Maverick™ from CIRRUS LOGIC

AN182

Application Note

WRITING CODE FOR ROM

Note: Cirrus Logic assumes no responsibility for the attached information which is provided “AS IS” without warranty of any kind (expressed or implied).

P.O. Box 17847, Austin, Texas 78760
(512) 445 7222 FAX: (512) 445 7581
http://www.cirrus.com

TABLE OF CONTENTS

LIST OF TABLES

1. INTRODUCTION

This application note will cover the steps required to build a binary image that can be loaded into FLASH memory for stand-alone, self-running applications. There are two examples provided. Each example includes the initialization code and a main C routine that also contains an interrupt service routine that was also written in C. The linker output is listed to illustrate how memory is allocated for these examples. Both examples use the internal SRAM for stack and data storage. The DRAM's are not used.

This application note assumes you are using the ARM® Tools, Version 2.5x. The concepts presented should be valid for other software development environments with some minor modifications. This application note should also include the project and source files for the examples.

2. THE STEPS TO BUILD A ROM

Follow these steps to build a ROM:

1) Determine your memory layout.
2) Write the C code
3) Link with an initialization program (such as `init.s`) that
    a) Sets up the processor mode
    b) Defines the stack area
    c) Performs some hardware initialization
    d) Loads the MMU tables and enable the MMU (if necessary)
    e) Defines entry points for your interrupt service routine and main C entry point
4) Set the Read-Only value to 0x0000 in the Linker. Set the Read-Write area to the first RAM location.
5) Edit the Project Template (Edit→Project Template) and double-click on Link. Add the following in the Command Line window:

    `<fromelf> -nozeropad <$projectname>.axf -bin <$projectname>.rom.`

    This action invokes the `fromelf` program which converts the ARM ELF file to a true binary file. This also creates a file with the `.ROM` extension which is the file that is programmed into flash.
6) Download the `.ROM` binary image to the system using the `DOWNLOAD.EXE` program.

It is a good idea to carefully look at the settings in the project files that are referenced in this Application Note to fully comprehend the procedure.

When the examples are compiled and downloaded correctly, the heartbeat LED will flash once a second for five times then enter the standby state. Pressing the WAKEUP button starts the processor and flashes the LED five more times, and then goes to standby. This continues forever.

3. DEBUGGING AN INTERRUPT

Debugging an interrupt is a bit tricky. In both the ARMulator and in Angel, an interrupt can be simulated by doing the following in the debugger:

1) Enable Interleaving View in the Execution Window (this displays the actual assembly code) and addresses. Use Ctrl-I or Options→Toggle Interleaving.
2) In the register window, change `psr` to %xxxxxxx_IRQ32 (don't change the current settings of xxxxxxxx).
3) Change spur to`%xxxxxx_SVC32` (otherwise, won't return in Supervisor Mode).
4) Set r14 (the link register) to current value +4
5) Set PC to `0x18` (the IRQ vector)

Now you can single step through the handler and upon exiting, the original state will be restored.

4. SOFTWARE EXAMPLE #1

This first software example does not use the MMU. This results in a program with a very small footprint (<1 kByte). Adding an MMU does improve performance but adds an additional 16 K or more of program space to hold the Translation Tables. Without invoking the MMU, the memory map of a 72xx embedded controller in External Boot Mode is shown in Table 1.

The project file for this example is called ROMCode.apj. This project file has two variants: Debug and ROM. The debug variant allows you to simulate the code in the ARMulator. The ROM variant builds the ROM image. The C code example is called C_ROM.C. This routine defines some "do-nothing" memory, checks that the RAM memory allocated in an array is valid, enables a timer interrupt, then loops forever. The interrupt handler routine simply toggles the state of the heartbeat LED each time the interrupt is called. The program init.s is an assembly routine that sets up the processor mode, defines the stack area, sets the ZI (zero initialized RAM area) to zero, and defines the calling procedure for the interrupt service C routine and the main C routine.

The Linker symbol output is shown in Listing 1. Comments are placed along side the output to clarify the meanings.

Address	Contents	Size
0xF000.0000	Reserved	256 Mbytes
0xE000.0000	Reserved	256 Mbytes
0xD000.0000	DRAM Bank 1	256 Mbytes
0xC800.0000 0xC000.0000	16MB DRAM, Bank 0 16MB DRAM, Bank 0	256 Mbytes
0x8000.4000	Unused	~ 1 Gbyte
0x8000.0000	Internal Registers	16K bytes
0x7000.0000	Boot ROM (nCS7)	128 bytes
0x6000.0000	On-chip SRAM (nCS6)	38,400 (0x9600) bytes
0x5000.0000	Expansion--Evaluation Board Peripherals (nCS5)	4 x 64 Mbytes
0x4000.0000	Expansion--Evaluation Board Peripherals (nCS4)	4 x 64 Mbytes
0x3000.0000	Expansion--Evaluation Board Peripherals (nCS3)	256 Mbytes
0x2000.0000	Expansion--Evaluation Board Peripherals (nCS2)	256 Mbytes
0x1000:0000	NAND Flash ROM Bank 1 (nCS1)	256 Mbytes
0x0000:0000	NAND Flash ROM Bank 0 (nCS0)	256 Mbytes

Table 1. EP72xx/71xx in External Boot Mode Memory Map for Example #1

5. SOFTWARE EXAMPLE #2

This next software example is identical to the previous example except that the MMU is enabled. This requires that the source listing, mmu.s, be included in the project file. It also requires that the compiler flag “MMU_enabled” be defined. This is accomplished in the project options rather than in the source code. This allows one set of C and assembly source to be maintained.

The project file for this example is called ROM_CMMU.APJ. The file that defines the memory configuration is called “mmu.s.” It will define a memory map as shown in Table 2. Note that only 1 Mb is defined. This is about as small a configuration possible. Only one Level 1 item is needed. But since Level 1 data must be aligned on a 14-bit address, this sets the minimum size of the ROM image to 16K plus the size of the Level I table (1 word, or 4 bytes, in this example). The Level II tables resided on a 10-bit aligned (4 K) boundary. So there exists substantial “empty” space in the ROM image. With careful design, these unused regions could contain constant data or code.

The MMU defines the virtual address space as described in Table 2.

The Linker output with explanations is shown in Section 7.

6. CONCLUSION

This application note explains how one might design and build code that can be placed into ROM (or FLASH). Two examples were used. The first generated code that was 440 bytes in length but does not take advantage of the MMU.

The second example is the same code but with the MMU enabled and a minimal virtual memory configuration was established. The size of the code in this case, is larger (16,412 bytes). This is due to the restrictions on where the Level I Translation Program resides. It is generally recommended that the MMU be enabled as this also enables the cache that maximizes performance in more demanding applications, such as MP3 decoding.

Address Range	Function
0x0002C000 - 0x0002FFFF	EP7209 internal registers
0x0002B000 - 0x0002BFFF	Parallel port interface (nCS1)
0x0002A000 - 0x0002AFFF	NAND FLASH interface (nCS1)
0x00020000 - 0x00029FFF	40K of internal SRAM (only 37.5K exists)
0x00000000 - 0x0001FFFF	128K of program ROM (nCS0)

Table 2. MMU Virtual Address Space

7. ALTERNATE MEMORY MAP

The file `mmu_alt.s` offers an alternative memory map that does not use a Level 2 translation table. This avoids the code at 0x0400 which lies between the code and the Level 1 table at 0x4000. In most embedded cases where an operating system such as Linux or Windows® CE is not required, you don't need to worry about Level 2 at all. Using `mmu_alt.s`, the memory map in Table 3 is defined:

Note how compact this memory model is. The size of the Translation table is relatively small since there are few entries to define the first 5 Mb of memory (five Level 1 entries for a total of 20 bytes). As an exercise, try using this with the example program. Just make certain you change the code to reflect the different memory addresses.

Address Range		Purpose
0x0050.0000	0xFFFF.FFFF	Undefined. Accesses generate Data Abort
0x0040.0000	0x004F.FFFF	USB interface (nCS4)
0x0030.0000	0x003F.FFFF	NAND FLASH interface (nCS1)
0x0020.0000	0x002F.FFFF	EP7209 internal registers
0x0010.0000	0x001F.FFF	1Mb Internal SRAM (only 38.4K exists)
0x0000.0000	0x000F.FFFF	1Mb of program ROM (nCS0)

Table 3. mmu_alt.s Memory Map

8. LISTING OF LINKER SYMBOL TABLE OUTPUT FOR SOFTWARE EXAMPLE # 1

The following information is a formatted version of the output produced by Software Example #1. The symbol table for Software Example #1 is shown in Table 4.

Linker Output	Units	Explanation
C$$code$$Base	000000	This is the starting location. Defined in Linker Read-Only
C$$code$$Limit	000224	Size of the C program.
CStack$$zidata$$Base	60000108	Beginning of zero-initialized RAM. Includes area for stack.
CStack$$zidata$$Limit	60000608	Top of Stack
C$$code	000000	Defined in init.s
InterruptHandler	0000ec	Location of InterruptHandler C routine
Image$$RO$$Limit	000238	Total size of ROM image
Image$$RW$$Base	60000000	Start of RAM. Defined in Linker Read-Write
Image$$ZI$$Base	60000008	Start of zero-initialized RAM (heap)
Image$$RW$$Limit	60000608	End of RAM used. Includes stack
C_entry	000148	Location of main C routine
CStack$$zidata	60000108	Bottom of stack
C$$constdata$$Base	000224	Start of storage for constant values such as “hex”
C$$constdata$$Limit	000138	End of constant storage
C$$data$$Base	60000000	Start location for C data storage. This is the static int “temp” which is 4 bytes in length
C$$data$$Limit	60000008	End of data. Only “temp” was declared.
C$$zidata$$Base	60000008	Start of the heap area used in the example, char array[256]
C$$zidata$$Limit	60000108	End of heap, 256 bytes.
hex	000224	Location where the const hex value is stored. It is stored at the end of the image. Adding 0x14 (20) gives a total of 0x1bc (same as Image$$RO$$Limit) and is the total size of the binary image (444 bytes)
j	60000000	Location of the volatile global variable “j”
Array	60000008	Location for start of array
Image$$RO$$Base	000000	Beginning address of the ROM image
Image$$ZI$$Limit	60000608	Size of RAM used.

Table 4. Symbol Table for Software Example #1

8.1 Summary of Memory Used

A summary of memory used in Software Example #1 is shown in Table 5:

Base	Size	Type	Read Only (RO) / Read Write (RW)	Name
0	ec	CODE	RO	C$$code from object file `init.o`
ec	138	CODE	RO	C$$code from object file `C_ROM.o`
224	14	DATA	RO	C$$constdata from object file `C_ROM.o`
60000000	8	DATA	RW	C$$data from object file `C_ROM.o`
60000008	100	ZERO	RW	C$$zidata from object file `C_ROM.o`
60000108	500	ZERO	RW	CStack$$zidata from object file `init.o`

Table 5. Software Example #1 Memory Use

8.2 Description of Results for Software Example #1

Image entry point: 0

Entry area: "C$$code" from object file `init.o`

	Code Size	Inline Data	Inline Strings	“const” data	RW Data	0-Init Data	Debug Data
Object Totals	512	36	0	20	8	1536	0

Table 6. Characteristics of Example #1

9. LISTING OF LINKER SYMBOL TABLE OUTPUT FOR SOFTWARE EXAMPLE # 2

The following information in this section is a formatted version of the output produced by Software Example #2.

Linker Output	Units	Explanation
Assembly$$PageTable$$Base	004000	This is the location for the first item in the Level 1 Translation Table
Assembly$$PageTable$$Limit	000668	Size of Level 1 Table
Assembly$$PageTable	000400	This is the location of the first item in the Level II Translation Table.
C$$code$$Base	000000	Start location
C$$code$$Limit	000258	Size of code
CStack$$zidata$$Base	020008	Start of ZI data in RAM
CStack$$zidata$$Limit	020108	Upper limit of ZI data in RAM
C$$code	000000	Defined in init.s
InterruptHandler	000120	Location of InterruptHandler C routine
Image$$RO$$Limit	004018	Total size of ROM image
Image$$RW$$Base	020000	Start of RAM. Defined in Linker Read-Write
Image$$ZI$$Base	020008	Start of zero-initialized RAM (heap)
Image$$RW$$Limit	020608	End of RAM used. Includes stack
C_entry	00017c	Location of main C routine
CStack$$zidata	020108	Bottom of stack
C$$constdata$$Base	004004	Start of storage for constant values such as “hex”
C$$constdata$$Limit	004018	End of constant storage. Size is 0x14 or 20.
C$$data$$Base	020000	Start location for C data storage. This is the static int “temp” and volatile j, both of which are 4 bytes in length
C$$data$$Limit	020008	End of data. 8 bytes for two values.
C$$zidata$$Base	020008	Start of the heap area used in the example, char array[256]
C$$zidata$$Limit	020108	End of heap, 256 bytes.
j	020000	Location of volatile variable j

Table 7. Symbol Table for Software Example #2

hex	004004	Location where the const hex value is stored. Note that it appears after the Level I entry. Adding 20 to 0x4004 gives 0x401C (16,412) which is the size of the binary image.
Array	020008	Location for start of array
Image$$RO$$Base	000000	Beginning address of the ROM image
Image$$ZI$$Limit	020608	Size of RAM used.

Table 7. Symbol Table for Software Example #2 (Continued)

9.1 Summary of Memory Used

A summary of memory used in Software Example #2 is found in Table 8.

Base	Size	Type	Read Only (RO) / Read Write (RW)	Name
0	120	CODE	RO	C$$code from object file init.o
120	138	CODE	RO	C$$code from object file C_ROM.o
258	1a8	PAD	RO	Assembly$$PageTable from object file Mmu.o
400	c0	DATA	RO	Assembly$$PageTable from object file Mmu.o
4c0	3b40	PAD	RO	Assembly$$PageTable_ from object file Mmu.o
4000	4	DATA	RO	Assembly$$PageTable_ from object file Mmu.o
4004	14	DATA	RO	C$$constdata from object file C_ROM.o
20000	4	DATA	RW	C$$data from object file C_ROM.o
20004	100	ZERO	RW	C$$zidata from object file C_ROM.o
20104	500	ZERO	RW	CStack$$zidata from object file init.o

Table 8. Software Example #2 Memory Use

9.2 Description of Results for Software Example #2

The characteristics of Software Example 2 are described in Table 9.

Image entry point: 0

Entry area: "C$$code" from object file init.o:

	Code Size	Inline Data	Inline Strings	"const" data	RW Data	0-Init Data	Debug Data
Object Totals	556	44	0	216	8	1536	0

Table 9. Characteristics of Example #2

• Notes •


Maverick™
from
CIRRUS LOGIC