Appendix D
Editing the Linker Script

Overview

Introduction

In Lesson 16, we mentioned that there were times when we might want to edit the linker script file. In this appendix we examine the linker script and discuss how we might want to change it.

In this section

Following is a list of topics in this section:

The Linker Script

Introduction	The MPLINK linker cannot do its job without a linker script. The script tells MPLINK most of what it needs to decide how to place the program into the PIC's memory. The default linker script essentially describes the particular PIC to the linker. It includes such things has how much general purpose register memory is available, how much program memory is available, and how much of the GP register memory is shared between banks. Even though the default linker script describes what would seem to be immutable hardware, it is possible to interpret the hardware capability in different ways. Microchip has chosen different views of different PICs, so sometimes capabilities that are essentially identical in hardware seem different when viewed by the software, because of the translation provided by the linker script.
What is in a linker script?	The linker script is a sequence of commands describing the PIC to the linker. There are four different categories of commands: • Commands that provide information about files to be linked. These could be provided on the MPLINK command line, but the command line can be shortened by using these linker script commands. • Memory region definitions – These commands form the core of describing the hardware to the linker • Logical Section Definitions – These commands describe how we would like the hardware to appear to the program • Stack Definitions – These commands are only relevant to the 17F and 18F PIC parts. We will not discuss them here. In the following sections, we will examine each of the commands that make up the first three categories.
Comments	Within a linker script, anything which appears on a line following two slashes, `//`, is ignored. Blank lines are permitted.

Command Line Information

Introduction	The command line group of commands help us shorten the MPLINK command line. When we are linking within MPLAB, these commands are largely redundant.
LIBPATH	`LIBPATH` tells MPLINK what directories to search for libraries. If we give MPLINK fully-qualified names for all the libraries we use, there is really no need for this command. The default linker scripts typically include the current directory in the `LIBPATH`. `LIBPATH .;C:\Projects\Libraries`
INCLUDE	The `INCLUDE` directive allows us to include additional linker scripts. While it might be useful to have a group of sections that we use across projects, linker script files are typically quite short, so this command is of relatively little use. `INCLUDE C:\Projects\Include\Mylibs.lkr`
LKRPATH	`LKRPATH` tells MPLINK what directories to search for include files. If the `INCLUDE` directive is relatively useless, the `LKRPATH` directive is even more so. `LKRPATH .;C:\Projects\Include`
FILES	`FILES` tells MPLINK what files to include when linking. We can list a combination of object (.o) and library (.lib) files in this list. When we are linking from within MPLAB, the IDE provides all the file names on the command line so this directive is unnecessary. `FILES main.o sub1.o mylib.lib`

Memory Region Definition

Introduction	The memory region set of commands describe the memory assets of the PIC. These commands ultimately determine how the linker views the hardware.
CODEPAGE	The `CODEPAGE` command describes a part of memory which will be used to store instructions. The command describes a starting and ending address for a piece of memory, and gives it a name. It may have an optional '`PROTECTED`' attribute which means that it may only be used for parts of the program that specifically request this region. The command may also include an optional fill value: `CODEPAGE NAME=page0 START=0x0005 END=0x0100 PROTECTED`
DATABANK	The `DATABANK` command describes file register memory with a syntax almost identical to the CODEPAGE command (except that a fill value is not available). The `DATABANK` refers to banked memory. The default 16F84 linker script describes the GPR's as banked, even though that memory looks more like a `SHAREBANK` to the program. On some other processors, Microchip chose the other alternative: `DATABANK NAME=gprs START=0xC END=0x4F`

SHAREBANK	On processors with more than 96 GPR locations, the top 16 locations are available in all banks (as are all GPR locations in the 16F84). To describe this behavior the `SHAREBANK` command is used which is identical, except for the command, to the `DATABANK`:
	
	`SHAREBANK NAME=gprnobnk START=0x170 END=0x17F`
	`SHAREBANK NAME=gprnobnk START=0x1F0 END=0x1FF`
	
	When different addresses share the same locations, they share the same name in their `SHAREBANK` commands.

ACCESSBANK	`ACCESSBANK` is used to describe a special type of memory on 18Fxx processors called Access Memory. The syntax is the same as the others: `ACCESSBANK NAME=accessram START=0x0 END=0x7F`

Logical Section Definition

Introduction	The logical sections assign sections as viewed by the program to memory regions in the hardware. There is only one command in this group; `SECTION`.

SECTION	The `SECTION` command associates a `CODEPAGE` or `BANK` name with a name visible inside the assembler. The command includes a `ROM=` or `RAM=` name to associate with a `CODEPAGE` or `xxxBANK` command earlier:
	
	`SECTION NAME=STARTUP ROM=vectors`
	`SECTION NAME=DISPLAY RAM=dispbuf`

Linker provided sections	MPLINK always provides two sections not mentioned in the linker script. The **`.code`** section is the default location for program code. The **`.cinit`** section is used to provide initialization values for initialized RAM locations. The **`.cinit`** section is not especially useful except for high level language programs. The code required to transfer values from the **`.cinit`** section to RAM in most cases would be more complex than specific code written for the application.
Assigning sections to regions	When a section is allocated to a specific `CODEPAGE` or `DATABANK`, any code assigned to that section will be placed in that region. However, for sections which are provided by the linker, i.e. **`.code`** and **`.cinit`**, those sections will be allocated within a region which does not have the `PROTECTED` attribute.

Examining p16f84a.lkr

Introduction

Examples always seem to help, and why not start by examining the linker script we have been using all along … p16f84a.lkr.

P16f84a.lkr

Just so we have it handy, here is the default script:

```
// Sample linker command file for 16F84A
// $Id: 16f84a.lkr,v 1.4 2002/01/29 22:10:01 sealep Exp $

LIBPATH  .

CODEPAGE   NAME=vectors  START=0x0      END=0x4      PROTECTED
CODEPAGE   NAME=page     START=0x5      END=0x3FF
CODEPAGE   NAME=.idlocs  START=0x2000   END=0x2003   PROTECTED
CODEPAGE   NAME=.config  START=0x2007   END=0x2007   PROTECTED
CODEPAGE   NAME=eedata   START=0x2100   END=0x213F   PROTECTED

DATABANK   NAME=sfr0     START=0x0      END=0xB      PROTECTED
DATABANK   NAME=sfr1     START=0x80     END=0x8B     PROTECTED

DATABANK   NAME=gprs     START=0xC      END=0x4F

SECTION    NAME=STARTUP  ROM=vectors    // Reset and interrupt vectors
SECTION    NAME=PROG     ROM=page       // ROM code space
SECTION    NAME=IDLOCS   ROM=.idlocs    // ID locations
SECTION    NAME=CONFIG   ROM=.config    // Configuration bits location
SECTION    NAME=DEEPROM  ROM=eedata     // Data EEPROM
```

The CODEPAGE commands

Notice that the script has allocated several codepages, some of which are unfamiliar. First, notice that the first four locations are set aside. Location 0x0 is the reset vector, and 0x4 is the interrupt vector. The script names this region, appropriately enough, **`vectors`**. The only region not protected is **`page`**, which is the entire remaining FLASH memory. The **`.idlocs`** region contains four words which are available to identify our particular PIC. **`.config`** is the region where the results of our `__config` directive are stored, and **`eedata`** is where the EEPROM lives. Simple enough.

The DATABANK commands

The `DATABANK` commands are a little different. You may have noticed that the assembler generates addresses in the 0x8x range for registers that are in bank 1 (`TRISA, EECON1`), and addresses 0x0b and below for special purpose registers in bank 0 (`PORTA, INTCON`). This results in regions **`sfr0`** and **`sfr1`**. The general purpose registers which run from 0xc to 0x4f are named, appropriately enough, **`gprs`**.

The SECTION commands

The `SECTION` commands define what we can see from the program. In Lesson 16 we used the **`STARTUP`** section which we can see is allocated to **`vectors`**. We typically don't directly reference **`PROG`** so we don't see that in our map. However, **`page`** is the only region which is not `PROTECTED`, so the **`.code`** and **`.cinit`** sections are allocated from **`page`**.

If we wanted to use the identification locations, we could store our program ID in **`IDLOCS`**. Similarly, we can use **`DEEPROM`** to initialize the EEPROM.

Placing Page Zero Tables

Introduction

There are times when we want to place some code specifically on a particular page. For example, when we have lookup table, we need to ensure that the tables do not cross 256-word page boundaries. In processors with limited memory, like the PIC16F84, we can save a few instructions from each table by placing the table on page zero.

Placement from Assembler

When we wish to control the location of a section of code, we can do that from the assembler by specifying the address in the code instruction:

```
Tables  Code  0x0005
```

However, this only allows the section to appear in a single .asm file, since only one section can share a particular address. We could assign a name and address to the tables from each .asm file, but this would require adjustments across .asm files if the length of one table changed. This defeats much of the advantage of using relocatable code.

Using the linker script

The problem can be solved by editing the linker script. To do this we need to copy the default linker script to our project. It is probably helpful if the script is renamed to reflect the project name.

We can assign tables from multiple .asm files to the same section, providing we do not assign an address:

```
Tables   Code
Tab1     addwf    PCL,F
           …
```

This will cause the linker to group all our tables together. To tell the linker to place the new section, **`Tables`**, at a particular place, we need to make two changes to the linker script.

Making CODEPAGE space

First, we must allocate some memory for the table with the `CODEPAGE` directive. In order to get some space, we must take space away from another region. Our only real choice here is the **`page`** region:

```
CODEPAGE   NAME=page0    START=0x5      END=0x4F
CODEPAGE   NAME=page     START=0x50     END=0x3FF
```

In this case, we started the **`page`** region a little higher in memory to make space for a new region we called **`page0`**, which will hold our tables. Note that we may need to make adjustments to the size of **`page0`** after we have seen our map.

Creating a new Section

Now that we have allocated space for the table, we need to assign the section name we have chosen above, **`Tables`**, to the region **`page0`**.

```
SECTION    NAME=Tables   ROM=page0
```

Linker Behavior

Depending on the size of our program, the linker may decide to allocate **`.code`** or **`.cinit`** from our **`page0`** region. As long as we have enough space for our tables, this really doesn't matter. However, if we wish to prevent this from happening, we can assign the `PROTECTED` attribute to our **`page0`** region.

Grouping functions on a page

Introduction

On processors with more than 2K of program memory, we need to take a little care with where routines are placed in memory. The `goto` and `call` instructions can only reference code within a 2K page. To `call` or `goto` outside this range requires the use of the `lcall` or `lgoto` special instructions. These instructions actually take up to three words instead of one, so it is to our advantage to avoid them to the extent possible.

Very often, applications decompose in to groups of related functions. These related functions tend to have many calls among themselves, and relatively few calls into the group. These groups of related routines are good candidates for libraries. Recall the LCD library example in Lesson 16; two calls into the library resulted in 7 functions getting loaded due to calls within the library. What we didn't see in the linker map is that some of those routines got called multiple times.

Placing routines together

In order to group these related routines, we do essentially what we did for a table; we assign a name for the section within our source, create a region were we want to store the routines, and assign the section name to the region.

Example

In the LCD library, all the code has been assigned to the section **`LCDLIB`**. If we use the default linker script, this section is allocated from the unprotected memory, **`page`**. On a processor with 2K or less of memory, this is just fine. But suppose we wanted to ensure that the routines stayed on one page. We could allocate the section to the beginning of a page, just as we did in the previous example:

```
CODEPAGE   NAME=page     START=0x5      END=0x2FF
CODEPAGE   NAME=page3    START=0x300    END=0x3FF
```

Then we could assign the **`LCDLIB`** section to that page:

```
SECTION    NAME=LCDLIB   ROM=page3
```

Note that although we have used a 16F84A example, this technique is not particularly useful for that processor since it only has 1k of program memory. However, were we to use a processor with more memory, this could have a significant impact on both storage required and performance.