Chapter 5
The Linker

Objectives

This chapter introduces the operation of the linker and gives some specifics about the CodeWarrior® Smart Linker.

5.1 Introduction

> A *linker program* combines *object* files to produce an *absolute* file to be loaded into the microcontroller's memory.

In this chapter we investigate the operation of a program used to link together separately assembled and compiled source files into one file. The *linker* produces an output file to be loaded into the microcontroller using the correct types of memory for each portion of the program.

5.2 Assumptions

In this chapter we assume you are using the CodeWarrior® tool set and that you will be running the linker from within the CodeWarrior Integrated Development Environment (IDE). Further, at least until we get to Chapter 8 where we consider C programs, we assume you are developing assembly language programs. In the following sections we describe how the linker locates each of the various parts of an assembly language program into the proper memory.

5.3 Why Use a Relocatable Assembler/Compiler and a Linker?

The relocatable assembler and compiler allow efficient and effective program development using modular programming methodologies much like you use for complex C++ or other high level language programs. You may develop functional modules separately, and independently, from one another and then to *link* them to *build* the final application to be loaded into the microcontroller's memory. Thus a large project can be allocated to different members of the development team. The linker also allows you to have local labels and to expose only a smaller subset of labels to the global program and other modules. Imagine the problems you could have with duplicate labels if you took many modules built by separate groups and tried to assemble them in one giant assembly language file of many thousand lines.

5.4 Memory Types, Sections, and Section Types

In Chapter 2 we described the operation of the relocatable assembler and compiler. We also described the different types of memory, RAM and ROM, and showed that each type is used for different purposes in our embedded system programs. When using the relocatable assembler, the placement of the various parts of our programs in the different memory is called *locating*. That is, we must locate the different parts of the program in the proper memory.

Another task to be done is linking together the separate modules. The need for this arises when code in one module calls or refers to code or data elements in another. This process is also achieved by the linker.

Every program, whether written in assembly language or C, will have various *segments* or *sections*.[1] These are the *code section*, *constant data section*, *variable data section*, and *stack section*.

Code Section

The code section contains all of the program code. The program MUST be located in read-only memory in an embedded system. In a development environment, where you may have a *development system* or an *evaluation board (EVB)*, you may be place your program and constants into RAM. This makes it easier to change and modify as you are developing and debugging your software. Ultimately, however, you will change the physical location of the code section to ROM for the embedded application.

> All parts of the program that must remain when power is turned off, including the *code* and *constant data*, are placed in the code section which is then located in ROM memory.

Constant Data Section

> Constants are to be located in ROM too.

Constant data elements are located also in ROM memory. Examples of constant data include variable data initialization values and messages to be printed.

Variable Data Section

The variable data section contains the storage locations for all variable data used in the program. There are several points to make about variable data storage used in a program:

> The *variable data* section is located in RAM memory.

- All variable data storage locations must be allocated in the program by the assembler or compiler using a define storage (DS) directive.
- All variable data storage must be located in RAM memory.
- The program code must initialize all variable data must be initialized, if needed, when the processor is running. You MUST not assume any data value will be in a RAM data storage location when the system is power on.

Stack Section

As we will see in later chapters, the stack is an allocation of RAM used for temporary information storage. This could be data saved temporarily or return addresses for subroutines. The stack pointer register must be initialized pointing to RAM before the stack can be used.

> The *stack* section is located also in RAM memory.

---

[1] We will generally use the term section, although you will see segment used in some of the CodeWarrior files. The two terms are synonymous.

5.5 Linker Operation

Locating in Proper Memory

Before we show how to set up the linker to create your program to be loaded into the microcontroller memory, let us have a look at what at what it is supposed to do.

Example 5-1 and Example 5-2 show the *main* and *subroutine* programs for two separately assembled modules. This program does not do anything sensible so we will not worry particularly about the assembly language code.

Main Module

In the main program, Example 5-1, we see there are four distinct uses of memory. The order in which they appear is not important although we will see some programming style suggestions in Chapter 7. The first is a variable data section shown between lines 15 – 17. A memory allocation for a single byte of data storage is made at line 17 by the assembler directive

```
main_data:  DS.B 1.
```

The label `main_data` marks a single byte of storage and will be assigned an address during the linking process. Any other *variable* data storage you might need must be defined in this section or other named sections that can be located in RAM. The section continues until another `SECTION` directive or something other than the define storage (`DS`) directive is found. Everything in this section must be located in RAM memory.

The next memory use is found starting at line 20. This is the *code section* and it starts with the directive

```
MainCode:   SECTION.
```

Everything in this section, from line 20 to the end of the program is to be located in ROM memory. You should choose the label `MainCode` to be something meaningful so you can see where it is located when you view the linker map file.

A *constant data section*, `MainConst: SECTION`, is defined at line 30 and a constant byte is defined by

```
main_const: DC.B $44
```

at line 33. Because constants go into the same type of memory as the code, you could simply define the bytes following the program code. However, it is far better to define a constant section to allow all constants to be grouped together in the linking process or to locate them in a different part of the ROM than the code.

A fourth memory use is implied in line 12 where the label `__SEG_END_SSTACK`[2] is defined to be external. The value of this label will be defined by the linker and is used with the stack pointer initialization instruction at line 22 in the main program. As we will learn later, the stack pointer register must be initialized before calling any subroutines. Also, the stack itself must be located in RAM memory.

[2] This label starts with two underbar (_) characters.

Example 5-1 Relocatable "Main" Program

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1
    2    2                      ; Rel_ex_1a_main.asm
    3    3                      ; The main module in a relocatable example.
    4    4                      ; This program does nothing that is sensible.
    5    5                      ; FM Cady 7/04
    6    6                      ; *****************************************
    7    7                      ; Define entry point for the main program.
    8    8                                  XDEF  Entry
    9    9                      ; Define symbols used in the subroutine.
   10   10                                  XREF  sub_1
   11   11                      ; *****************************************
   12   12                      ; Stack section
   13   13                                  XREF  __SEG_END_SSTACK
   14   14                      ; *****************************************
   15   15                      ; Variable Data Section
   16   16                      MainData:   SECTION
   17   17   000000             main_data:  DS.B 1
   18   18                      ; *****************************************
   19   19                      ; code section
   20   20                      MainCode:   SECTION
   21   21                      Entry:
   22   22   000000 CFxx xx                 lds   #__SEG_END_SSTACK
   23   23                      ; Initialize the variable main_data.
   24   24   000003 180C xxxx               movb  main_const,main_data
             000007 xxxx
   25   25   000009 B6xx xx     loop:       ldaa  main_data ; Get the data.
   26   26   00000C 16xx xx                 jsr   sub_1     ; Pass to sub
   27   27                      ; Increment data for the next loop.
   28   28   00000F 72xx xx                 inc   main_data
   29   29   000012 06xx xx                 jmp   loop    ; Continue forever.
   30   30                      ; *****************************************
   31   31                      MainConst:  SECTION
   32   32                      ; Define constants in the code section
   33   33   000000 44          main_const: DC.B  $44
```

Subroutine Module

Let us now look at how memory is used in the subroutine shown in Example 5-2. There are three memory uses in this example. There is a section of code starting at line 16 and continuing through line 23. We see a constant data section at line 26 and a variable data section at line 31.

Example 5-2 Relocatable Subroutine

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ; rel_ex_1a_sub_1.asm
    2    2                      ; Relocatable subroutine example with
    3    3                      ; variable and constant data.
    4    4                      ; FM Cady 7/04
    5    5                      ; Input Registers:
    6    6                      ;   A = some input data
    7    7                      ; Output Registers:
    8    8                      ;   A = some output data
    9    9                      ; Registers modified:
   10   10                      ;  A, CCR
   11   11                      ; ****************************************
   12   12                      ; Define symbols used in the subroutine.
   13   13                                XDEF  sub_1
   14   14                      ; ****************************************
   15   15                      ; Code Section
   16   16                      SubCode:  SECTION
   17   17                      sub_1:
   18   18                      ; On entry, the data is in Accumulator A.
   19   19                      ; Store the data in the subroutine.
   20   20   000000 7Axx xx               staa  sub_data
   21   21                      ; Get some data to return to the main.
   22   22   000003 B6xx xx               ldaa  sub_const
   23   23   000006 3D                    rts       ; Return to main program.
   24   24                      ; ****************************************
   25   25                      ; Define constant data.
   26   26                      SubConst: SECTION
   27   27   000000 33          sub_const:DC.B  $33
   28   28                      ; ****************************************
   29   29                      ; Variable Data Section.
   30   30                      SubData:  SECTION
   31   31   000000             sub_data: DS.B  1
```

Memory Needs

Our analysis of memory requirements for the main and subroutine is summarized in Table 5-1. Notice that we have included an estimate of the size of the stack needed for this program. We will defer how we find this out to a later chapter.

Table 5-1 Program Memory Requirements

Module	ROM		RAM	
Main	Code	21 bytes	Data	1 byte
	Constant Data	1 byte	Stack	2 bytes
Subroutine	Code	7 bytes	Data	1 byte
	Constant Data	1 byte		

The Hardware Memory Map

As programmers, now we know how much ROM and RAM our program will require. As embedded system engineers, we must look at the hardware upon which our program is going to be installed. We do that by looking at the *memory map* of the system. In a single-chip microcontroller, such as we described in Chapter 2, the manufacturer determines the memory map when the chip is designed and manufactured. Figure 5-1 shows a MC9S12C32 microcontroller operating in *single*-chip mode in an embedded system with no external RAM memory.

> Software engineers must know the system memory map to be able to load their code into the microcontroller's memory.

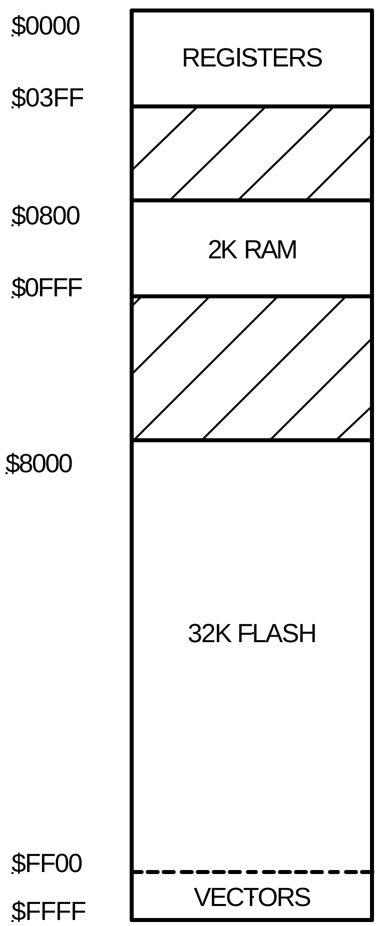

Figure 5-1 Memory Map for MC9S12C32 Microcontroller

Linking the Code

Before we show how to instruct the linker to locate our code correctly, we need to discuss the linking part of this job. Linking refers to establishing the final address for certain instructions, such as a *jsr – jump-to-subroutine*, when the starting address of the subroutine is in another module. This address resolution must be performed also when one module refers to a data location defined in another module. Line 26 in Example 5-1 has a `jsr sub_1` instruction where `sub_1` is in the second, separately assembled module. The assembler has three directives that organize the information needed to accomplish this linking task.

XREF, XDEF, and XREFB

When a symbol is used in one module but is defined in another, it is said to be an *external* symbol. In this case, the *XREF – External Reference* directive is used. In the module where the symbol is defined, the *XDEF – External Definition* directive is used. In Example 5-1 the symbol `sub_1` is the address for the jump-to-subroutine instruction on line 26. Because that symbol is not defined in the main module, it is declared to be external by `XREF sub_1` on line 10. Similarly, the `sub_1` symbol must be made visible in the subroutine module. This is done by the `XDEF sub_1` on line 13. For every XREF in our program modules there must be a partner XDEF in some other module.[3]

> Every XREF must have a companion XDEF in some other module.

Example 5-1 shows `XDEF Entry` at line 8. This makes the code entry point (where the E`ntry` label is at line 21) visible to the linker.

There is another directive, *XREFB – External Reference for Symbols on the Direct Page* included in the assembler directives. This is used when an external symbol is on the direct addressing page and thus fewer addressing bits are required.

5.6 The Linker Parameter File

The linker is controlled by a *linker parameter file (*.prm)*. You put the memory map information shown in Figure 5-1 in the linker parameter file so the linker can locate each section properly. Example 5-3 shows a linker parameter file used to link (and locate) the two modules in Example 5-1 and Example 5-2.

> The linker parameter file allows us to specify easily where code and data are located.

Example 5-3 Linker Parameter File

```
NAMES
END

SECTIONS
    RAM = READ_WRITE 0x0800 TO 0x0FFF;
/* unbanked FLASH ROM */
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
END

PLACEMENT
/* Place the code and constant sections in ROM */
    .init, MainCode, SubCode,MainConst, SubConst,
    DEFAULT_ROM  INTO  ROM_C000;
/* Place the Variable Data */
    MyData, MySubData,DEFAULT_RAM  INTO  RAM;
END
/* Define the stack size needed */
STACKSIZE 0x100
/* Specify the initial reset vector entry point */
VECTOR 0 Entry  /* reset vector */
/* INIT is needed for assembly applications */
INIT Entry
```

[3] If you forget to declare a symbol as external with XREF, the assembler will give you an error message. If you forget the XDEF, the linker will generate an error message.

Parameter File Commands

Although the CodeWarrior linker is very powerful with many features, we will describe only those few necessary to link assembly and C program modules. Keywords and commands in the parameter file must be capitalized. Each section in the parameter file starts with the command name in capitals and ends with the `END`. All commands and qualifiers are case sensitive.

Names:

The *NAMES* command lists the files building the application. For an application linked in the CodeWarrior tool set, the files that have been "added" to the application are automatically linked. Other modules may be linked by including their names in the NAMES section.

Sections:

```
SECTIONS
    RAM = READ_WRITE 0x0800 TO 0x0FFF;
/* unbanked FLASH ROM */
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
END
```

The *SECTIONS* part defines the memory map for your application. It allows you to assign meaningful names to address ranges. These can be used in the subsequent placement section to increase the readability of your parameter file. The syntax of a SECTIONS entry is the following:

`<your_name> = MEMORY_TYPE  address TO address;`

where

`<your_name>` is the name you wish to use for the particular address range, and
`MEMORY_TYPE` defines the type of memory as shown by the qualifiers in Table 5-2.

Table 5-2 Memory Type Qualifiers

Qualifier	Initialized Variables	Non-Initialized Variables	Constants	Code
READ_ONLY	Not applicable.[4]	Not applicable.[4]	Content written to target address.	Content written to target address.
READ_WRITE	Content written into copy down area with information to allow it to be copied at startup.[5]	Area contained in zero out information.[6]	Content written into copy down area.[5,6]	To allocate code in a RAM area for system development, you should declare this area as READ_ONLY. In an embedded system, this is not applicable.[4]
NO_INIT	Not applicable.[4]	Handled as allocated.	Not applicable.[4]	Not applicable.[4]
PAGED	Not applicable.[4]	Handled as allocated.	Not applicable.[4]	Not applicable.[4]

[4] These cases are not intended but the linker does allow them. When used, the qualifier controls what is written into the application.

[5] Initialized objects and constants in READ_WRITE must be initialized at the start of the program at run-time. In a C program this is done by the start-up code but in an assembly language program you must ensure this is done. The copy down area contains information to initialize the constants in the start-up code.

[6] The zero out information specifies areas that must be initialized with 0 at startup.

Placement:

```
PLACEMENT
/* Place the code and constant sections in ROM */
    .init, MainCode, SubCode, MainConst, SubConst,
    DEFAULT_ROM  INTO  ROM_C000;
/* Place the Variable Data */
    MyData, MySubData, DEFAULT_RAM  INTO  RAM;
END
```

The *PLACEMENT* section places your various sections into the proper type of memory as defined in the SECTIONS section. You must include `DEFAULT_RAM` and `DEFAULT_ROM` placements. The linker, by default, will place your sections into the proper placement. For example, the linker knows to place any memory allocation (DS) parts of the program into DEFAULT_RAM and any code and constants (DC) into DEFAULT_ROM. You may, for the sake of documentation readability or to control where sections are placed, "place" the named sections from your program as shown in Example 5-3. This is optional but is a good documentation practice.

Stacksize:

```
/* Define the stack size needed */
STACKSIZE 0x100
```

An important part of your code development is the allocation of storage and placement of the stack. As we will learn in Chapter 6, the stack is an area of RAM used for temporary variable storage and for return addresses for subroutines. Because both the stack and the variable data storage sections are in READ_WRITE memory, it is important that neither one interfere with or overwrite the other. You must estimate the amount of stack storage that is needed and specify at least this amount. Because stack overflow (using more stack memory than is allocated) can cause many problems in your running program, most embedded system engineers are very conservative and allocate plenty of stack space. In this example, `STACKSIZE` allocates $0100 bytes, places the stack into RAM and defines the variable `__SEG_END_SSTACK` to be used in your program to initialize the stack pointer register (as shown in line 22 in Example 5-1.)

Vector:

```
/* Specify the initial reset vector entry point */
VECTOR 0 Entry  /* reset vector */
```

The vector section allows you to initialize vectors. This example shows how to initialize the reset vector to allow the microcomputer to be *vectored* to the start of the program when power is turned on. We will see other vectors and their uses for interrupts in Chapter 12.

Init:

```
/* INIT is needed for assembly applications */
INIT Entry
```

When linking an assembly application, an *entry point* is required. The *INIT* command with its operand, in this case *Entry*, provides this information.

Comments:

Comments are entered into the parameter file using standard C or C++ syntax for comments.

5.7 The Linker Output Files

Two output files created by the linker are important to us after we have located and linked our object files. These are the linker map file, which shows us where the linker has placed our code, and the absolute code file to be loaded into the microcontroller's memory.

The Linker Map File

The *Linker Map File* shows us where our code, constant and variable data are located, along with other information. Following the linking controlled by the linker parameter file in Example 5-3, the linker map file shows the following sections.

> The *linker map file* shows us where all elements of our program are located.

Target Section:

```
TARGET SECTION
---------------------------------------------------------------------------------------------
Processor   : Motorola HC12
Memory Model: SMALL
File Format : ELF\Dwarf 2.0
Linker      : SmartLinker V-5.0.22 Build 4047, Feb 17 2004
```

The target section shows the processor, the memory model (more important when programming in C), the file format, and the linker version.

File Section:

```
FILE SECTION
-----------------------------------------------------------------------------------
rel_ex_1a_main.asm.o                     Model: SMALL,         Lang: Assembler
rel_ex_1a_sub_1.asm.o                    Model: SMALL,         Lang: Assembler
```

This lists all files that have been linked.

Startup Section:

```
STARTUP SECTION
-----------------------------------------------------------------------------------
Entry point: 0xC000 (Entry)
```

The startup section shows the entry point, i.e. where the program will start.

Section-Allocation Section:

```
SECTION-ALLOCATION SECTION
Section Name                    Size  Type     From       To       Segment
----------------------------------------------------------------------------------
.init                             21     R     0xC000     0xC014   ROM_C000
SubCode                            7     R     0xC015     0xC01B   ROM_C000
MainConst                          1     R     0xC01C     0xC01C   ROM_C000
SubConst                           1     R     0xC01D     0xC01D   ROM_C000
MainData                           1   R/W      0x800      0x800   RAM
SubData                            1   R/W      0x801      0x801   RAM
.stack                           256   R/W      0x802      0x901   RAM
.vectSeg0_vect                     2     R     0xFFFE     0xFFFF   .vectSeg0

Summary of section sizes per section type:
READ_ONLY (R):         20 (dec:      32)
READ_WRITE (R/W):     102 (dec:     258)
```

Each of the software components that have been allocated into the various sections is described. Note that the code in the module that is first executed, that is the main module, is placed into a section named *.init*. In a C program, the startup code will be placed in .init. All other named sections, such as SubCode (in the subroutine), MainConst, MainData and Subdata are listed with their size, type of memory, the range of address, and the segment assignment.

Vector-Allocation Section:

```
VECTOR-ALLOCATION SECTION
    Address     InitValue   InitFunction
-------------------------------------------------------------------------------------
     0xFFFE        0xC000   Entry
```

This section shows how vectors have been initialized. We will see more of this when we discuss interrupts in Chapter 12.

Object-Allocation Section:

```
OBJECT-ALLOCATION SECTION
     Name                Module                   Addr   hSize   dSize     Ref Section  RLIB
-----------------------------------------------------------------------------------------
MODULE:                  -- rel_ex_1a_main.asm.o --
- PROCEDURES:
     Entry                                        C000       9       9       0 .init
     loop                                         C009       C      12       1 .init
- VARIABLES:
     main_data                                     800       1       1       3 MainData
     main_const                                   C01C       1       1       1 MainConst
- LABELS:
     __SEG_END_SSTACK                              902       0       0       1
MODULE:                  -- rel_ex_1a_sub_1.asm.o --
- PROCEDURES:
     sub_1                                        C015       7       7       1 SubCode
- VARIABLES:
     sub_const                                    C01D       1       1       1 SubConst
     sub_data                                      801       1       1       1 SubData
```

One of the most important sections is the object-allocation section. It shows, for each of the modules (in this case the main and subroutine) where the procedures, labels, and data elements are located. We will need this information when it comes time to debug our program. The hSize and dSize give the number of bytes in hexadecimal and decimal.

Module Statistic:

```
MODULE STATISTIC
  Name                                       Data   Code  Const
-------------------------------------------------------------------------------------
  rel_ex_1a_main.asm.o                          2     42      2
  rel_ex_1a_sub_1.asm.o                         2     14      2
  other                                       256      2      0
```

This shows for each module the number of code, data, and constants bytes that have been allocated. In this listing, *other* refers to the space allocated to the stack.

Section Use in Object-Allocation Section:

```
SECTION USE IN OBJECT-ALLOCATION SECTION
SECTION: ".init"
  Entry loop
SECTION: "SubCode"
  sub_1
SECTION: "MainConst"
  main_const
SECTION: "SubConst"
  sub_const
SECTION: "MainData"
  main_data
SECTION: "SubData"
  sub_data
```

This section shows which elements (or objects) of our program use which memory section.

Object List Sorted by Address:

```
OBJECT LIST SORTED BY ADDRESS
     Name                                      Addr   hSize   dSize     Ref Section RLIB
---------------------------------------------------------------------------------------
     main_data                                  800       1       1       3 MainData
     sub_data                                   801       1       1       1 SubData
     Entry                                     C000       9       9       0 .init
     loop                                      C009       C      12       1 .init
     sub_1                                     C015       7       7       1 SubCode
     main_const                                C01C       1       1       1 MainConst
     sub_const                                 C01D       1       1       1 SubConst
```

The address of each object (denoted by a label) is given here.

Unused Object Section:

This lists any objects found in the object files that were not linked, such as a label not referenced.

Copydown Section:

This section lists all blocks that are copied from ROM to RAM at program startup. This is more relevant in C programming.

Object-Dependencies Section:

```
OBJECT-DEPENDENCIES SECTION
------------------------------------------------------------------------------------
Entry                     USES __SEG_END_SSTACK main_const main_data
loop                      USES main_data sub_1 loop
sub_1                     USES sub_data sub_const
```

This section lists the names of global objects which every function and variable.

Dependency Tree:

An object dependency tree shows in a tree format all detected dependencies between functions.

Statistic Section:

```
STATISTIC SECTION
------------------------------------------------------------------------------------
ExeFile:
--------
Number of blocks to be downloaded: 5
Total size of all blocks to be downloaded: 32
```

This section gives the number of bytes of code in the application.

Absolute Files

The linker can produce two kinds of output files to be loaded in the microcontroller memory. These are absolute, *.abs*, and Motorola S files.

Absolute files, which have the extension .abs, contain the program code plus some debugging information. We will see how to use this information in Chapter 8. Motorola S files, which have extensions .s1, .s2, .s3 or .s19, contain all code from the READ_ONLY sections of the application. This, then, can be downloaded to the microcontroller or programmed into the ROM.

5.8 Remaining Questions

In this chapter we have described enough of the features of the linker to get started in assembly language programming. We still have a number of issues to cover and we will do that in succeeding chapters. Some of these details yet to be explained include the following topics:

- What is startup code? *Startup code is the code that is executed when the processor is first powered-up or the program is first run. In C programs a standard startup code sequence is used. We will discuss this issue further in Chapter 8.*

5.9 Conclusion and Chapter Summary Points

In this chapter we have discussed the operation of the linker. We have included enough details to be able to assemble and link assembly language programs using separately assembled relocatable modules.

- There are two kinds of memory in a microcontroller system – RAM and ROM.
- There can be two types of ROM – EEPROM and FLASH EEPROM.
- You must know what the memory map is to be able to locate the various parts of the program in the proper memory types.
- Variable data storage and the stack use RAM.
- The program code and constants use ROM.
- The linker is controlled by a linker parameter file.
- It is very easy to change the parameter file to relocate the code for different hardware configurations.

5.10 Bibliography and Further Reading

Cady, F. M., *Microcontrollers and Microprocessors*, Oxford University Press, New York, 1997.

Smart Linker, Metrowerks Corporation, Austin, TX, 2003.

5.11 Problems

Table of Contents

Chapter Listings:
Revision Chapters