XAPP642 (v1.0) October 21, 2002

Relocating Code and Data for Embedded Systems

Author: Kraig Lund

Summary

This application note describes a method for building a ROM firmware image residing in one location of memory and executing from/in another location. The examples given in this application note use the widely available GNU tools targeted for the PowerPC™ processor.

Introduction

Embedded systems and general-purpose computers differ in many ways. One key difference is how programs are loaded and executed out of memory. Software written on personal computers are typically compiled into a completely relocatable format with no absolute addresses embedded in the instructions. This file is archived on a disk. To access the program, the operating system finds available memory, the loader (an operating system function) retrieves the program, inserts all of the necessary absolute addresses for the program, and the program is put into available memory. This process is called dynamic linking, because address resolution occurs dynamically at runtime.

However, many embedded systems programs are written in a different manner. First, the program is written on a host machine for a target system. The host machine and the target system can be very different. Second, the program is not stored on a file system or disk. A code image is built on the host system containing all the necessary addresses embedded in the program, and is then downloaded to non-volatile read-only memory (FLASH or EEPROM). In the end application, the embedded system runs independently. This type of linking is called static linking because address resolution occurs before the program is ever run.

Embedded programs contain sections that are read/write in nature; however, these sections are typically initialized and stored in read-only memory (FLASH or EEPROM). At boot time these items need to be copied to read/write memory before they can be used. Additionally, executable program code is typically stored in read-only memory, however, it is sometimes desirable to execute this code from a faster read-write memory e.g., SRAM. Therefore, the code needs to be copied from ROM to RAM at boot time before it is executed. This process is generally known as relocation. Figure 1 illustrates this concept.

© 2002 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and further disclaimers are as listed at http://www.xilinx.com/legal.htm. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Xilinx is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Xilinx makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

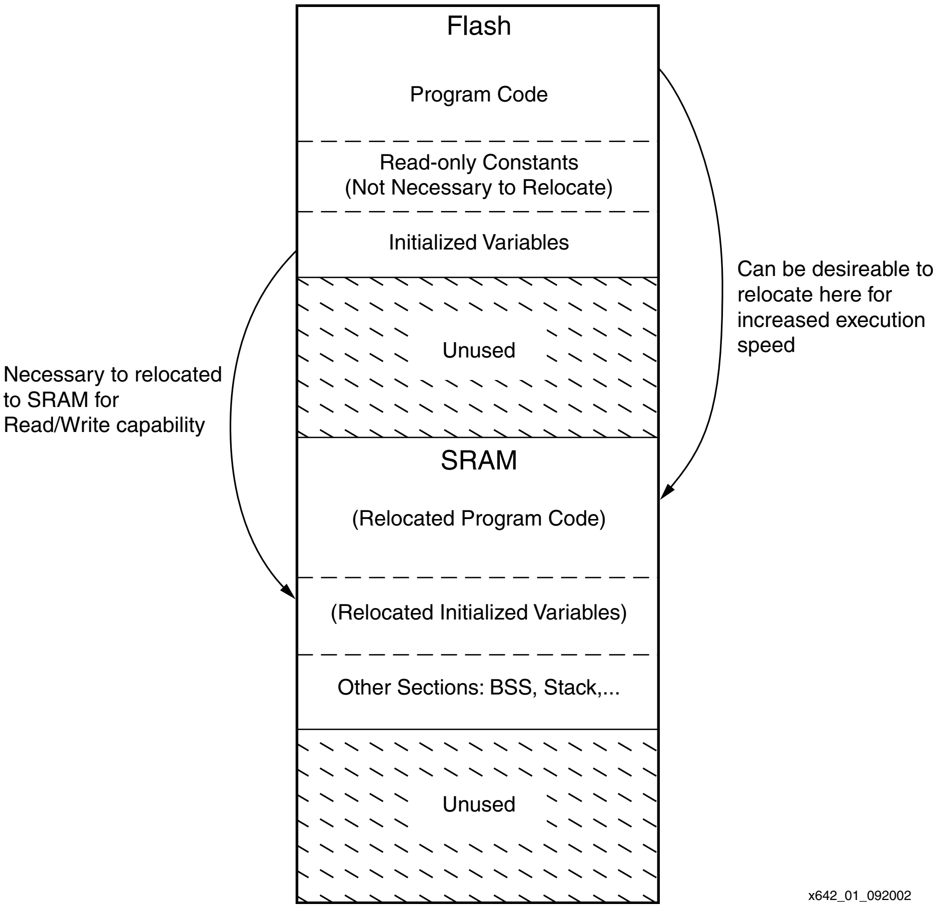

Figure 1: **Memory Map**

This application note describes a method to relocate both code and data from non-volatile ROM into read-write memory using the widely available GNU tools targeting the PowerPC processor.

Compilation and Address Resolution

This section discusses the compilation and linking process for embedded systems. When functions are written and compiled there are usually calls within those functions to other external functions or data entities. Those other functions and data entities can be located in a location unknown to the compiler during the compilation process. When this happens, the compiler inserts a placeholder for the unknown memory address. An example would be a C language "int" variable defined in one file but is accessed in the file being compiled. During compilation, the compiler will generate a "load from memory" instruction in order to load the variable from memory into one of the processors registers. This requires the address of the stored "int" variable. Since the compiler does not yet know the location of the stored variable in memory, a placeholder is used for the address.

The purpose of the linker is to determine the addresses. This is called address resolution. The linker takes all of the separately compiled files, locates them in user defined locations in memory, and then fills in the placeholders. Typically the user defines these locations in a "linker script" file, sometimes known as a "mapfile". The linker script tells the linker how to build the executable program.

During compilation, the GNU compiler GCC creates many different types of code, called segments. Some segments are built-in types and others are user definable. Some of the common built-in types of segments are:

BSS: Uninitialized Data (Read/Write)
DATA: Initialized Data (Read/Write)
RODATA: Constants (Read Only)
TEXT: Program Code (Read Only)

A simple C program shows how different entities can be segmented in the compilation process.

```
#include <stdio.h>
  int array_data[10] =    {0,1,2,3,4,5,6,7,8,9};    // DATA
  const int array_const[10] = {9,8,7,6,5,4,3,2,1,0};// RODATA

  int main(){
    int i;                         //BSS

    i = i + 10;                    //TEXT
    printf("%d\n", array_data[0]); //TEXT should print 0 followed by newline
    array_data[0] =  array_const[0];//TEXT
    printf("%d\n",array_data[0]);  //TEXT should print 9 followed by newline

return 0;
}
```

The problem of relocation now becomes obvious. The previous example contains four different segments; some are Read/Write and some Read only. If the linker locates each initialized segment in ROM, as is typically necessary in most embedded systems, the program will not work. The statement "`array_data[0] = array_const[0];`" breaks the system because data can not be written to a ROM. A method is needed to have the initialized values of the array `array_data[]` initially reside in ROM, and then be relocated to RAM at run time.

Example 1

Relocating Data

Sometimes it is necessary to relocate data, instructions, or both. The simplest case is to relocate data. This first example relocates a simple initialized piece of data from one location in memory to another. The following listings are code snippets showing the main functions necessary to relocated data. The complete code listing is included in the file XAPP642.zip on the Xilinx FTP site at:

ftp://ftp.xilinx.com/pub/applications/xapp/xapp642.zip

Code Listing for Example 1

The following snippet is from the boot code located in *crt0.S*. It shows the function calls to copy data and main.

--snip

```
/* set up initial stack frame */
  addi 1,1,-8/* location of back chain */
  lis  0,0
  stw  0,0(1)/* set end of back chain */



      bl    CopyData           /* copy the data from flash to ram */
      bl    main
```

--end snip

The function CopyData is located in the *copy.c* file.

```
extern int  _etext, _data, _edata;
void CopyData(void){

    int *src = &_etext; // pointer to beginning of DATA, end of TEXT
    int *dst = &_data;  // pointer to where we want to copy DATA

    while (dst < &_edata) {
        *dst++ = *src++;   // copy the data from its physical address
    }                      // to its logical address
 }
```

Main() is located in the *datarelo.c* file.

```
int myarray[10] = {0,1,2,3,4,5,6,7,8,9}; // create an initialized array
                                         // which we will relocate
int main()
{
     int i;

     while(1){                            // do something with the array
       for(i=0; i<10; i++){
         myarray[i] = i + 10;}
     }
     return 0;
}/*end of main*/
```

A simplified version of the linker script, the following file is called *mapfile*.

```
STACKSIZE = 4k;
MEMORY
{
  ddr   : ORIGIN = 0x00000000, LENGTH = 32m
  sram  : ORIGIN = 0x10000000, LENGTH = 2m
  flash : ORIGIN = 0xFE000000, LENGTH = 32m - 4
  boot  : ORIGIN = 0xFFFFFFFC, LENGTH = 4
}

ENTRY(_boot)
SECTIONS
{
  .text :
  {
   _text = . ;
    *(.text)
   _etext = . ;
  } > flash

.data  :
   AT ( ADDR (.text) + SIZEOF (.text) )
  {

   /* since the code uses type "int" to copy the data
      make sure that the data section is aligned on an
      integer (4-byte) boundary                        */

    . = ALIGN(4);
   _data = . ;
     *(.data)
     *(.COMMON);
   _edata = . ;
    . = ALIGN(4);

  } > sram

       .
       .
       .

    .boot    :
  {
    *(.boot)
  } > boot
}
```

Explanation of Example 1

To understand the linking step, linker script format is examined here. The MEMORY section describes the system memory map. In the SECTIONS section, the code segments are linked and located to different portions of the memory map. The following statement locates all text (program code) as denoted by the following code snippet into the FLASH memory region. It also provides a constant, "_etext", for use by the C function CopyData, the address of the end of the text segment in memory.

```
.text :     /* Name of the "output section */
  {
    _text = . ;
      *(.text)
    _etext = . ;
  } > flash
```

The linker script describes where data goes. This segment will be relocated. To understand how the linker works, two addresses for this piece of code are defined; a physical address and a logical address. The physical address is the memory address where code is loaded, in this case it is loaded into a non-volatile memory. A logical address is sometimes known as a virtual address or a run-time address. This type of virtual address should not be confused with the PowerPC Memory Management Unit (MMU) and virtual addressing mode. The logical address is the location where code is executed from, in this case SRAM. The physical address is defined by using the AT command. The statement AT (`ADDR (.text) + SIZEOF (.text)`) loads the data segment following the text segment located in the FLASH memory in this system. The logical address annotated by the "`> sram`" statement informs the linker that the .data section will eventually reside in SRAM. Therefore, all references in the program code should address "data" in the SRAM, not the FLASH.

```
  .data  :
    AT ( ADDR (.text) + SIZEOF (.text) )
  {

    /* since the code uses type "int" to copy the data make sure that the data
    section is aligned on an integer (4-byte) boundary*/

  . = ALIGN(4);
  _data = . ;
    *(.data)
    *(.COMMON);
  _edata = . ;
  . = ALIGN(4);
   } > sram
```

The programmer ensures that the initial values of the data is copied from the FLASH to the SRAM before it is ever used. In this example, this is done by calling the function CopyData before executing the statement array_data[0] = array_const[0]; instruction.

In the following code, the function CopyData is examined to determine how it works.

```
extern int  _etext, _data, _edata;
void CopyData(void){

    int *src = &_etext; // pointer to beginning of DATA, end of TEXT
    int *dst = &_data;  // pointer to where we want to copy DATA to
    while (dst < &_edata) {
        *dst++ = *src++;   // copy the data from its physical address
    }                      // to its logical address
 }
```

Three variables are defined in the linker script used by the function to copy the data from the FLASH to the SRAM: _etext, _data, and _edata.

_etext

End physical and logical address of the text segment. Also, it is the beginning of the data section's physical address. (Note: physical = logical address in the absence of the AT command)

_data

Beginning logical address of the .data segment.

_edata

End logical address of the .data segment.

Two pointers, src and dst are initialized to point to the addresses _etext and _data respectively. The following statement dereferences these pointers and copies the data from _etext to _data until the address _edata is reached.

```
while (dst < &_edata) {
        *dst++ = *src++;   // copy the data from its physical address
    }                      // to its logical address
```

Example 2

Relocating Instructions and Data

This example relocates both instructions and data to SRAM. This is a slightly more complicated procedure than just relocating data. If the program code is to be relocated, how can it be executed before it is relocated? By having the boot code located in a non-relocated section of ROM, the boot code relocates the application software.

In this example the boot code calls the function CopyText(), that copies all of the code to RAM except the boot code. After CopyText is finished copying the application, the boot code calls CopyData(). After the relocation of text and data is finished, the boot code finally calls main(). The code listing for the boot code and CopyText() is shown below. CopyData() and main() are the same as in the previous example.

Code Listing for Example 2

The following snippet is from the boot code located in *crt0.S*. This snippet shows the function calls to CopyText, CopyData, and main.

```
-----------------------------snip
/* set up initial stack frame */
  addi 1,1,-8/* location of back chain */
  lis  0,0
  stw  0,0(1)‘/* set end of back chain */

/* call CopyText to relocate the program code */
  bl  CopyText

/* call CopyData; must use a long call since CopyData is located in RAM */
  lis  30,CopyData@h    /* load upper bits of r30 w/ address of CopyData */
  ori  30,30,CopyData@l /* load lower bits of r30 w/ address of CopyData */
  mtlr 30               /* move address of CopyData to LR */
  blrl /* branch to contents of LR and update LR with current instruction  */
       /* address + 4  */

/* call main; must use a long call since main is located in RAM */
  lis  30,main@h
  ori  30,30,main@l
  mtlr 30
  blrl
-----------------------------end snip
```

The function CopyText is located in the *copy.c* file called.

```
void CopyText(void)  __attribute__ (( section(".rom")));

extern int _etext,  _text, __ltext;

void CopyText(void){
    int *src = &__ltext;  // pointer to the physical address of TEXT
    int *dst = &_text;    // pointer to the logical address of TEXT

    while (dst < &_etext) {
        *dst++ = *src++;
    }
 main();
}
```

Explanation of Example 2

The function prototype tells the linker to map the input section to *.rom* rather than the default *.text* using an attribute. Every embedded compiler has a different type of *__attribute__ function*. Some compilers use the *#pragma* keyword to accomplish the same thing.

A simplified version of the Linker Script for this example follows:

```
STACKSIZE       = 4k;
MEMORY
{
  ddr   : ORIGIN = 0x00000000, LENGTH = 32m
  sram  : ORIGIN = 0x10000000, LENGTH = 2m
  flash : ORIGIN = 0xFE000000, LENGTH = 32m - 4
  boot  : ORIGIN = 0xFFFFFFFC, LENGTH = 4
}
ENTRY(_boot)
SECTIONS
{
  .rom :
  {
  *(.rom)
  } > flash
  .text :
  AT ( ADDR(.rom) + SIZEOF(.rom) )
  {
  _text = . ;
  *(.text)
  _etext = . ;
  } > sram
__ltext = LOADADDR(.text);
__letext = LOADADDR(.text) + SIZEOF(.text);
.data  :
  AT ( ADDR (.rom) + SIZEOF (.rom) + SIZEOF(.text) )
  {

  /* since the code uses type "int" to copy the data make sure that the data
section is aligned on an integer (4-byte) boundary                       */

  . = ALIGN(4);
  _data = . ;
    *(.data)
    *(.COMMON)
_edata = . ;
  . = ALIGN(4);
  } > sram


    .
    .
    .

  .boot    :
  {
    *(.boot)
  } > boot
}
```

In Example 2, the AT (ADDR(`.rom`) + SIZEOF(`.rom`)) loads the `.text` section directly after the `.rom` section into the FLASH memory. The logical address is, however, located in the SRAM as indicated by the "`> sram`" statement. Similarly the data section is loaded directly after the `.rom` and `.text` sections in the FLASH as is indicated by the AT (ADDR (`.rom`) + SIZEOF (`.rom`) + SIZEOF(`.text`)). Again the logical address is in the SRAM as indicated by the "`> sram`" notation.

Debugging the Examples

Another useful tool for the embedded programmer is the "dump" utility. This tool can be used to extract useful information from object files, both linked and unlinked. For instance, it can be used to disassemble an object file or to display program header information. The code from Example 2 and the GNU dump utility (OBJDUMP) is used to display information about the different sections of code. For each section a VMA and LMA are given. These acronyms correspond to logical address (VMA) and physical address (LMA). The following listing was generated using the OBJDUMP tool from the command line:

"`powerpc-eabi-objdump -h textrelo.elf`".

The "`-h`" tells OBJDUMP to dump all of the section header information.

```
Sections:
Idx Name           Size      VMA       LMA       File off  Algn
  0 .rom           000000f8  fe000000  fe000000  00010000  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text          000000e4  10000000  fe0000f8  00020000  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data          00000028  100000e4  fe0001dc  000200e4  2**2
                   CONTENTS, ALLOC, LOAD, DATA
  3 .fdata         0000003c  1000010c  1000010c  0000010c  2**2
                   CONTENTS, ALLOC, LOAD, DATA
  4 .sdata         00000000  10000148  10000148  00000148  2**2
                   CONTENTS, ALLOC, LOAD, DATA
  5 .sdata2        00000000  10000148  10000148  00000148  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .sbss          00000000  10000148  10000148  00030000  2**0
                   CONTENTS
  7 .bss           00001008  10000148  10000148  00000148  2**0
                   ALLOC
  8 .boot          00000004  fffffffc  fffffffc  0002fffc  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .comment       00000068  00000000  00000000  00030000  2**0
                   CONTENTS, READONLY
```

The previous listing shows the logical address (VMA) of the .text section is `0x10000000` and the physical address (LMA) is `0xfe0000f8`. Likewise for the .data section, the logical address is `0x100000e4` and the physical address is `0xfe0001dc`. The size of each section is also given for easy verification of linker script instruction completion.

The linker script tells the linker to physically locate .data with the following command:

```
AT ( ADDR (.rom) + SIZEOF (.rom) + SIZEOF(.text) )
```

From the previous listing it is possible to calculate and verify that the linker physically located the .data section in the right location.

```
ADDR(.rom)              0xfe000000

SIZEOF(.rom)     +      0x000000f8

SIZEOF(.text)    +      0x000000e4

                 =      0xfe0001dc
```

The calculated address, `0xfe0001dc` is equal to the LMA reported by OBJDUMP for the `.data` section.

There are several other methods to verify addresses. One method is to use the "`--print-map`" in the linker command line. Another option is to use GNU's *NM* tool to provide the information.

Conclusion

It is usually both desirable and necessary in embedded systems to have program code and data located physically in one memory-mapped address but run from entirely different addresses. Embedded tool chains provide this functionality. Embedded linkers allow the programmer to define two addresses for each section of code. The physical address defines where the code is physically loaded into the non-volatile memory. The logical address defines where the code eventually resides. The programmer provides the functionality to copy the code from it's a physical to a logical address during the boot sequence. The examples in this application note show how this is done for the GNU toolset. Other toolsets may differ in syntax, however, the concepts are the same.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/21/02	1.0	Initial Xilinx release.