

# Lesson 16 Relocatable Code

## Overview

|                        |  |                 |
|------------------------|--|-----------------|
| <b>Introduction</b>    | Trying to use code from previous projects can be very tedious with the absolute code model discussed so far. Relocatable code helps with these issues. |                 |
| <b>In this section</b> | Following is a list of topics in this section:   |                 |
|                        | <b>Description</b>   | <b>See Page</b> |
|                        | Reusing Code   | 2               |
|                        | What is relocatable code   | 4               |
|                        | The Linking Process  | 5               |
|                        | Assembler Directives   | 6               |
|                        | A Simple Example   | 8               |
|                        | Sharing Data Locations   | 11              |
|                        | Re-using File Register Locations   | 13              |
|                        | Libraries  | 15              |
|                        | Using a library  | 16              |
|                        | Further Experiments  | 18              |
|                        | Wrap Up  | 19              |

## Reusing Code

|  |   |
|--|---|
| <b>Introduction</b>                        | <p>One of the very appealing things about programming the PIC is that everything is absolutely predefined. We tell the assembler precisely what to put in the PIC and <i>where to put it</i>. This makes PIC programming virtually devoid of the nasty surprises that we sometimes see in programming in more complex environments.</p> <p>This absolute determinism has its dark side; when we want to use code from a previous project, it can become very tedious merging that old code with the new code. It would be nice if some of the hard work could be automated. That is exactly what the linker, MPLINK, does for us. However, before we can use MPLINK to help us, we need to provide clues in our code as to our intentions.</p>  |
| <b>Use of include files</b>                | <p>We have already seen how we can break larger programs apart into several files, and include those additional files in our assembly. While this can help with organizing our code, it doesn't help with some of the challenges we have when we want to reuse the code.</p> <p>If we place a group of routines we want to re-use into an include file, we can avoid cutting and pasting it into our source file. However, we need to adjust the general purpose register locations used so as not to clash with our existing program. If the include file uses names for symbols that are easy to read, we are likely to have also used those names somewhere else in the program. Finally, if we have a group of functions in a single include file, the odds are that the next program doesn't need all of the functions. So, we must either waste the memory for those unneeded routines, or edit the include file to remove those routines. This requires studying, and re-understanding, the functions to be sure they aren't required elsewhere.</p> |
| <b>Allocating File Register Locations</b>  | <p>Up until now, we have used the <code>cblock</code> directive to identify where we want to place specific variables in the file register. Normally, we don't really care what particular location we use, we simply don't want to inadvertently use that location for something else. It would be nice if those locations could be allocated automatically. That way each include file could have its own declarations for general purpose register usage and somehow they could be magically merged so that they don't conflict with the main programs or other include files.</p>   |
| <b>Allocating Program Memory Locations</b> | <p>Program memory generally isn't as much of a problem as GP register memory. We care about what goes into locations zero and four (reset and interrupt vectors), but beyond that we normally just use one location after another. Including some code with the include directive follows this model.</p> <p>Sometimes, though, we care about keeping some program on a particular page. When we write lookup tables it is simpler to write these tables for the first 256 words of program memory. If an include file has such a table, getting that to coexist with tables in the main program, or in other include files, can get messy.</p>   |

*Continued on next page*

## Reusing Code, Continued

### Managing Symbol Scope

Typically, if we write a routine for some particular purpose, the only program memory label<sup>1</sup> within that routine that needs to be known outside the routine is its entry point. However, if we include the routine through the include directive, all of the labels in the function have the same priority as the labels in the main program. We need to ensure that we don't have name clashes with any of the labels we have already used.

The same applies to general purpose (file) register symbols. Within a routine, its memory use is generally only of concern to the routine itself. Occasionally we may pass parameters into the routine or results out of the routine with the GP registers, and these symbols would need to be known outside the function. But for the most part, general purpose register use tends to be local to a particular routine.

### Reusing file register locations

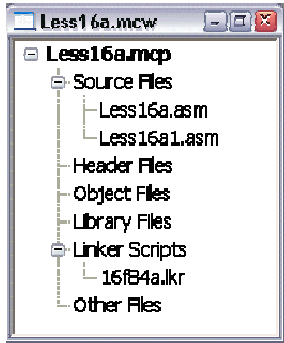
On the PIC16F84A, there are only 56 general purpose register (GPR) locations. As we pointed out in the previous section, the use of these locations tends to be local to a particular function. To reuse those locations for other functions we tend to name those scratch locations with names like temp, temp1, scratch, or some other innocuous name. This makes our program harder to read since we can't use meaningful names that might be different across routines. We could equate another symbol to that address, but now we have built ourselves a trap for when we add the next include file.

<sup>1</sup> To be picky, we generally refer to a label as something that we wrote in column 1 of the source. These become symbols in the assembler result for a relocatable assembly, but the "symbol" is a slightly broader term that encompasses not only our program and data labels, but also some other symbols generated by the assembler that might not be visible in our program source. For now, it is helpful to think of a symbol as simply a tag for something in the program that might need to be referenced later. It only becomes important now because when we were doing absolute assembly, there was no "later". Now there is.

## What is relocatable code

|                                 |  |
|---------------------------------|--|
| <b>Introduction</b>             | When we assemble absolute code, the assembler generates a hex file that contains exactly the code that will be loaded into the PIC. If we want the linker to be able to move that code around, any addresses need to be somehow marked so that the linker knows to fix them up. This includes both program and GPR locations. There might even be different ways of handling different symbols.  |
| <b>Local and Global symbols</b> | If we are going to be able to call a routine that wasn't included in the current assembly, the assembler needs to be told that the routine's name is going to be found somewhere else. Similarly, when that routine is assembled, the entry point needs to be marked so that the assembler knows that the routine's entry point is a symbol that needs to be exposed to the outside; that is, the symbol is <i>global</i> . Any other symbols can be made <i>local</i> , which means that they are only known within the current assembly, and can't be accessed by other routines.  |
| <b>Program Sections</b>         | <p>A relocatable program is broken up into a number of program <i>sections</i>. These sections each contain code with different rules for where it may be placed. The section may also have a name.</p> <p>Most of the time we won't give a section a name, so the assembler will provide a name for us. By default, our program code goes into a section named <code>.code</code> and our data goes into a section named <code>.udata</code>. The assembler also provides default names of <code>.config</code> where the configuration data is stored (H'2007'), <code>eedata</code> where the EEPROM data is stored (H'2100'), and <code>.idlocs</code> where the ID data is stored (H'2000'). (We haven't mentioned the ID locations but they are simply a place where we can store things like the version number of a program so we can easily identify a programmed PIC. And your author has no idea why Microchip chose to put a dot before everything except <code>eedata</code>.)</p>  |
| <b>Assembler Output</b>         | <p>When we assemble an absolute program, the assembler outputs a <code>.hex</code> file. This file contains exactly the code to be written to the PIC, in a format readable by the programming software.</p> <p>When we build relocatable code, the assembler outputs a <code>.o</code> (object) file, which contains the assembled codes like the <code>.hex</code> file, but whenever an address is referenced, the file contains rules for the linker to resolve that address. The linker can then take that output, along with other <code>.o</code> files, and link them together to get the <code>.hex</code> file to load into the PIC.</p> <p>Suppose, for example, we have code like:</p> <pre> Loop     incfsz   abc,F     goto    Loop </pre> <p>In an absolute assembly, the assembler would know the address of <code>Loop</code> and could then assign that address as the target of the <code>goto</code>. When we are generating relocatable code, however, the address of <code>Loop</code> is not known. Instead, the assembler outputs the offset of <code>Loop</code> from the front of the program section, along with instructions to the linker to add the starting address of the section to the address of <code>Loop</code>.</p> |

## The Linking Process

|                                 |  |
|---------------------------------|--|
| <p><b>Introduction</b></p>      | <p>When making relocatable code, the developer has the option to break the program into any number of individual files. The programmer could choose to keep the entire program in a single file. However, it is often better to have a separate file for each subroutine in the program. Sometimes it may make sense to have several related routines in one file. The linker now has the task of linking these files together to make the final .hex file.</p>  |
| <p><b>Linking in MPLAB</b></p>  | <p>When we create a project in MPLAB, somehow MPLAB needs to know whether it should create relocatable code or absolute code. So far, we have always ended up with absolute code.</p> <p>However, if we were to add more than one .asm file to the project, MPLAB<sup>2</sup> detects that we are going to have to link those files together, and would generate relocatable code. In addition, MPLAB will try to link those assemblies together. However, it does not have enough information to do this.</p>   |
| <p><b>The Linker Script</b></p> | <p>Before the linker can piece together all the bits of our program, it needs to know what to do with the various program sections. The linker script tells the linker what to do with the program sections. The default linker script simply gives the linker a map of the hardware for the particular PIC.</p> <p>The linker script takes a bit of thought. Fortunately, we can almost always steal an already prepared linker script. In the MPLAB program directory, in the subdirectory<sup>3</sup>:</p> <p style="text-align: center;">MCHIP_Tools\LKR</p> <p>will be found a series of linker scripts, one for each processor. We can simply add the linker script for our processor to our project under the “Linker Scripts” heading.<sup>4</sup></p> <div style="text-align: center;">  </div> <p>This works just like adding source files. Right-click on Linker Scripts, select Add Files..., and navigate to the linker script directory.</p> |

- 2 This applies only to MPLAB version 6.x. In earlier versions of MPLAB, the linker must be specifically called out in the project properties. In 7.x, we must have a linker script for MPLAB to realize it must call MPLINK.
- 3 For MPLAB 4.x, these scripts are in the install directory. In 5.x, they are in LKR under the install directory, in 7.x, they are in Microchip\MPASM Suite\LKR.
- 4 Again, this is different for earlier versions.

## Assembler Directives

### Introduction

When we write a program targeted at generating relocatable code, there are a few differences in the directives we use. In this section, we will talk about the most commonly used directives.

### The code directive

When we want to write instructions to be placed in the PIC's program memory, we need to precede those instructions with a `code` directive:

```

code
Start
    movlw    H'01'
    movwf   Count

```

The `code` directive tells the assembler that the following code belongs in the program section containing instructions. Because we didn't include a name in front of the `code` directive, the assembler assumes that we don't have any special concerns about how this code is handled. If we wanted, for some reason, to group these instructions together with some instructions from another file, we could assign a name to the code segment:

```

MySeg    code

```

This is generally only interesting for processors with over 2K of program memory. `goto` and `call` statements which cross 2K boundaries require an extra instruction, and a group of routines with many references between them may want to be grouped together to avoid this.

There is a special code segment named `STARTUP` that resides at location zero. We normally have a `goto` instruction there so we might do something like:

```

STARTUP    code
           goto    Start

```

There are a few cases where we care about where a particular piece of code goes. In this case, we can assign an address to the segment:

```

Table2    code    H'0200'

```

In general, this should be avoided. However, sometimes it can be used to avoid editing the linker script, which can be somewhat tricky.

### The udata and res directives

What `code` does for instructions, `udata` does for GPR addresses. `udata` stands for uninitialized data. Up until now, we have set aside GPR locations with the `cblock` and `cend` directives. These don't really reserve memory for us, they merely assign sequential values to symbols so that we can easily refer to them. The `res` directive actually reserves places in the file register for us:

```

           udata
Var1      res    1
Var2      res    1

```

The value after the `res` indicates how many bytes to reserve. Like the `code` directive, we can assign a name and/or an address with the `udata` directive. By default, space is allocated beginning at the first available GPR location and the section is named `.udata`.

*Continued on next page*

## Assembler Directives, Continued

### global and extern

With the directives so far, we could put together a working program using relocatable code. However, the program would need to be entirely within one file, because the symbols are all local. If we try to reference something in another file, the assembler will complain that the symbol is undefined.

The `global` directive tells the assembler to make a symbol available outside this file.

```
global MySub
code
MySub xorlw H'05'
```

While the `extern` directive tells the assembler that the symbol mentioned will be resolved later by the linker:

```
extern MySub
code
call MySub
```

Both directives may list several symbols:

```
extern LCDinit,LCDletr,LCDclear
```

These directives may be placed almost anywhere, but it is best to place them near the front of the file. That way, the reader can quickly see what capabilities this file provides to the rest of the world (`global`) and what external assets it relies on (`extern`).

### Additional Directives

There are quite a number of other directives which are available, but most are rarely used. Of interest when we are pressed for GPR space is the `udata_ovr` (overlaid data) directive which allows us to share GPR locations between program sections without sharing the symbols. The `udata_shr` (data shared among banks) directive forces GPR locations into the top 16 locations, which is important on PICs with more than 96 file register locations. The `idata` directive, in combination with the `data` directive, provides for initialized data.

Other than `udata_ovr` however, these directives aren't commonly used, and most programs can get along with the five directives, `code`, `udata`, `res`, `global` and `extern`.

We will do an example with `udata_ovr`, even though it is a fairly uncommon directive.

The `udata_shr` directive is not available on the PIC16F84A because the chip doesn't have the shared GP register hardware<sup>5</sup>. The `idata` directive is probably more complicated than it is worth, and is mostly intended to help with implementing higher level languages like Pascal. We will not cover either of these in this lesson. They are only mentioned here in because the student will see them in the assembler documentation.

<sup>5</sup> On some newer PICs, especially newer 16C parts, Microchip has chosen to define the entire GPR space of the part as "share". The GPR behavior of these parts is identical to the F84, so this can be a cause of some confusion. This treatment can be changed by editing the linker script file, but the developer is cautioned to review the datasheet carefully when using a new part.

## A Simple Example

|                                      |   |
|--------------------------------------|---|
| <b>Introduction</b>                  | In this section, we will make a trivial program using relocatable code. We will see how the <code>global</code> directive can make a label available for the <code>extern</code> directive.   |
| <b>Setting up the project</b>        | <p>Create a project, <code>Less16a</code>, just like you have done many times before. However, this time, make two <code>.asm</code> files, <code>Less16a.asm</code>, and <code>Less16a1.asm</code>. Include both of them in your project.</p> <p>Right click on “Linker Scripts” and select “Add Files...” just like you do for Source Files. (See the figure on page 5). Navigate to your MPLAB directory, and then to <code>MCHIP_Tools</code> and finally, <code>LKR</code>, and select <code>16f84a.lkr</code>.<sup>6</sup></p>  |
| <b>Adding code to our subroutine</b> | <p>In <code>Less16a1.asm</code>, we will add the code for our subroutine. In this case, we really just want to demonstrate linking, so we really don’t need much fancy in this subroutine. Let’s just do something that complements whatever is in the <code>W</code> register:</p> <pre>                 global      aSub                  code aSub           xorlw      h'ff'                 return                 end </pre> <p>Notice that our entry point, <code>aSub</code>, has been made available to the rest of the world through the <code>global</code> directive.</p> <p>We can only include the <code>__config</code> directive once per program, so it is best to put it in the mainline and not here. We also didn’t need anything from the processor include file. Rather than introduce an unnecessary dependency, we have omitted it here. The reader is cautioned, however, that lacking this include can produce some troublesome errors.</p> |
| <b>The mainline</b>                  | <p>Now, lets add some code to the main program, <code>Less16a.asm</code>, to put something into the <code>W</code> register, call <code>aSub</code>, and then store the result in the file register.</p> <pre>                 processor   pic16f84a                 include     p16f84a.inc                 __config    _XT_OSC &amp; _WDT_OFF &amp; _PWRTE_ON                  extern      aSub                 udata var1           res         1  STARTUP code                 goto       Start                  code Start                 movlw      h'17'                 call       aSub                 movwf      var1  aa            goto       aa                 end </pre>  |

*Continued on next page*

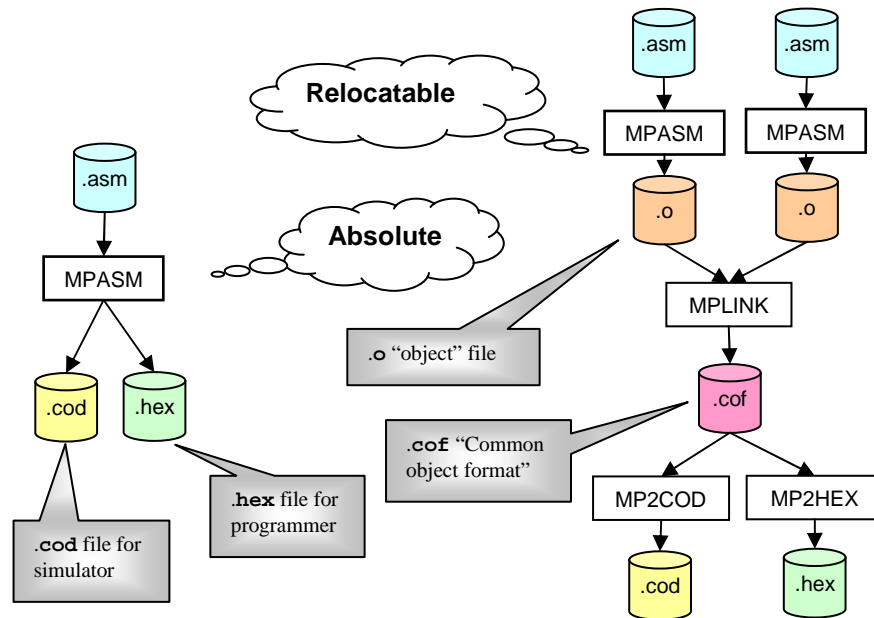
<sup>6</sup> In earlier versions of MPLAB, you must highlight the `.hex` file in the Edit Project dialog, and click on “Node Properties”. You must then select MPLINK as the “Language Tool”. You must also provide the name of the `.lkr` file in the “Additional command line options”. The location of the default `.lkr` files is different in different versions of MPLAB other than 6.x.



## A Simple Example, Continued

### Assembling

When we click on the build button, notice that the assembler runs twice, once for each .asm file. Then some other programs run, including the linker. The other two programs convert the linker output to a .hex file for loading into the PIC, and also to a .cod file for simulation. These are the same files that are produced when we do an absolute assembly.



### Simulating the program

If we choose MPLAB SIM as our simulator and step through the program, we will see that there are no nasty surprises – the program executes as we would expect, and the result gets deposited in location h'c' in the file register, just as if we had used a cblock to define var1.

*Continued on next page*

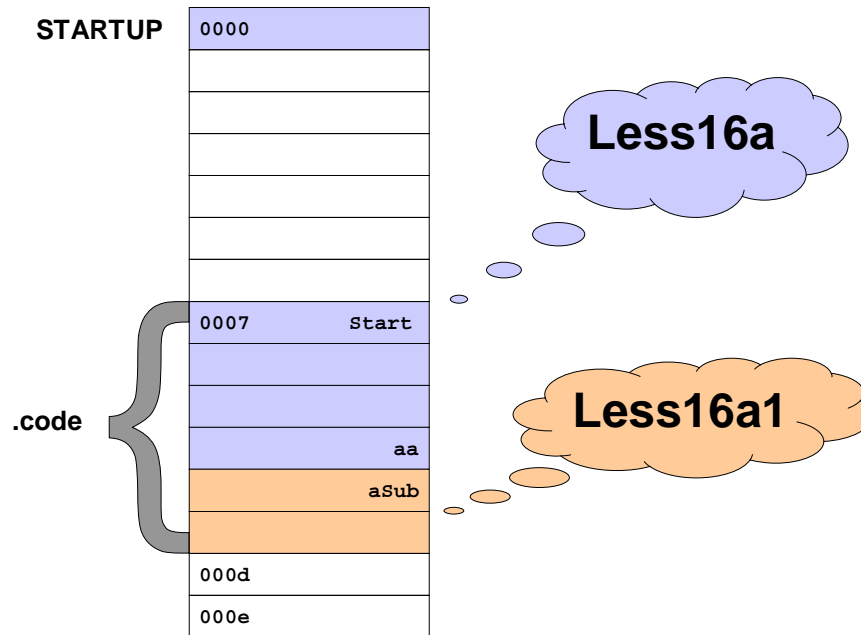
## A Simple Example, Continued

### Seeing where things got placed

Go to the Project menu, and select Build Options and then Project. Select the MPLINK Linker tab in the dialog that comes up, and check the “Generate map file” check box.<sup>7</sup> Now re-build the project.

Go to File->Open and in the “Files of type” dropdown select Map Files. Then open Less16a.map.

Scrolling down to “Symbols – Sorted by Address” we can see how the linker decided to place things in memory. The “Start” symbol ended up at location H'7'. The loop at “aa” ended up at h'a'. If you count instructions from Start you will see that makes sense. Since “aa” was the last location in our mainline, and “aSub” was the first location in our subroutine, aSub ends up at h'b', right where you would expect it. The only variable in our program, “var1”, got placed at the first available file register location, h'c'.



Note that the order might be different, depending on how files were added to the project, and whether you have a slightly different version of the linker. The actual order really doesn't matter all that much since all the addresses get fixed to reflect how things really turned out.

<sup>7</sup> In earlier versions of MPLAB, you must select Edit Project from the Project menu. Then highlight the .hex file and click on “Node Properties”. Check “Map file” to cause the linker to create a map file, and in versions prior to 5, you must provide a name for the map file.

## Sharing Data Locations

### Introduction

In the previous example, we saw how the `extern` directive could tell the assembler that a symbol would be resolved elsewhere, and how the `global` directive could make a symbol available outside the assembly. This allows us to make subroutines which are independent of the routines that call them.

However, it is also possible for a routine to make its data global, and thus shareable with other routines. In this example, we will use a pair of routines to store and retrieve a bit of data in a location hidden from the calling program

### Less16b

This time we will use a mainline and two subroutines. Again, create a project, Less16b, and add the `.lkr` file just like in the previous example. Our mainline will load a value into `W`, call a routine, then clear the `W` and call another routine:

```

processor      pic16f84a
include        p16f84a.inc
__config      _XT_OSC & _WDT_OFF & _PWRTE_ON

extern         Sub1,Sub2

STARTUP       code
              goto      Start

Start         movlw     h'3a'
              call      Sub1

              clrw
              call      Sub2

aa            goto      aa

              end
    
```

Notice that we will declare the two subroutines `extern`, just as we did with the one routine in the previous example.

### The Subroutines

The first subroutine uses a file register location to save the data, much like the earlier example. However, `Less16b1.asm` declares that location `global`:

```

global        Sub1
global        Shared

Shared       udata
            res      1

Sub1         code
            movwf    Shared
            return

            end
    
```

*Continued on next page*

## Sharing Data Locations, Continued

### The Subroutines (continued)

The second subroutine, `Less16b2.asm`, does not declare a `udata` section. Instead, it declares as `extern` the variable made `global` by `Less16b1`.

```

                                global      Sub2
                                extern      Shared

                                code

Sub2:
                                movf      Shared,W
                                return

                                end

```

### Assembling and simulating

When we assemble and link this program, it is worth looking at the map:

| Symbols - Sorted by Name |          |          |         |                                       |  |
|--------------------------|----------|----------|---------|---------------------------------------|--|
| Name                     | Address  | Location | Storage | File                                  |  |
| Start                    | 0x000007 | program  | static  | C:\Projects\PIC\Lesson16\Less16b.asm  |  |
| Sub1                     | 0x00000c | program  | extern  | C:\Projects\PIC\Lesson16\Less16b1.asm |  |
| Sub2                     | 0x00000e | program  | extern  | C:\Projects\PIC\Lesson16\Less16b2.asm |  |
| aa                       | 0x00000b | program  | static  | C:\Projects\PIC\Lesson16\Less16b.asm  |  |
| Shared                   | 0x00000c | data     | extern  | C:\Projects\PIC\Lesson16\Less16b1.asm |  |

Notice that there is only one memory location reserved for the variable `Shared`, and it is the one declared by `Less16b1`. If we were to simulate this program and step through it, there would be no surprises.

### Re-use

In this example, again we left off the `processor` and `include` statements from our subroutines. This actually requires a certain amount of thought. Much of the time we cannot omit the `include` because there are symbols in that file that we need. For example, the definitions for `PORTA`, `INDF`, and the various configurations bits are all provided by the processor include file.

However, when we are building relocatable code, it is much easier to use our routines in other programs than it is when we are building absolute code. Since different PICs are very similar, we might very well want to re-use our subroutines on a project with some other processor. As a result, we may want to avoid any *unnecessary* dependencies on a particular PIC model.

## Re-using File Register Locations

|                                   |  |
|-----------------------------------|--|
| <p><b>Introduction</b></p>        | <p>In the previous example, two routines shared a file register cell, because they needed to share information. However, very often, a subroutine's file register use is temporary. Once the routine exits the memory is no longer needed. It would be handy if several subroutines could re-use the same block of memory.</p> <p>This is exactly what the <code>udata_ovr</code> directive is for.</p>  |
| <p><b>udata_ovr</b></p>           | <p>The <code>udata_ovr</code> directive specifies that we are to reserve general purpose register locations, just like the <code>udata</code> directive. However, it allows any other routine which specifies a <code>udata_ovr</code> section of the same name to share these memory locations. As a result, the linker will assign all of these to the same memory location.</p>   |
| <p><b>A udata_ovr example</b></p> | <p>In this example, we will write two delay routines, pretty much identical except that they will use different names for their loop counters. The loop counters will be stored in a <code>udata_ovr</code> section named <code>Counters</code>. The first subroutine, <code>Less16c1.asm</code>:</p> <pre> global          Sub1  Counters       udata_ovr Loopc1         res          1 Loopc2         res          1  Sub1           code                 movlw        5                movwf        Loopc1  Loop1          movlw        3                movwf        Loopc2  Loop2          decfsz       Loopc2, F                goto         Loop2                decfsz       Loopc1, F                goto         Loop1                 return                end     </pre> <p>Notice that we have a section named <code>Counters</code> and we have reserved two locations for our loop counters, <code>Loopc1</code> and <code>Loopc2</code>. We have made the subroutine entry point <code>global</code>, just as in the previous examples.</p> |

*Continued on next page*

## Re-using File Register Locations, Continued

### A udata\_ovr example (continued)

The second routine, Less16c2.asm, will be virtually identical, except it will have a different name, and it will use the creatively renamed loop counters, Loopc3 and Loopc4. However, the section name will be the same:

```

                                global      Sub2

Counters      udata_ovr
Loopc3        res                    1
Loopc4        res                    1

                                code

Sub2

                                movlw      5
                                movwf     Loopc3

Loop1

                                movlw      3
                                movwf     Loopc4

Loop2

                                decfsz   Loopc4,F
                                goto     Loop2
                                decfsz   Loopc3,F
                                goto     Loop1

                                return
                                end

```

Finally, the mainline will do nothing more than call these two subroutines over and over:

```

                                processor   pic16f84a
                                include     p16f84a.inc
                                __config   _XT_OSC & _WDT_OFF & _PWRTE_ON

                                extern      Sub1,Sub2

STARTUP      code
                                goto      Start
                                code

Start

                                call      Sub1
                                call      Sub2
                                goto      Start
                                end

```

### Assembling and testing

When we build this program, it is probably a little easier to use the animate feature of MPSIM to test it. But since the two subroutines don't use their data at the same time, nothing inappropriate happens, even though, as we can see from the map, the variables share the same memory locations:

```

Loopc1  0x00000c    data    static H:\PIC\Lesson16\Less16c1.asm
Loopc2  0x00000d    data    static H:\PIC\Lesson16\Less16c1.asm
Loopc3  0x00000c    data    static H:\PIC\Lesson16\Less16c2.asm
Loopc4  0x00000d    data    static H:\PIC\Lesson16\Less16c2.asm

```

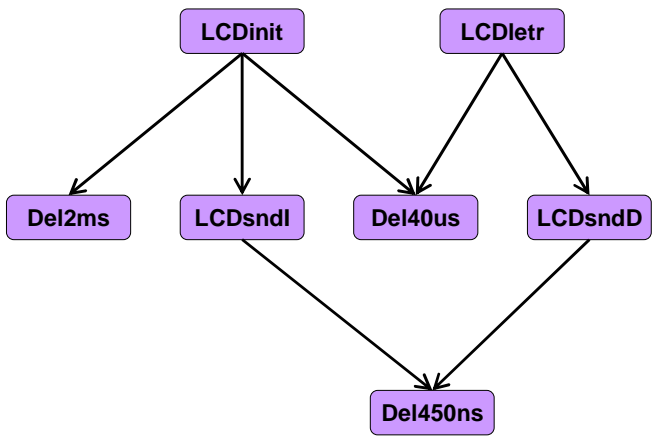
### A note on udata\_ovr

Notice that each subroutine's use of the udata\_ovr section must stand alone. Because a subroutine cannot know what might have happened to that memory before, it must take care to initialize every location it uses before it can count on the contents of that location.

## Libraries

|                           |   |
|---------------------------|---|
| <b>Introduction</b>       | <p>So far we have seen how using relocatable code can help us break up our program into a number of pieces that are somewhat more independent than they are with absolute code. Much of the dogwork of adjusting memory locations is handled by the linker. But some of the real power of this technique becomes evident when we use a library.</p>   |
| <b>What is a library?</b> | <p>In it's simplest terms, a library is simply a collection of already assembled routines that haven't yet been linked, and are stuck together all in one file. A library is a convenient way to package a group of subroutines that have something in common. For example, in the next experiment, we will use a library that contains a group of routines that manipulate the LCD.</p>  |
| <b>Why a library?</b>     | <p>As we begin to do more and more PIC projects, we discover ourselves solving the same problem over and over. The LCD is the stereotypical case. Probably half the PIC projects out there are nothing more than a PIC, an LCD, and some sort of input circuitry.</p> <p>While the LCD is classic, we could see similar situations in code for the DDS, for communicating with general I2C devices, for extended precision math, for Morse generation, and on and on.</p> <p>Often, handling some particular subject requires a number of routines to work in concert. The library provides a convenient way to package a group of related routines, and keep them handy to be used in future projects.</p>   |
| <b>How it works</b>       | <p>When we use a library, the linker does some special things. During linking, the linker sticks together all the object files we mention in the project, and resolves any references between them. Only then does it go looking into the library, and it retrieves only those object modules which include symbols which are unresolved.</p> <p>The linker uses those modules to resolve any unresolved references from the program. However, those modules may have references to other modules. So the linker now goes back to the library (or libraries) again to find the routines needed to resolve those references. It keeps repeating this process until all the loose ends are tied up.</p> <p>As a result of this, the linker will collect anything we need from the library, and nothing else.</p> <p>The linker looks at the object files in the project first, then the libraries, in the order the libraries are listed in the project. This allows us to override individual library modules with our own code without modifying the library.</p> <p>In the example, we will look at how all this actually happens.</p> |

## Using a library

|                               |  |
|-------------------------------|--|
| <b>Introduction</b>           | In this section, we will do another simple program. This time, however, we will use a library to provide some other routines.  |
| <b>Setting up the project</b> | Set up a Less16d project with only one source, Less16d.asm, but don't forget to include 16f84a.lkr as you did in Less16a. Copy LCDlib.lib from the Lesson16.zip file to your project directory. Right click on Libraries in your project window, and add LCDlib.lib to the project.  |
| <b>The project source</b>     | <p>This time the source will start like any other, with our processor, include, and __config directives. We are going to use two routines from the library. The first, LCDinit, initializes the LCD so it will accept commands. The second, LCDletr, displays a letter on the LCD:</p> <pre style="text-align: center;">extern      LCDinit,LCDletr</pre> <p>The actual work will be done by calling LCDinit, then loading a character into the W register and calling LCDletr to display that character:</p> <pre> ; Initialize the LCD call      LCDinit  ; Put something on the LCD movlw    '2' call     LCDletr </pre> <p>We will put our tight aa goto aa at the end to prevent the program from running off into the weeds when it is done, and assemble the program and load it into the PIC-EL.</p> |
| <b>What just happened?</b>    | <p>If you make a map and look at it, you will see that quite a few routines got included, even though we only asked for two. In fact, LCDinit calls a routine that calls a routine, and so on. We didn't need to concern ourselves with any of this, because the linker took care of it for us.</p>  <pre> graph TD     LCDinit --&gt; Del2ms     LCDinit --&gt; LCDsndI     LCDinit --&gt; Del40us     LCDletr --&gt; Del40us     LCDletr --&gt; LCDsndD     LCDsndI --&gt; Del450ns     LCDsndD --&gt; Del450ns </pre>   |

*Continued on next page*



## Using a library, Continued

### What just happened? (continued)

Let's look in a little more detail at just what the linker went through.

First, it took `Less16d.o`, the first object file in our project. (In this case, it happens to be the only object file). Our object file defines two sections, `STARTUP` and `.code`. The linker also knows about the `.cinit` section that is described in the linker script. The linker always tries to put a `.cinit` section in there, even if we don't need it. `.cinit` is always at least two words long.

Our `STARTUP` section must be placed at location `H'0000'`, so the linker places it there. The code is only one word long (which shows as two bytes on the map). However, the linker script tells the linker that locations `H'0000'` through `H'0004'` are reserved for `STARTUP`, so the first available location is `H'0005'`. The linker places `.cinit` there. This makes the next available location `H'0007'` (since `.cinit` is two words long), so that is where the linker places `.code`.

Now the linker looks at the symbols. There are three symbols in the list ... `Start`, `LCDinit`, and `LCDletr`. The linker has just placed `Start` at `H'0007'`, so it can go back to the `goto Start` instruction, and assign `H'0007'` as the target of the `goto`. We have told the linker, via the `extern` directive, that `LCDinit` and `LCDletr` are external, so it is content for now that it doesn't know the targets of those two `call` instructions.

Now that the first object file has been processed, the linker is left with two symbols that are unresolved. It now moves on to the next object file. But in this case, there is no 'next' object file. Having completed all the object files, it moves on to the first library.

Looking through `LCDlib`, the linker discovers that the symbol `LCDinit` is found in the module `LCDinit` so it loads `LCDinit` next. The next available location is `H'0000b'`, so that is where it ends up. The linker now adds the symbols in `LCDinit` to its list of symbols that need to be dealt with. These include `LCDsndI`, `Del2ms`, and `Del40us`. (There is also a reference with no name within `LCDinit`, `_.code_0008`).

Now it looks at its updated symbol list to see what can be crossed off. It discovers that it can now fix up the `call LCDinit` instruction to call location `H'000b'`, so it makes that fixup and crosses `LCDinit` off its list of missing things.

Now it repeats the process of looking for things it needs in the library, next discovering `Del2ms`, which it places at the next available location, `H'0023'`.

While it is doing this, it also is doing the same thing with data locations. It turns out that `Del2ms` and `Del40us` need two general purpose register locations each, and `LCDinit`, `LCDsend`, and `LCDletr` need one location each, which happen to be shared (`H'10` in the GPRs).

If all the possibilities in `LCDlib` were exhausted and there were still unresolved names, the linker would now move on to the next library. But in this case, there is only one library, and all the loose ends are tied up, so the linker can call it a day.

## Further Experiments

|                       |  |
|-----------------------|--|
| <b>Introduction</b>   | We have used two routines from the twenty that make up the LCDlib library. The student may want to explore some of the other routines.   |
| <b>LCDlib.inc</b>     | Included with the sources is an include file, LCDlib.inc. This file contains <code>extern</code> directives for the routines in the library. For the most part, the names of the routines are self-explanatory. The student may want to explore some of these routines.  |
| <b>LCDdig</b>         | On the map for Less16d, the routine LCDdig showed up, even though it wasn't called. LCDdig is actually another entry point in LCDletr. The ASCII representation for the digits 0 through 9 is actually the digit, plus a H'30'. Were we to call LCDletr with a H'6', for example, instead of seeing the digit 6, we would see a special character. LCDdig simply adds H'30' to the W register and falls through to LCDletr.  |
| <b>LCDaddr</b>        | <p>LCDaddr allows the caller to position the next character to be displayed. Thus, if one were to place a 4 in the W register and call LCDaddr, then a call to LCDletr would cause the character to be placed at the fifth character position on the display (the addresses start at zero).</p> <p>For those PIC-EL's with the 8 character display, this works as expected. However, the "16 character" display on some PIC-EL's is actually a 2 line by 8 character display, so characters won't show up on the right half of the display. If you look at LCDmacs.inc in the library source zip you will get some clues as to how to write to the second line of the display.</p> |
| <b>Del1s</b>          | When you are experimenting, it is handy to slow things down so you can see what is displayed. Del1s simply waits around for about a second so you can see what is on the display.  |
| <b>LCDmsg</b>         | LCDmsg is a tricky one, but useful. If you prepare a buffer in the file register that contains ASCII text, preceded by a length, place the <u>address</u> of that buffer into the W register, and then call LCDmsg, the message will be displayed. This routine handles the 8/16 character display issues.   |
| <b>Other Routines</b> | There are a number of other interesting routines in the library. Most have relatively obvious names. If simply guessing how they work doesn't end up with good results, the sources are included for your study.   |

## Wrap Up

---

**Summary**

In this lesson, we have examined how create relocatable code, and how we can use it to make it easier to re-use code that we have written. We have used a library, and were able to perform relatively complex operations while writing very little code, by taking code from a library.

---

**Coming Up**

In the next lesson, we will study the LCD and see how the routines in the LCD library were developed.

---