MASTERs 2012

LAB Manual for 1658 BTL

Bootloading, Application Mapping and Loading Techniques on PIC32

Table of Contents

MICROCHIP

MASTERs Conference

LAB 1:
Getting to know the PIC32 Linker Scripts

Purpose:

This lab helps you to understand the followings.

- Arrangement of PIC32 linker script files.
- selecting the right linker script for the chosen PIC32 device part number and
- contents of the linker scripts

Overview:
In this lab you will walk through the PIC32 linker script files and understand the contents of linker script files. At the end of this lab there is an exercise where you will answer some objective questions.

Procedure :

1. Locating procdefs.ld file

The procdefs.ld file is device specific linker script file and is found inside the folder where PIC32 compiler tools are installed.

In this step you will go to the following path where *procdefs.ld* file for various PIC32 part numbers are located inside their respective folders.

C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\proc

2. Open the correct procdefs.ld file

In the subsequent labs of this class, we will be using PIC32MX795F512L. Therefore; the right linker script file for us is the one that is located in the folder 32MX795F512L. Open the procdefs.ld file in MPLAB X.
To open a file in MPLAB X, in the menu choose **File** -> **Open File** and then browse to the folder where procdefs.ld file is located.

3. Understanding the contents of procdefs.ld file

You will go through the contents of procdefs.ld file. Procdefs.ld file contains following categories.

- Inclusion of Processor-Specific Object File(s)
- Inclusion of Peripheral Libraries
- Base Exception Vector Address and Vector Spacing Symbols
- Memory Address Equates
- Memory Regions
- Configuration Words Input/Output Section Map

Inclusion of processor specific Object File

This section of the processor definitions linker script ensures that the processor specific object file(s) get included in the link. The “processor.o” contains SFR definitions.

```
/*****************************************************************
 * Processor-specific object file.  Contains SFR definitions.
 *****************************************************************
INPUT("processor.o")
```

The INPUT line specifies that processor.o should be included in the link as if this file were named on the command line. The linker attempts to find this file in the current directory. If it is not found, the linker searches through the library search paths (i.e., the paths specified with the -L command line option for the linker).

Inclusion of processor-specific peripheral libraries

This section of the processor definitions linker script ensures that the processor specific peripheral libraries get included.

```
/*****************************************************************
 * Processor-specific peripheral libraries are optional
 *****************************************************************
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512L.a")
```

Base Exception Vector Address and Vector Spacing Symbols

This section of the processor definitions linker script defines values for the base exception vector address and vector spacing.

```
/*****************************************************************
 * For interrupt vector handling
 *****************************************************************
PROVIDE(_vector_spacing = 0x00000001);
_ebase_address = 0x9FC01000;
```

The *_ebase_address* specifies the base address of the interrupt vector table (IVT). By default the IVT is mapped to the address value of *0x9FC01000* in KSEG0 region. PIC32 device supports 64 interrupt vectors. The linker keyword *_vector_spacing* defines the space between any two vectors of IVT. By default it is set to 32 bytes by setting the value of *_vector_spacing* to 1. The vector spacing is calculated using the below formula.

Vector spacing = (_vector_spacing << 5) = (0x00000001 << 5) = 32 bytes

Memory Address Equates

This section of the linker script provides information about certain memory addresses required by the default linker script.

```
/*************************************************************
 * Memory Address Equates
 * _RESET_ADDR      -- Reset Vector
 * _BEV_EXCPT_ADDR  -- Boot exception Vector
 * _DBG_EXCPT_ADDR  -- In-circuit Debugging Exception Vector
 * _DBG_CODE_ADDR   -- In-circuit Debug Executive address
 * _DBG_CODE_SIZE   -- In-circuit Debug Executive size
 * _GEN_EXCPT_ADDR  -- General Exception Vector
 *************************************************************/
_RESET_ADDR              = 0xBFC00000;
_BEV_EXCPT_ADDR          = 0xBFC00380;
_DBG_EXCPT_ADDR          = 0xBFC00480;
_DBG_CODE_ADDR           = 0xBFC02000;
_DBG_CODE_SIZE           = 0xFF0     ;
_GEN_EXCPT_ADDR          = _ebase_address + 0x180;
```

The `_RESET_ADDR` defines the processor's Reset address. This is the address from where an application begins running.

On all forms of reset the processor enters into Bootstrap mode. While the processor is in Bootstrap mode, all interrupts are disabled and all general exceptions are redirected to one interrupt vector address, 0xBFC00380. The `_BEV_EXCPT_ADDR` defines this bootstrap exception address.

The `_DBG_EXCPT_ADDR` defines the address that the processor jumps to when a debug exception is encountered, when using the debugger.

The `_DBG_CODE_ADDR` defines the start address of the debug executive. The debug executive is a small program downloaded into the target device by the debugger along with the user program and is responsible for debugging the user program.

The `_DBG_CODE_SIZE` defines the flash size reserved for debug executive.

The `_GEN_EXCPT_ADDR` defines the address that the processor jumps to when a general exception is encountered and when the processor is not in bootstrap mode.

(Note: Once the interrupt controller of PIC32 is configured for desired mode of operation, The bootstrap mode is exited by setting the control bit BEV to 0)

Memory Regions

This section of the procdefs.ld file provides information about the memory regions that are available on the device.

```
/*************************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for orphaned sections.
 * Only sections specifically assigned to these regions can be allocated
 * into these regions.
 *************************************************************************/
MEMORY
{
  kseg0_program_mem    (rx)  : ORIGIN = 0x9D000000, LENGTH = 0x80000
  kseg0_boot_mem             : ORIGIN = 0x9FC00490, LENGTH = 0x970
  exception_mem              : ORIGIN = 0x9FC01000, LENGTH = 0x1000
  ksegl_boot_mem             : ORIGIN = 0xBFC00000, LENGTH = 0x490
  debug_exec_mem             : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                    : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                    : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  configl                    : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                    : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  ksegl_data_mem       (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x20000
  sfrs                       : ORIGIN = 0xBF800000, LENGTH = 0x100000
  configsfrs                 : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
}
```

Following memory regions are defined with an associated start address and length.

1. Program memory region for application code (`kseg0_program_mem`)
2. C startup code (`kseg1_boot_mem`)
   Any application will begin running from its C start-up code. Therefore; the address value assigned to `_RESET_ADDR` and the value of `kseg1_boot_mem` origin must be same.
3. Interrupt vector table (`exception_mem`).
   The exception_mem must align on a 4KB address boundary. Note that the base address of `exception_mem` and the address assigned to `_ebase_address` must be same.
4. Data memory region (`kseg1_data_mem`)
5. Memory region reserved for debugger code (`debug_exec_mem`)
6. Individual configuration words (`config0, config1, config2` and `config3`).
7. Configuration word memory region (`configsfrs`).
8. Special function registers – peripheral registers (`sfrs`)

The `kseg0_boot_mem` is not used and is reserved for future use.

It is to be noted that `kseg0_program_mem`, `kseg0_boot_mem` and `exception_mem` are mapped into KSEG0 address space. All other memory regions are mapped into KSEG1 address space.

The attributes (`rx`) specify that read-only sections or executable sections can be located into the program memory regions. Similarly, the attributes (`w!x`) specify that sections that are not read-only and not executable can be located in the data memory region. Since no attributes are specified for the boot memory region, the configuration memory regions, or the SFR memory region, only specified sections may be located in these regions.
(i.e., orphaned sections may not be located in the boot memory regions, the exception memory region, the configuration memory regions, the debug executive memory region, or the SFR memory region).

CONFIGURATION WORDS INPUT/OUTPUT SECTION MAP

This section in *procdefs.ld* is not important for the application mapping. This section is input/output section map for Configuration Words. It defines how input sections for Configuration Words are mapped to output sections for Configuration Words. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. All output sections are specified within a SECTIONS command in the linker script.

```
/*********************************************************
 * Configuration-word sections
 *********************************************************
SECTIONS
{
  .config_BFC02FF0 : {
    KEEP(*(.config_BFC02FF0))
  } > config3
  .config_BFC02FF4 : {
    KEEP(*(.config_BFC02FF4))
  } > config2
  .config_BFC02FF8 : {
    KEEP(*(.config_BFC02FF8))
  } > config1
  .config_BFC02FFC : {
    KEEP(*(.config_BFC02FFC))
  } > config0
}
```

For each Configuration Word that exists on the specific processor, a distinct output section named `.config_address` exists, where address is the location of the Configuration Word in memory. Each of these sections contains the data created by the `#pragma config` directive for that Configuration Word in the source code. Each section is assigned to their respective memory region (`confign`).

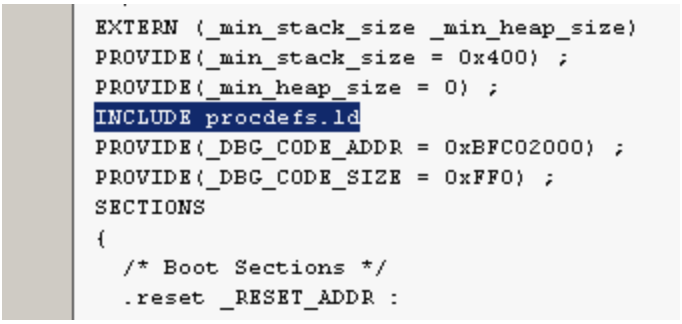
4. Locating elf32pic32mx.x file

The elf32pic32mx.x contains the template of the main linker script file. This is located in the following path where XC32 compiler is installed. Note that this is only a template of the actual linker script. The main linker script which linker uses is internal to the linker.

C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\ldscripts\elf32pic32mx.x

5. Inclusion of Procdefs.ld file in elf32pic32mx.x

Open the file elf32pic32mx.x file in an editor (say MPLAB X). See how the file procdefs.ld file is included into the elf32pic32mx.x using the linker command INCLUDE at line 17.



```
EXTERN (_min_stack_size _min_heap_size)
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
INCLUDE procdefs.ld
PROVIDE(_DBG_CODE_ADDR = 0xBFC02000) ;
PROVIDE(_DBG_CODE_SIZE = 0xFF0) ;
SECTIONS
{
  /* Boot Sections */
  .reset _RESET_ADDR :
```

6. Other contents of elf32pic32mx.x

This file mainly contains the input/output section map, where code sections are mapped into the memory regions defined in procdefs.ld file. The rest of the contents of elf32pic32mx.x is not important for this class and for application/bootloader mapping. However, after this class, it is recommended to read the document “MPLAB C32 User Guide.pdf” to understand more about these sections. The document comes as part of the XC32 installation package and is found in the following path.
C:\Program Files\Microchip\xc32\v1.00\doc\MPLAB-XC32-Users-Guide.pdf

Exercise: Try to answer these questions

1) Which of the following memory regions contain C-Startup code?

a) kseg0_program_mem
b) kseg1_boot_mem
c) kseg0_boot_mem
d) exception_mem

2) Which of the following memory regions contain interrupt vector table?

a) kseg0_program_mem
b) kseg1_boot_mem
c) kseg0_boot_mem
d) exception_mem

3) The reset address of an application is defined by

a) _RESET_ADDR.
b) _BEV_EXCPT_ADDR
c) _ebase_address
d) _DBG_EXCPT_ADDR

4) The _ebase_address must point to base address of _____

a) kseg0_program_mem
b) kseg1_boot_mem
c) kseg0_boot_mem
d) exception_mem

5) The text and data sections (all C files) of an application are mapped in

a) kseg0_program_mem
b) kseg1_boot_mem
c) kseg0_boot_mem
d) exception_mem

LAB 2:
Application Mapping and Bootloader Mapping

Purpose:
Map bootloader and application into non-overlapping memory regions

Overview:
In this lab you will learn how to modify the linker scripts of bootloader and application projects to map the resulting image into non-overlapping memory regions.

Following are the objectives of this lab
- Edit the linker script of the application project to map the application.
- Edit the linker script of the bootloader project to map the bootloader
- Build and program the bootloader into PIC32 starter kit.
- Program the application using the bootloader

The bootloader is a USB HID bootloader and the application is a small application which blinks two LEDs on the starter kit.

Procedure :

1. The Hardware Setup

The hardware setup for this class is shown in figure 2.1.

PROCEDURE

Setup the hardware as per the arrangement shown in figure 2.1

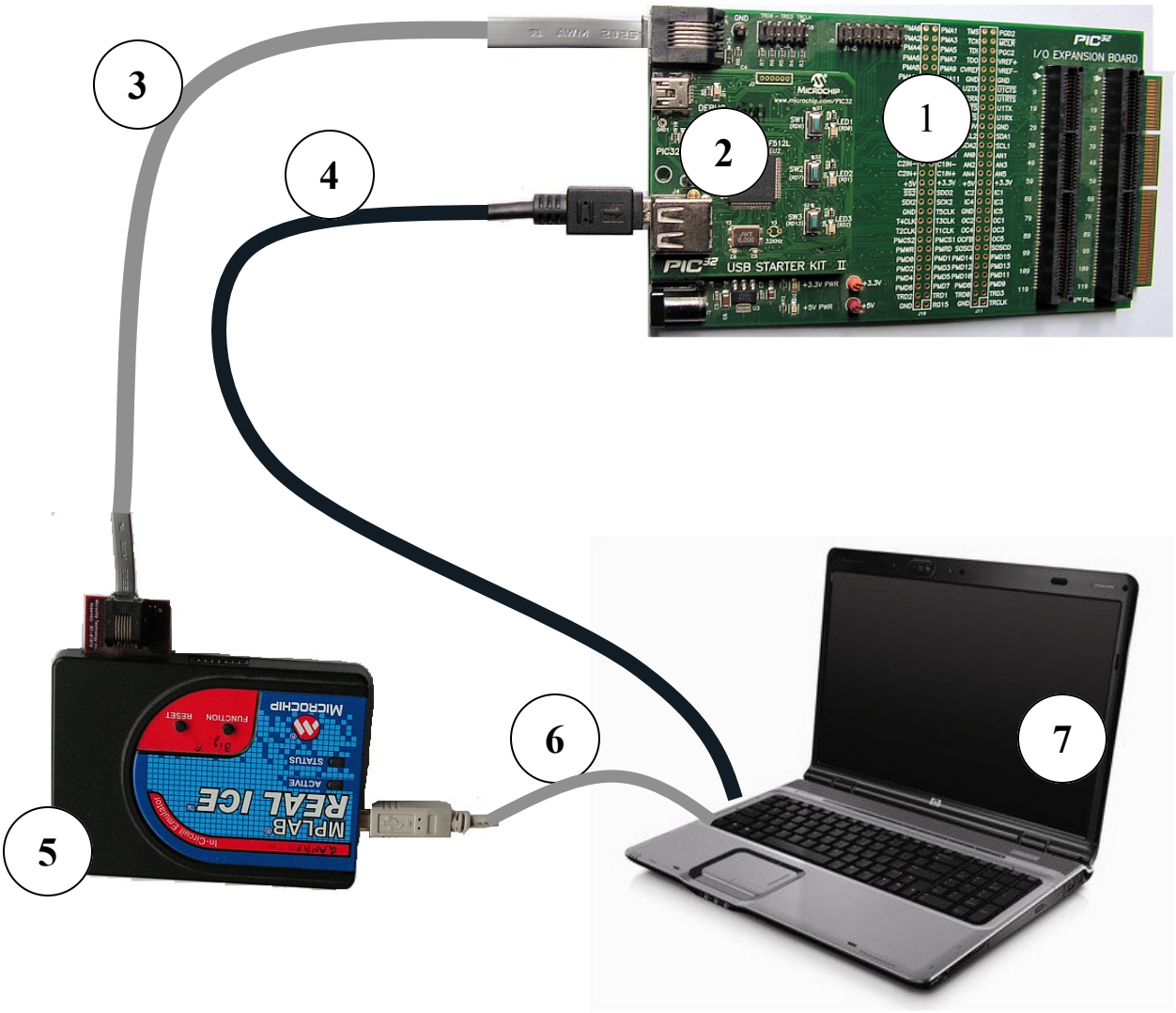


Figure 2.1: Hardware Setup for this Class

1. PIC32 I/O Expansion Board
2. PIC32 USB Starter Kit II
3. RJ11 cable connecting Real ICE and the hardware.
4. USB A to Micro B Cable connecting PC to starter kit (for USB communication). The hardware is powered by this USB connection.
5. MPLAB REAL ICE (Debugger)
6. USB A to B cable (connecting PC and Real ICE)
7. PC

2. Copy the correct procdefs.ld file into the Application project

CHOOSING THE CORRECT LINKER SCRIPT

The "PIC32 starter kit" is mounted with "PIC32MX795F512L". The linker script header file "procdefs.ld" for this part number is found in the following path.

C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\proc\32MX795F512L

PROCEDURE

Copy the procdefs.ld file into the application project "*LAB2/application/application.x*". The LAB2 folder can be found inside 1658 BTL class folder.

3. Application Mapping Scheme for this Lab

There are no procedures in this step. Read the following information.

APPLICATION MAPPING SCHEME

Figure 2.2 in the next page shows the application mapping scheme.
The bootloader chosen for this lab is a USB HID bootloader. The size of the USB HID bootloader is considerably large and cannot be fit entirely inside 12KB boot flash. Hence, the bootloader is split into two parts. One part is mapped into boot flash and the other part is mapped in to lower 24KB of program flash.

The application must be mapped into the remaining part of the program flash, not overlapping with the bootloader. Recall: To remap the application you need to rearrange its memory regions kseg1_boot_mem, exception_mem and kseg0_program_mem.

In the subsequent steps of this lab, you will be editing the linker script to map the application image as per the scheme shown in the figure 2.2.

RULES

Follow these rules while mapping the memory regions of the application.

1. *exception_mem* must align on a 4KB flash page. This is very important for the interrupts to work.
2. It is recommended not to alter the length of exception_mem and kseg1_boot_mem as they contain the interrupt vector table and C-start up code for the application.
3. The rearranged memory regions must not overlap with other memory regions defined in procdefs.ld

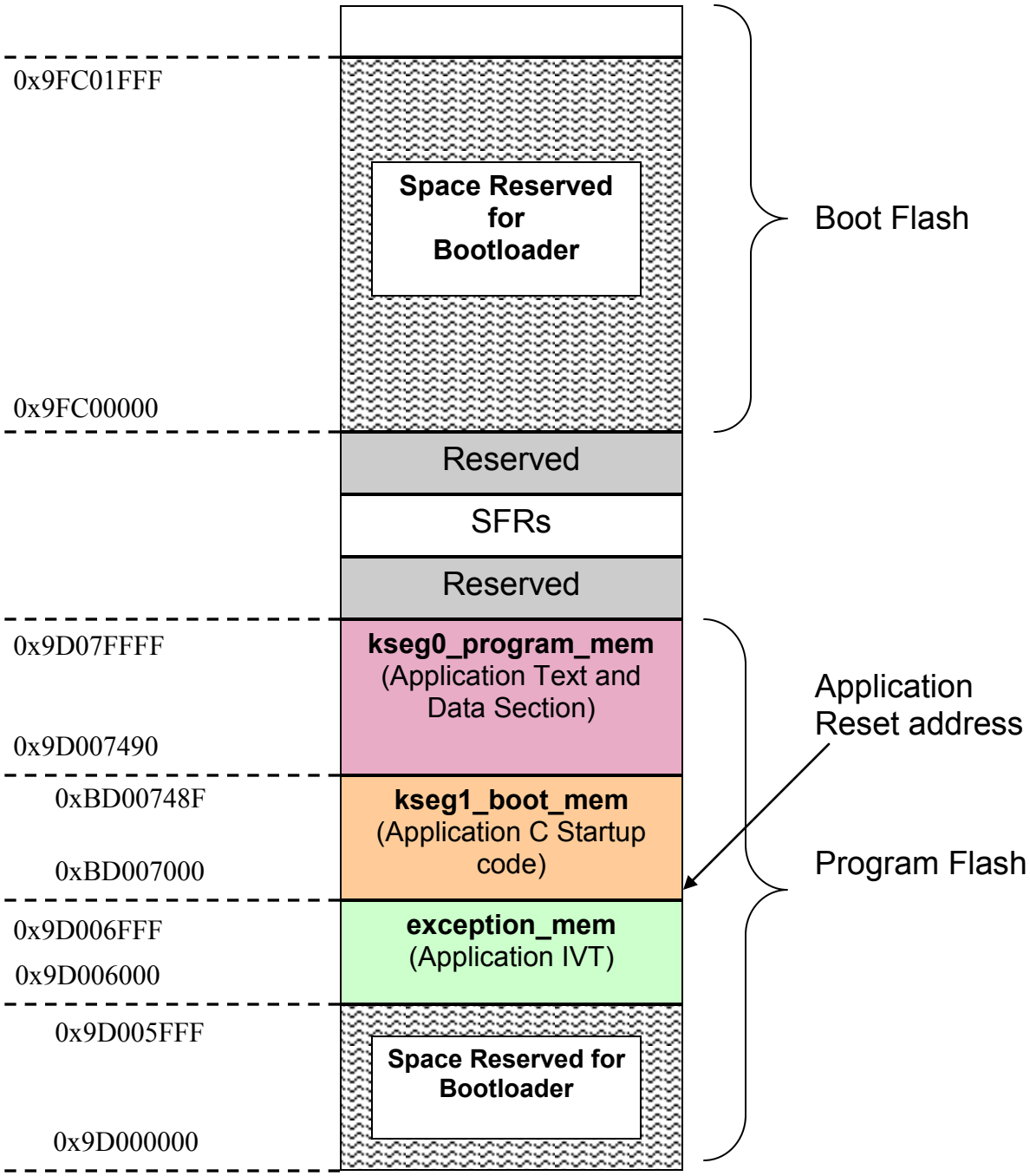


Figure 2.2: Application mapping scheme

4. Mapping the application's IVT

Mapping the IVT involves following two steps.

- Mapping "exception_mem"
- Changing the value of exception base address

PROCEDURE

Mapping exception_mem

You will map the *exception_mem* as per the application mapping scheme shown in figure 2.2. Go to the folder ""*LAB2/application/application.x*"". Open the procdefs.ld file in MPLAB X and change the ORIGIN value of *exception_mem* to 0x9D006000. Do not change the value of LENGTH. You will need this length to support all the 64 interrupt vectors.

```
41  MEMORY
42  {
43    kseg0_program_mem    (rx)   : ORIGIN = 0x9D000000, LENGTH = 0x80000
44    kseg0_boot_mem              : ORIGIN = 0x9FC00490, LENGTH = 0x970
45    exception_mem               : ORIGIN = 0x9D006000, LENGTH = 0x1000
46    kseg1_boot_mem              : ORIGIN = 0xBFC00000, LENGTH = 0x490
```

PROCEDURE

Changing the value of exception base address

The *_ebase_address* must point to the base address of the exception_mem. The value assigned to *_ebase_address* will be loaded into controller's EBASE register by C-startup code as part of the IVT initialization. Change the value of *_ebase_address* to 0x9D006000.

```
11
12  /*****************************************
13   * For interrupt vector handling
14   *****************************************
15  PROVIDE(_vector_spacing = 0x00000001);
16  _ebase_address = 0x9D006000;
17
```

5. Mapping the C-Startup code

PROCEDURE

C-startup code is mapped into kseg1_boot_mem. As per the application mapping scheme shown in figure 2.2, change the ORIGIN of kseg1_boot_mem to 0xBD007000. Note that the address 0xBD007000 falls in KSEG1 region. Keep the value of LENGTH to its default value 0x490.

```
41  MEMORY
42  {
43    kseg0_program_mem    (rx)   : ORIGIN = 0x9D000000, LENGTH = 0x80000
44    kseg0_boot_mem              : ORIGIN = 0x9FC00490, LENGTH = 0x970
45    exception_mem               : ORIGIN = 0x9D006000, LENGTH = 0x1000
46    kseg1_boot_mem              : ORIGIN = 0xBD007000, LENGTH = 0x490
47    debug_exec_mem              : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
```

6. Changing the reset vector

The application always begins running from its C-startup code. After remapping the C-startup code it is necessary to change the reset address value to the base address of remapped kseg1_boot_mem.

PROCEDURE

Change the value of _RESET_ADDR to base address of kseg1_boot_mem i.e. 0XBD007000.

```
25  * _GEN_EXCPT_ADDR  -- General Exception Vector
26  ************************************************
27  _RESET_ADDR                = 0xBD007000;
28  _BEV_EXCPT_ADDR            = 0xBFC00380;
29  _DBG_EXCPT_ADDR            = 0xBFC00480;
30  _DBG_CODE_ADDR             = 0xBFC02000;
```

7. Changing the bootstrap exception vector address and debug exception address

The bootstrap exception vector address must always be at an offset of 0x380 to reset address and must be with in kseg1_boot_mem region. Similarly, the debug exception address must be at an offset of 0x480 to reset address. These are requirements for linker. Linker generates an error if these requirements are not met.

PROCEDURE

- Change the value of _BEV_EXCPT_ADDR to _RESET_ADDR + 0x380.
- Change the value of _DBG_EXCPT_ADDR to _RESET_ADDR + 0x480.

```
25   * _GEN_EXCPT_ADDR  -- General Exception Vector
26   *********************************************************
27  _RESET_ADDR                 = 0xBD007000;
28  _BEV_EXCPT_ADDR             = 0xBD007380;
29  _DBG_EXCPT_ADDR             = 0xBD007480;
30  _DBG_CODE_ADDR              = 0xBFC02000;
31  _DBG_CODE_SIZE              = 0xFF0     ;
32  _GEN_EXCPT_ADDR             = _ebase_address + 0x180;
```

8. Mapping the kseg0_program_mem

The text and data sections of the application is mapped into kseg0_program_mem.

PROCEDURE

As per the application mapping scheme shown in figure 2.2, change the origin and length of kseg0_program_mem to 0x9D007490 and 0x78B70 respectively.

```
40   ************************************************************************
41  MEMORY
42  {
43    kseg0_program_mem     (rx)   : ORIGIN = 0x9D007490, LENGTH = 0x78B70
44    kseg0_boot_mem               : ORIGIN = 0x9FC00490, LENGTH = 0x970
45    exception_mem                : ORIGIN = 0x9D006000, LENGTH = 0x1000
46    kseg1_boot_mem               : ORIGIN = 0xBD007000, LENGTH = 0x490
47    debug_exec_mem               : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
```

9. Copy the main linker script template file and rename it

The main linker script template file is found in the following path.
C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\ldscripts\elf32pic32mx.x

PROCEDURE

Copy *elf32pic32mx.x* file into the application project folder "**LAB2/application/application.x**" found inside the class folder. After copying the file rename the file to **elf32pic32mx.ld**.

10. Add the linker script file into the project workspace

PROCEDURE

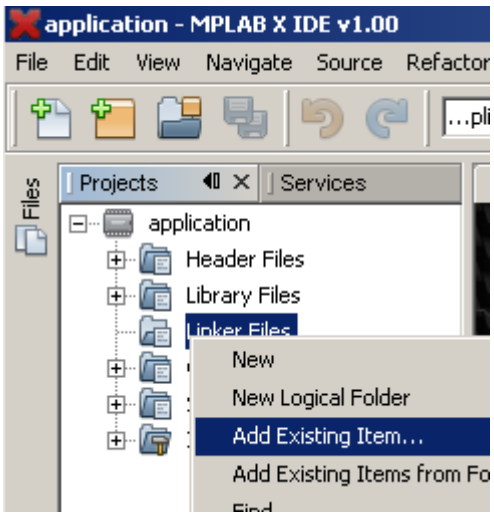
Open MPLAB X IDE. Close any open projects by selecting **File** -> **Close All Projects** from the IDE menu. Open the application project "**LAB2/application/application.x**".

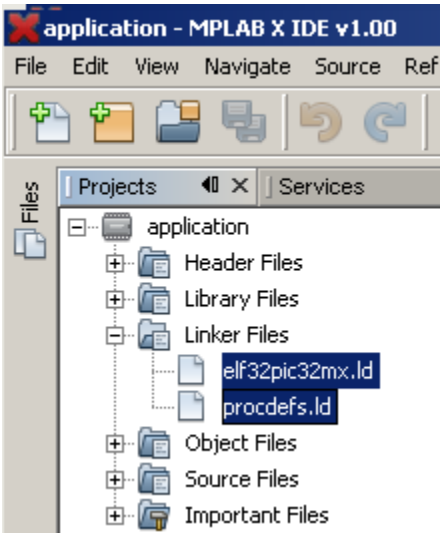
- To open a project, from the IDE menu select **File** -> **Open Project**.
- Navigate to the directory **"LAB2/application"** and select the project directory **"application.x"**

PROCEDURE

To add the linker scripts you will have to follow the below mentioned steps.

- In the projects view, select "Linker Files" and right click.
- In the pop-up menu, choose "Add Existing Item".
- In the subsequent window that appears, browse to "**LAB2/application/application.x"**. Select and add the files "elf32pic32mx.ld" and "procdefs.ld".





11. Clean and build application project

PROCEDURE

Clean and build the application project by pressing  in the MPLAB X IDE. Make sure the project builds with no errors.

12. Verify the application map

PROCEDURE

Open the ".map" file in the path **"LAB2/application/application.x/ application.map".** Verify the mapping of remapped memory regions kseg0_program_mem, exception_mem and kseg0_program_mem. Make sure they are generated as per our application mapping scheme shown in figure 2.2.

application.map

```
132
133 Memory Configuration
134
135 Name             Origin             Length             Attributes
136 kseg0_program_mem 0x9d007490          0x00078b70         xr
137 kseg0_boot_mem    0x9fc00490         0x00000970
138 exception_mem     0x9d006000         0x00001000
139 kseg1_boot_mem    0xbd007000         0x00000490
140 debug_exec_mem    0xbfc02000         0x00000ff0
141 config3           0xbfc02ff0         0x00000004
142 config2           0xbfc02ff4         0x00000004
143 config1           0xbfc02ff8         0x00000004
144 config0           0xbfc02ffc         0x00000004
145 kseg1_data_mem    0xa0000000         0x00020000         w !x
146 sfrs              0xbf800000         0x00100000
147 configsfrs        0xbfc02ff0         0x00000010
148 *default*         0x00000000         0xffffffff
```

CONGRATULATIONS: You just Completed Application Mapping

You just completed the application mapping. In the subsequent section of this lab you will be performing bootloader mapping. Bootloader must be mapped such that it does not overlap with the application in the PIC32 flash. To map the bootloader you shall repeat most of the steps followed for the application mapping.

13. Copy the procdefs.ld file into the bootloader project

PROCEDURE

Copy the procdefs.ld file into the bootloader project folder "*LAB2/bootloader/bootloader.x*" found inside the class folder from the following path

C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\proc\32MX795F512L

14. Bootloader Mapping Scheme

There are no procedures in this step. Read the following information.

BOOTLOADER MAPPING SCHEME

Figure 2.3 shows the mapping scheme for bootloader. You need to map the bootloader such that it does not overlap on the application.

Since the USB HID bootloader is considerably large, the bootloader cannot be entirely fit into the boot flash. The bootloader C startup code and IVT is mapped into the boot flash. The text and data sections of the bootloader is mapped into the 24kB of program flash (in the lower address range).

The origin and length of *kseg1_boot_mem* and *exception_mem* memory regions are not altered. Only the length of *kseg1_program_mem* is shrunk, so that it does not overlap with the memory region reserved for the application.

In the subsequent steps of this lab you will map the bootloader as per the scheme shown in the figure 2.3

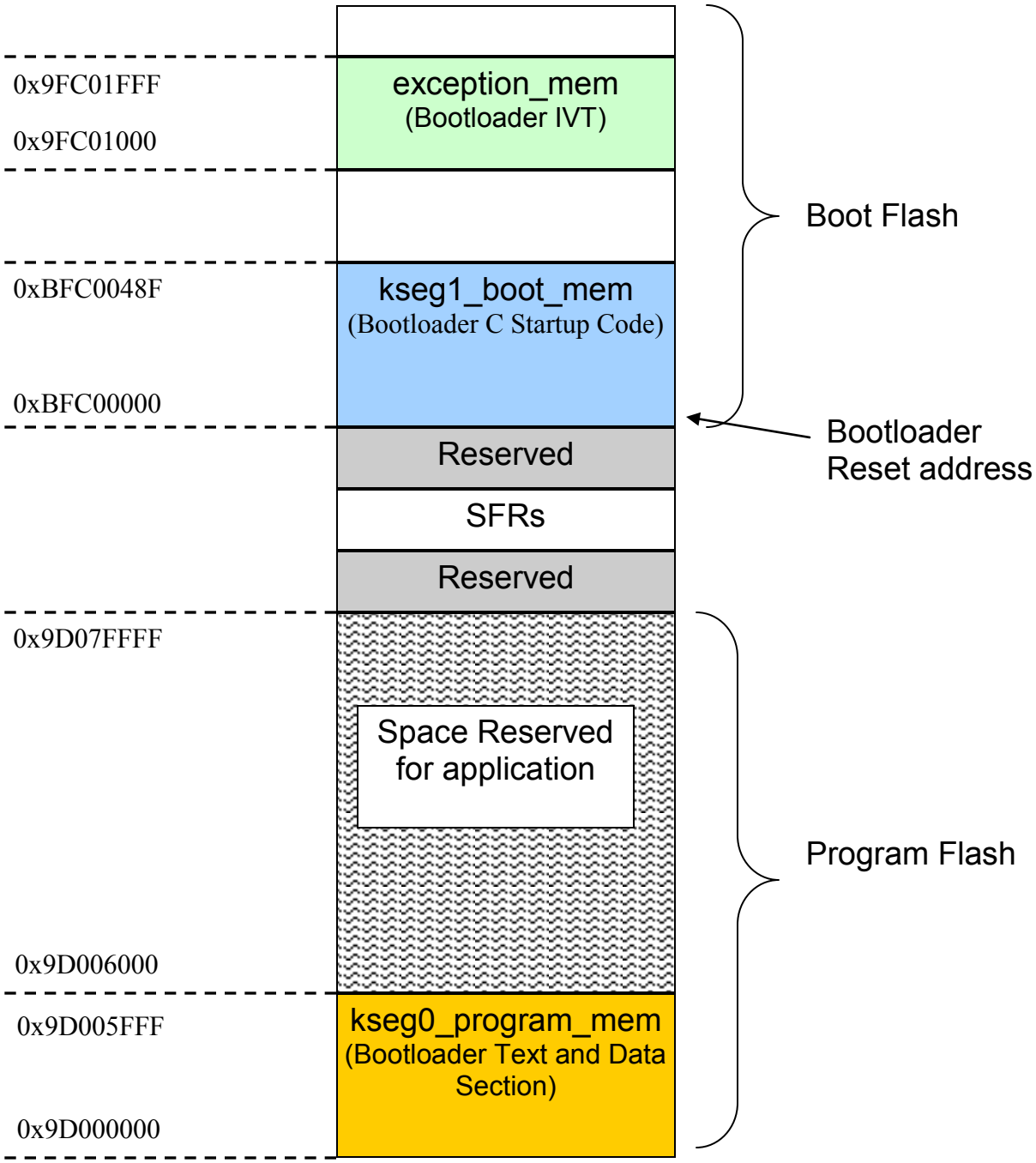


Figure 2.3: Bootloader mapping scheme

15. Shrink the length of kseg1_program_mem

Do not alter the origin and length of kseg1_boot_mem and exception_mem. Alter only the length of kseg0_program_mem.

PROCEDURE

Go to the folder ““*LAB2/bootloader/bootloader.x*”. Open the procdefs.ld file in MPLAB X. Change length of kseg0_program_mem to 0x6000 as per the boot-loader mapping scheme shown in figure 2.3.

```
37   * Memory regions without attributes cannot be used for orphaned secti
38   * Only sections specifically assigned to these regions can be allocat
39   * into these regions.
40   ************************************************************************
41  MEMORY
42  {
43    kseg0_program_mem     (rx)   : ORIGIN = 0x9D000000, LENGTH = 0x6000
44    kseg0_boot_mem               : ORIGIN = 0x9FC00490, LENGTH = 0x970
45    exception_mem                : ORIGIN = 0x9FC01000, LENGTH = 0x1000
```

16. Copy the main linker script template file and rename it

The main linker script template file is found in the following path.
C:\Program Files\Microchip\xc32\v1.00\pic32mx\lib\ldscripts\elf32pic32mx.x

PROCEDURE

Copy *elf32pic32mx.x* file into the bootloader project directory “**LAB2/ bootloader/bootloader.x**”. The LAB2 folder is located inside the MASTERs class directory.

PROCEDURE

After copying the file rename the file to **elf32pic32mx.ld**.

17. Add the linker script file into the project workspace

PROCEDURE

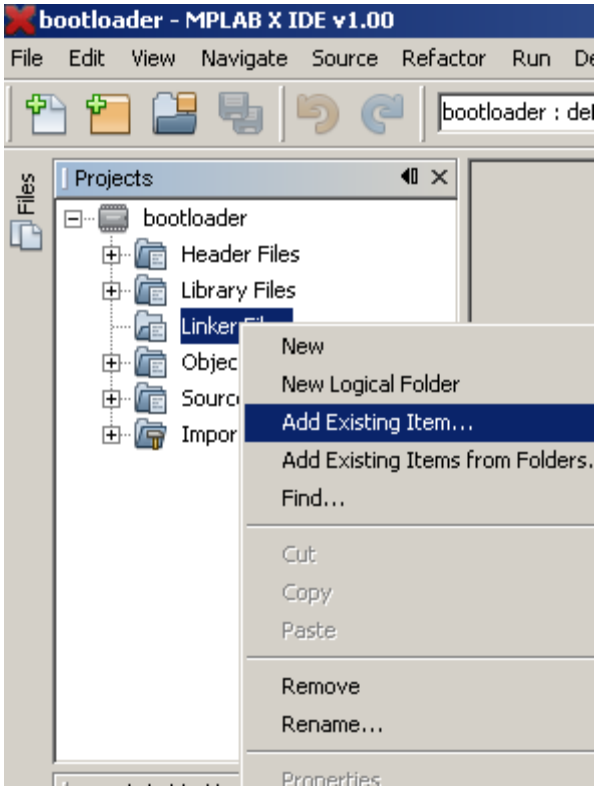
Open MPLAB X IDE. Close any open projects by selecting **File** -> **Close All Projects** from the IDE menu. Open the bootloader project "**LAB2/bootloader/bootloader.x**".

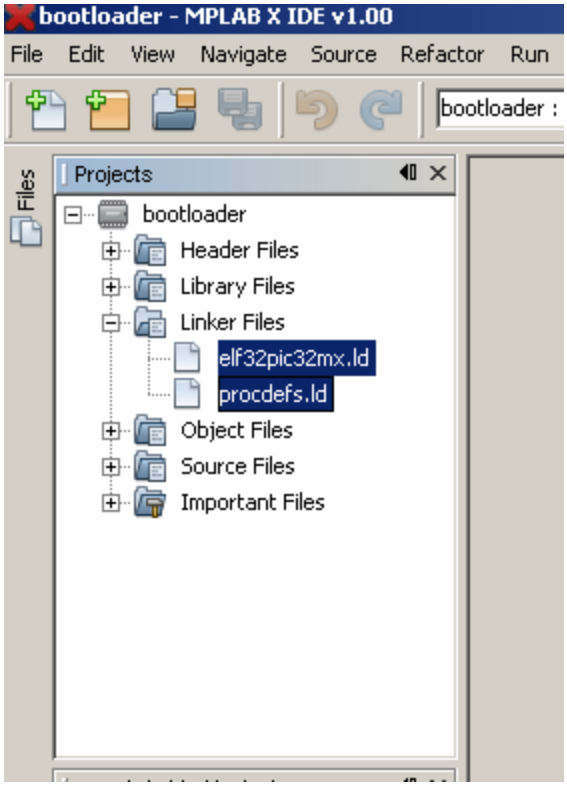
- To open the project, from the IDE menu select **File** ->**Open Project**.
- Navigate to the directory **"LAB2/bootloader"** and select the project directory **"bootloader.x"**

PROCEDURE

To add the linker scripts you will have to follow these steps.

- In the projects view, select and right click on "Linker Files"
- In the pop-up menu, choose "Add Existing Item".
- In the subsequent window that appears browse to "**LAB2/bootloader/bootloader.x"**. Select and add the files "elf32pic32mx.ld" and "procdefs.ld".



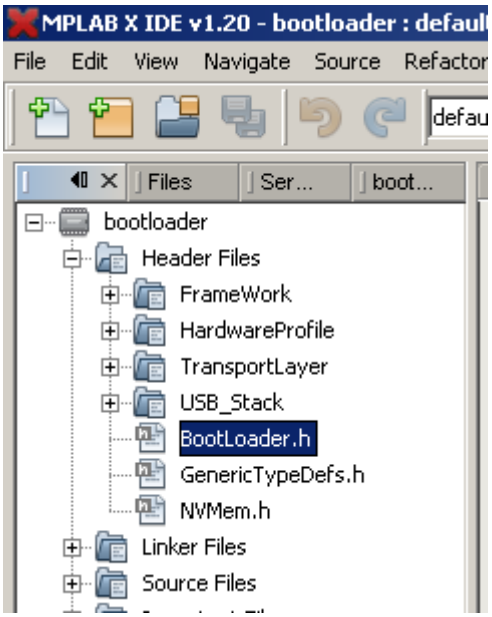


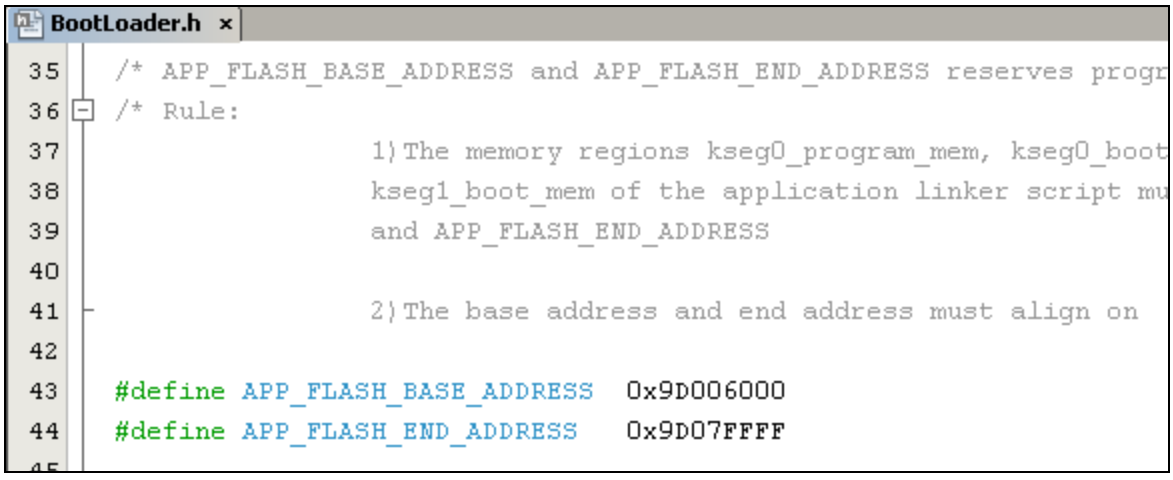
18. Inform the bootloader about the location of the application in program flash

Bootloader needs to know the location of the application to perform the erase and program operations. The macros APP_FLASH_BASE_ADDRESS and APP_FLASH_END_ADDRESS in the bootloader code defines the base and end addresses of the memory region reserved for the application. As per the bootloader mapping scheme shown in figure 2.3, application occupies the flash address range from 0x9D006000 to 0x9D07FFFF respectively.

PROCEDURE

In the "bootloader.h" file, set the value of macros APP_FLASH_BASE_ADDRESS and APP_FLASH_END_ADDRESS to 0x9D006000 to 0x9D07FFFF respectively.





19. Inform the bootloader about the reset address of the application

The bootloader must jump to the reset address of the application when there is a condition to run the application. You have to inform the bootloader about this reset address by setting the macro USER_APP_RESET_ADDRESS. The reset address of the application must point to the base address of its *kseg1_boot_mem* as it contains C startup code. Refer to the application mapping scheme shown in figure-2.2, the base address of the application kseg1_boot_mem is 0xBD007000.

PROCEDURE

In the "bootloader.h" file, set the value of USER_APP_RESET_ADDRESS to 0xBD007000.

BootLoader.h

```
45
46  /* Address of  the Flash from where the application starts execut
47  /* Rule: Set APP_FLASH_BASE_ADDRESS to _RESET_ADDR value of appli
48  #define USER_APP_RESET_ADDRESS  0xBD007000
49
```

20. Clean and build bootloader project

PROCEDURE

Clean and build the bootloader project by pressing  in the MPLAB X IDE. Make sure the project builds with no errors.

21. Program the bootloader into the Hardware setup

In this step, you will use Real ICE to program the bootloader into the hardware.

PROCEDURE

Before programming the hardware, make sure of the followings….

- Real ICE is connected to PC and the hardware.
- Make sure that the USB A to micro B cable is connected between the PC and USB starter Kit-II. The hardware must be powered by this USB connection.

Program the Bootloader by clicking  on MPLAB X IDE. Make sure that the bootloader is programmed into the hardware with no errors.

Note: Programming may take some time.

22. Reset the device and put the bootloader in firmware upgrade mode

In this step you will reset the device and put the bootloader in firmware upgrade mode.

PROCEDURE

Follow these procedures to put the bootloader in firmware upgrade mode.

- Remove the Real ICE connection from the hardware.
- Unplug the USB cable from the PIC32 starter kit.
- Keeping the switch SW3 on PIC32 USB starter kit pressed, plug the USB cable back to the starter kit. This will put the bootloader in firmware upgrade mode.
- On successfully entering the firmware upgrade mode, the bootloader blinks the LED3 on starter kit.

23. Program the application using the bootloader

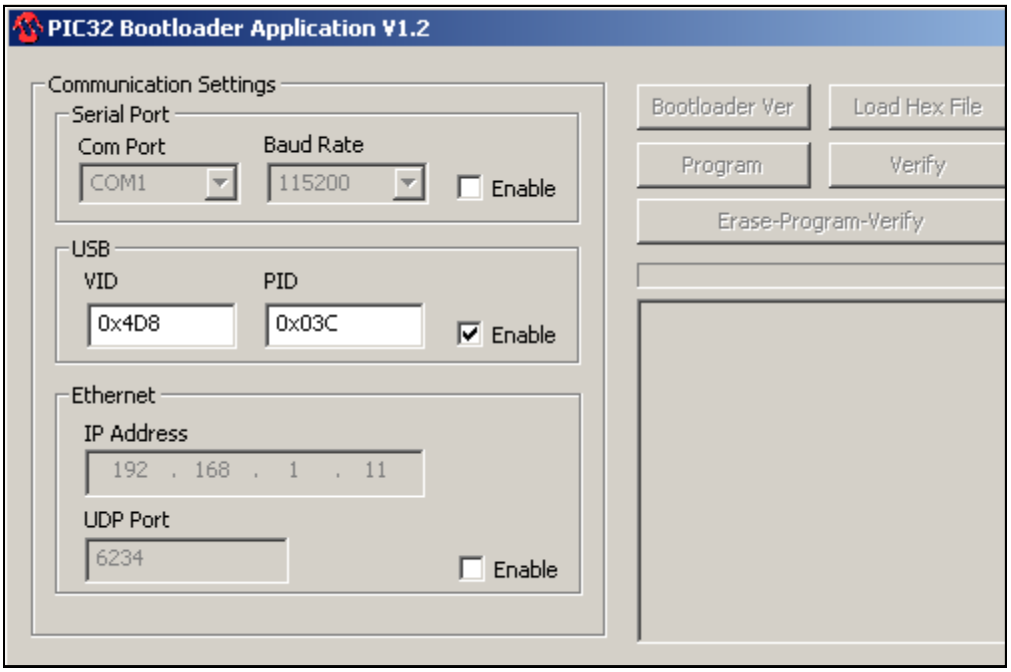
In this step you will program the application using the bootloader.

PROCEDURE

Double click and run the PC application PIC32UBL.exe. The PC application can be found in the path “LAB2/pc_application/” inside the class folder.

PROCEDURE

In the PC application, enable USB under the communication settings.

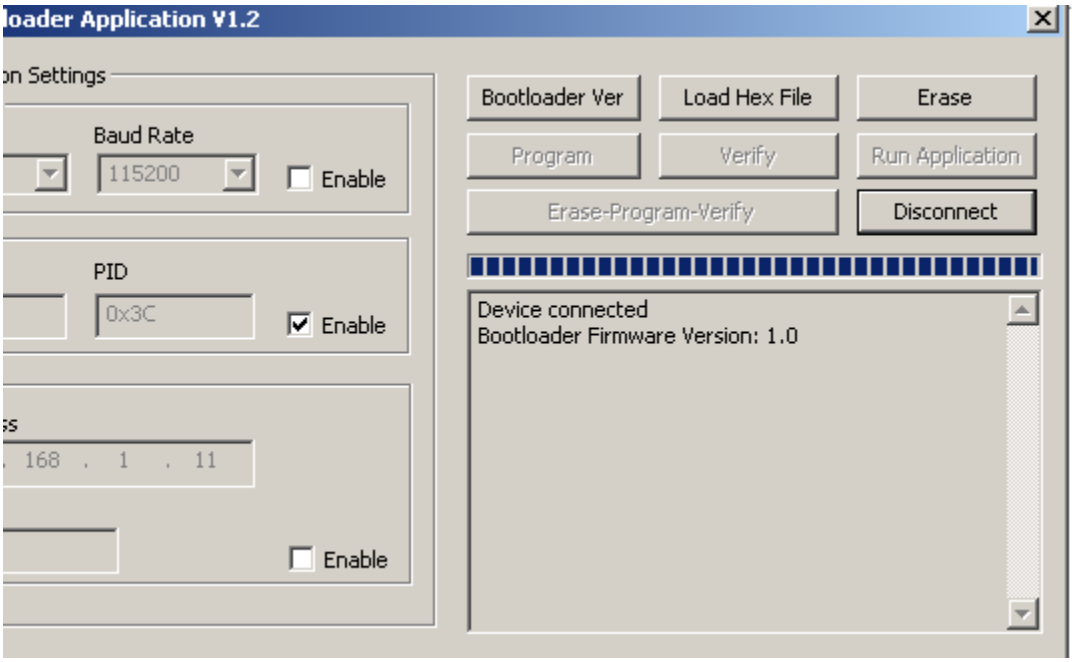


PROCEDURE

Connect to the bootloader by clicking on “Connect” in the PC application.

PROCEDURE

On successful connection the PC application reads the bootloader version as shown in the screenshot below.



PROCEDURE

Erase the application memory region by clicking on “Erase”. Erase will take some time before the console updates the result of erase operation. On successful erase you will see “Flash Erased” message on the PC application console.

PROCEDURE

You will be loading the application hex file into the PC application. Click on the “Load Hex File”.
Browse to “LAB2/application/application.x/dist/default/production” and load the “application.x.production.hex” file.

PROCEDURE

Now program the loaded application hex file by clicking “Program”. On successful programming you will see “Programming Completed” message on the PC.

PROCEDURE

After the completion of programming, click “Verify”. Verification succeeds if application is programmed correctly.

PROCEDURE

Click on “Run Application” to begin running the application. The application causes LED1 and LED2 to blink on the starter kit.

Try resetting the hardware. To reset the hardware unplug and plug the USB cable back to PIC32 starter kit. You will see that bootloader directly jumps to application without going into the firmware upgrade mode.

To put the bootloader back into firmware upgrade mode, reset the hardware keeping the switch SW3 on PIC32 starter kit pressed. You will see the LED LED3 blinking, which indicates that the bootloader is in firmware upgrade mode. Once in firmware upgrade mode, you can again program the application.

Conclusion

In this lab you learnt how to map the application and bootloader images in to different non-overlapping memory regions. You also learnt how to program an application using the bootloader.

LAB 3:
Run Time Library Loading

Purpose:

Using Run Time Library Loading (RTLL) technique link a simple math library to an application.

Overview:

In this lab you will understand how to use Microchip provided framework to link a library with an application at run time.

Following are the objectives of this lab

- Export the APIs from a simple Math Library.
- Compile and program the library and application separately.
- Open the library and call the library APIs from application using RTLL technique.

The hardware setup for this class is same as lab-2 shown in figure 2.1.

Procedure :

1. Application and Library Mapping

The “application” and “library” must be mapped into non overlapping flash memory regions. This is similar to bootloader and application mapping procedure that we learnt in Lab-2. The library and application must have custom linker scripts to map them into different memory locations.

The library is mapped in the program flash address region from 0x9D06E000 to 0x9D07FFFF. The application occupies the remaining portion of the memory region as shown in Figure 3.1.

Since you have already learnt the mapping technique in the previous lab, you will skip those steps in this lab. This lab will focus on RTLL technique only.

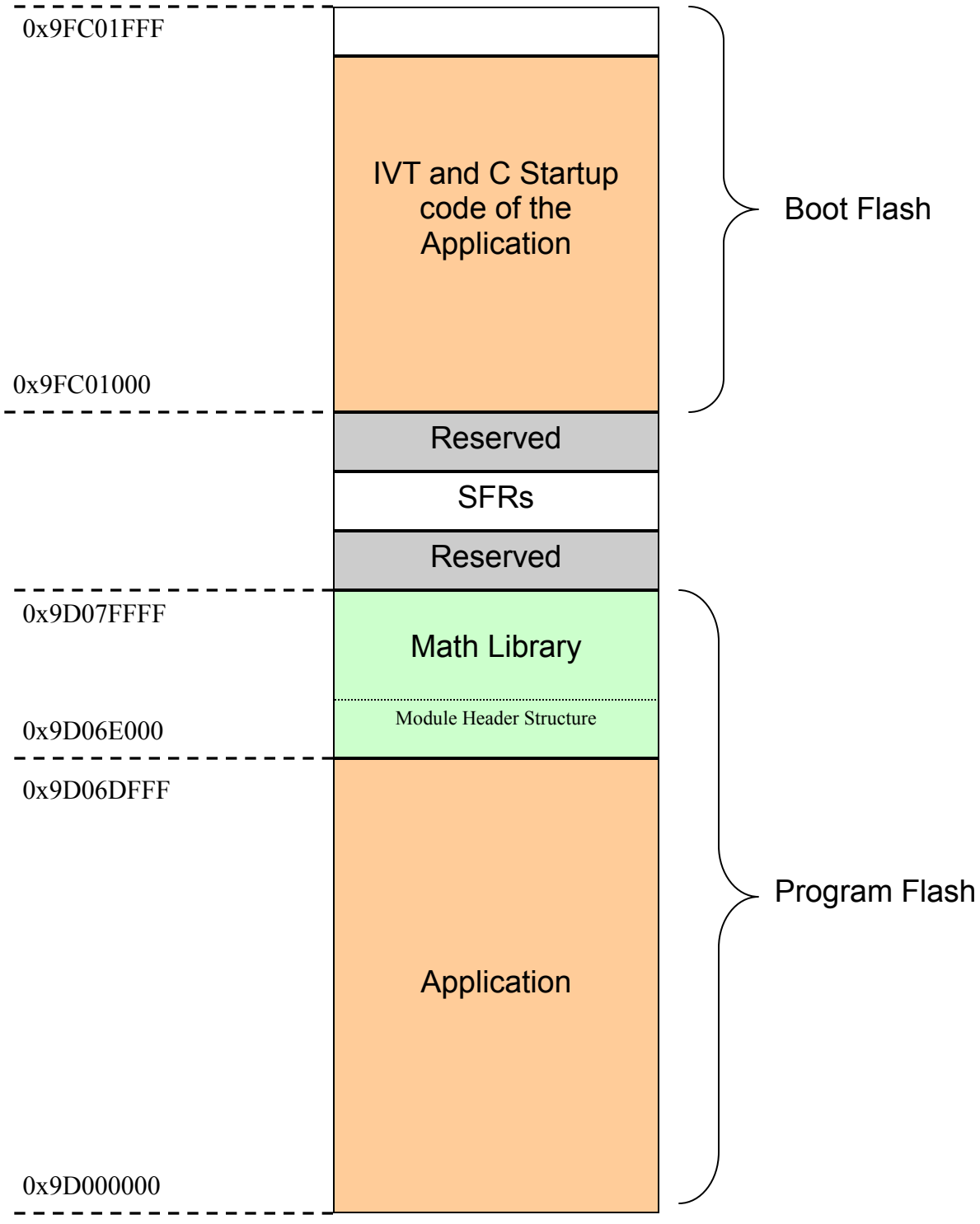


Figure 3.1: Application and Library mapping

2. Get to know the Math Library

In this step you will open the math library project and observe the functions implemented in the library.

PROCEDURE

You may have to close any open projects by selecting **File** -> **Close All Projects** in MPLAB X. Open the library project.

- To open the project, from the IDE menu select **File** -> **Open Project**.
- Navigate to the directory **"\LAB3\Demo\my_lib",** select and open the project **"my_lib.x"**

OBSERVE THE LIBRARY FUNCTIONS

In the project view, under the source files, open the "my_math_lib.c" file. Observe the functions implemented in this file. The file contains two functions.

- myAdd(): Adds two parameters.
- myMul(): Multiplies two parameters.

The function *myAdd()* takes two arguments and returns its sum.

```
/*************************************************
   Library Add Function.

*************************************************/
int myAdd(int fArg, int sArg)
{
        // Return the sum.
        return(fArg + sArg);
}
```

OBSERVE THE LIBRARY FUNCTIONS

The function *myMul()* takes three arguments, two arguments are pointers to operand and the third argument is pointer to the result.

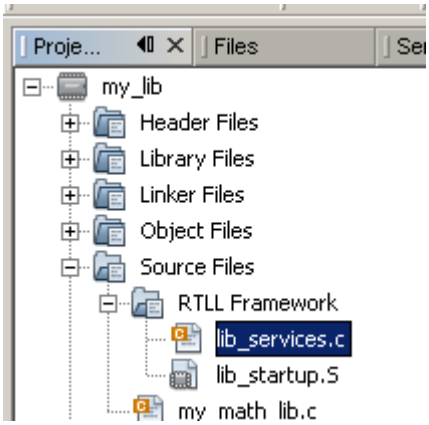
```
/*************************************************
 Library Multiply Function
*************************************************/
void myMul(int *fArg, int *sArg, int *sRes)
{

        // Return the multiplication result.
        *sRes = (*fArg) * (*sArg);

}
```

3. Add Module Header Name in to the Library

PROCEDURE

In this step you will embed a unique string into the library. You will have to remember this string as it is needed to open the library from the application.

- In the project view, under *source files-> RTLL Framework*, open the file "lib_services.c".
- In the "lib_services.c" there is a macro "_MODULE_NAME_". You will have to assign a string to this macro. For example; you can give a name like "@MyMathLibrary". This should be a string, put it in quotes.




```
45
46    // TODO: Define the library name here
47    #define _MODULE_NAME_     "@MyMathLibrary"
48
```

4. Exporting the procedures from the library

PROCEDURE

This step will expose the procedures *myAdd()* and *myMul()* of the library to the application.

- In “lib_services.c” there is an array of structure “*_exportProcTbl[ ]*” that contains the procedure name (as string) and handle to the procedure. To export the procedures you have to initialize this array with procedure name and procedure handle. Note; Procedure name must be a string enclosed in quotes.

```
50    const T_PROC_DCPT _exportProcTbl[] =
51    {
52      // TODO: Add Procedure Name and
53      // Procedure Handle to be exported.
54
55            //Proc Name, Proc Handle
56            {"myAdd",     (void *)myAdd },
57            {"myMul",     (void *)myMul }
58    };
```

5. Clean and Build the Library Project

PROCEDURE

Clean and build the library project "*my_lib*" by pressing  in the MPLAB X IDE. Make sure the project builds with no errors.
You can ignore the warning "`cannot find entry symbol _reset; not setting start address`".

6. Program the library hex file into the hardware

In this step you will program the library image into the hardware.

PROCEDURE

Before programming the hardware, make sure of the followings….

- Real ICE is connected to PC and the hardware.
- Make sure that the USB A to micro B cable is connected to PC and USB starter Kit-II. The hardware must be powered by this USB connection.

Program the library hex file by clicking  in MPLAB X IDE.

Make sure that the library is programmed into the hardware with no errors.
You can ignore the warning "`cannot find entry symbol _reset; not setting start address`".

7. Open the application project

In this step you will open the application project. In the subsequent steps you will call the library APIs from the application.

PROCEDURE

Open the application project in MPLAB X IDE. Note that you may have to close the library project by selecting **File** -> **Close All Projects**.

- To open the project, from the IDE menu select **File** -> **Open Project**.
- Navigate to the directory **"\LAB3\Demo\main_app",** select and open the project **"MainApp.X"**

8. Declare the prototypes of library function

There is nothing to do in this step. Just observe how the prototype of the library functions are declared in the application.

OBSERVE LIBRARY FUNCTION PROTOTYPES

The application must know the prototypes of library function. In the project view, under the source files, open the “main.c” file. Observe how the prototype of the “add()” and “mul()” functions are declared just before the main().The prototype must match the functions defined in the library.

```
52  #include "main_services.h"
53  #include "RTLL/app_services.h"
54
55  //Declare the prototypes of the librar
56  int (*add)(int  , int );
57  void (*mul)(int * , int *, int * );
```

9. Opening the Library from the application

This step shows the usage of dlopen() function provided by RTLL framework. This step is a coding step, where you will open the library using *dlopen().*

PROCEDURE

The *dlopen()* function is already included in the *main.c* file. The *dlopen()* function returns a valid handle to the library if library exists.

You just have to pass math library name and library address to *dlopen()* function as parameters.

- Library name is the string assigned to the macro _MODULE_NAME in step -3 of this lab. This is case sensitive.
- The library address is the address location of the “Library Module Header”. Refer ***Figure3.1: Application and Library Mapping*** in step 1. The Module Header is placed at address 0x9D06E000.

```
79      // get the library handle using dlopen() function.
80      // TODO: Input library name and library address to dlopen()
81      hMyLib = dlopen( "@MyMathLibrary" , (void*)0x9D06E000 ); // I
82
```

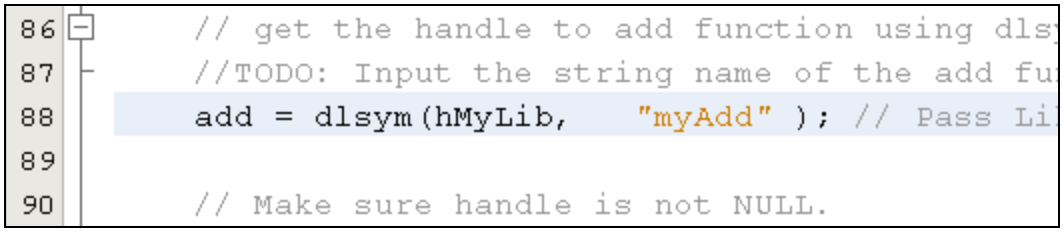
10. Get the handle to “add” function of the library

This step shows the usage of *dlsym()* function provided by RTLL framework. The *dlsym()* function is already included in the *main.c* file. The *dlsym()* function returns a valid handle to “add” function of the library.

PROCEDURE

You just have to pass library handle and function name as string to *dlsym().*

- Library handle is the handle returned from *dlopen()* in step 9.
- Function name is the name given to myAdd() function inside *“_exportProcTbl[ ]”* in step 4. This is case sensitive.



11. Get the handle to “mul” function of the library

PROCEDURE

Similar to previous step, get the handle to “mul” function.

```
93      // get the handle to add function using
94      //TODO: Input the string name of the mu
95      mul = dlsym(hMyLib,    "myMul" ); // Pas
96      ASSERT(mul != 0);
97
```

12. Clean and build the application project

PROCEDURE

Clean and build the application project “*MainApp*” by pressing  in the MPLAB X IDE. Make sure the project builds with no errors.

13. Program and Debug the application project

In this step you will program and debug the application project to realize how the application opens the library and call its functions using RTLL technique.

PROCEDURE

Before programming the hardware, make sure of the followings….

- Real ICE is connected to PC and the hardware.
- Make sure that the USB A to micro B cable is connected to PC and USB starter Kit-II. The hardware must be powered by this USB connection.

Set Breakpoints

Set breakpoints on the following lines in main.c. Note that you can toggle breakpoint on a selected line by pressing “Ctrl+F8”.

- The line where “dlopen()” function is called.
- The lines where “dlsym()” function are called.

Debug the project

To debug the project click on  in the MPLAB X IDE. Make sure that the debugger programs the hardware with no errors. You can also debug the program step by step by pressing function key F8. Verify the result of add and multiplication operation.

Conclusion

In this lab you learnt ...

- Build and program application and library images separately.
- Making use of Microchip’s RTLL framework to link application and library at run time.

DO NOT FORGET TO READ THE APPENDICES

After this lab, take sometime to read the Appendices found in the subsequent sections of this lab material.

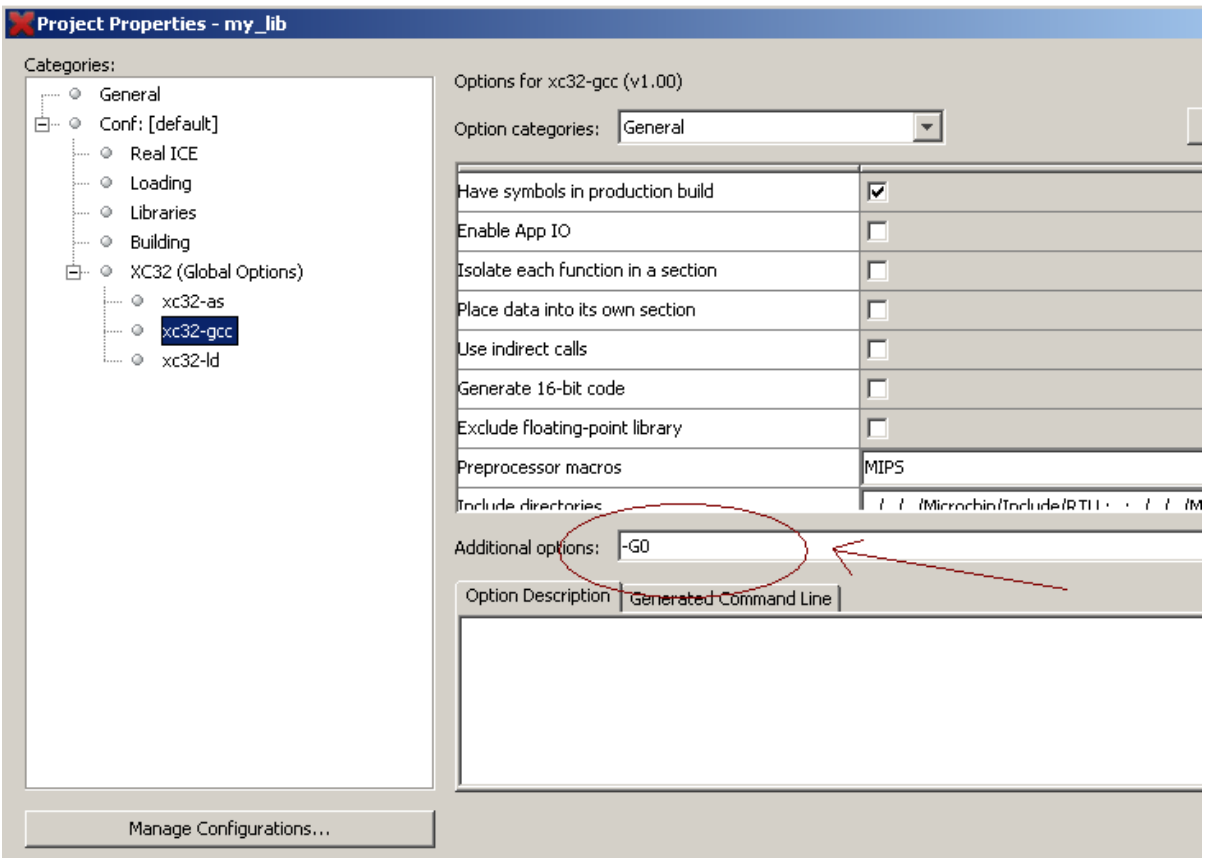
Appendix A: Library Project Settings for the RTLL technique

This appendix describes the project settings for the library. These settings were already saved in the workspace (my_lib.x) and hence were not included in the lab. However, if you wish to create a new library project you must not forget to add these settings.

Disable gp-relative adressing in the library

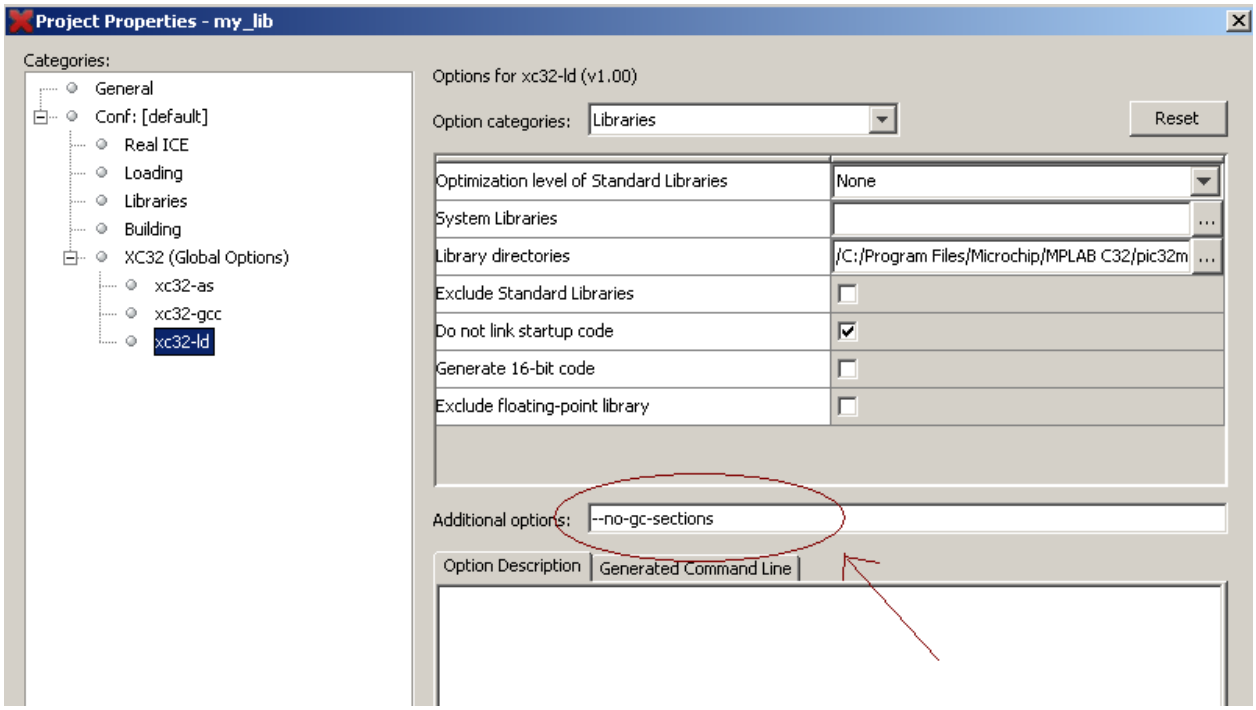
The C compiler supports global-pointer relative (gp-rel) addressing. By using gp-relative addressing, microcontroller saves an instruction cycle to access global data. The gp-relative addressing uses gp register of the microcontroller. If both application and library try to use the gp-relative addressing, the gp register will have to be shared between application and library. This will complicate the RTLL design since the gp register has to be saved during every context switch.

To keep the design simple, gp-relative addressing is disabled in the library. The gp-relative addressing is disabled by setting the compiler option –G0 in the library project properties (my_lib.x).



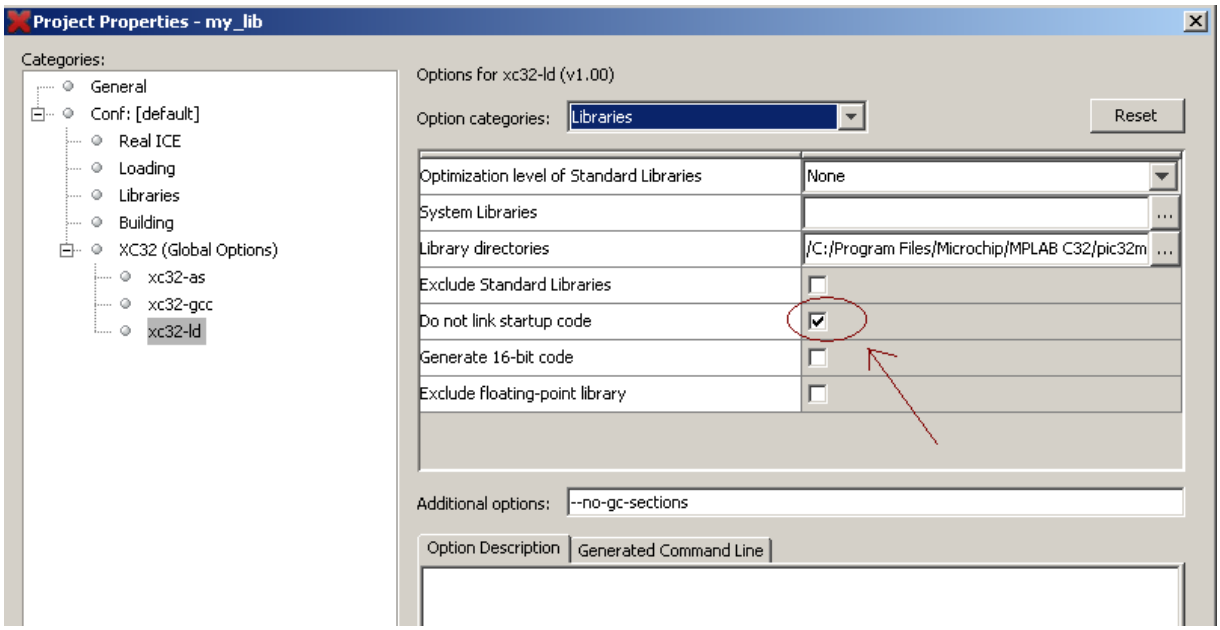
Disable Garbage Collection

The library does not contain *main()*. Hence, linker treats most of the functions inside the library as unreferenced code sections. The “Garbage Collection” feature of the linker eliminates them at the time of linking. In order to retain all the code sections, you need to disable “Garbage Collection” feature by setting the linker option “—no-gc-sections” in the project properties.



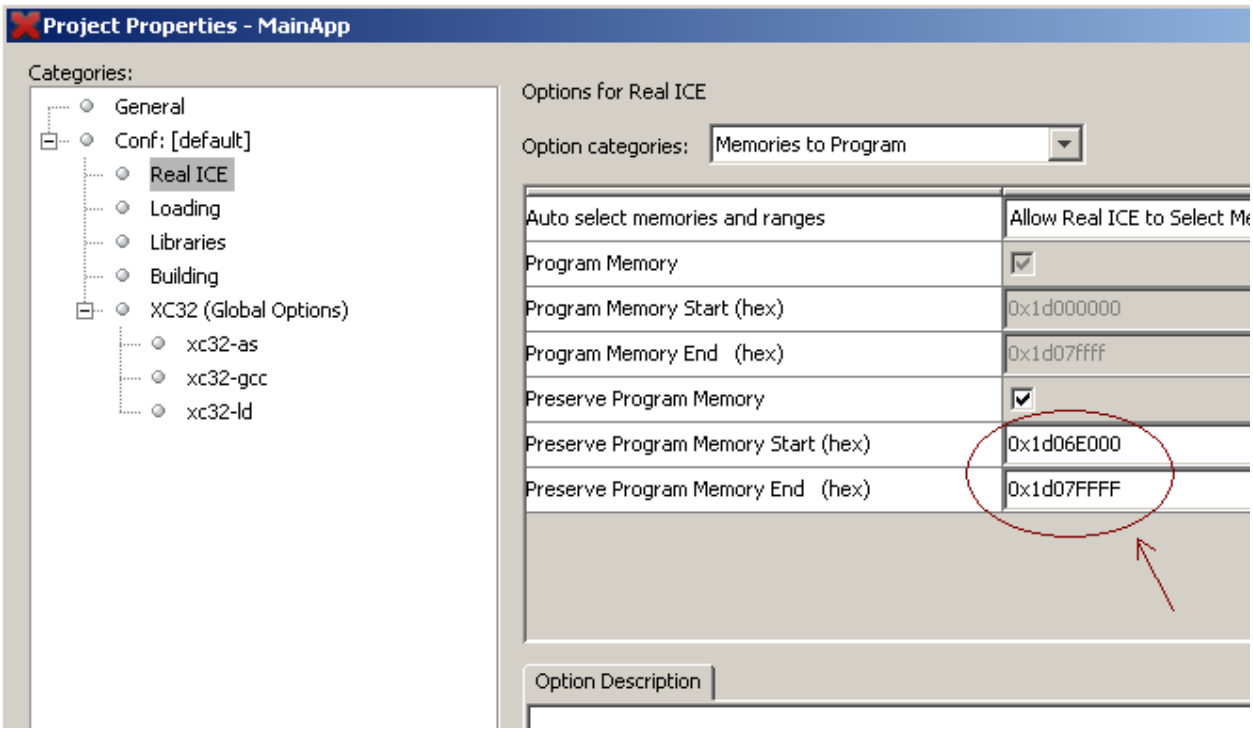
Delink startup code from library project

The library does not contain *main()*. Therefore you will have to delink the C-Startup code from library project. To delink the C-startup code, in the project properties window, under the categories select **XC32** -> **xc32-ld**. In the option categories select "Libraries" from the drop down list. Tick the checkbox, "*Do not link startup code*".



Appendix B: Application Project Settings for the RTLL technique

In the RTLL lab (LAB-3), “library” is programmed first followed by the “application”. While programming the application, you need to make sure that the application doesn’t erase/overwrite the memory region occupied by the library. You need to configure the Real-ICE to preserve the memory region reserved for the library. In the project settings for “MainApp”, under the *Categories*, select *Conf->Real ICE*. Select “*Memories to Program*” in the option categories. Tick the check box “*Preserve Program Memory*”. Set *preserve program memory start* and *preserve program memory end* addresses to the memory range occupied by the library. These must be physical memory addresses.



Appendix C: Placing the Library Module Header in the absolute address

The “Library Module Header” structure is initialized in the source file RTLL.c. You can use compiler provided address attribute to place the module header structure in the required address location. Make sure that this address is in the memory region reserved for the library.

```
// Standard header which must be exposed by the library.
const T_MODULE_DYN_HDR __attribute__((address(0x9D06E000))) _ModuleLoadHdr =
{

    _MODULE_NAME_,          // name of the Module
    _libInit,               // Start up code
    _nProcs,                // Number of procedures
    _exportProcTbl          // Export Procedure Table
};
```