MICROCHIP

TLS 2130

Getting Started with MPLAB® C for dsPIC® and PIC24

Author: *Rob Ostapiuk*
Microchip Technology Inc.

Agenda

- **C in an Embedded Environment**
- **Very Short Review of 16-bit Architecture**
- **MPLAB® C30 Compiler Toolset Overview**
  - Lab 1: *Creating C30 Projects in MPLAB*
- **How to Set Configuration Bits in Code**
  - Lab 2: *Setting Configuration Bits in Code*
- **How to Read and Write Registers**
- **How to Read and Write I/O Pins**
  - Lab 3: *"Hello, world" for Microcontrollers*

Agenda

- **C Runtime Environment**
- **Memory Models**
- **Attributes**
- **Interrupts**
  - Lab 4: *Writing Interrupt Service Routines*
- **Working with Libraries**
  - Lab 5: *Using Peripheral Libraries*
  - Lab 6: *Creating Custom Libraries*

Agenda

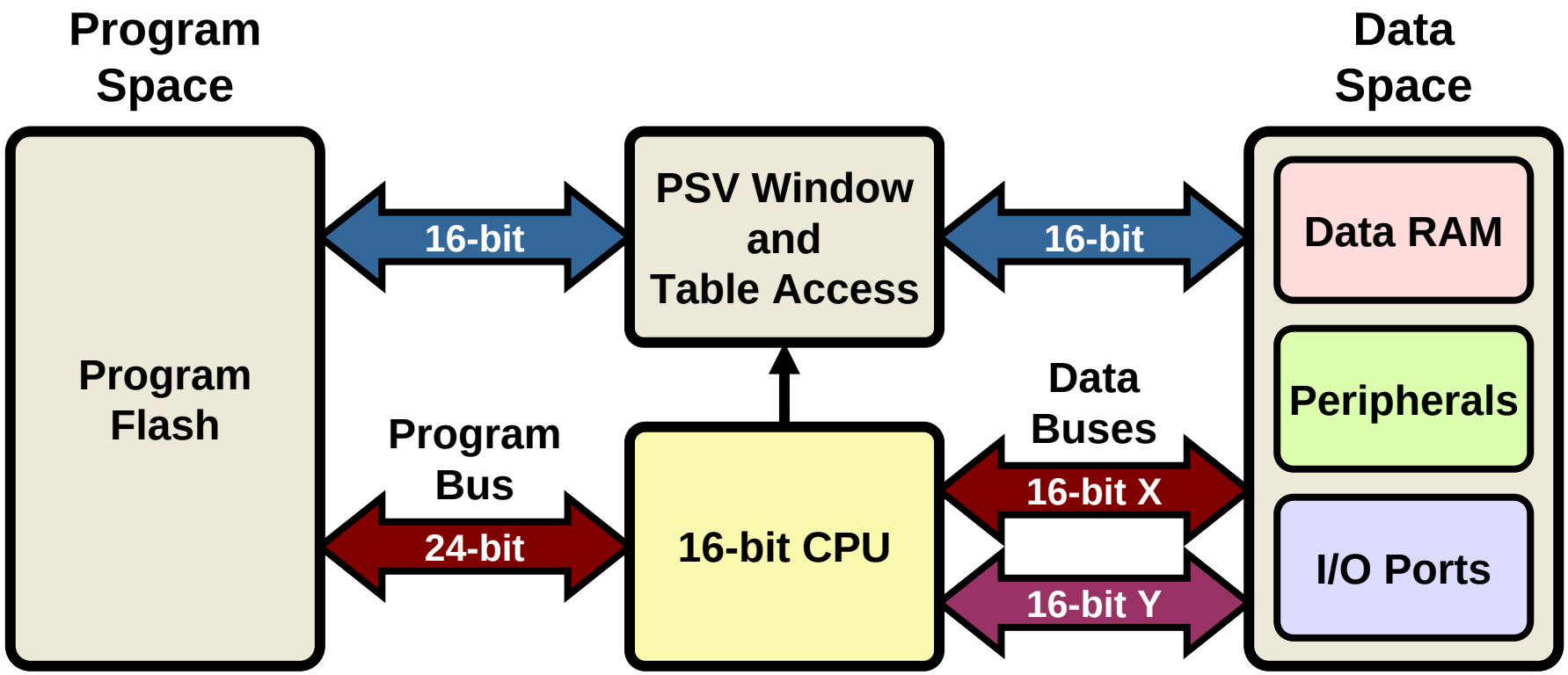
- **Mixing C and Assembly**
  - Lab 7: *Calling Assembly Functions from C*
- **Optimization Techniques**
  - **Compiler Optimization**
  - **Coding Tips for Generating Optimal Code**
  - **Built-in Functions**

16-bit Architecture

A Brief Overview

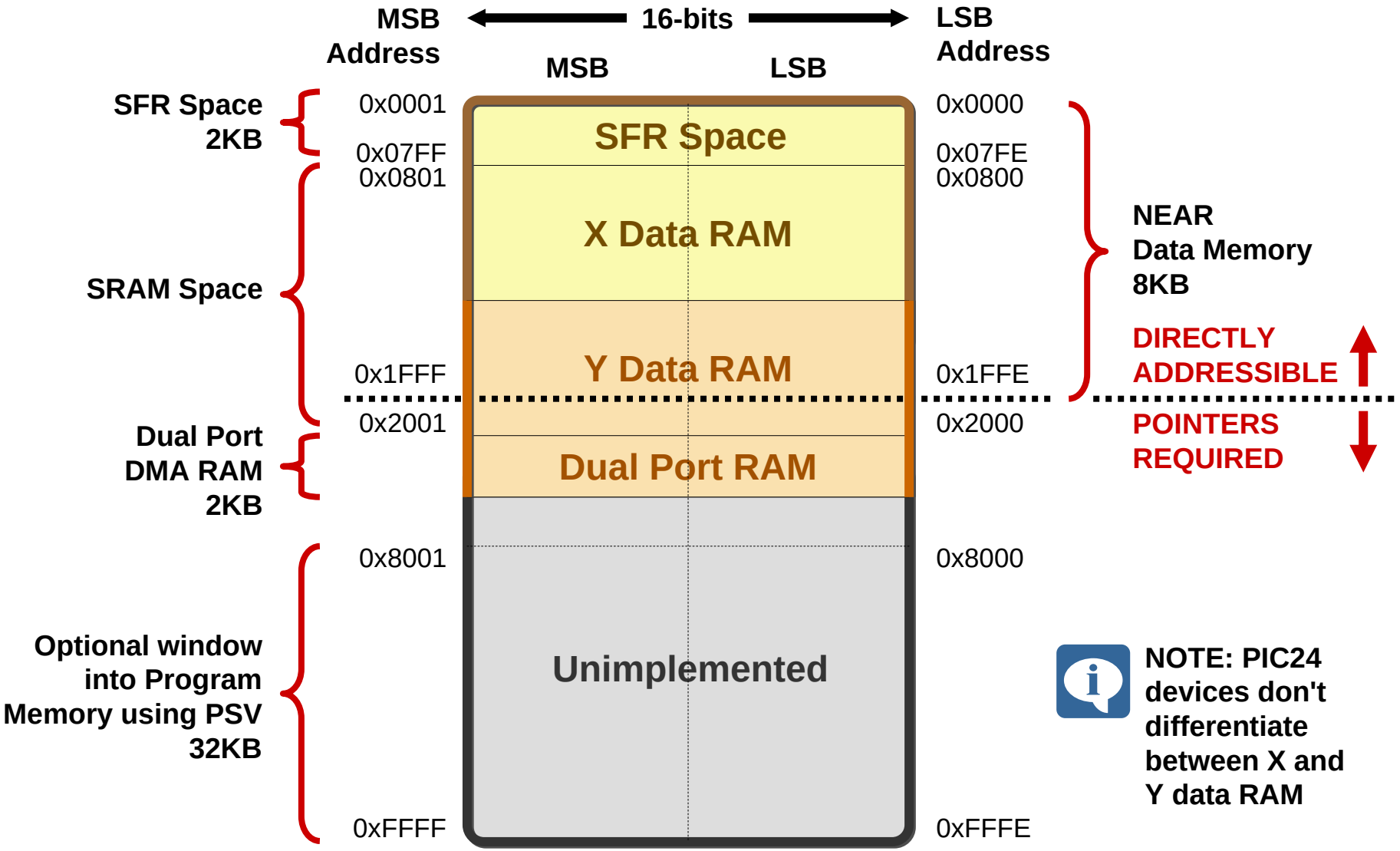
Simplified Block Diagram

16-bit PIC® Architecture

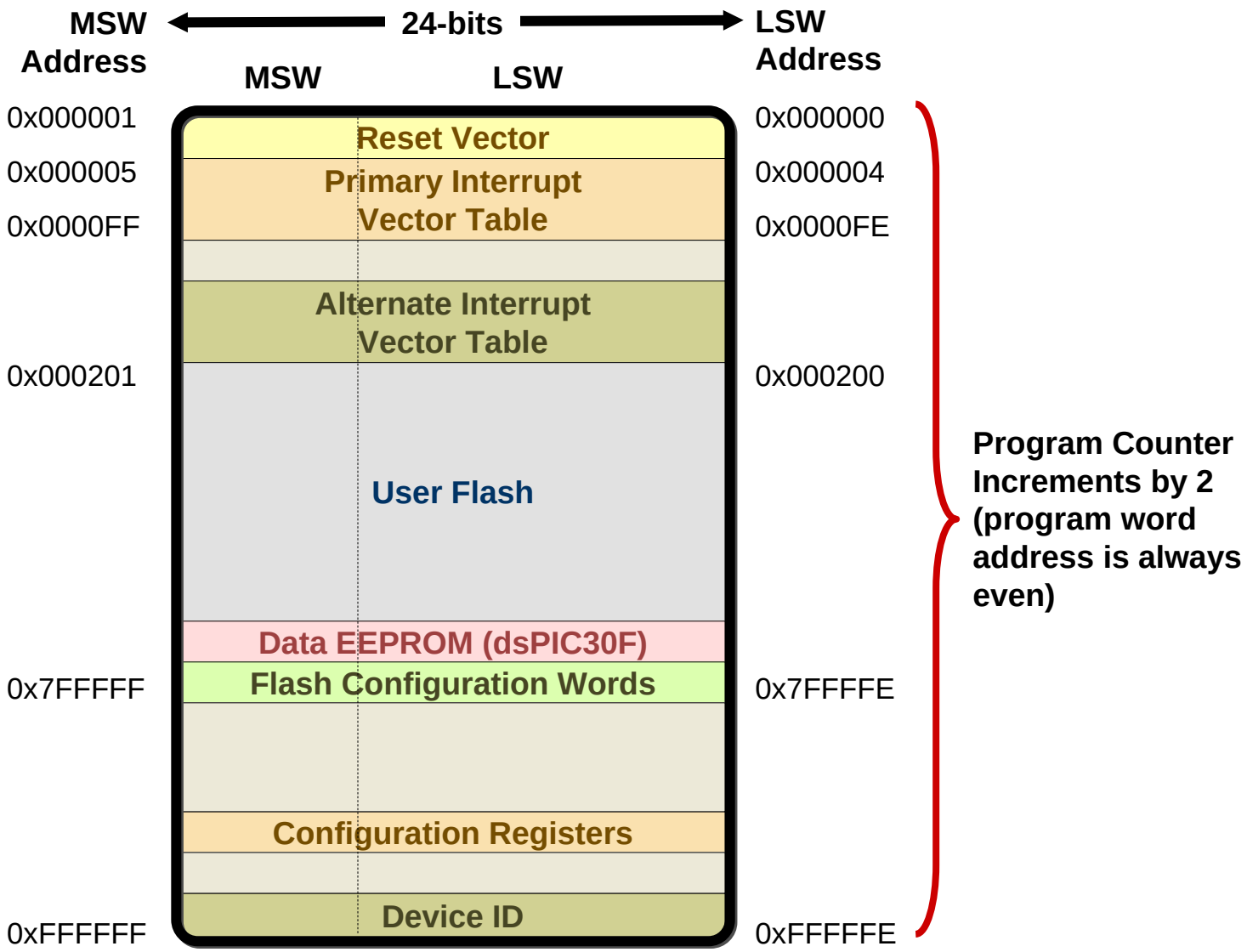


The Y data bus is only available on the dsPIC30 and dsPIC33 families.

Data Memory Map
16-bit PIC® Architecture

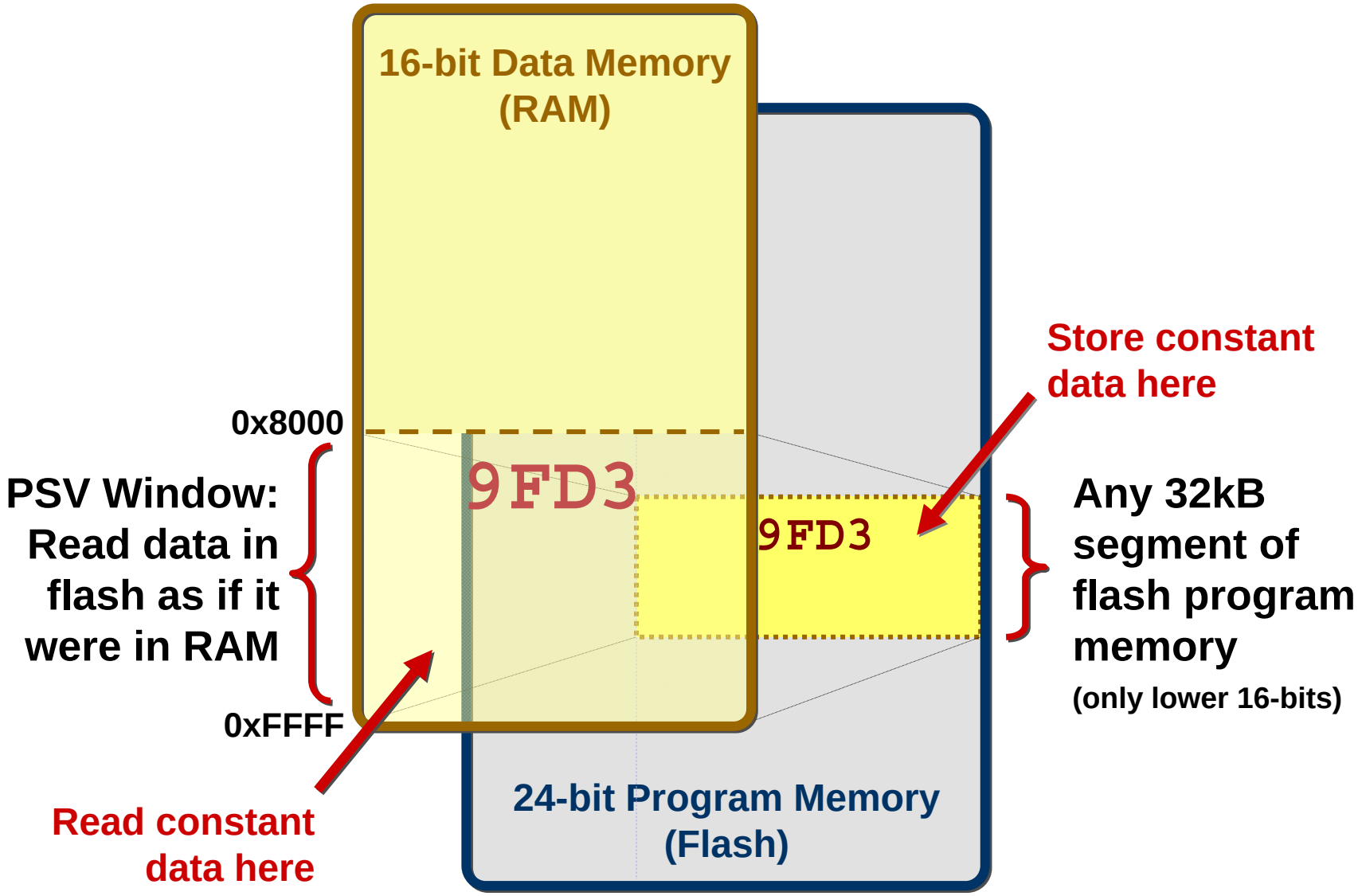


Program Memory Map
16-bit PIC® Architecture

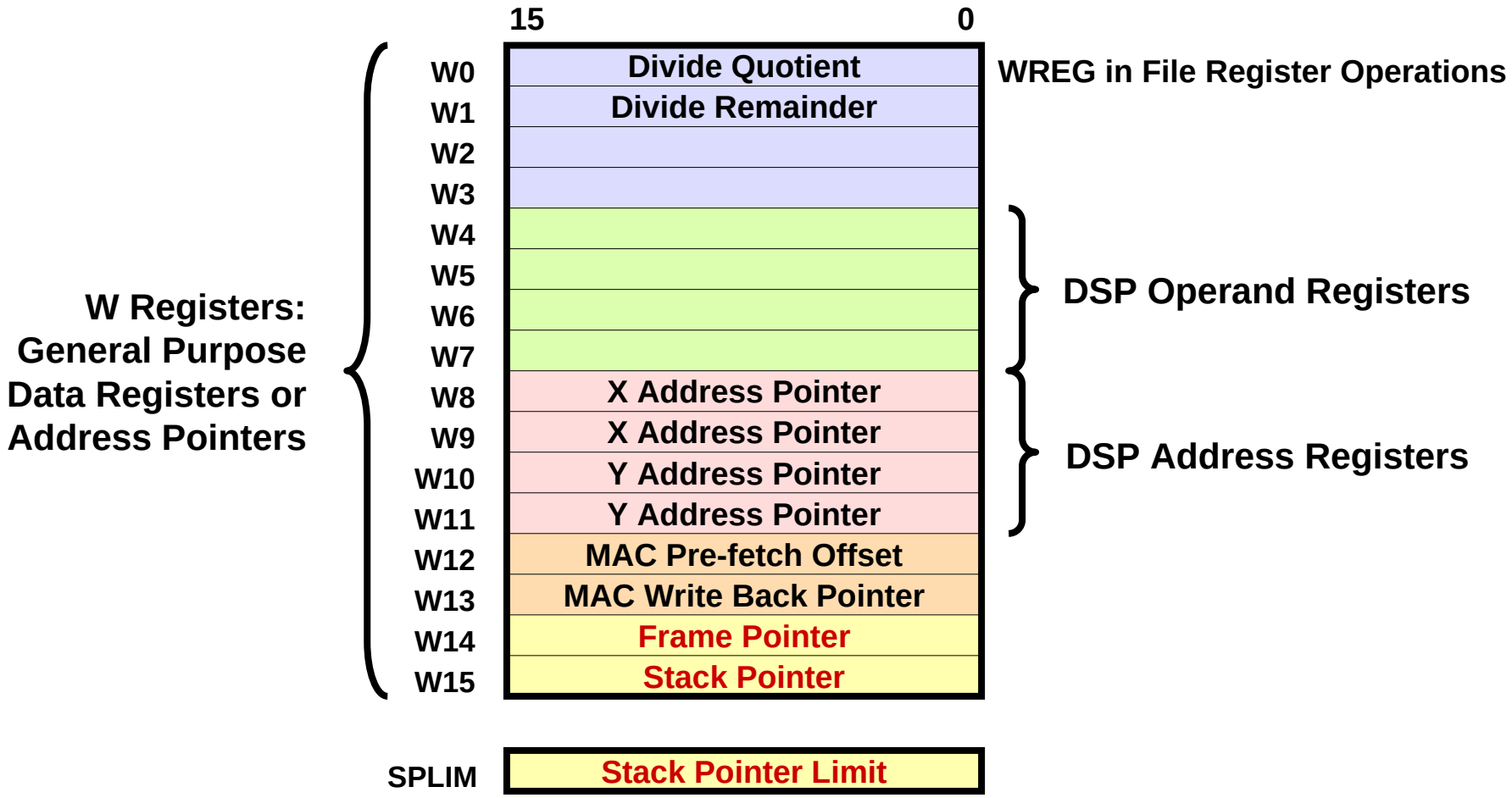


Program Space Visibility Window

16-bit PIC® Architecture



Programmer's Model
16-bit PIC® Architecture



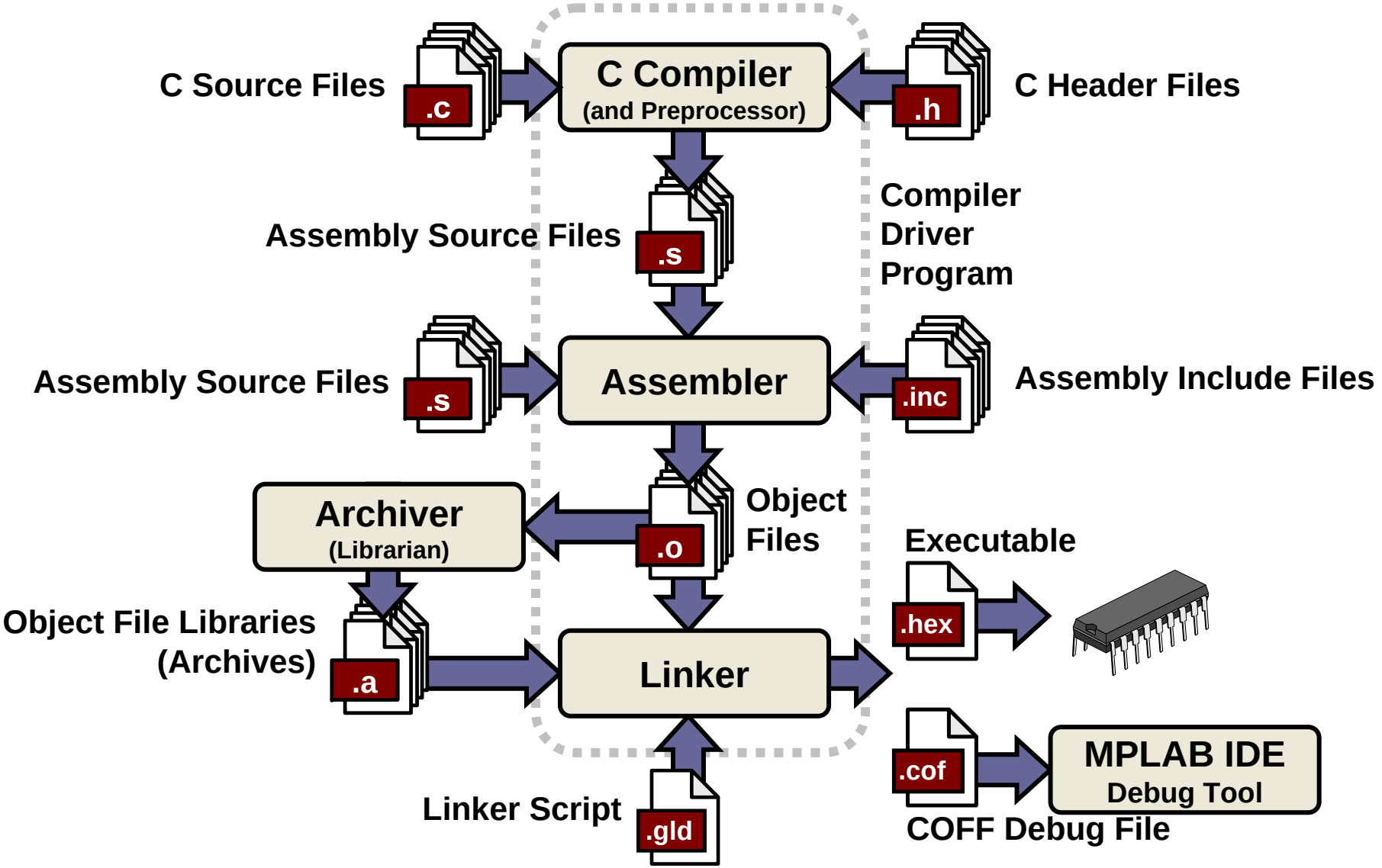
MPLAB® C for dsPIC® and PIC24

Toolset Overview

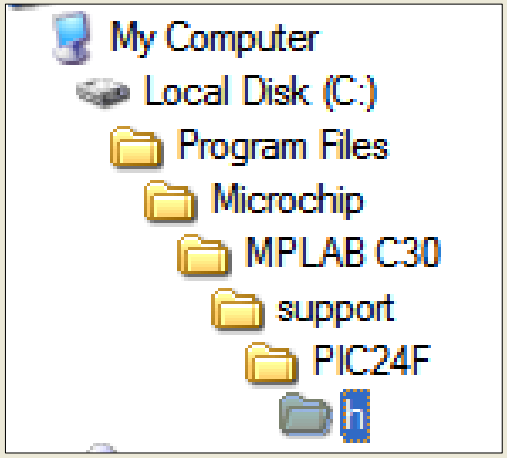
MPLAB® C for dsPIC® and PIC24
Overview

- ANSI x3.1989 compliant
- Optimizing compiler
- Includes language extensions for Microchip's 16-bit architectures
- Ported from GCC (GNU) compiler from the Free Software Foundation
- Works as a component of MPLAB® IDE

Development Tools Data Flow



MPLAB® C for dsPIC® and PIC24
Header Files



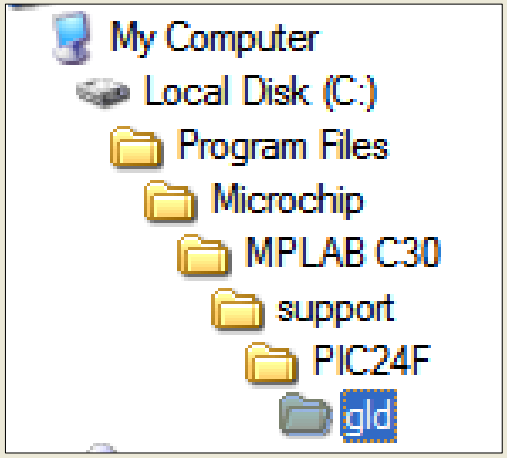
Header files are included as part of the MPLAB C30 installation and are located in the following directory:

C:\Program Files\Microchip\MPLAB C30\Support\PIC24F\h

p24FJ128GA010.h

- **One header file per device:**
  - **Provides access to registers as C variables**
  - **Defines labels for bit manipulation**
  - **Defines macros to utilize instructions not normally accessible from C**

MPLAB® C for dsPIC® and PIC24
Linker Scripts



Linker Script files are included as part of the MPLAB C30 installation and are located in the following directory:

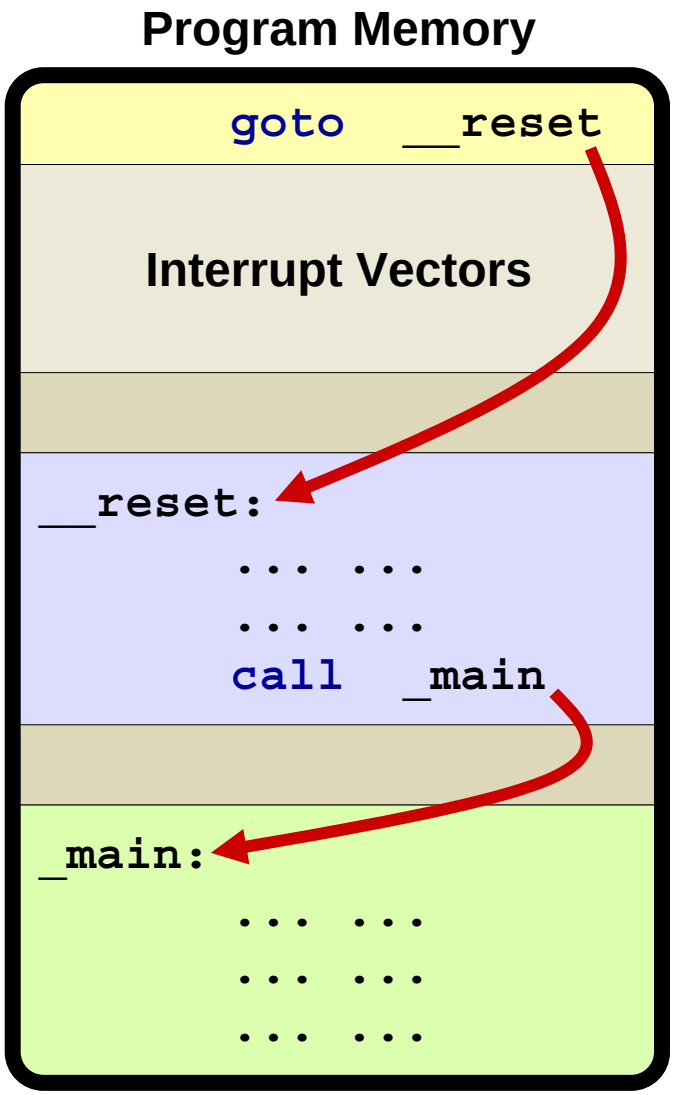
C:\Program Files\Microchip\MPLAB C30\Support\PIC24F\gld

p24FJ128GA010.gld

- **One linker script file per device:**
  - **Defines memory sections and boundaries**
  - **Associates ISR names with interrupt vectors**
  - **Equates register variables with addresses**

MPLAB® C for dsPIC® and PIC24

Startup and Initialization



Reset Vector (Address 0x000000)

Populated automatically by LINK30 Linker

Calls runtime environment setup code in crt0.o (`__reset` label)

.o **crt0.o** or **crt1.o** (from libpic30.a)

C Runtime Environment Setup Code

Inserted automatically by LINK30 Linker

(Source files: crt0.s and crt1.s)

.c **main.c**

Your C code's `main()` routine.

Included by you in your project and placed in memory by LINK30 Linker

MPLAB® C for dsPIC® and PIC24
Data Representation

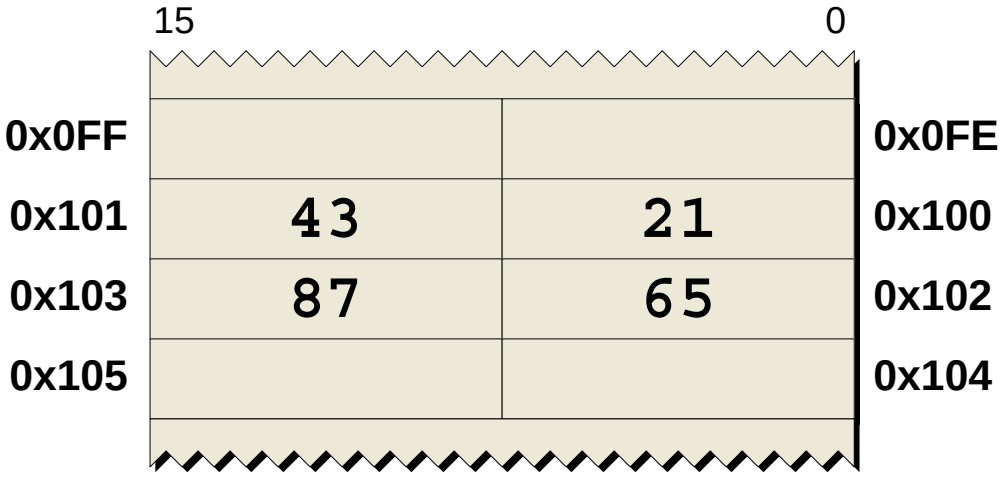
- **Multibyte quantities are stored in "little endian" format:**
  - **LSB is stored at lowest address**
  - **LSb is stored at lowest numbered bit position**

How the value `0x87654321` is stored

In Working Registers W4 & W5

W4	4321
W5	8765

In Data Memory (RAM) @ 0x100



MPLAB® C for dsPIC® and PIC24

Integer Data Types

Type	Bits	Min	Max
`char`, `signed char`	8	-128	127
`unsigned char`	8	0	255
`short`, `signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`, `signed int`	16	-32768	32767
`unsigned int`	16	0	65535
`long`, `signed long`	32	-2^{31}	$2^{31} - 1$
`unsigned long`	32	0	$2^{32} - 1$
`long long`, `signed long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long long`	64	0	$2^{64} - 1$

MPLAB® C for dsPIC® and PIC24

Floating Point Data Types

Type	Bits	E Min	E Max	N Min	N Max
`float`	32	-126	127	2^{-126}	2^{127}
`double *`	32	-126	127	2^{-126}	2^{127}
`long double`	64	-1022	1023	2^{-1022}	2^{1023}

E = Exponent
N = Normalized (approximate)
* `double` is equivalent to long double if `-fno-shortt-double` command line option is used

Type	Bits	|Min|	|Max|
`float`	32	1.175494×10^{-38}	$3.40282346 \times 10^{38}$
`double *`	32	1.175494×10^{-38}	$3.40282346 \times 10^{38}$
`long double`	64	$2.22507385 \times 10^{-308}$	$1.79769313 \times 10^{308}$

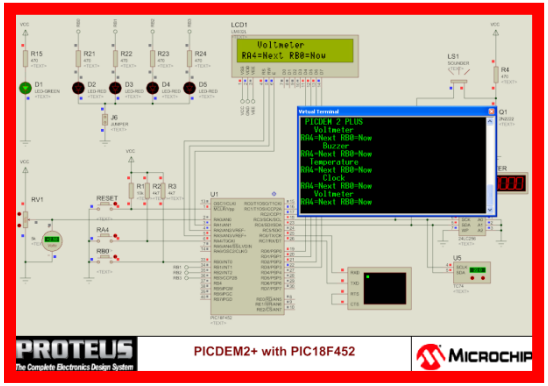
MPLAB® C for dsPIC® and PIC24
Pointers

- All pointers are 16-bits in size, regardless of whether they are data pointers or program pointers
- Constants in flash viewed via PSV
- Jump tables sometimes used for program pointers (handles)

Labcenter Proteus VSM

- Virtual Prototyping for all PIC MCU families
- Simulate complete embedded systems
- Develop and debug your firmware on virtual hardware
- Test and verify before ordering your prototype

www.labcenter.com



Schematic simulation of Microchip evaluation board

Lab Exercise 1

Creating an MPLAB® C Based Project
with MPLAB IDE Step-by-step

Lab Exercise 1

Creating an MPLAB® C Based Project

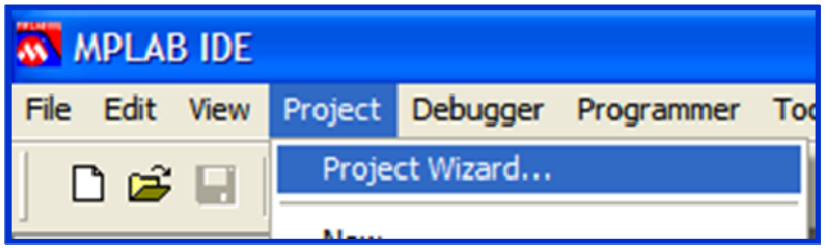
Purpose

The purpose of this lab is to illustrate the steps required to create an MPLAB® C30 based project within the MPLAB Integrated Development Environment. You will learn how to select the compiler as the build tool, which files must be included in your project, how to allocate a heap and what code must be included in your source file.

Procedure

1. Open MPLAB and start the project wizard by selecting from the menu:

   Project ▸ Project Wizard…

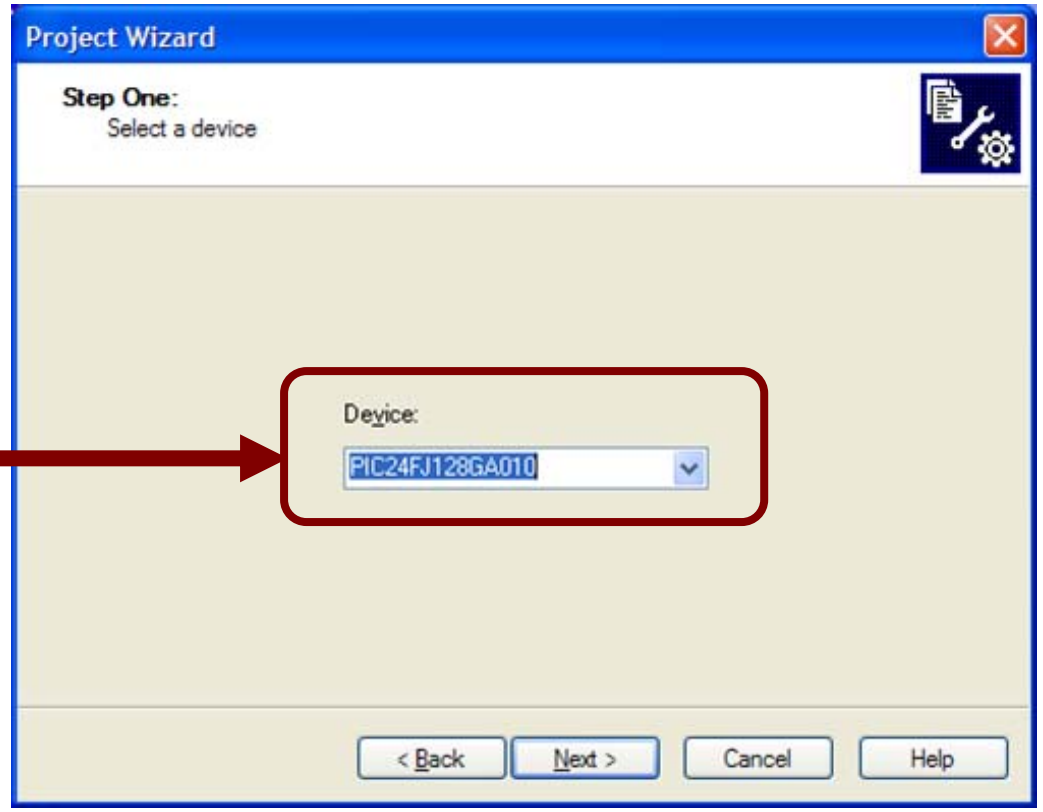


After the Project Wizard opens,

Click Next > to continue…

Lab Exercise 1
Creating an MPLAB® C Based Project

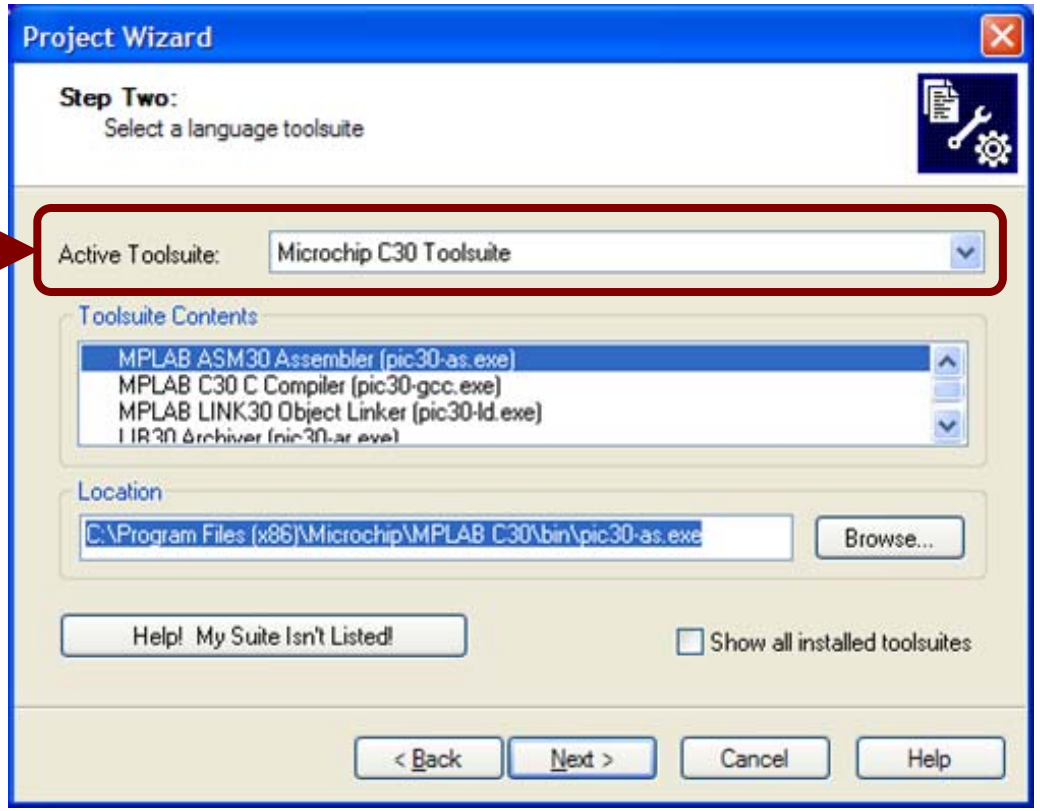
2 In the **Device** combo box, select:
PIC24FJ128GA010



Click Next > to continue…

Lab Exercise 1
Creating an MPLAB® C Based Project

3 In the **Active Toolsuite** combo box, select:
Microchip C30 Toolsuite



Click **Next >** to continue…

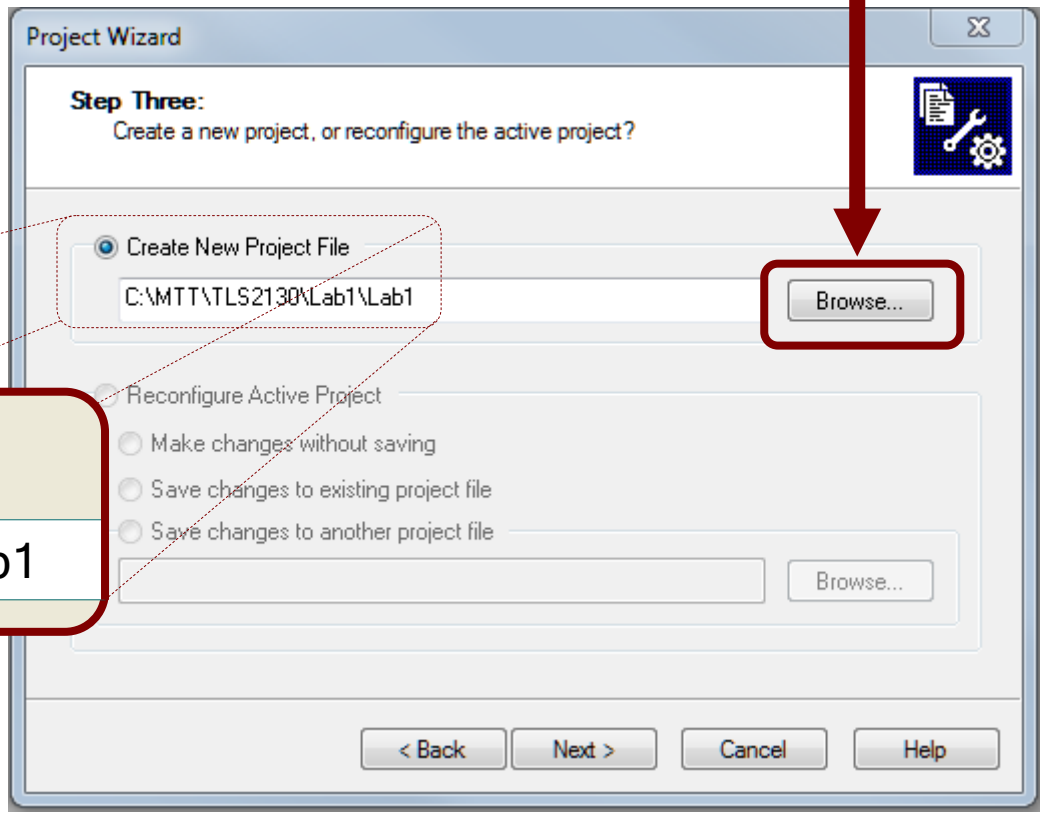
Lab Exercise 1

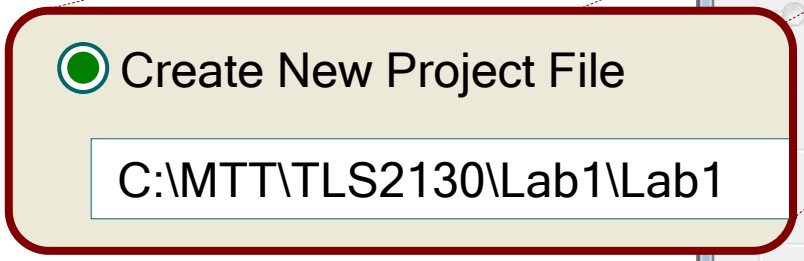
Creating an MPLAB® C Based Project

4 Click **Browse…** and navigate to:

C:\MTT\TLS2130\Lab1

Then name the file **Lab1.mcp**





Click **Next >** to continue…

Lab Exercise 1

Creating an MPLAB® C Based Project

5 Add source files to the project. In the left hand list box, navigate to:

C:\MTT\TLS2130\Lab1

Select the file **Lab1.c**

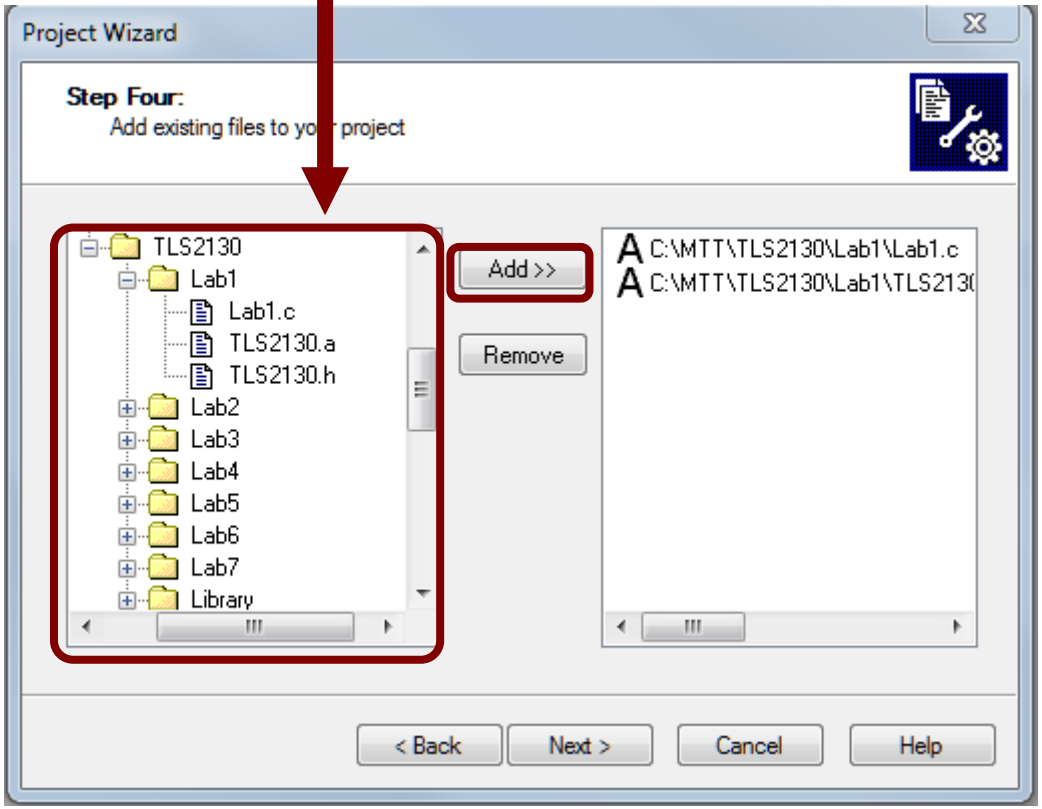
Click **Add >>**

6 Add library file to the project:

Select the file **TLS2130.a**

Click **Add >>**

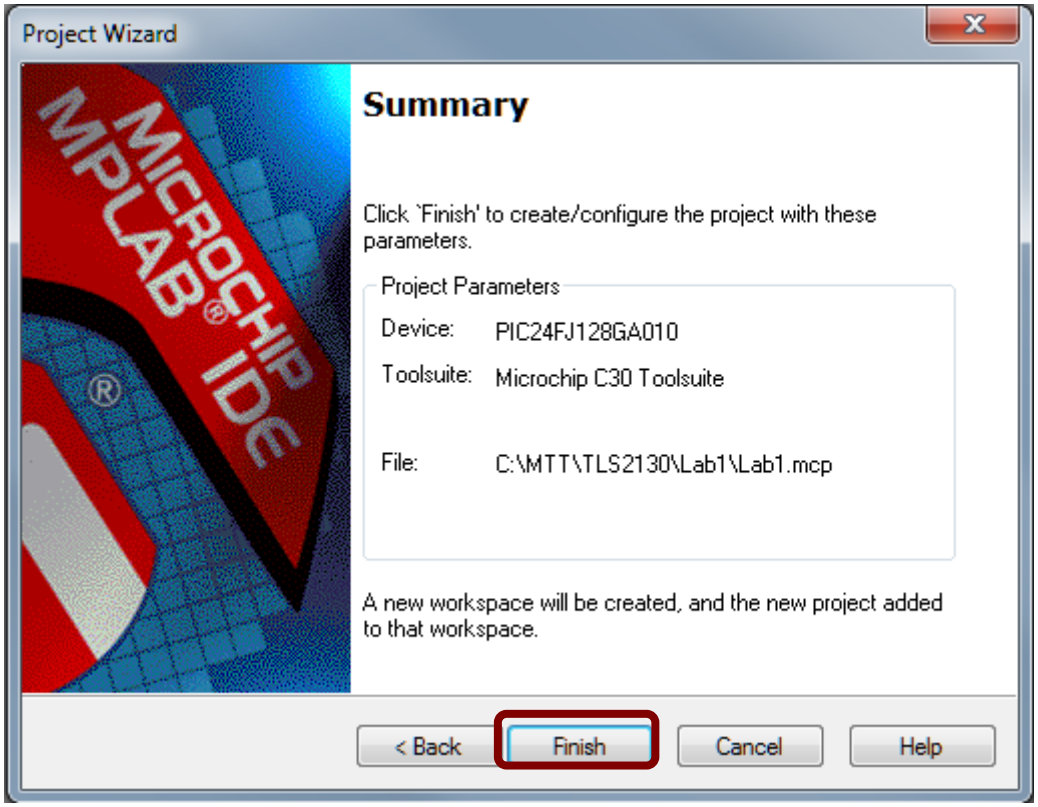
Click **Next >** to continue…



Lab Exercise 1

Creating an MPLAB® C Based Project

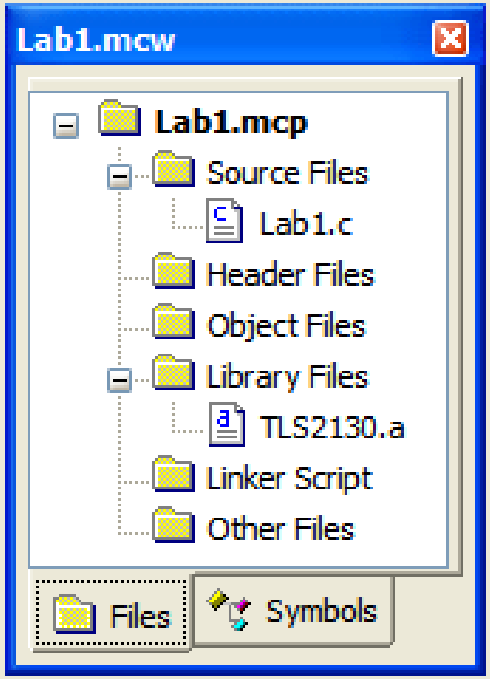
7 Click **Finish**



If the project tree isn't visible, select from the menu:

View ▸ Project

Lab 1 Project Tree

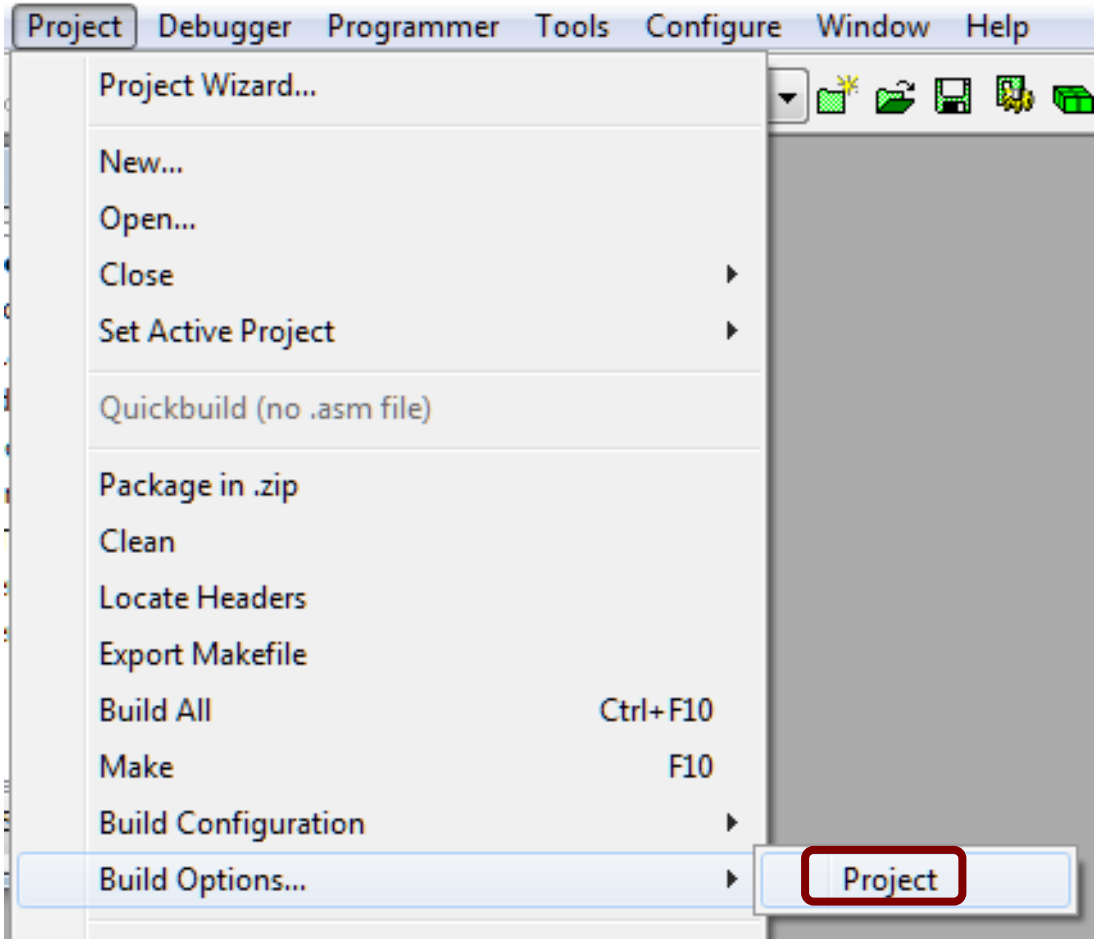


Lab Exercise 1

Creating an MPLAB® C Based Project

8 Open the project build options by selecting from the menu:

Project ▸ Build Options... ▸ Project

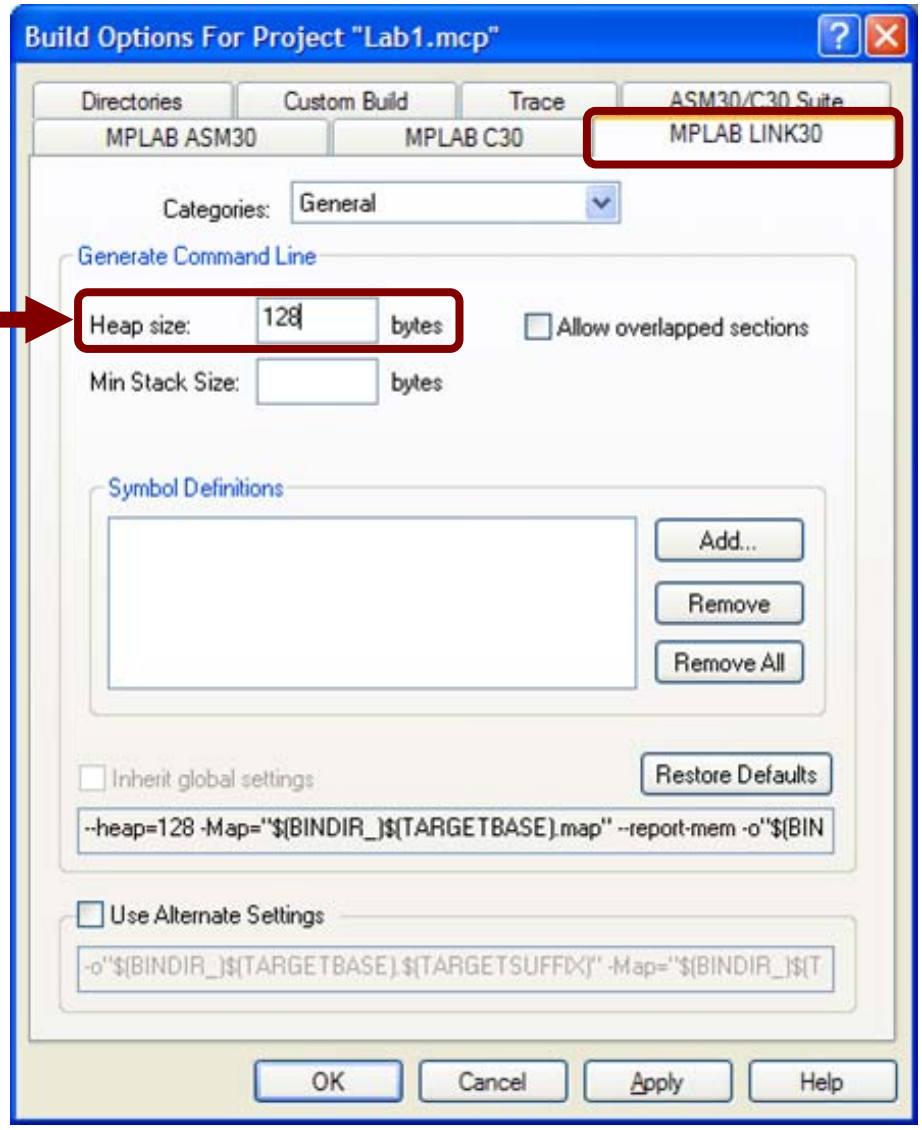


Lab Exercise 1

Creating an MPLAB® C Based Project

9 Select the **MPLAB LINK30** tab at the top of the window.

In the **Heap size** box, enter a value of **128** bytes.



The value of 128 bytes is somewhat arbitrary. Your application may require a larger or smaller heap, or possibly no heap at all.

Click **OK**

Lab Exercise 1
Creating an MPLAB® C Based Project

10 Add the minimum required framework to your soruce code in Lab1.c.
In the project tree, double click on the file Lab1.c

`#include` the header files required by your application.

- **Device specific header file (required to access device features)**
- **Standard C and Microchip library header files (if used)**
- **Header files for your own libraries (if used)**

Lab1.c

```
15  #include <p24fj128ga010.h>
16  #include <stdio.h>
17  #include "TLS2130.h"
```

Lab Exercise 1
Creating an MPLAB® C Based Project

11 Add the minimum required framework to your soruce code in Lab1.c.

Setup device configuration bits in code

- Use `_CONFIG1` and `_CONFIG2` macros (defined in header file)
- Bitwise AND list of option constants (defined in header file)
- Unspecified options will use their default settings

Lab1.c

```
19  _CONFIG1(FWDTEN_OFF & JTAGEN_OFF)
```

This is the minimum required configuration for the Explorer 16 Demo Board.

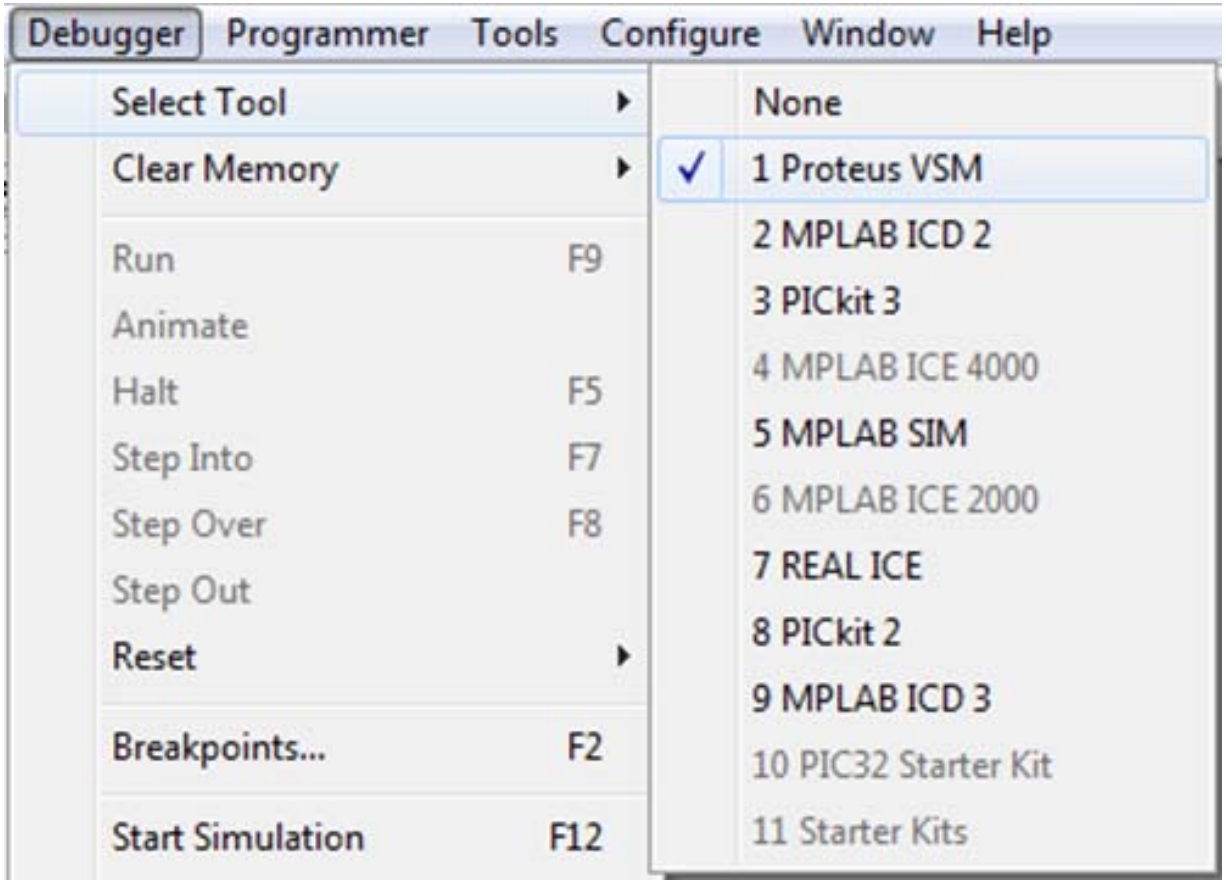
Lab Exercise 1

Creating an MPLAB® C Based Project

12

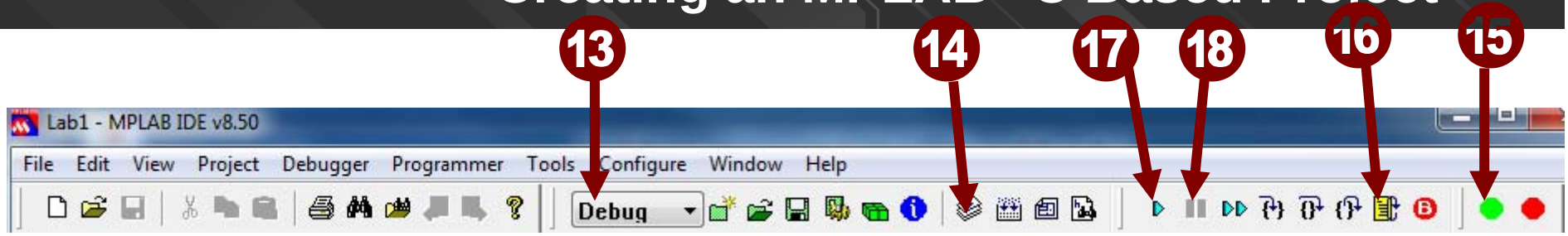
Enable MPLAB® REAL ICE by selecting from the menu:

Debugger ▸ Select Tool ▸ Proteus VSM



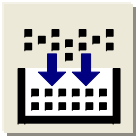
Lab Exercise 1

Creating an MPLAB® C Based Project

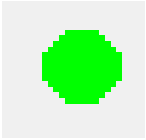


13 Select Debug mode.

14 Click Build All button.

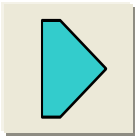


15 If no errors reported, Click Start Simulation button.

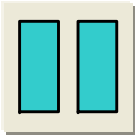


16 When programming completes,click Reset button.

17 Click Run button.



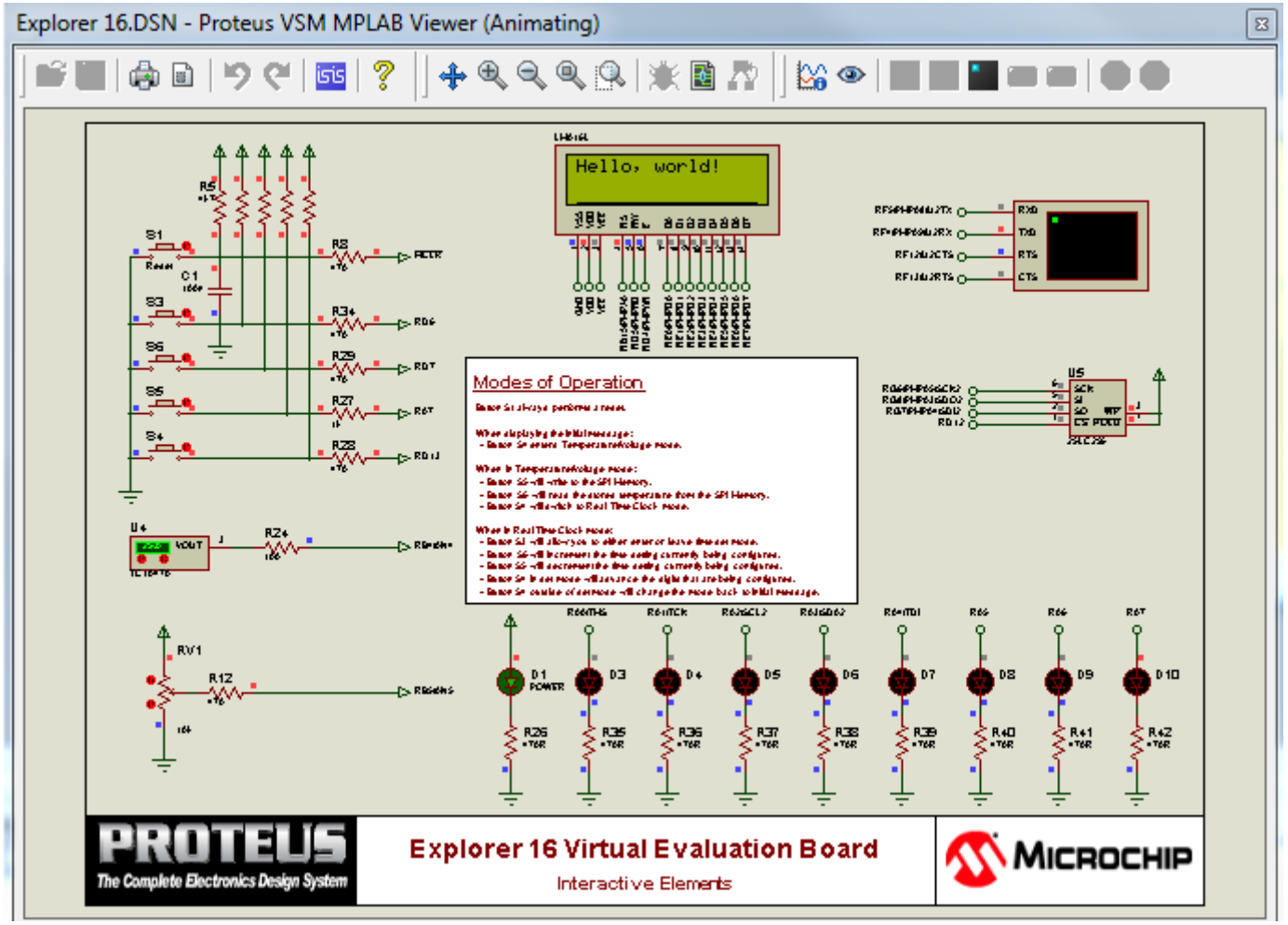
18 Click Halt button.



Lab Exercise 1

Creating an MPLAB® C Based Project

Results



Lab Exercise 1
Creating an MPLAB® C Based Project

Conclusions

- **Minimum steps to setup a C based project:**
  - **Follow standard MPLAB® project setup steps**
  - **Select C30 as build tool**
  - **Add linker script for your device to project**
  - **#include device header file in source code**
  - **#include library header files (if required)**
  - **Add configuration bits setup in code**
  - **Allocate a heap (if required)**

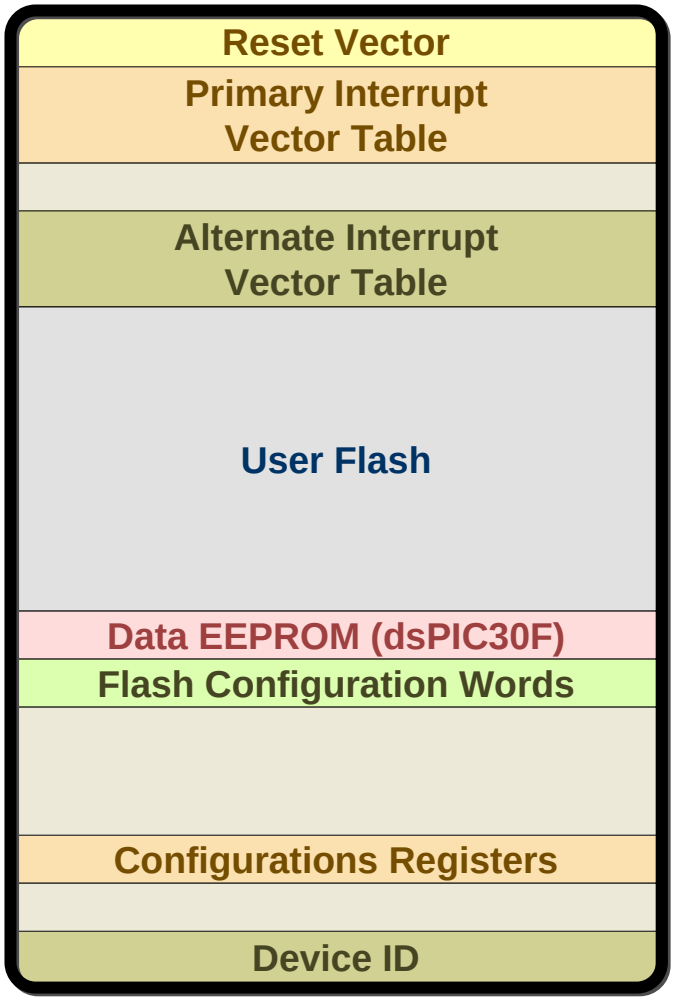
How to Set PIC® Configuration Bits

Using the `_CONFIG1()` and `_CONFIG2()` Macros

What are Configuration Bits?

- Used to setup device features:
  - Code Protect
  - Watchdog Timer
  - JTAG
  - Oscillator Options
  - Debug Options
  - More…

CONFIG1 and CONFIG2 Located in program memory space, outside range of executable code space



Flash Configuration Word 1
CONFIG1

CONFIG1 (PIC24FJ128GA010)

Upper Third:

U-1	U-1	U-1	U-1	U-1	U-1	U-1	U-1
-	-	-	-	-	-	-	-
bit 23							bit 16

Middle Third:

r-0	R/PO-1	R/PO-1	R/PO-1	R/PO-1	R/PO-1	U-1	R/PO-1
r	JTAGEN	GCP	GWRP	$\overline{\text{DEBUG}}$	COE	-	ICS
bit 15							bit 8

Lower Third:

R/PO-1	R/PO-1	U-1	R/PO-1	R/PO-1	R/PO-1	R/PO-1	R/PO-1
FWDTEN	WINDIS	-	FWPSA	WDTPS3	WDTPS2	WDTPS1	WDTPS0
bit 7							bit 0

Flash Configuration Word 1
_CONFIG1(x) Macro

_CONFIG1(x) Macro Definition

```
#define _CONFIG1(x) ➲
        __attribute__((section("__CONFIG1.sec,code"))) ➲
        int _CONFIG1 = (x);
```

- Defined in device specific header files
- x is formed by anding together constants representing configuration bit values
- Omitted bits retain their default value

Example

```
_CONFIG1(FWDTEN_OFF & JTAGEN_OFF)
```

_CONFIG1(x)
Parameter List Options

_CONFIG1(x) Macro Parameter List Options for PIC24FJ128GA010 (Part 1)

JTAG	`JTAGEN_OFF`	Disabled
	`JTAGEN_ON`	Enabled
Code Protect	`GCP_ON`	Enabled
	`GCP_OFF`	Disabled
Write Protect	`GWRP_ON`	Enabled
	`GWRP_OFF`	Disabled
Background Debug	`BKBUG_ON`	Enabled
	`BKBUG_OFF`	Disabled
Clip-On Emulation	`COE_ON`	Enabled
	`COE_OFF`	Disabled
ICD Pins Select	`ICS_PGx1`	EMUC/EMUD share PGC1/PGD1
	`ICS_PGx2`	EMUC/EMUD share PGC2/PGD2
Watchdog Timer	`FWDTEN_OFF`	Disabled
	`FWDTEN_ON`	Enabled
Windowed WDT	`WINDIS_ON`	Enabled
	`WINDIS_OFF`	Disabled

_CONFIG1(x)
Parameter List Options Continued…

_CONFIG1(x) Macro Parameter List Options for PIC24FJ128GA010 (Part 2)

Watchdog Prescaler	`FWPSA_PR32`	1:32
	`FWPSA_PR128`	1:128
Watchdog Postscaler	`WDTPS_PS1`	1:1
	`WDTPS_PS2`	1:2
	`WDTPS_PS4`	1:4
	`WDTPS_PS8`	1:8
	`WDTPS_PS16`	1:16
	`WDTPS_PS32`	1:32
	`WDTPS_PS64`	1:64
	`WDTPS_PS128`	1:128
	`WDTPS_PS256`	1:256
	`WDTPS_PS512`	1:512
	`WDTPS_PS1024`	1:1024
	`WDTPS_PS2048`	1:2048
	`WDTPS_PS4096`	1:4096
	`WDTPS_PS8192`	1:8192
	`WDTPS_PS16384`	1:16384
	`WDTPS_PS32768`	1:32768

Flash Configuration Word 2
CONFIG2

CONFIG2 (PIC24FJ128GA010)

Upper Third:

U-1	U-1	U-1	U-1	U-1	U-1	U-1	U-1
-	-	-	-	-	-	-	-
bit 23							bit 16

Middle Third:

R/PO-1	U-1	U-1	U-1	U-1	R/PO-1	R/PO-1	R/PO-1
IESO	-	-	-	-	**FNOSC2**	**FNOSC1**	**FNOSC0**
bit 15							bit 8

Lower Third:

R/PO-1	R/PO-1	R/PO-1	U-1	U-1	U-1	R/PO-1	R/PO-1
FCKSM1	**FCKSM0**	**OSCIOFCN**	-	-	-	**POSCMD1**	**POSCMD0**
bit 7							bit 0

Flash Configuration Word 2
_CONFIG2(x) Macro

_CONFIG2(x) Macro Definition

```
#define _CONFIG2(x) 
        __attribute__((section("__CONFIG2.sec,code"))) 
        int _CONFIG2 = (x);
```

- Defined in device specific header files
- x is formed by anding together constants representing configuration bit values
- Omitted bits retain their default value

Example

```
_CONFIG2(FNOSC_PRIPLL & POSCMOD_HS & OSCIOFNC_OFF)
```

_CONFIG2(x)
Parameter List Options

_CONFIG2(x) Macro Parameter List Options for PIC24FJ128GA010

Two Speed Startup	`IESO_OFF`	Disabled
	`IESO_ON`	Enabled
Oscillator Selection	`FNOSC_FRC`	Fast RC Oscillator
	`FNOSC_FRCPLL`	Fast RC Oscillator with divide and PLL
	`FNOSC_PRI`	Primary Oscillator (XT, HS, EC)
	`FNOSC_PRIPLL`	Primary Oscillator (XT, HS, EC) with PLL
	`FNOSC_SOSC`	Secondary Oscillator
	`FNOSC_LPRC`	Low Power RC Oscillator
	`FNOSC_FRCDIV`	Fast RC Oscillator with divide
Clock Switching & Monitor	`FCKSM_CSECME`	Both Enabled
	`FCKSM_CSECMD`	Only Clock Switching Enabled
	`FCKSM_CSDCMD`	Both Disabled
OSC2/RC15 Function	`OSCIOFNC_ON`	RC15
	`OSCIOFNC_OFF`	OSCO or $F_{OSC}/2$
Oscillator Type	`POSCMOD_EC`	External Clock
	`POSCMOD_XT`	XT Oscillator
	`POSCMOD_HS`	HS Oscillator
	`POSCMOD_NONE`	Primary Disabled

How to Set PIC® Configuration Bits

- **`_CONFIGn(x)` macros must be used anywhere *after* the `#include` for the device specific header file**

Example

```
/*************************************************************
* PROGRAM: main.c
* AUTHOR:  Clem Finch
*************************************************************/
#include <p24fj128ga010.h>
#include "TLS2130.h"

_CONFIG1(FWDTEN_OFF & JTAGEN_OFF);
_CONFIG2(FNOSC_PRIPLL & POSCMOD_HS & OSCIOFNC_OFF);
```

Where is all this stuff defined?

- **Parameter labels for _CONFIGn(x) macros:**
  - **At the end of the device specific header files (e.g. p24fj128ga010.h)**
- **CONFIGn Register Descriptions:**
  - **In the data sheet under "Special Features" in the "Configuration Bits" subsection**
- **CONFIG Bits Functional Descriptions:**
  - **In the data sheet under the relevant section (e.g. WDT under "Special Features" section in "Watchdog Timer" subsection)**

Lab Exercise 2

Setting PIC® Configuration Bits in Code

Lab Exercise 2

Setting PIC® Configuration Bits in Code

Objective

Use the `_CONFIGn()` macros to setup the configuration registers and view the results in the Configuration Bits window inside the MPLAB® IDE

- **Setup the configuration registers with the following options:**

  - **Oscillator = *HS*, *Primary with PLL***
  - **OSC2/RC15 Function = *OSC2***
  - **Clock Switching & Monitor = *Both Disabled***
  - **JTAG = *Disabled***
  - **Watchdog Timer = *Disabled***

Lab Exercise 2

Setting PIC® Configuration Bits in Code

Procedure

Follow the directions in the lab manual starting on page 2-1.

On the lab PC…

If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

Open the lab workspace by selecting from the menu:
File ▶ Open Workspace…
and select the file:
C:\MTT\TLS2130\Lab2\Lab2.mcw

Lab Exercise 2

Setting PIC® Configuration Bits in Code

Conclusions

- **Configuration Bits:**
  - **Should be set in code using the `_CONFIG1(x)` and `_CONFIG2(x)` macros**
  - **Parameters consist of a series of option labels bitwise ANDed together**
  - **Option labels are defined in the device specific header file**
  - **Unspecified options retain their default settings (as per the data sheet)**

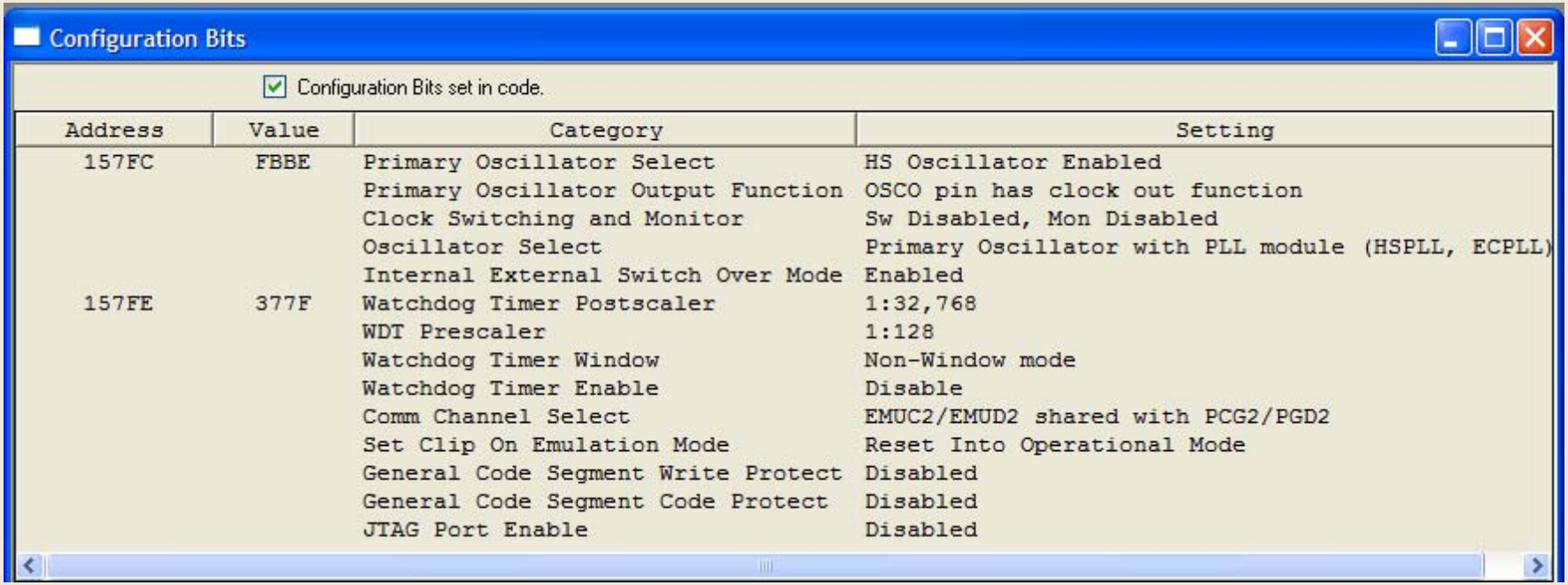
Lab Exercise 2

Setting PIC® Configuration Bits in Code

Results

From the menu, open the configuration bits window:

Configure ▸ Configuration Bits…



Address	Value	Category	Setting
157FC	FBBE	Primary Oscillator Select	HS Oscillator Enabled
		Primary Oscillator Output Function	OSCO pin has clock out function
		Clock Switching and Monitor	Sw Disabled, Mon Disabled
		Oscillator Select	Primary Oscillator with PLL module (HSPLL, ECPLL)
		Internal External Switch Over Mode	Enabled
157FE	377F	Watchdog Timer Postscaler	1:32,768
		WDT Prescaler	1:128
		Watchdog Timer Window	Non-Window mode
		Watchdog Timer Enable	Disable
		Comm Channel Select	EMUC2/EMUD2 shared with PCG2/PGD2
		Set Clip On Emulation Mode	Reset Into Operational Mode
		General Code Segment Write Protect	Disabled
		General Code Segment Code Protect	Disabled
		JTAG Port Enable	Disabled

How to Read and Write Registers

Word and Bit Access

How to Read and Write Registers
Full 16-bit Word

Syntax

REGNAME

- **Use the name of the register as you would an ordinary `int` type variable**
- **Variables defined in device specific header file with register name as shown in datasheet**

Examples

```
PORTA = 0xFE31;                 // Write 0xFE31 to PORTA
AtoD_Result = ADC1BUF0;         // Read A/D Result
TX1REG = 'a';                   // Send 'a' out UART
if (RX1REG == 'x') { … }        // If received char is 'x'
while (RX1REG) { … }            // While char is not '\0'
```

Register Variable Declaration
Example for PIC24FJ128GA010

PORTA Variable Declaration from p24fj128ga010.h Header File

```
extern volatile unsigned int PORTA __attribute__((__sfr__));
```

- `extern`: PORTA is actually defined in linker script…
- `volatile`: This variable may be altered by something other than the code (i.e. the hardware or an interrupt routine)
- `__attribute__((__sfr__))`: Tag to indicate that this is a special function hardware register (more about attributes later)

How to Read and Write Registers
Individual Bits and Bit Fields

Syntax

```
REGNAMEbits.BITNAME
```

- Use the name of the register with the word 'bits' in lower case attached to it
- Use the name of the bit or bit field as specified in the data sheet

Examples

```
LATAbits.LATA5 = 1;                  // Set bit 5 of PORTA
Flag = PORTAbits.RA5;                // Read bit 5 of PORTA
while (!AD1CONbits.DONE) { … } // While A/D converting
AD1CONbits.SSRC = 5;                 // Set 3-bit field = 5
//3-bits in bitfield: SSRC2 = 1, SSRC1 = 0, SSRC0 = 1
```

Bit Field Variable Declaration

Example for PIC24FJ128GA010

AD1CONbits Variable Declaration from p24fj128ga010.h Header File

```
__extension__ typedef struct tagAD1CON1BITS {
  union {
    struct {
      unsigned DONE:1;
      unsigned SAMP:1;
      unsigned ASAM:1;
      unsigned :2;
      unsigned SSRC:3;
      unsigned FORM0:1;
      unsigned FORM1:1;
      unsigned :3;
      unsigned ADSIDL:1;
      unsigned :1;
      unsigned ADON:1;
    };
    struct {
      unsigned :5;
      unsigned SSRC0:1;
      unsigned SSRC1:1;
      unsigned SSRC2:1;
      unsigned FORM:2;
    };
  };
} AD1CON1BITS;
extern volatile AD1CON1BITS AD1CON1bits __attribute__((__sfr__));
```

Bit Field Structure Definition

3-bit wide field

Primary Bit Names

Secondary Bit Names

Bit Field Variable Declaration

Register and Bit Field Variables
Definition in Linker Script

- **Both *REGNAME* and *REGNAME*bits defined in linker script**
- **Both are allocated at the same address**

Excerpt from P24FJ128GA010.gld Linker Script File

```
 TRISA          = 0x2C0;
_TRISA          = 0x2C0;
_TRISAbits      = 0x2C0;
 PORTA          = 0x2C2;
_PORTA          = 0x2C2;
_PORTAbits      = 0x2C2;
 LATA           = 0x2C4;
_LATA           = 0x2C4;
_LATAbits       = 0x2C4;
```

Assembly Label

C Labels

Eliminates need for a `union` in header file

How to Read and Write Registers
Individual Bits and Bit Fields – Shorthand Syntax

Shorthand Syntax

```
_BITNAME
```

- Use the bit name preceded by an underscore
- Defined as a macro in the header file
- May be used instead of structure syntax

Examples

```
_LATA5 = 1;                          // Set bit 5 of PORTA
Flag = _RA5;                         // Read bit 5 of PORTA
while (!_DONE) { … }                 // While A/D converting
_SSRC = 5;                           // Set 3-bit field = 5
//3-bits in bit field: SSRC2 = 1, SSRC1 = 0, SSRC0 = 1
```

Register Variable Declaration
Example for PIC24FJ128GA010

- For every bit in the device, there is a `#define` in the device specific header file in the form of:

Bit Access Shorthand Macro Definition

```
#define _BITNAME REGNAMEbits.BITNAME;
```

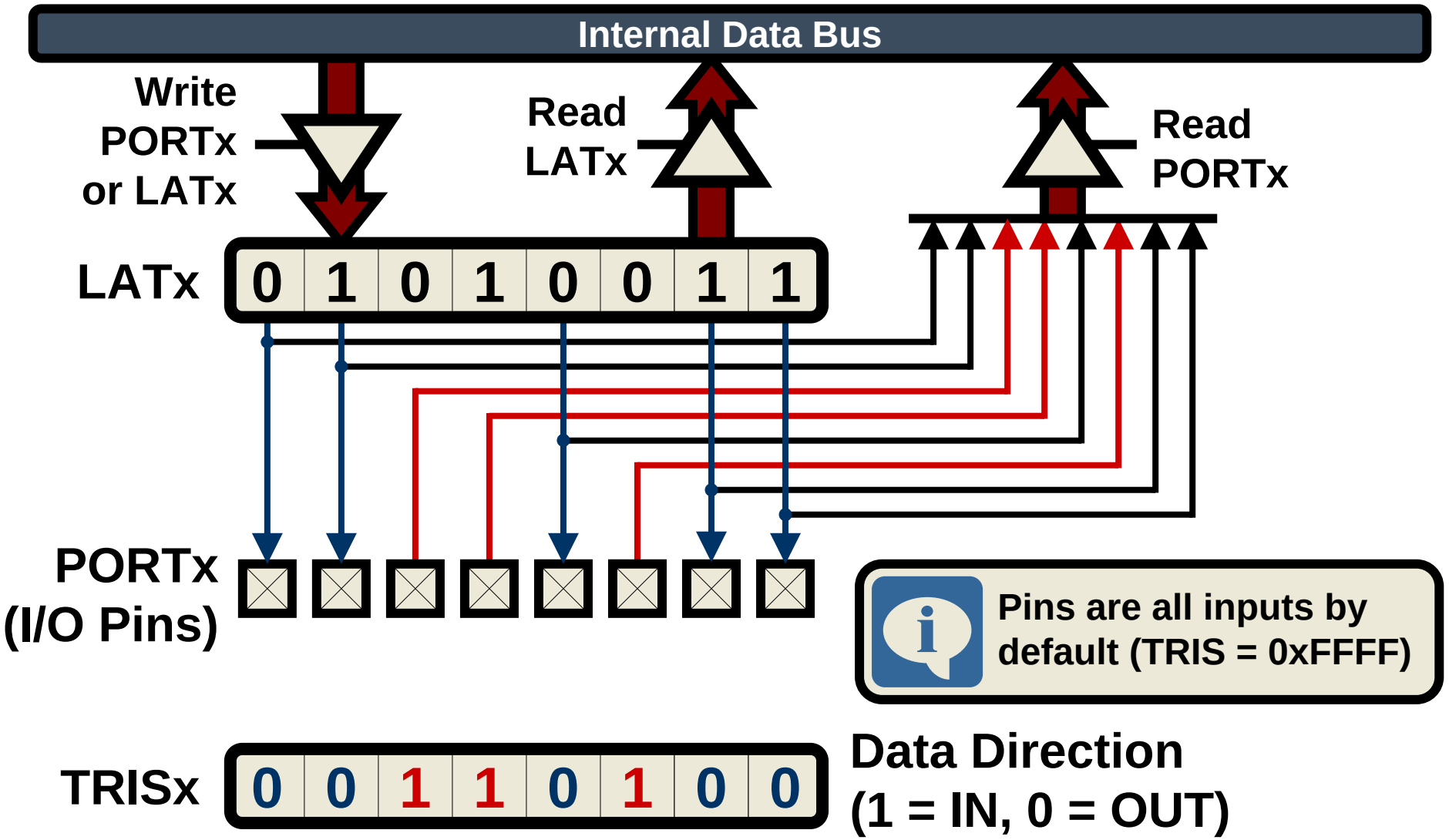
- Only works if header file is included
- Allows access to all bits by bit name only

Data Input and Output

Working with I/O Ports

Data Input and Output

TRIS, PORT and LAT Registers



Data Input and Output
TRIS, PORT and LAT Registers

- **Steps for working with PIC® I/O ports**

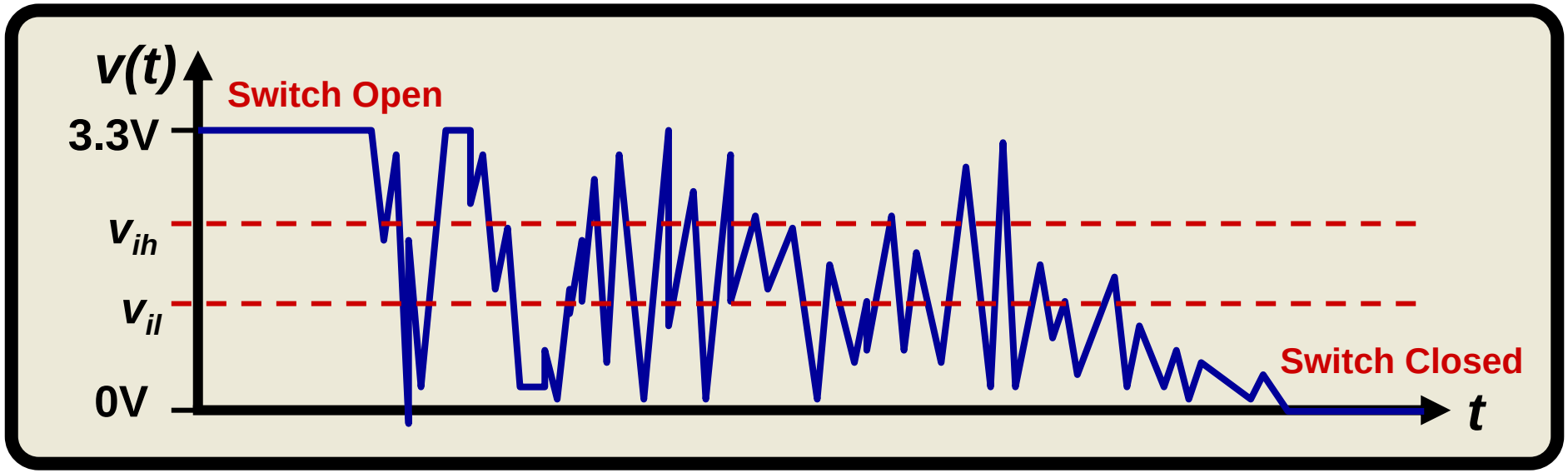
1	Initialize Output Latches to Known State	Write to: **LAT**
		`LATx = 0;`
		`LATxbits.LATxn = 0;`
		`_LATxn = 0;`

2	Configure Data Direction of Pins	Write to: **TRIS**
		`TRISx = 0x0023;`
		`TRISxbits.TRISxn = 1;`
		`_TRISxn = 1;`

3a	Write to outputs	Write to: **LAT**
		`LATx = 0x00F0;`
		`LATxbits.LATxn = 0;`
		`_LATxn = 0;`

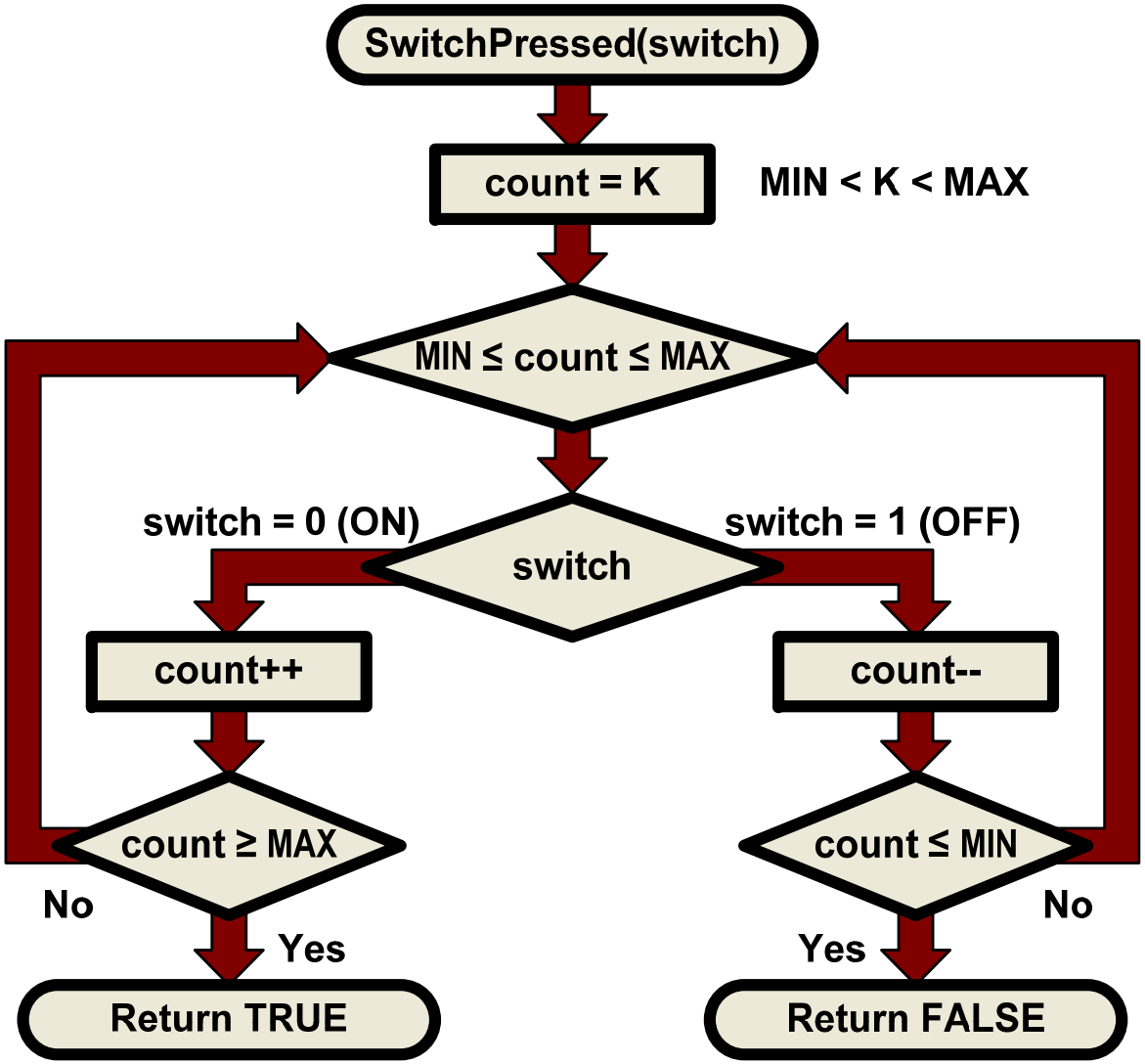
3b	Read from inputs	Read from: **PORT**
		`myVar = PORTx;`
		`myVar = PORTxbits.Rxn;`
		`myVar = _Rxn;`

How to Read Input Pins
Switch Debouncing



- Switch inputs must be debounced to prevent multiple false detections while switch contact settles
- Debounce routine provided in TLS2130.a
- Function prototype provided in TLS2130.h

How to Read Input Pins
Switch Debouncing



How to Read Input Pins
Switch Debouncing

Function Prototype

```
unsigned char SwitchPressed(volatile unsigned *sw, int bit);
```

- Written especially for this class
- Takes port pin variable as parameter
- Returns TRUE if switch is pressed
- Returns FALSE if switch is not pressed

Example

```
if (SwitchPressed(&PORTD, 6))
{
    // Do something if switch is pressed
}
```

Lab Exercise 3

Working with I/O Ports

Lab Exercise 3
Working with I/O Ports

Objective

Use the SwitchPressed() function to read S3. As long as S3 is pressed, LED D3 should be on. When S3 is released, LED D3 should be off.

- **Use the provided function SwitchPressed() to read and debounce S3 which is connected to bit 6 of PORTD**
- **While S3 pressed, turn on LED on RA0**
- **While S3 not pressed, turn off LED on RA0**

Lab Exercise 3
Working with I/O Ports

Procedure

Follow the directions in the lab manual starting on page 3-1.

On the lab PC…

If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

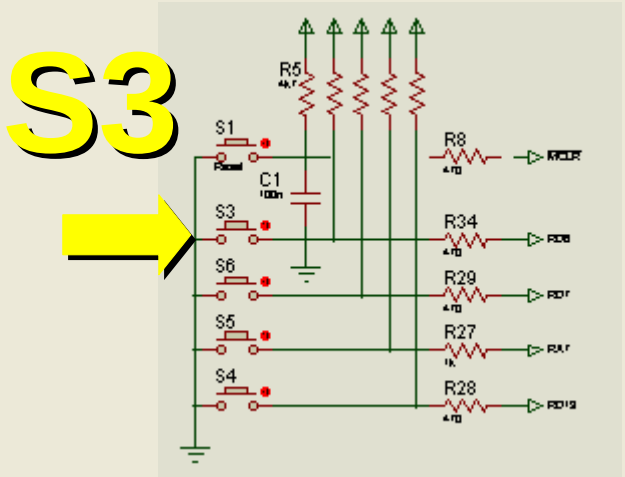
Open the lab workspace by selecting from the menu:
File ▶ Open Workspace…
and select the file:
C:\MTT\TLS2130\Lab3\Lab3.mcw

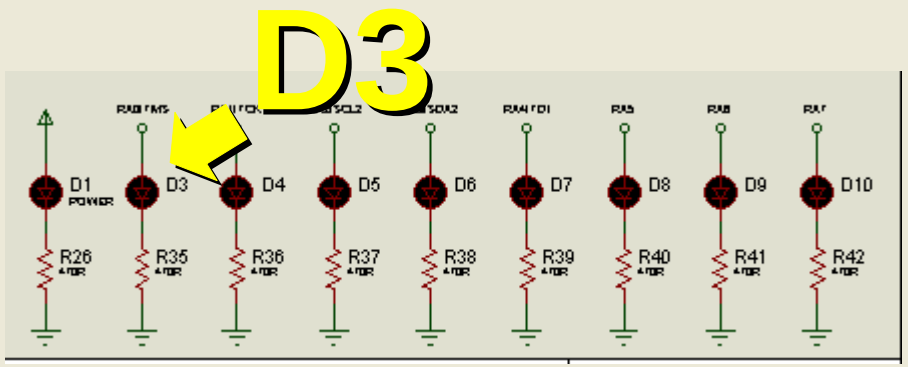
Lab Exercise 3
Working with I/O Pins

Results

Read RD6 via provided function: `SwitchPressed(&PORTD, 6)`

Write to RA0 using LATA: `LATAbits.LATA0 = 1 OR _LATA0 = 1`





Lab Exercise 3
Working with I/O Pins

Lab3.c – One possible solution…

```
#include <p24fj128ga010.h>
#include "TLS2130.h"

_CONFIG1(FWDTEN_OFF & JTAGEN_OFF)

int main(void)
{
    _TRISA0 = 0;
    while(1)
    {
        _LATA0 = 0;
        while(SwitchPressed(&PORTD, 6))
            _LATA0 = 1;
    }
}
```

Make bit 0 of PORTA an output

Turn off LED on bit 0 of PORTA

Read switch on bit 6 of PORTD

Turn on LED on bit 0 of PORTA

Lab Exercise 3
Working with I/O Pins

Conclusions

- TRIS registers determine I/O pin direction
- LAT registers used to write outputs
- PORT registers used to read inputs
- Switch inputs must be debounced
- Header files make it possible to work with an entire register (*REGNAME*) or individual bits (*REGNAME*`bits.`*BITNAME* or *_BITNAME*)

The C Runtime Environment

A Foundation for C Programs

Program Memory (Flash)

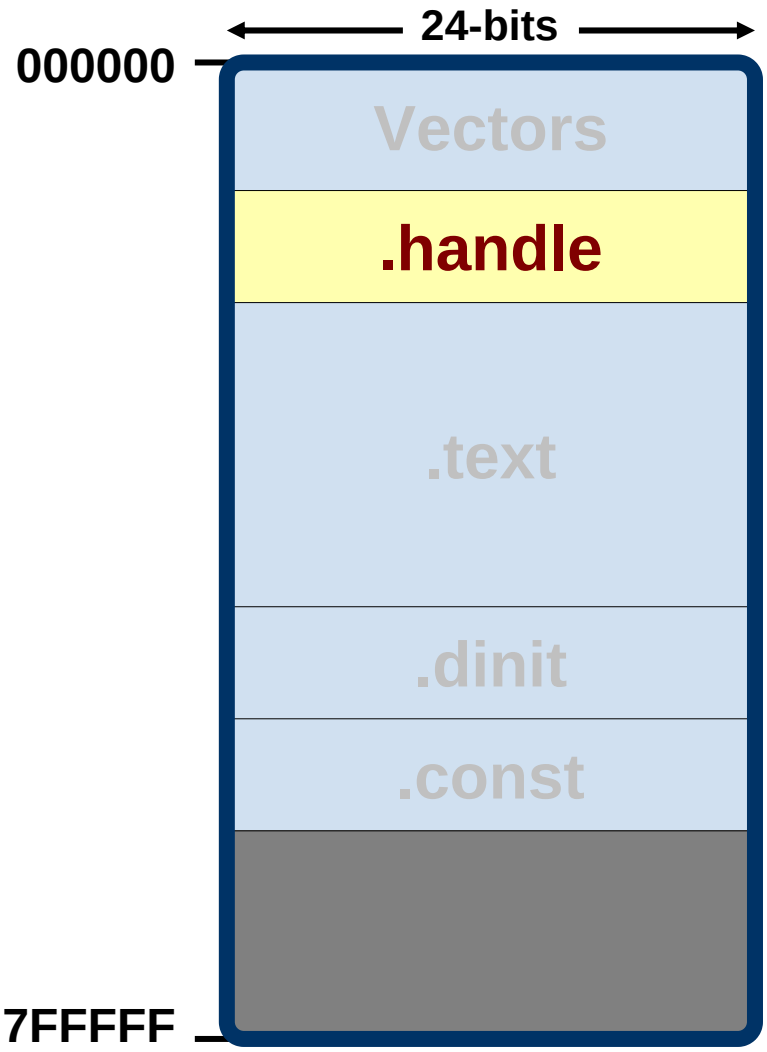
The Compiler's Viewpoint

Vectors

- Fixed locations in hardware
  - Reset Vector
  - Traps
  - Alternate Traps
  - Interrupt Vectors
  - Alternate Interrupt Vectors

Program Memory (Flash)

The Compiler's Viewpoint

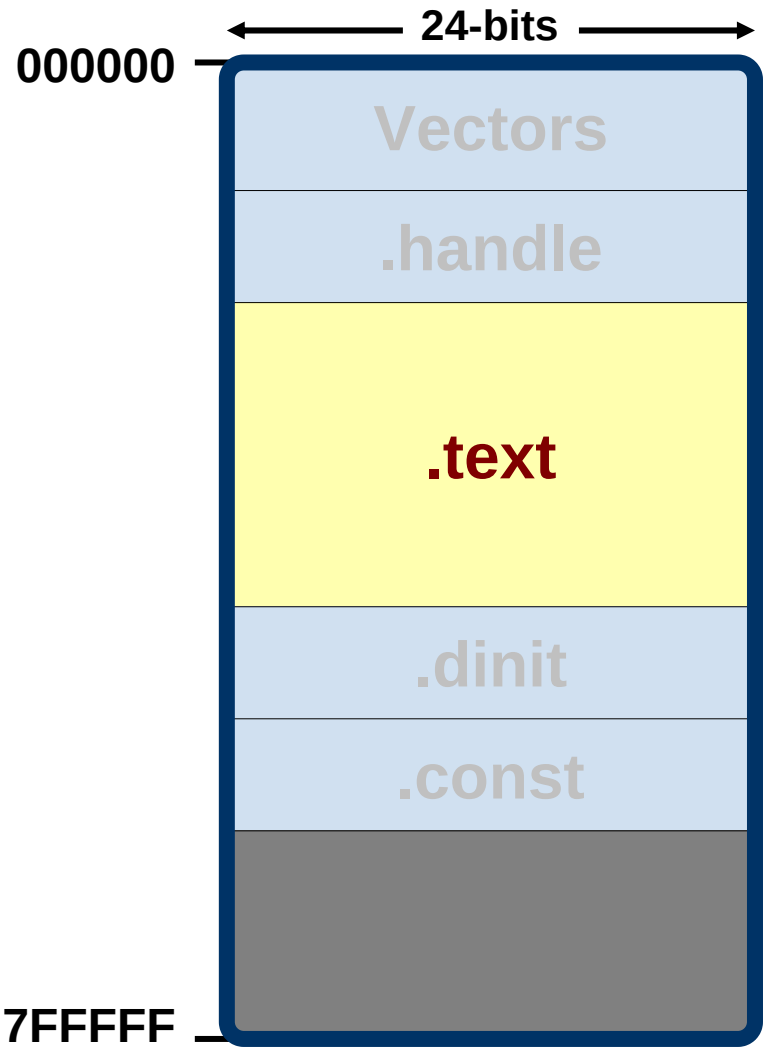


Far Code Handles

- Function Pointers
  - Allows use of 16-bit pointers in RAM to access 24-bit program memory addresses
  - Implements table of GOTO instructions

Program Memory (Flash)

The Compiler's Viewpoint

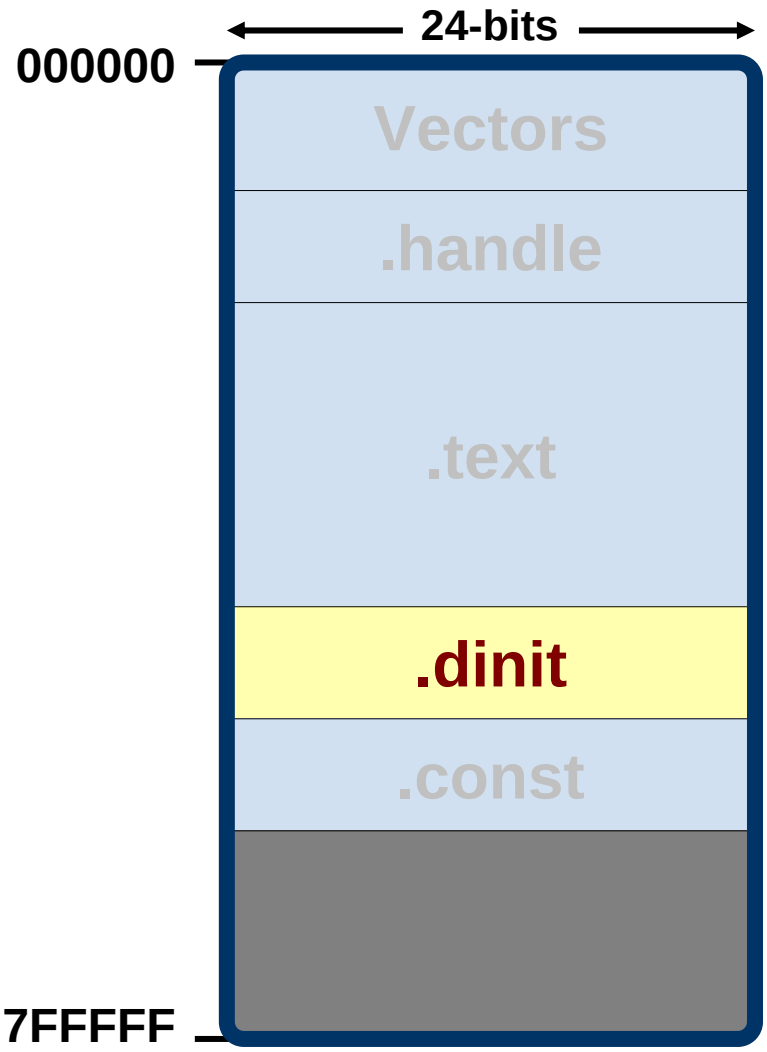


General Program Storage

- User Code
  - All executable code is placed here
- Initialization code placed here under the section name `.init`

Program Memory (Flash)

The Compiler's Viewpoint

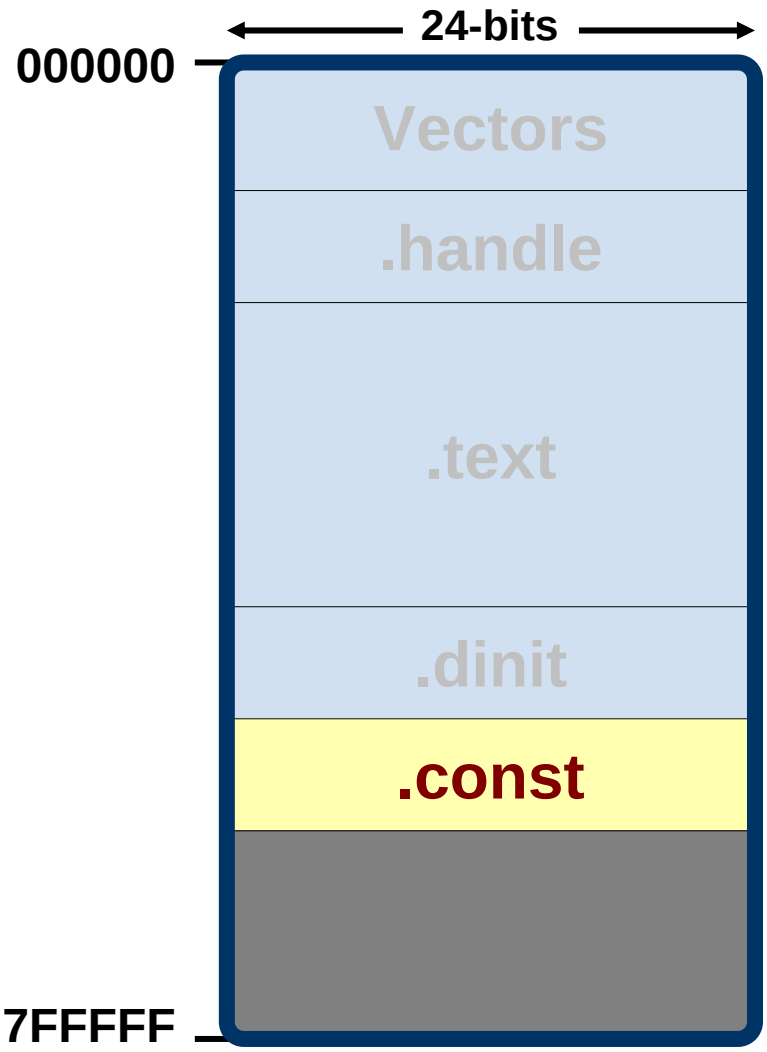


Data Memory Initializers

- Constants used to initialize:
  - Initialized Variables (e.g. `int x = 10;`)
  - Constants in RAM
- Values copied into RAM by startup code

Program Memory (Flash)

The Compiler's Viewpoint

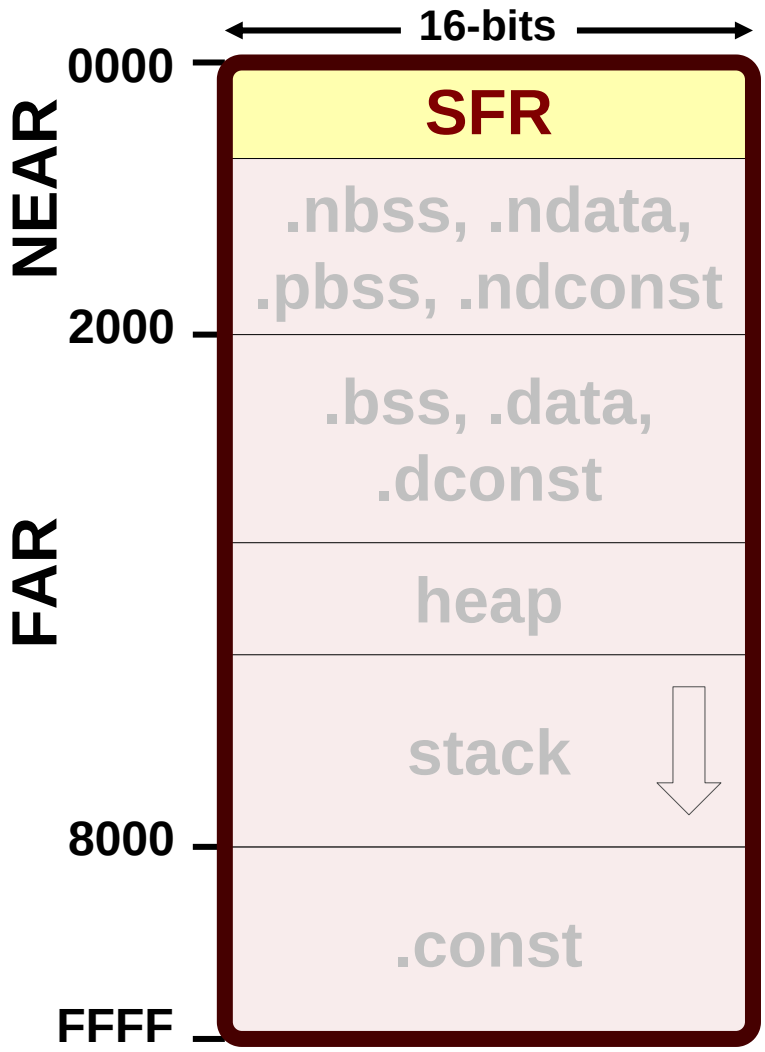


Constants in Flash

- Constants accessed via 32kB PSV window in RAM
- Constants declared with `const` keyword
- Managed via auto PSV feature

Data Memory (RAM)

The Compiler's Viewpoint

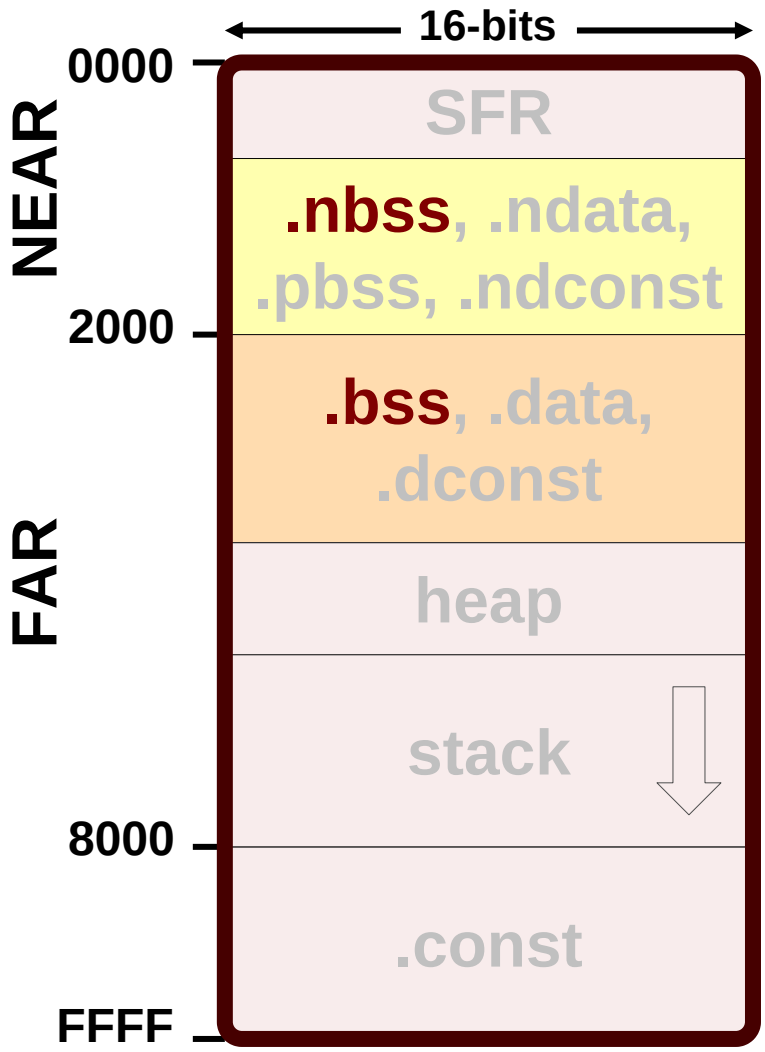


Special Function Registers

- **Hardware Registers**
- **Fixed Addresses**

Data Memory (RAM)

The Compiler's Viewpoint

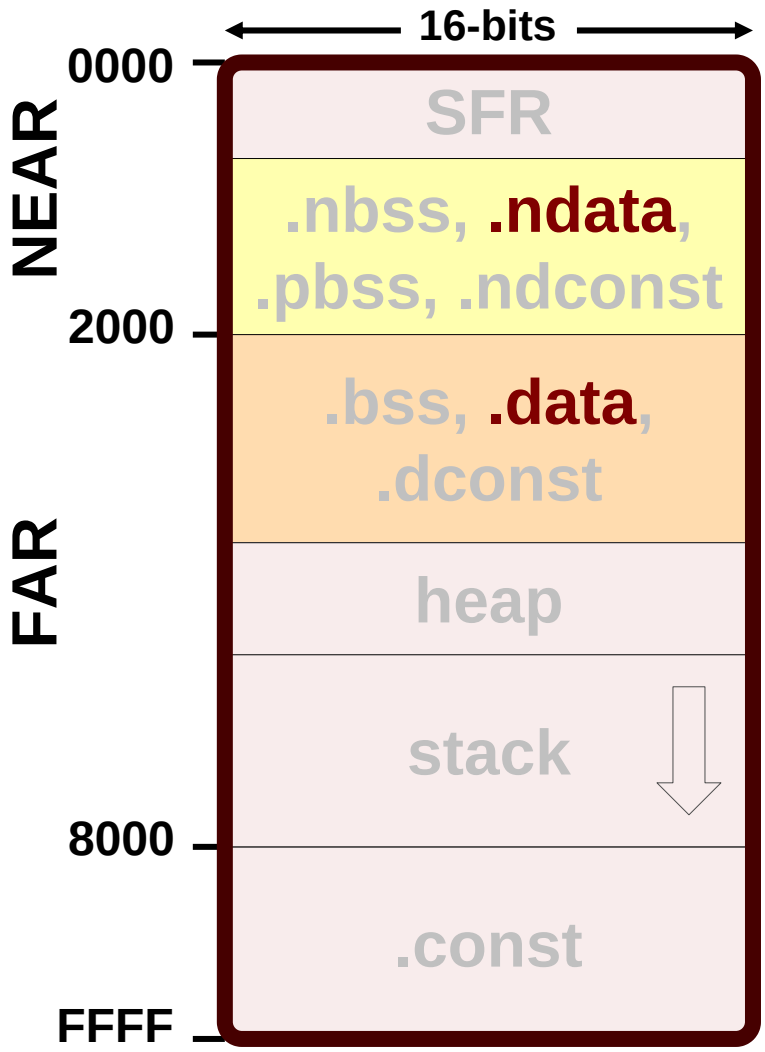


Uninitialized Variables

- e.g. `int x;`
- `.nbss` in `near` space
- `.bss` in `far` space

Data Memory (RAM)

The Compiler's Viewpoint

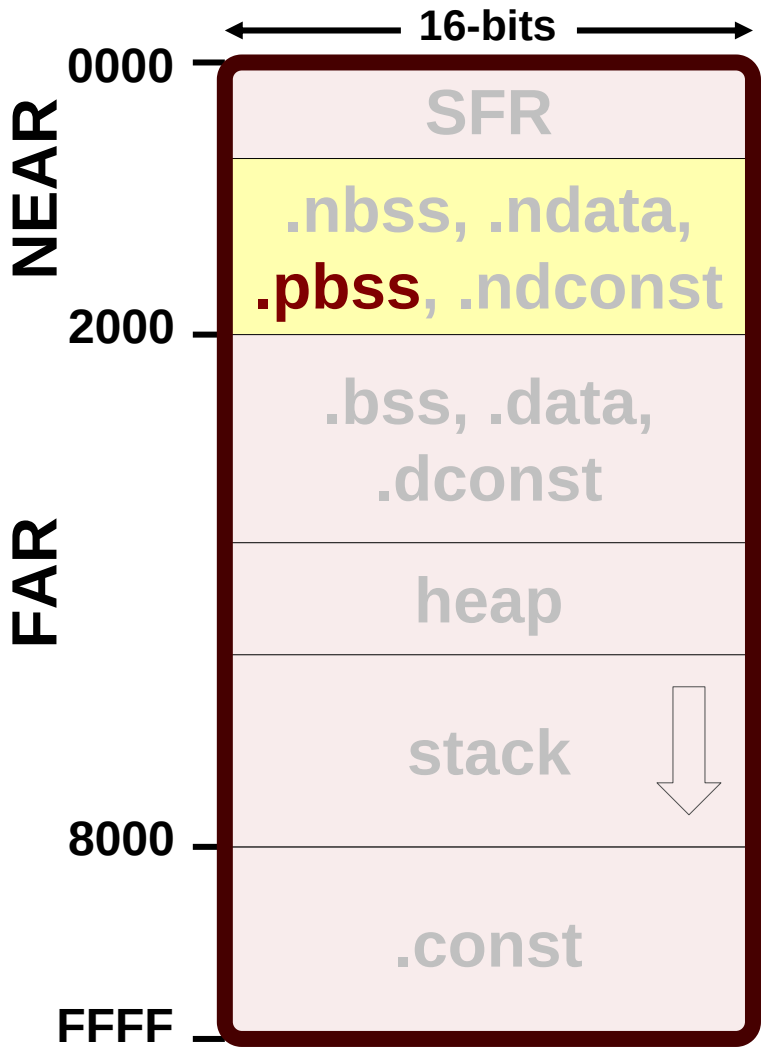


Initialized Variables

- e.g. `int x = 10;`
- `.ndata` in `near` space
- `.data` in `far` space
- Initial values copied from `.dinit` in flash by startup code

Data Memory (RAM)

The Compiler's Viewpoint

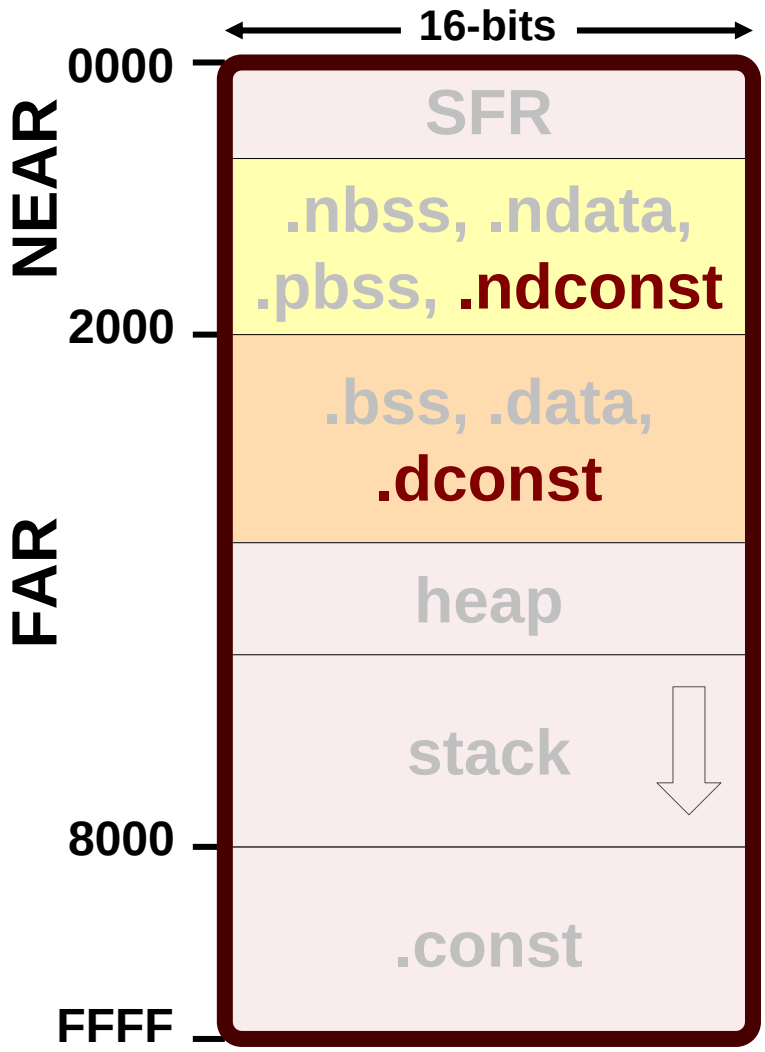


Persistent Variables

- Data unaffected by reset
- Not modified by startup code
- Located in `near` memory

Data Memory (RAM)

The Compiler's Viewpoint

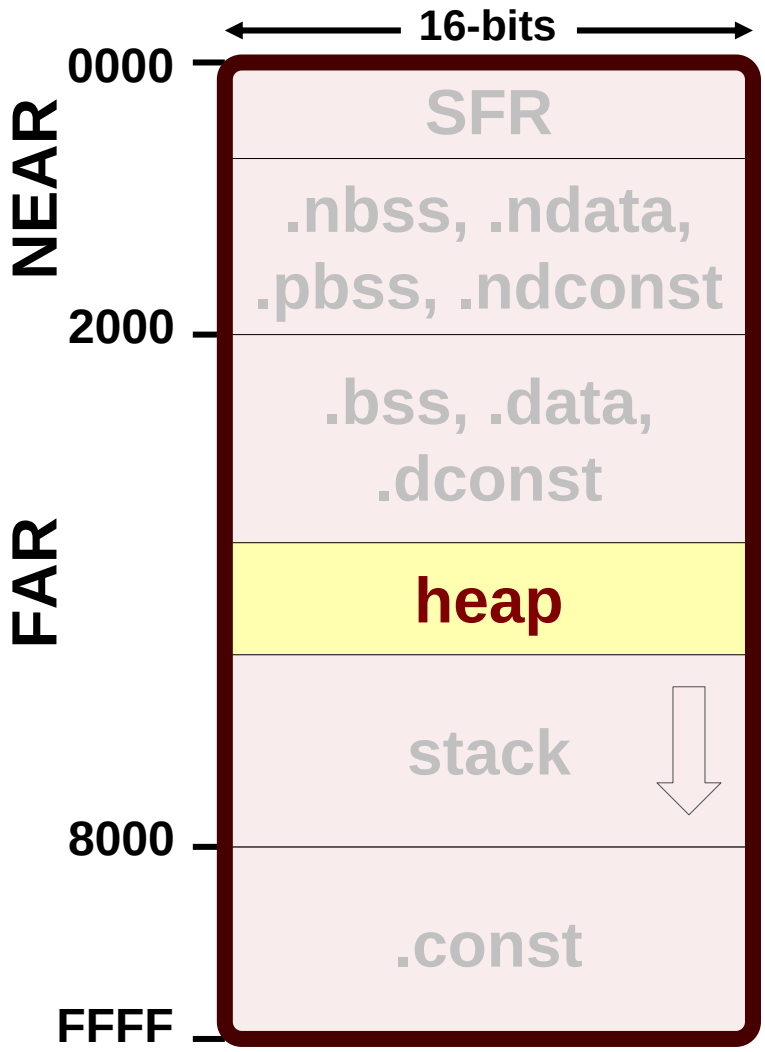


Constants in RAM

- `.ndconst` in `near` if small model used
- `.dconst` in `far` if large model used
- Created if `-mconst-in-code` switch not used
- Values copied from `.dinit` in flash by startup code

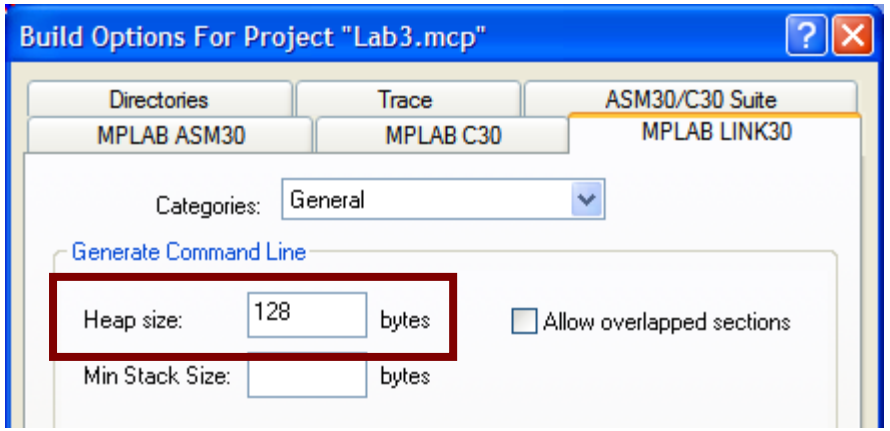
Data Memory (RAM)

The Compiler's Viewpoint



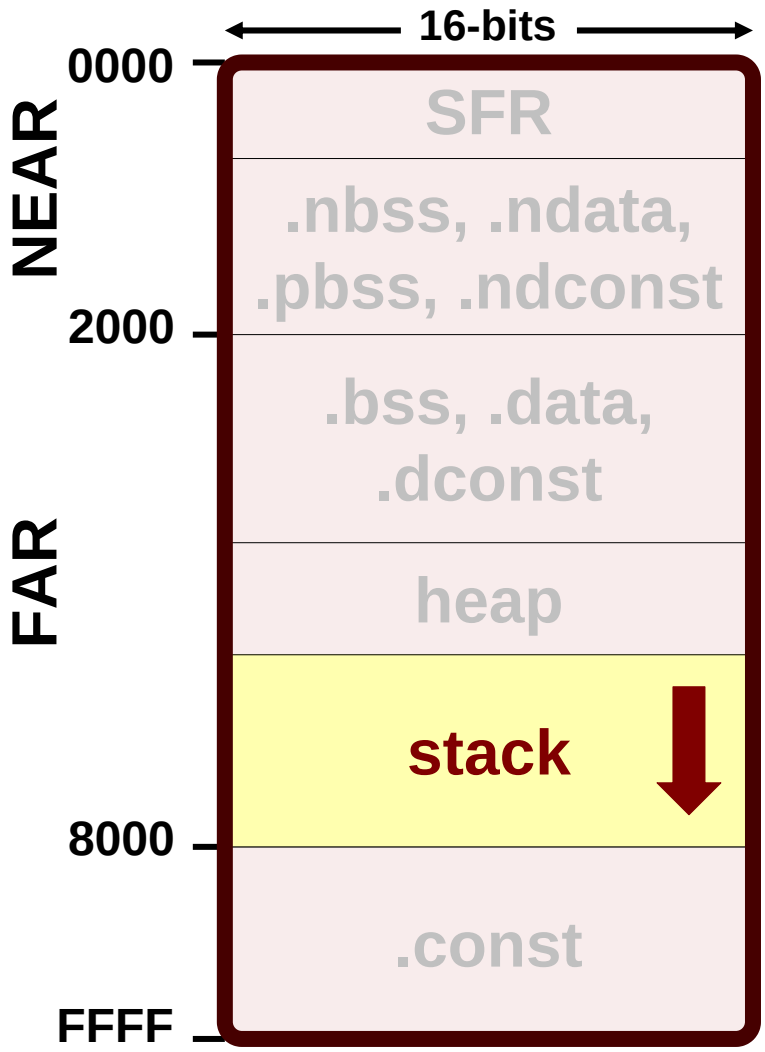
Heap

- Allocated if a heap declared in MPLAB®
- Size determined by user



Data Memory (RAM)

The Compiler's Viewpoint

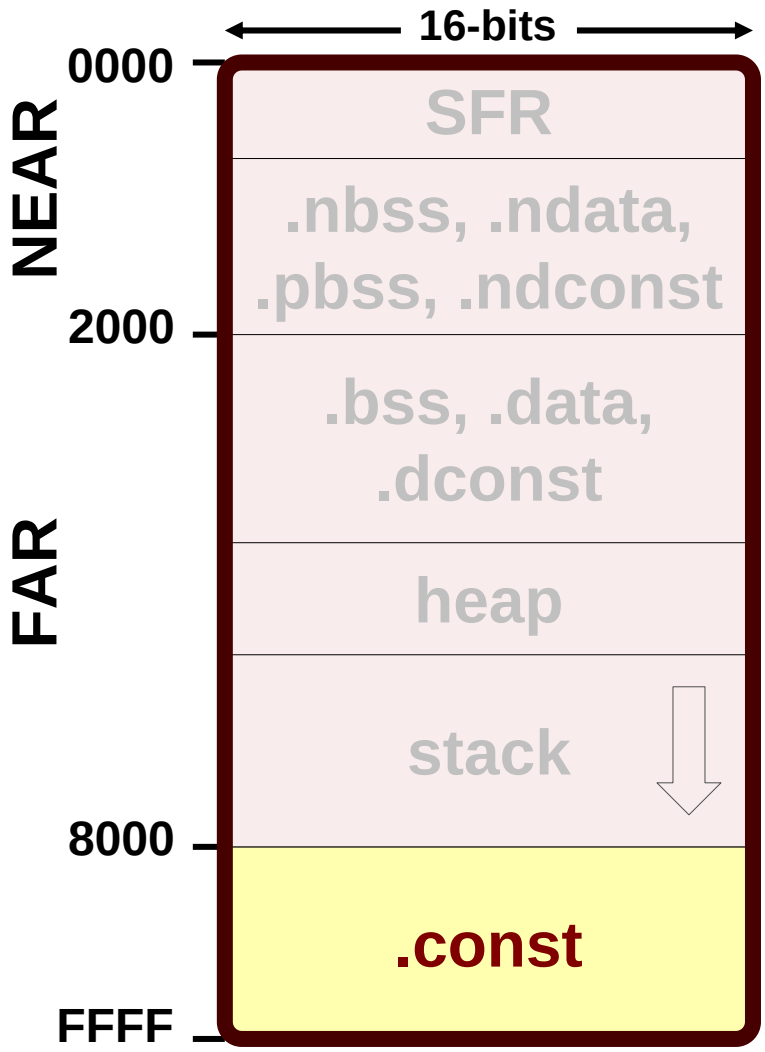


Stack

- **Automatically allocated by linker**
- **Occupies all unallocated RAM (by default)**
- **Grows toward higher addresses**

Data Memory (RAM)

The Compiler's Viewpoint

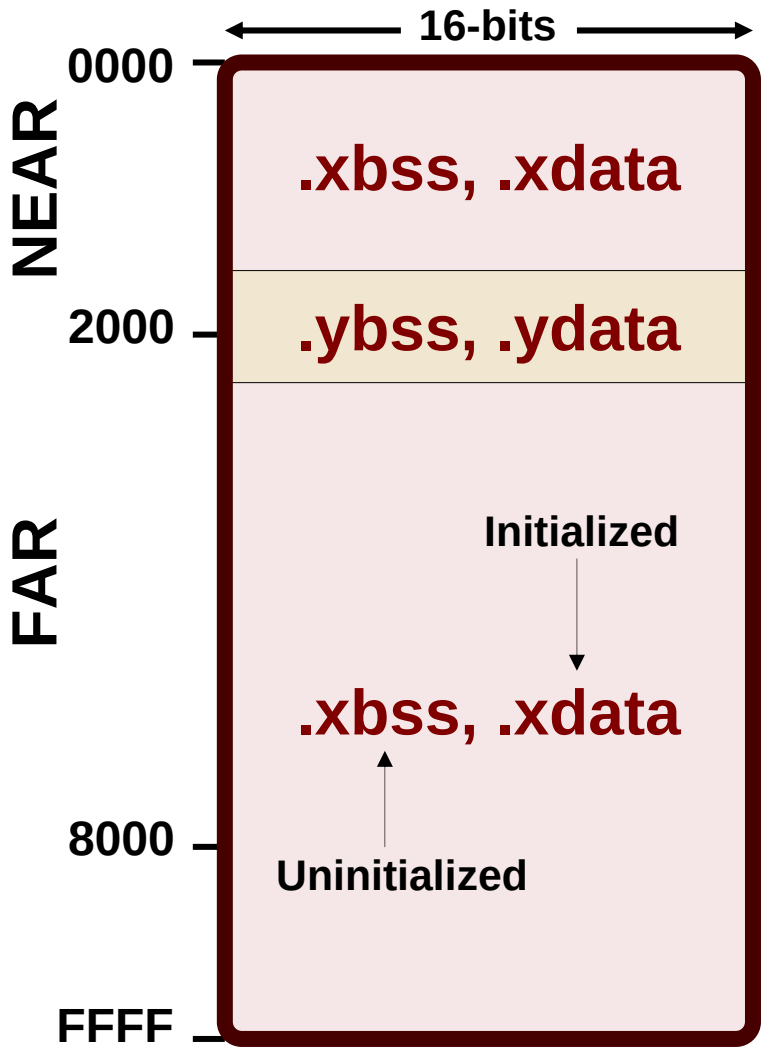


Constants in Code

- **32kB PSV window used to read constants from `.const` in flash as if they were in RAM**
- **No RAM exists above address 8000h**

Data Memory (RAM) for DSP

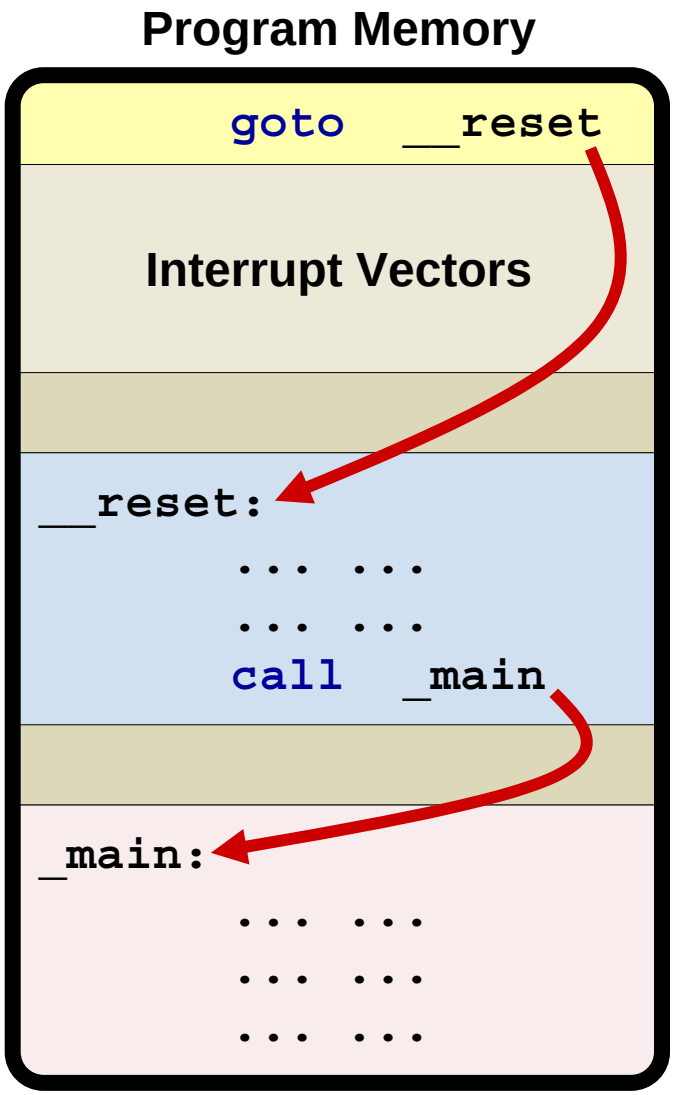
The Compiler's Viewpoint



X and Y Memory

- Used by MAC Class Instructions
- Non-DSP instructions see all memory as X
- Size and address range of Y memory varies from one device to another

Startup and Initialization



Reset Vector (Address 0x000000)
Populated automatically by LINK30 Linker
Calls runtime environment setup code in crt0.o (`__reset` label)

.o **crt0.o** or **crt1.o** (from libpic30.a)
C Runtime Environment Setup Code
Inserted automatically by LINK30 Linker
(Source files: crt0.s and crt1.s)

.c **main.c**
Your C code's `main()` routine.
Included by you in your project and placed in memory by LINK30 Linker

Startup and Initialization

Tasks Performed by Default Startup Module `crt0.o`

- Initialize Stack Pointer (W15)
- Initialize Stack Pointer Limit Reg. (SPLIM)
- Setup PSV window to view `.const` in flash
- Clear uninitialized data sections
- Copy data from `.dinit` in flash to initialized data sections in RAM
- Call items like `user_init`
- Call `main()` with no parameters
- If `main()` returns, reset processor

Startup and Initialization
Tasks Performed by Alternate Startup Module `crt1.o`

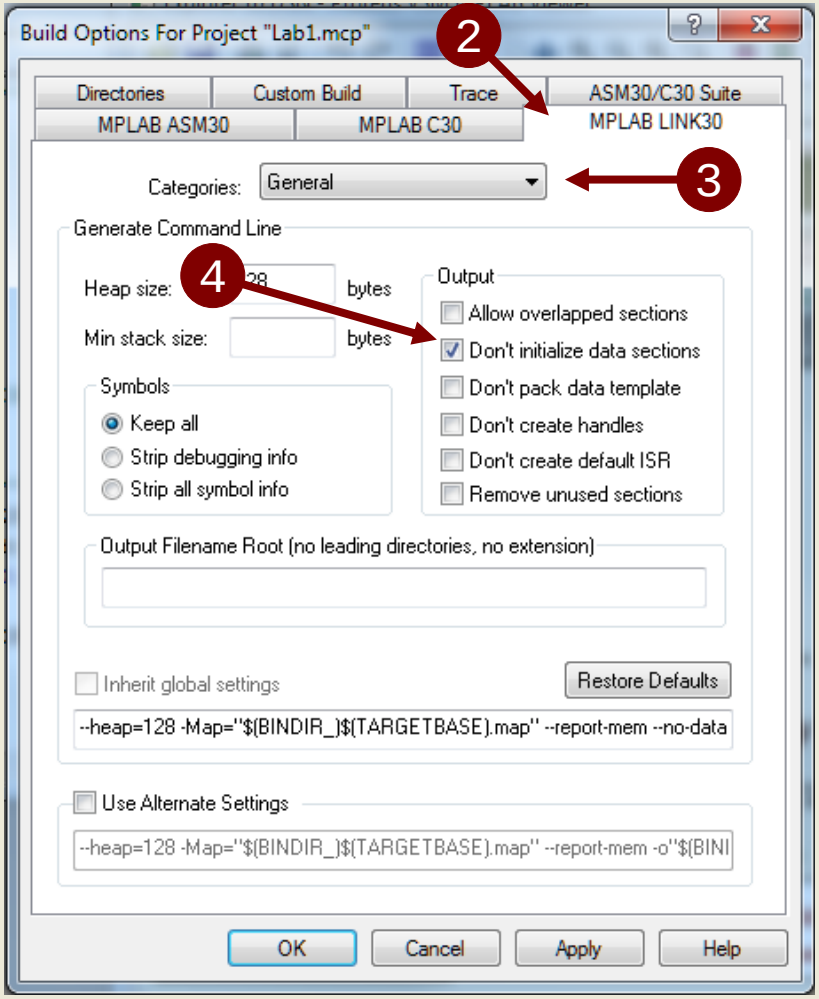
- Performs all steps taken by crt0.o except:
  - Does NOT clear uninitialized data sections
  - Does NOT load initialized data sections with values from `.dinit`
- Alternate startup module is much smaller
- Can be selected to conserve program memory if data initialization is not required

The persistent data section `.pbss` is never cleared or initialized by either startup module.

Startup and Initialization

How to Select the Alternate Startup Module crt1.o

Selecting the Alternate Startup Module crt1.o



1. **Open Project Settings**

   From the menu bar, select:
   Project ▸ Build Options... ▸ Project

2. **Select Linker Options**

   From the tabs, select:
   MPLAB LINK30

3. **Select Symbols & Output**

   From the Categories combo box select:
   General

4. **Disable data initialization**

   In the Output check box group, check:
   Don't initialize data sections

Startup and Initialization
Modifying the Startup Module

- Startup modules may be modified if needed
  - Source code (assembly) provided in:

    C:\Program Files\Microchip\MPLAB C30\src\pic30\crt*.s
  - If `main()` needs to be called with parameters, a conditional assembly directive may be switched to provide this support
  - Custom code may be run before startup code

Memory Models

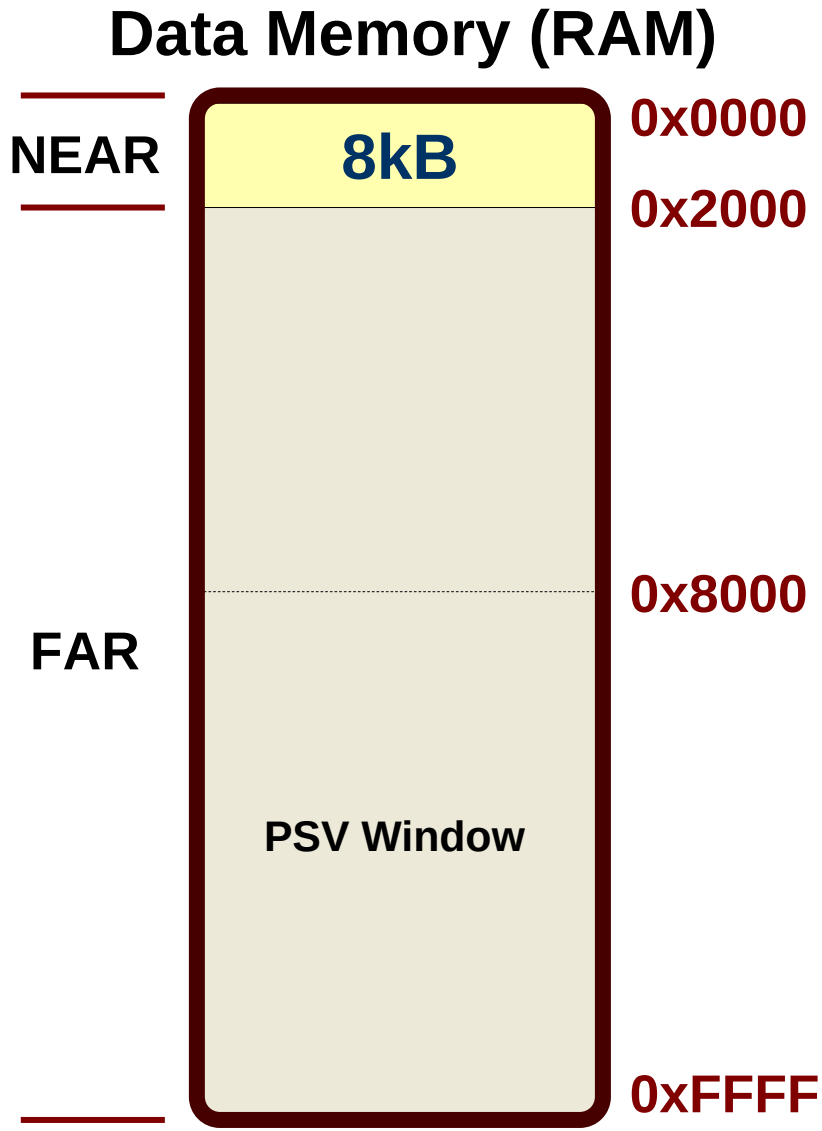
Object Allocation Schemes

Memory Models

Overview

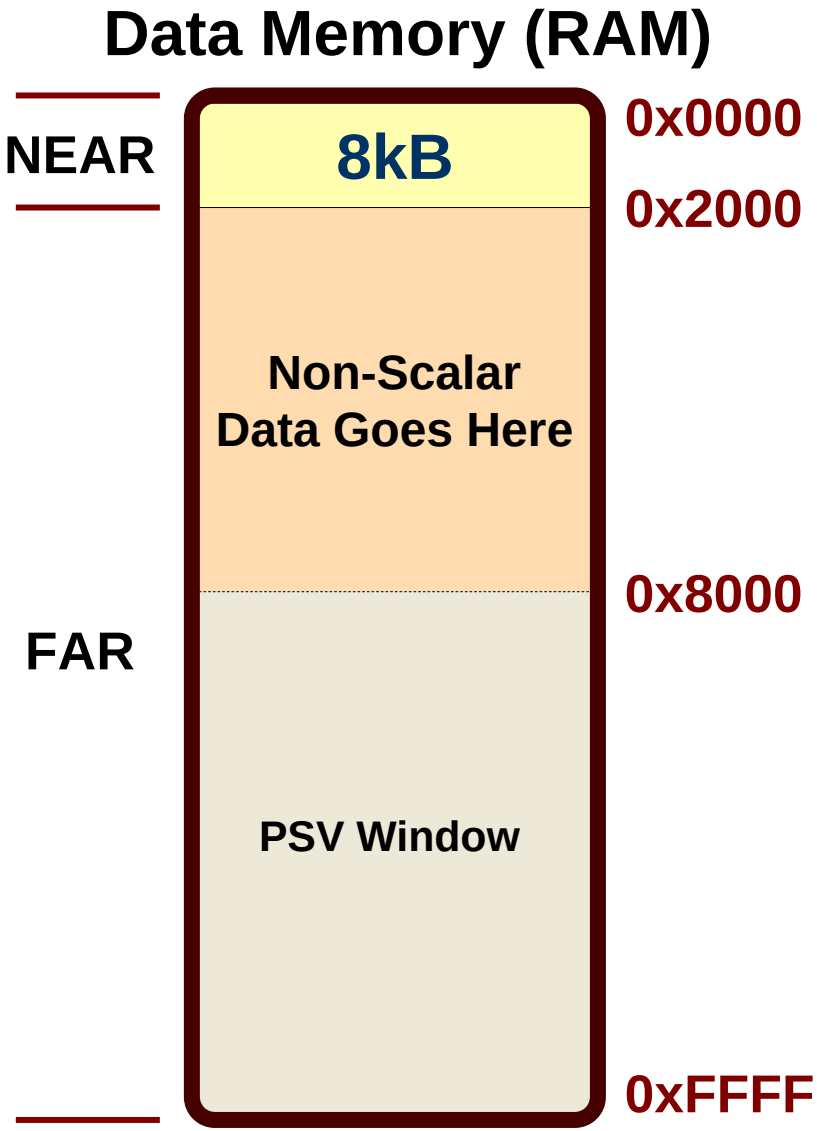
Option	Memory Definition	Description
Small Data `-msmall-data`	Up to 8 KB of data memory. **This is the default.**	Permits use of direct addressing for accessing data memory.
Small Scalar `-msmall-scalar`	Up to 8 KB of data memory. **This is the default.**	Permits use of direct addressing for accessing scalars in data memory.
Large Data `-mlarge-data`	Greater than 8KB of data memory.	Uses indirection for data references.
Small Code `-msmall-code`	Up to 32 Kw of program memory. **This is the default.**	No jump table for function pointers. Function calls use RCALL instruction.
Large Code `-mlarge-code`	Greater than 32 Kw of program memory.	Function pointers might use jump table. Function calls use CALL instruction.
Constants in Data `-mconst-in-data`	Constants located in data memory.	Values copied from program memory by startup code.
Constants in Code `-mconst-in-code`	Constants located in program memory. **This is the default.**	Values are accessed via Program Space Visibility (PSV) data window.

Memory Models
Small Data



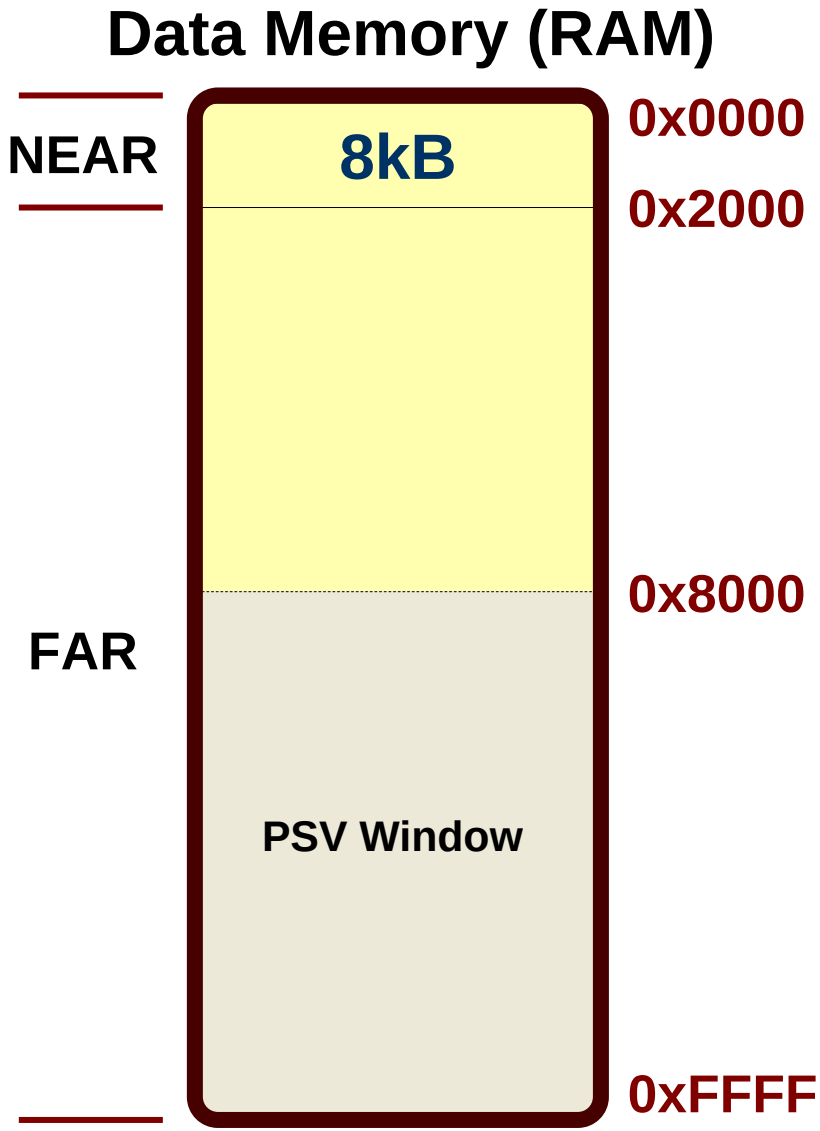
- **Default Model**
- **All data fits in first 8kB**
- **Direct Addressing Used (no pointers required)**
- **Fastest data access**
- **Smallest code to access data**

Memory Models
Small Scalar Data



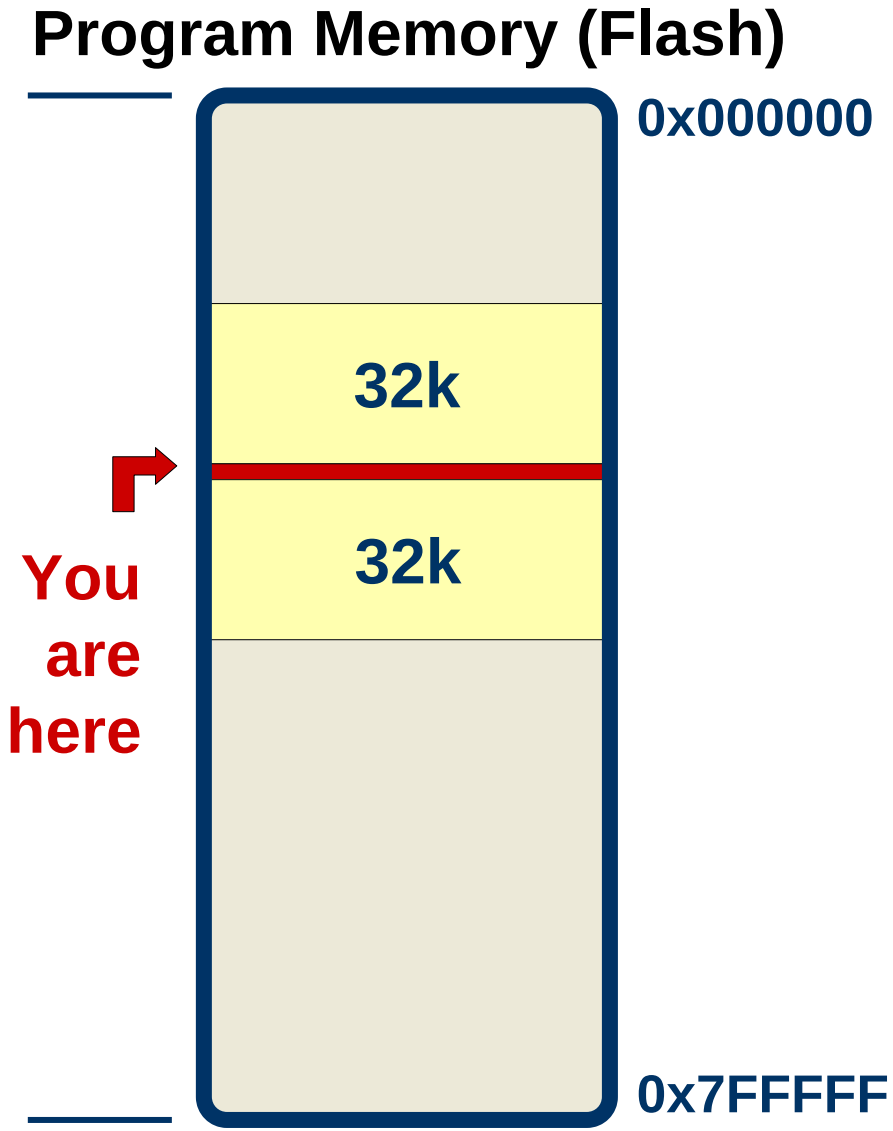
- Scalar data in first 8kB
- Direct addressing used with scalar data (no pointers required)
- Non-scalar data (arrays, structures) in far space
- Indirect addressing (pointers) used with non-scalar data

Memory Models
Large Data



- All data *cannot* fit in first 8kB
- All data treated as if it is in far space
- All data accessed indirectly (pointers)

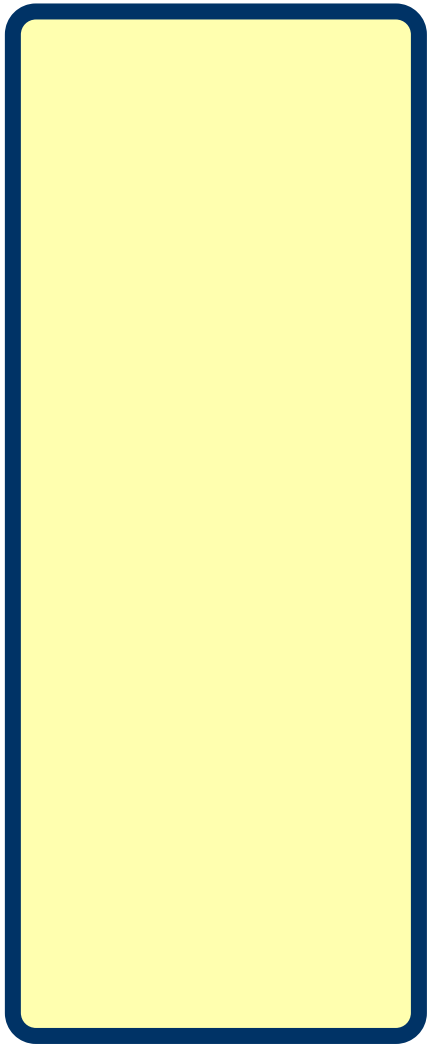
Memory Models
Small Code



- **Default Model**
- **All calls may jump only ±32k words**
- **No jump table for function pointers**
- **Function calls use `rcall` instruction**

Memory Models
Large Code

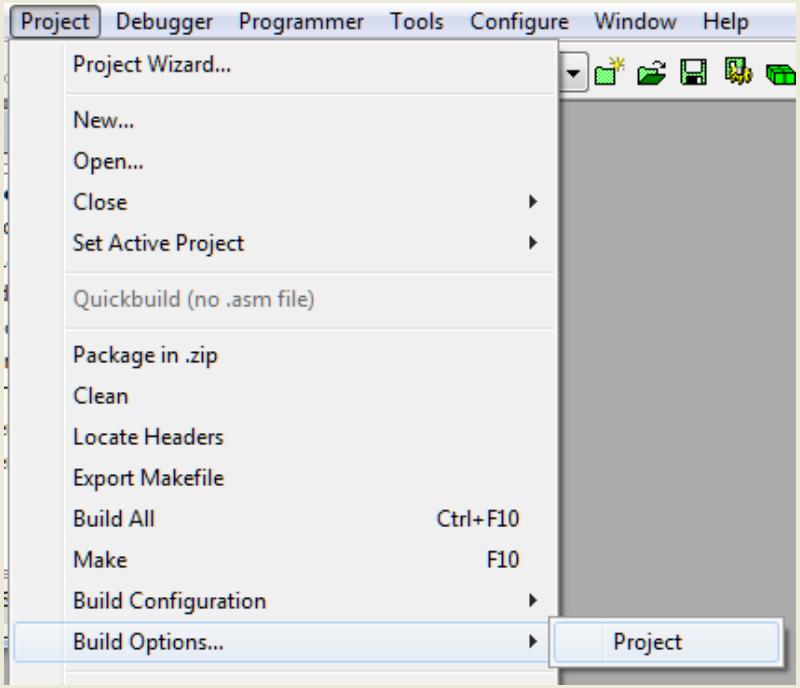
Program Memory (Flash)



- **Calls are not within ±32k words**
- **Jump table may be used for function pointers**
- **Function calls use `call` instruction**

Memory Models
How to Select the Memory Model

C30 Project Creation – Build Optoins



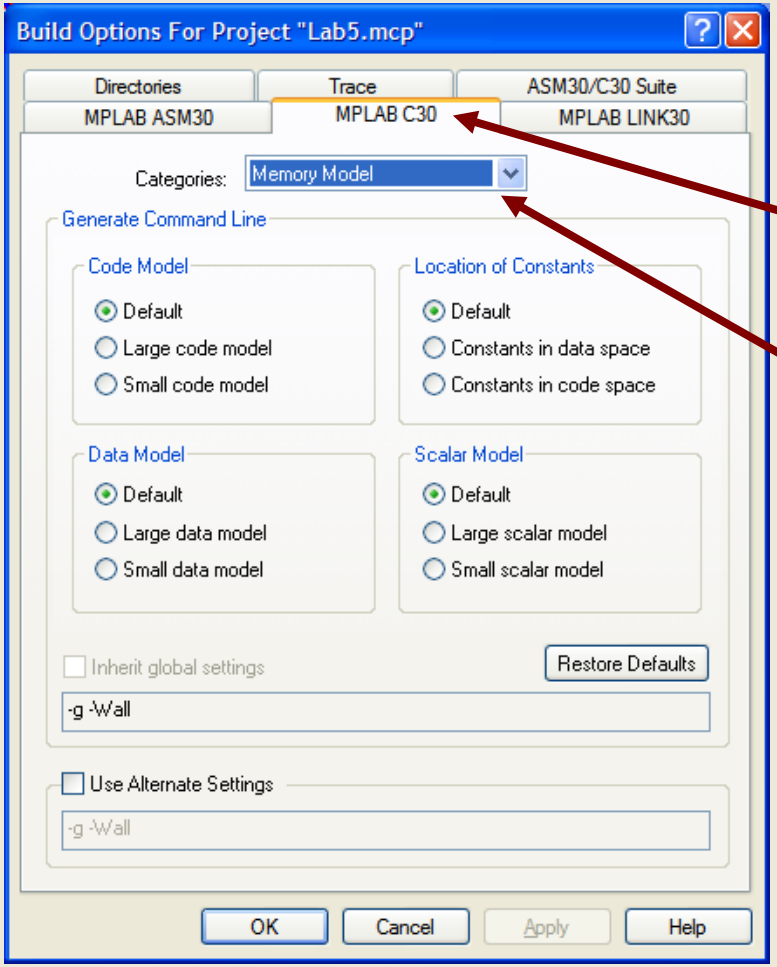
1 **Open Project Build Options**

- From the menu bar, select:
  Project ▶ Build Options... ▶ Project

Memory Models

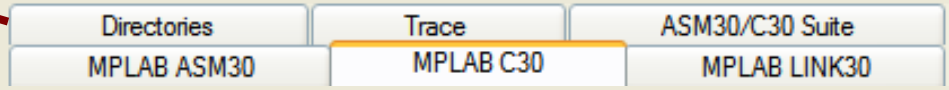
How to Select the Memory Model

C30 Project Creation – Build Options



2 **Go to Memory Models**

- Select the **MPLAB C30** tab

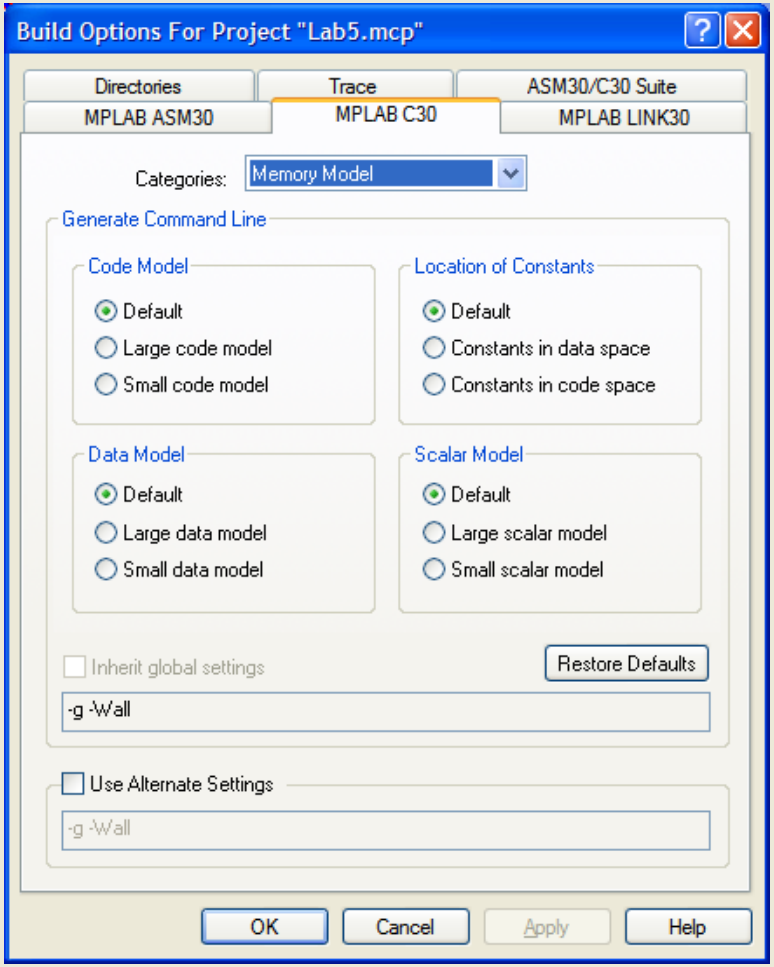


- Select the **Memory Model** category

Memory Models

How to Select the Memory Model

C30 Project Creation – Build Options



3 **Select Desired Models**

Code Model
- Default (small)
- Large code model
- Small code model

Location of Constants
- Default (code)
- Constants in data space
- Constants in code space

Data Model
- Default (small)
- Large data model
- Small data model

Scalar Model
- Default (small)
- Large scalar model
- Small scalar model

Memory Models
Tips & Tricks

- Inappropriate model for your program can cause compile or link errors
- As your program grows, you may need to change the memory model
- If desired, you have full control over where objects are placed in memory
  - Use small model, but force some objects into far memory
  - Compile different modules with different models

Memory Models
Tips & Tricks

- **Compiler can often generate more compact code if variables in near data**

Option	Tips for Optimal Memory Use
Small Data	Use if all variables for the application can fit within 8KB
Small Scalar	Use if all scalar type variables (no arrays or structures) for the application can fit within 8KB
Small Data or Small Scalar	If all data doesn't fit in near space, tag some variables with the `far` attribute so others have space to fit in near data.
Large Data	1. Compile some individual modules using the Small Data or Small Scalar option. Then include their compiled object modules in the Large Data project. 2. Tag individual variables with the `near` attribute

Memory Models
Tips & Tricks

- **Functions that are near (within 32k radius) may call each other more efficiently**

Option	Tips for Optimal Memory Use
Small Code	1. Use if all functions are within 32kw of each other. 2. Compile some modules using Small Code and include their object files in a Large Code project. 3. If not all functions are within 32kw of each other, tag some functions with the `far` attribute.
Large Code	1. Tag some functions with the `near` attribute. An error will be generated if the function cannot be reached by one of its callers using the more efficient form of the function call.

Attributes

GCC’s Replacement for #pragma

Attributes

Definition

Attributes are used to tell the compiler about certain behaviors or features of a variable, a function or a type. Attributes are specified using the keyword `__attribute__` followed by a list of attributes within double parentheses.

- **Used to describe variables or functions**
- **Help compiler to generate optimized code**
- **Help compiler to optimize memory use**
- **Establish rules for how particular variables or functions are to be handled with respect to the C runtime environment**

Attributes

Why attributes are used instead of ISO C's `#pragma`

- According to the GCC developers:
  - It is impossible to generate `#pragma` commands from a macro
  - "There is no telling what the same `#pragma` might mean in another compiler"
- In short, attributes are much more versatile than the traditional `#pragma`

Variable Attributes

How to Declare a Variable with Attributes

Syntax

```
type identifier __attribute__((attribute-list));
```

- Multiple attributes are separated by commas within the double parentheses
- Attributes may be before or after *identifier*
- Note: There are *two* underscores before and after the keyword `__attribute__`

Examples

```
int a[10] __attribute__((section(".xdata")));
int __attribute__((aligned(16))) b[10];
int x __attribute__((near));
```

Variable Attributes

Supported Attributes for Variables

- `address`
- `aligned`
- `deprecated`
- `far`
- `secure`
- `near`
- `noload`
- `packed`
- `persistent`
- `reverse`
- `section`
- `sfr`
- `space`
- `transparent_union`
- `unordered`
- `unused`
- `weak`

and many more…

Attributes may be specified with a leading and trailing double underscore to distinguish them from other entities in your code with the same name (e.g. `__aligned__`).

Function Attributes

How to Declare a Function with Attributes

Syntax

```
type identifier() __attribute__((attribute-list)) {…}
```

- Multiple attributes are separated by commas within the double parentheses
- Attributes may be before or after *identifier*
- Note: There are *two* underscores before and after the keyword `__attribute__`

Examples

```
int foo(void) __attribute__((address(0x100))) { … }

int __attribute__((address(0x500))) bar(void) { … }
```

Function Attributes

Supported Attributes for Functions

- `address`
- `alias`
- `const`
- `deprecated`
- `far`
- `format`
- `format_arg`
- `interrupt`
- `near`
- `no_instrument_function`
- `noload`
- `noreturn`
- `section`
- `shadow`
- `unused`
- `weak`

and many more…

Attributes may be specified with a leading and trailing double underscore to distinguish them from other entities in your code with the same name (e.g. `__address__`).

How to Override the Default Characteristics of Variables and Functions

Using Attributes

Locating Objects in Memory

How to place an object in near memory

`near` Attribute Syntax

```
__attribute__((near))
```

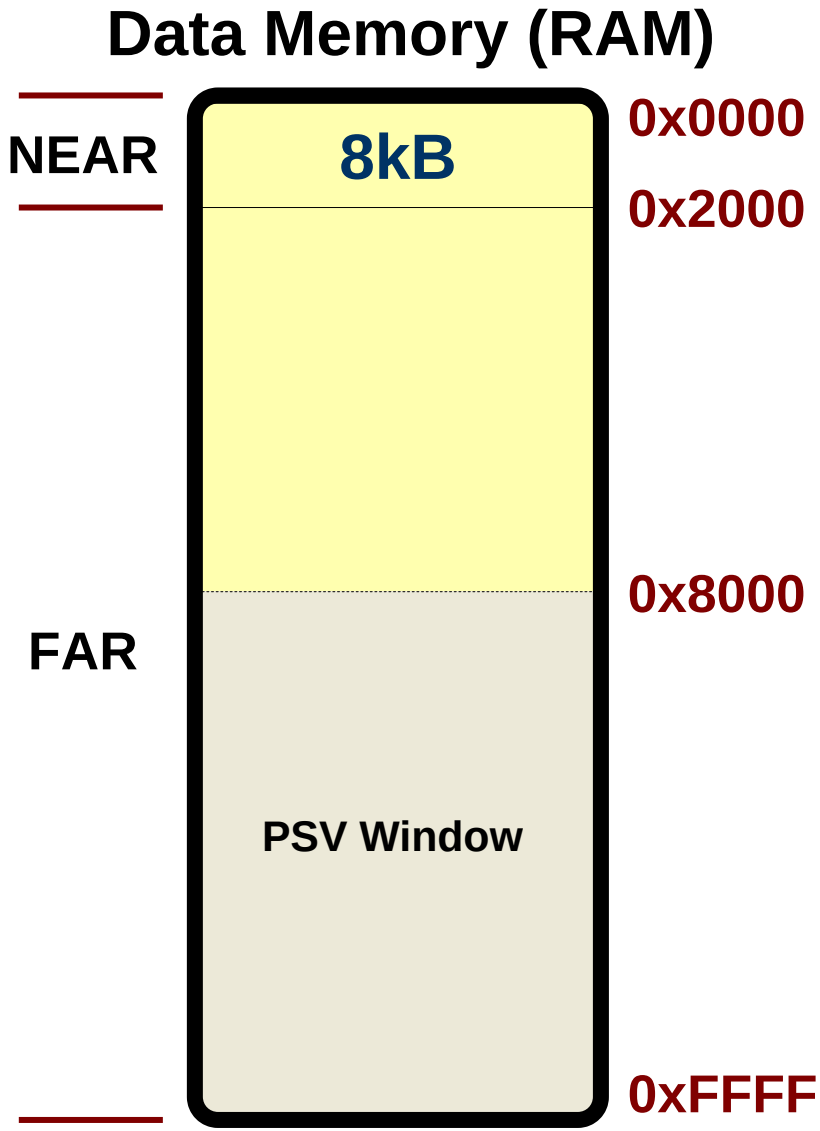
- Specifies that a function or variable should be located in near memory
- Treats a function or variable as if one of the small memory models were being used
- Should be accessed more efficiently

Examples

```
int x __attribute__((near));
int foo(void) __attribute__((near)) { … }
```

Locating Objects in Memory

Manually Optimizing Data Memory Use



- Use large data model
- Tag frequently accessed variables with `near` attribute

Locating Objects in Memory

How to place an object in far memory

`near` Attribute Syntax

```
__attribute__((far))
```

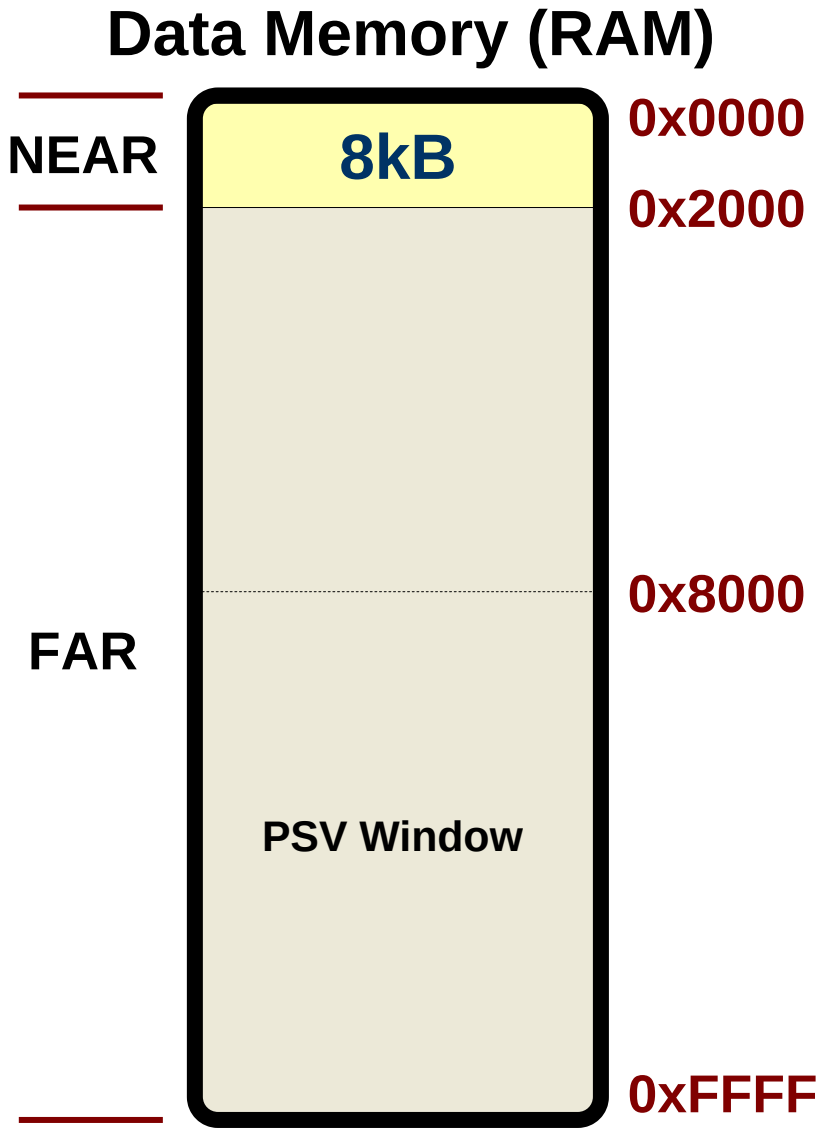
- Specifies that a function or variable should be located in far memory
- Treats a function or variable as if one of the large memory models were being used
- Might require extra overhead

Examples

```
int x __attribute__((far));
int foo(void) __attribute__((far)) { … }
```

Locating Objects in Memory

Manually Optimizing Data Memory Use



- Use small data model
- Let linker place most variables in near space (up to 8kB)
- Tag infrequently accessed variables with `far` attribute
  - Makes room in near space for frequently accessed variables
  - Allows more than 8kB of data with small model

Locating Objects in Memory

How to place a variable in X or Y data memory

`space` Attribute Syntax

```
__attribute__((space(space)))
```

- **Specifies that a variable should be located in X (`xmemory`) or Y (`ymemory`) data memory**
- **For dsPIC® devices only**
- **Useful for variables operated on by MAC class instructions**

Examples

```
int coefficient[101] __attribute__((space(xmemory)));
int input[255] __attribute__((space(ymemory)));
```

Locating Objects in Memory

How to place a variable in EEPROM data memory

`space` Attribute Syntax

```
__attribute__((space(eedata)))
```

- **Specifies that a variable should be located in EEPROM data memory**
- **For dsPIC30 devices only**

Examples

```
int coefficient[101] __attribute__((space(eedata)));
int x __attribute__((space(eedata)));
```

Locating Objects in Memory

How to place a variable in DMA memory

`space` Attribute Syntax

```
__attribute__((space(dma)))
```

- Specifies that a variable should be located in DMA data memory
- For some PIC24 and dsPIC33 devices only
- Many peripherals may read/write DMA memory without intervention by program

Examples

```
volatile int x __attribute__((space(dma)));
volatile int input[16] __attribute__((space(dma)));
```

Locating Objects in Memory

How to align an object on a byte/word boundary

`aligned` Attribute Syntax

```
__attribute__((aligned(alignment)))
```

- **Specifies the minimum alignment for the variable, measured in bytes**
- **Useful on dsPIC® in conjunction with assembly operations that require aligned operands**
- **Can improve the efficiency of some asm operations**

Example

Align `x` on a 16-byte boundary:

```
int x __attribute__((aligned(16)));
```

`aligned` can only increase the alignment. To reduce it, use `packed` instead.

Locating Objects in Memory

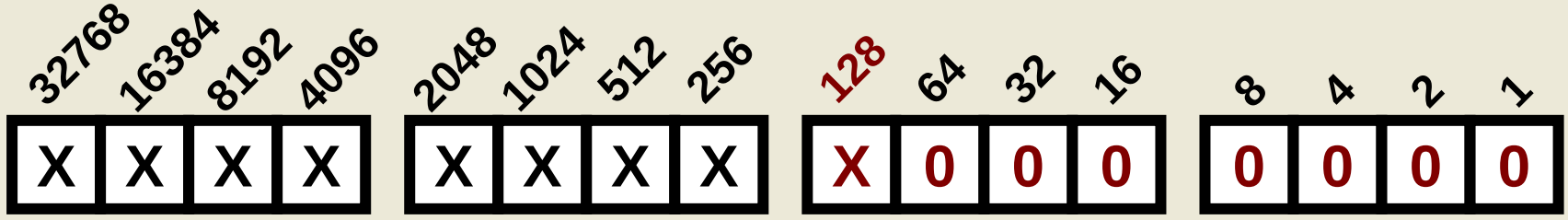
How to align an object on a byte/word boundary

- **Some operations require specific data alignment**
  - **Modulo Addressing (circular buffers)**

What does it mean to align an address on a byte boundary?

```
int __attribute__((aligned(128))) x[50];
```

128 chosen because it is the smallest power of 2 that can hold 100 bytes (50 words)



Possible addresses are of the form:
0xNN00 or 0xNN80 where N is any Hexadecimal Digit

Locating Objects in Memory

How to place an object at a specific address

`address` Attribute Syntax

```
__attribute__((address(address)))
```

- **Specifies the address where a function or variable should be located**
- **Use sparingly – program will be harder to optimize by linker**

Examples

```
int x __attribute__((address(0x1840)));
int foo(void) __attribute__((address(0x3000))) { … }
```

Overriding Default Behavior

How to make a variable persistent across resets

`address` Attribute Syntax

```
__attribute__((persistent))
```

- Prevents a variable from being overwritten upon device reset
- Places variable in the `.pbss` section which is unaffected by the runtime startup code

Examples

```
int x __attribute__((persistent));
int x _PERSISTENT;   //Shorthand macro defined in *.h
```

Overriding Default Behavior
Pre-defined Macros –PIC24 Devices

Memory Allocation Macros (PIC24)

```
#define _BSS(N) __attribute__((aligned(N)))
#define _DATA(N) __attribute__((aligned(N)))
#define _PERSISTENT __attribute__((persistent))
#define _NEAR __attribute__((near))
```

- **Defined in the header files**
- **Simplifies the task of identifying where you want to place an object in memory**

MPLAB® C30 Compiler User's Guide (Rev E): Section 6.6.3, Page 83

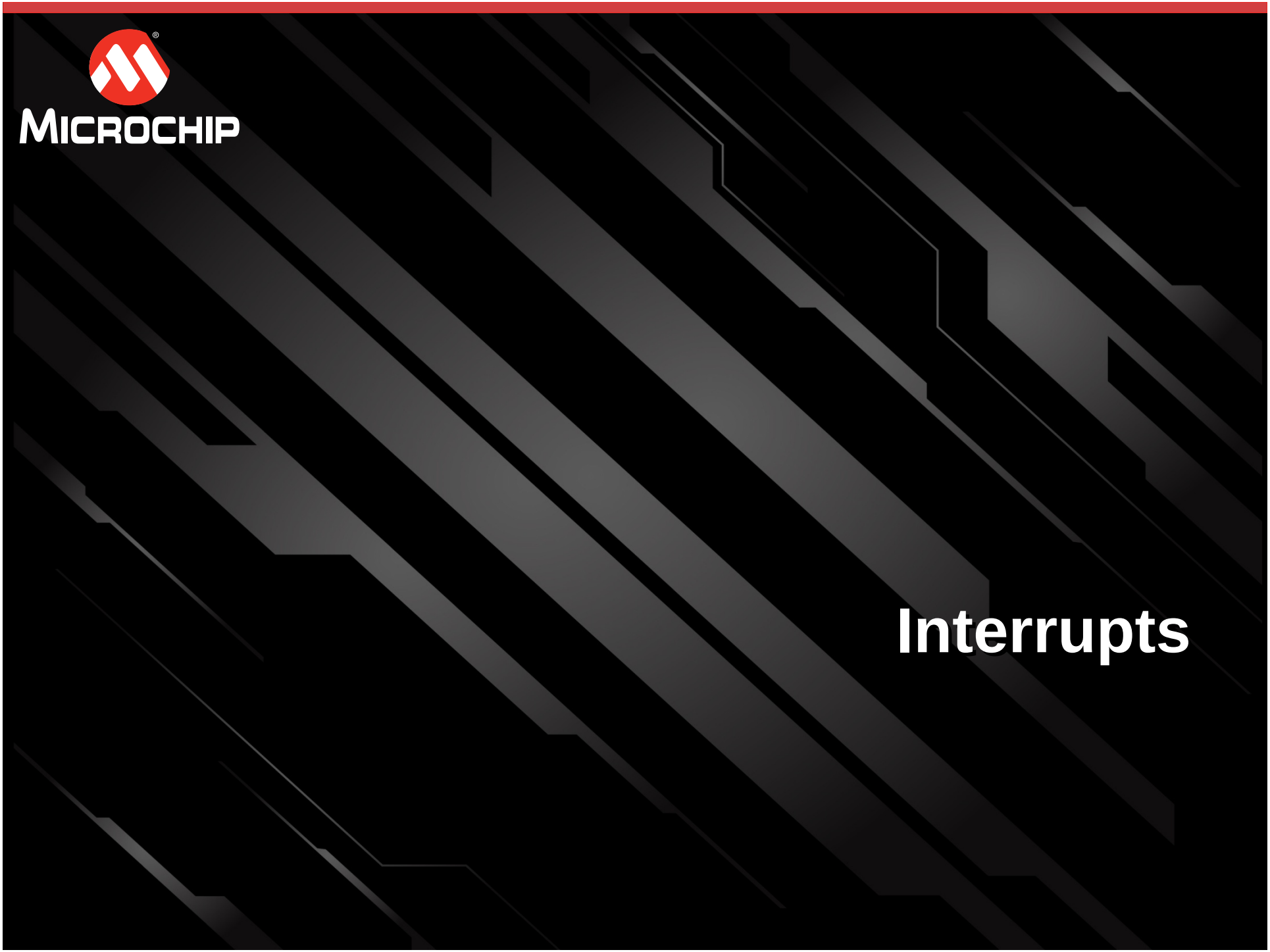
Overriding Default Behavior
Pre-defined Macros – dsPIC® Devices

Memory Allocation Macros (dsPIC®)

```
#define _XBSS(N) __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N) __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N) __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N) __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata), aligned(N)))
#define _PERSISTENT __attribute__((persistent))
#define _NEAR __attribute__((near))
```

- **Same as PIC24 macros but…**
  - **Provides separate macros for X and Y space**
  - **Provides macro for EEPROM Data Memory (dsPIC30 only)**

MPLAB® C30 Compiler User's Guide (Rev E): Section 6.6.3, Page 83


MICROCHIP
Interrupts

Interrupts

Definition

Interrupts are events that cause your program to stop what it is doing in order to run an Interrupt Service Routine which will handle the event by taking whatever action is required before finally returning control to your main program.

- **PIC24 / dsPIC interrupts are vectored**
- **Interrupts require special functions to service the events that cause them:**
  - **ISRs must not have any parameters**
  - **ISRs must not be called by the main code**
  - **ISRs should not call other functions**

Interrupts
How to Declare an Interrupt Service Routine

Interrupt Attribute Basic Syntax

```
void __attribute__((interrupt)) _ISRName(void)
{ …          Function Code Here          … }
```

- No parameters and `void` return type (required)
- Use pre-defined name (required)
- Do NOT call from main line code (required)
- Do not call other functions (*recommended*)

Example Interrupt Service Routine

```
void __attribute__((interrupt)) _INT0Interrupt(void)
{
    //Ordinary C code goes here to handle interrupt
}
```

Interrupts
Interrupt Function Names

- Interrupt Function names may be found in:
  - Device's Linker Script (e.g. p24fj128ga010.gld)
  - MPLAB® C30 User's Guide (Section 7.4)
  - MPLAB® C30 Online Help
- Used by LINK30 to associate interrupt function with the appropriate location in the interrupt vector table
- Linker puts the address of the interrupt function in the appropriate location in the interrupt vector table

Interrupt Functions

Partial List of Interrupt Function Names

IRQ #	Primary Name	Alternate Name
N/A	_ReservedTrap0	_AltReservedTrap0
N/A	_OscillatorFail	_AltOscillatorFail
N/A	_AddressError	_AltAddressError
N/A	_StackError	_AltStackError
N/A	_MathError	_AltMathError
N/A	_ReservedTrap5	_AltReservedTrap5
N/A	_ReservedTrap6	_AltReservedTrap6
N/A	_ReservedTrap7	_AltReservedTrap7
0	_INT0Interrupt	_AltINT0Interrupt
1	_IC1Interrupt	_AltIC1Interrupt
2	_OC1Interrupt	_AltOC1Interrupt
3	_T1Interrupt	_AltT1Interrupt

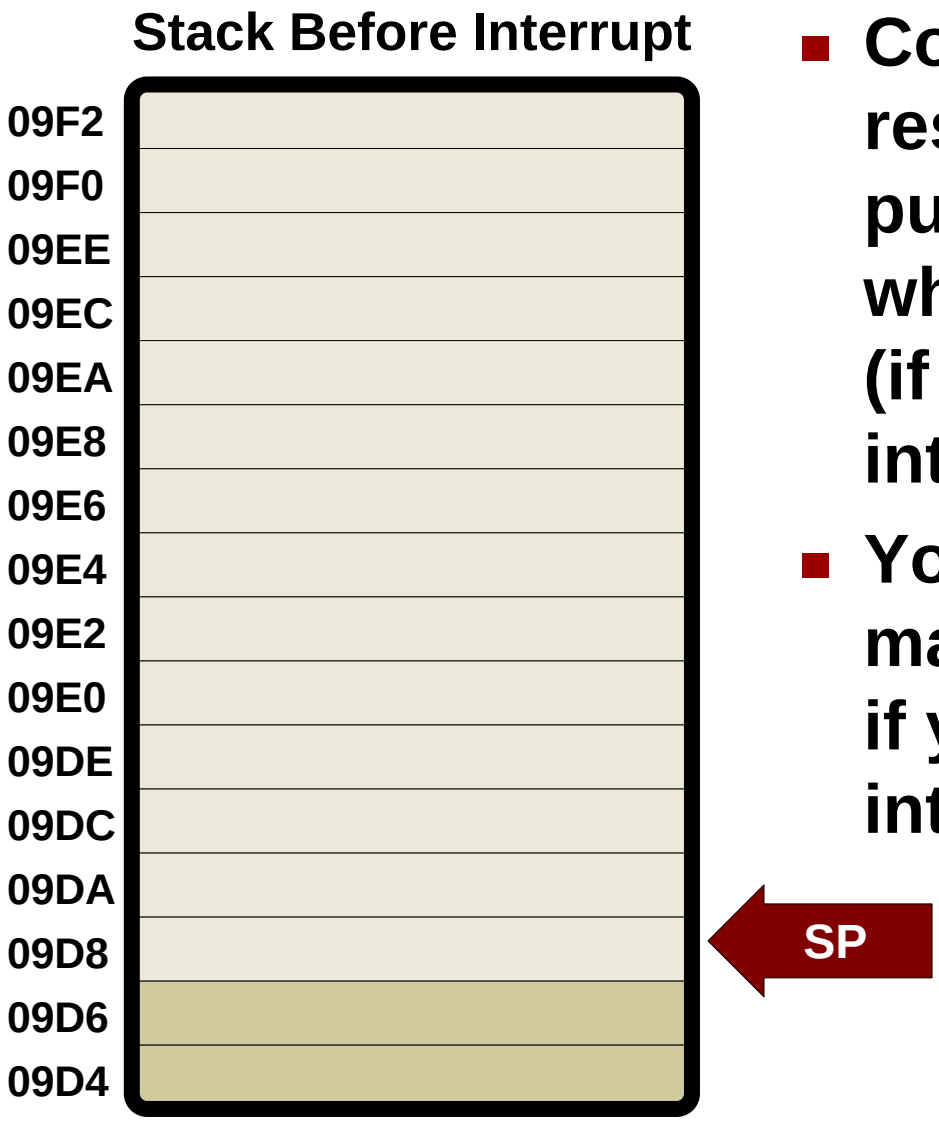
Interrupt Functions

Partial List of Interrupt Function Names

IRQ #	Primary Name	Alternate Name
4	_IC2Interrupt	_AltIC2Interrupt
5	_OC2Interrupt	_Alt OC2Interrupt
6	_T2Interrupt	_AltT2Interrupt
7	_T3Interrupt	_AltT3Interrupt
8	_SPI1Interrupt	_AltSPI1Interrupt
9	_U1RXInterrupt	_AltU1RXInterrupt
10	_U1TXInterrupt	_AltU1TXInterrupt
11	_ADCInterrupt	_AltADCInterrupt
12	_NVMInterrupt	_AltNVMInterrupt
13	_SI2CInterrupt	_AltSI2CInterrupt
14	_MI2CInterrupt	_AltMI2CInterrupt
15	_CNInterrupt	_AltCNInterrupt

Interrupts

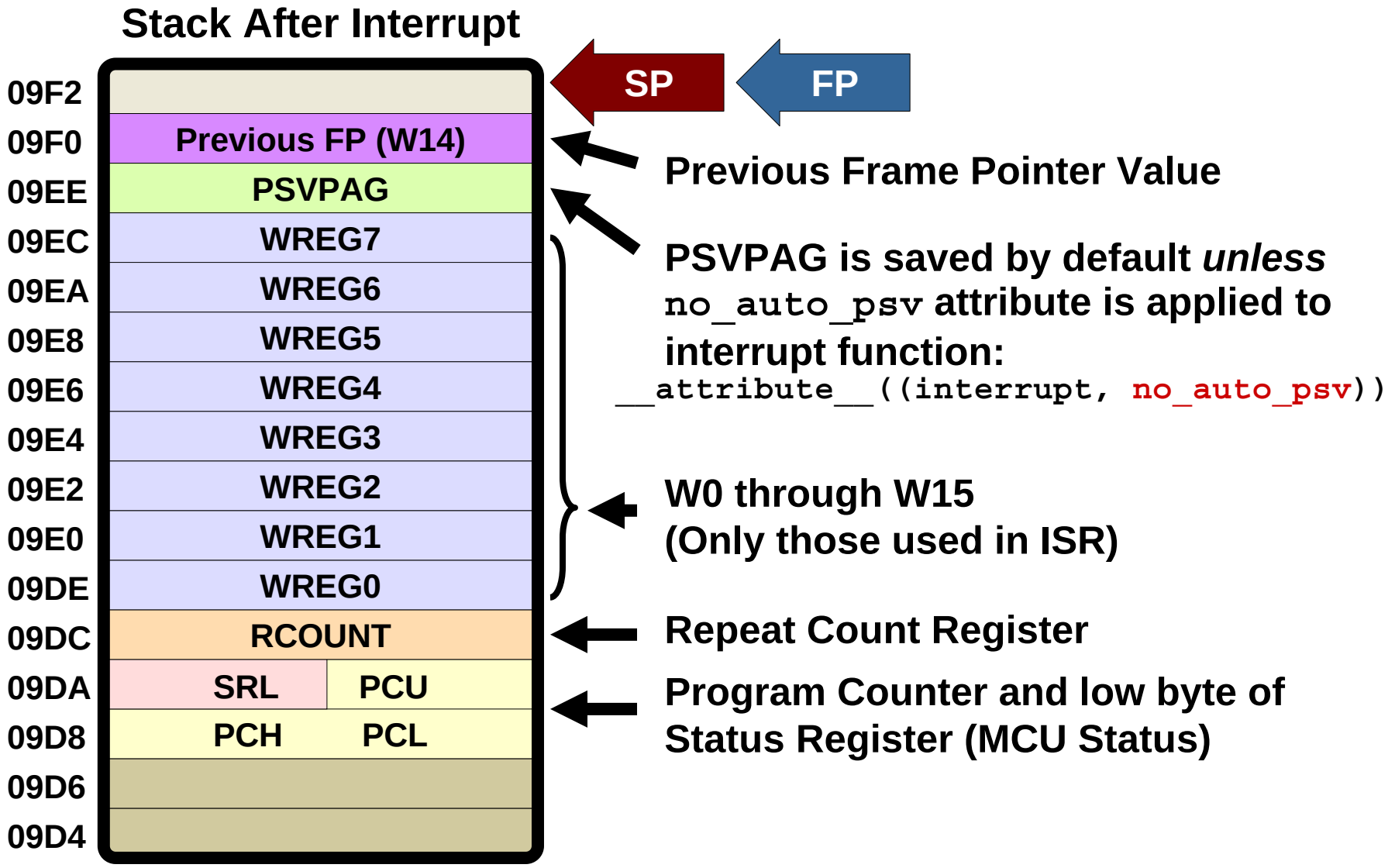
What happens when an interrupt occurs?



- Compiler managed resources are automatically pushed onto the stack when an interrupt occurs (if they are modified by the interrupt)
- You are responsible for managing other resources if you use them in your interrupt service routine

Interrupts

What happens when an interrupt occurs?



Interrupts

What happens when an interrupt occurs?

Example: Disassembly Listing of an ISR – Context Save

```
4:      void __attribute__((interrupt)) _INT0Interrupt(void)
5:      {        Code generated for opening brace '{' of ISR function:
011B2   F80036   push.w 0x0036                    Save RCOUNT
011B4   BE9F80   mov.d  0x0000,[0x001e++]         Save W0, W1
011B6   BE9F82   mov.d  0x0004,[0x001e++]         Save W2, W3
011B8   BE9F84   mov.d  0x0008,[0x001e++]         Save W4, W5
011BA   BE9F86   mov.d  0x000c,[0x001e++]         Save W6, W7
011BC   F80034   push.w 0x0034                    Save PSVPAG
011BE   B3C000   mov.b  #0x0,0x0000               Set  PSVPAG
011C0   8801A0   mov.w  0x0000,0x0034
011C2   FA0000   lnk    #0x0                      Allocate stack frame of 0 bytes
                                                  (Saves W14 onto stack)
        -- Your ISR Code Here --
```

Push to Top of Stack

0x001e is W15 (Stack Pointer)

This code is only present if the `no_auto_psv` attribute is not specified.

Interrupts

What happens when an interrupt occurs?

Example: Disassembly Listing of an ISR – Context Restore

-- Your ISR Code Here --

Code generated for closing brace '}' of ISR function:

```
12:     }
011D0   FA8000   ulnk                          Deallocate stack frame
011D2   F90034   pop.w   0x0034               Restore PSVPAG
011D4   BE034F   mov.d   [--0x001e],0x000c     Restore W6, W7
011D6   BE024F   mov.d   [--0x001e],0x0008     Restore W4, W5
011D8   BE014F   mov.d   [--0x001e],0x0004     Restore W2, W3
011DA   BE004F   mov.d   [--0x001e],0x0000     Restore W0, W1
011DC   F90036   pop.w   0x0036                Restore RCOUNT
011DE   064000   retfie                        Return From Interrupt
```

Pop from Top of Stack

This code is only present if `no_auto_psv` attribute is not specified.

Interrupts

Using Shadow Registers for Context Save/Restore

Syntax

```
void __attribute__((interrupt, shadow)) ISRname(void)
```

- The PUSH.S and POP.S instructions will be used to save and restore using the shadow registers:
  - W0 – W3
  - C, Z, OV, N, and DC bits in the SRH:SRL (STATUS) register

Example

```
void __attribute__((interrupt, shadow))
  _INT0Interrupt(void)
{
    /* ISR Code Here */
}
```

Interrupts

Using Shadow Registers for Context Save/Restore

Example: Disassembly Listing of an ISR Using Shadow

```
4:      void __attribute__((interrupt, shadow)) _INT0Interrupt(void)
5:      {          Code generated for opening brace '{' of ISR function:
011B2   FEA000     push.s                          Save W0 – W3, status bits C, Z, OV, N, DC
011B4   F80036     push.w 0x0036                   Save RCOUNT
011B6   BE9F84     mov.d 0x0008,[0x001e++]         Save W4, W5
011B8   BE9F86     mov.d 0x000c,[0x001e++]         Save W6, W7
011BA   F80034     push.w 0x0034                   Save PSVPAG
011BB   B3C000     mov.b  #0x0,0x0000              Set PSVPAG
011CC   8801A0     mov.w  0x0000,0x0034
011CE   FA0000     lnk    #0x0                     Allocate stack frame of 0 bytes
```

-- Your ISR Code Here --

0x001e is W15 (Stack Pointer)

Interrupts

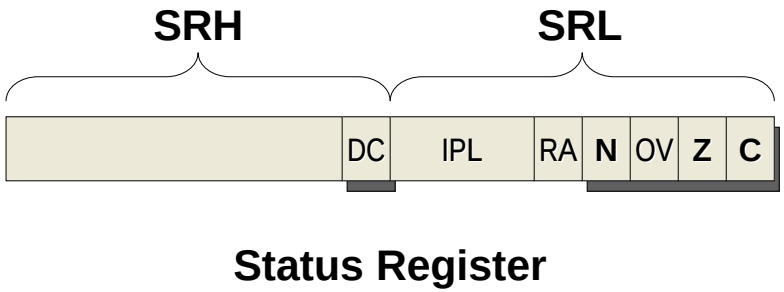
Using Shadow Registers for Context Save/Restore

Example: Disassembly Listing of an ISR Using Shadow

```
        -- Your ISR Code Here --

12:     }           Code generated for closing brace '}' of ISR function:
011CE   FA8000      ulnk                        Deallocate stack frame
011D0   F90034      pop.w 0x0034                Restore PSVPAG
011D0   BE034F      mov.d [--0x001e],0x000c     Restore W6, W7
011D2   BE024F      mov.d [--0x001e],0x0008     Restore W4, W5
011D4   F90036      pop.w 0x0036                Restore RCOUNT
011D6   FE8000      pop.s                       Restore W0 – W3, status bits C, Z, OV, N, DC
011D8   064000      retfie                      Return From Interrupt
```

Interrupts
Using Shadow Registers for Context Save/Restore

- Shadow registers must be used with care
- Shadow registers are only one level deep
- Contents can be lost if two interrupts use shadow registers and one is higher priority than the other



Interrupts
How to Save Variables Not Managed by the Compiler

Syntax

```
void __attribute__((interrupt(save(list))) _ISRname(void)
```

- **Any variables listed in the save list will be pushed onto the stack along with the compiler managed resources**

Example

```
void __attribute__((interrupt(save(x, y))))
  _INT0Interrupt(void)
{
    /* ISR Code Here */
}
```

Interrupts
Macros for Simplified ISR Syntax

Built-in ISR Macro Definitions

```
#define _ISR __attribute__((interrupt))

#define _ISRFAST __attribute__((interrupt, shadow))
```

- **The macros may be used when the interrupt being defined doesn't require any special attributes beyond `interrupt` and `shadow`**

Example

```
void _ISR _INT0Interrupt(void)
{
    /* ISR Code Here */
}
```

Interrupts
Firmware Engineer's Responsibilities

- You must clear the interrupt flag manually
  - Example: IFS*n*bits.*FlagName* = 0;
- You must save any registers/variables you access in the ISR's code if they are not handled automatically by the compiler
  - Add a save list to the interrupt attribute
  - Save them manually in your ISR code
- Variables modified by an interrupt should be tagged with the `volatile` keyword

Lab Exercise 4

Interrupts

Lab Exercise 4
Interrupts

Objective

Using the Timer 1 interrupt, blink LED D3 (RA0) at a rate determined by the internal RC oscillator's default frequency and Timer 1's period with a 1:8 prescaler and Fosc/2 as its clock source.

- **Timer 1 has been configured for you to interrupt approximately twice per second**
- **You need to write the interrupt service routine to toggle the LED and clear the interrupt flag**

Lab Exercise 4
Interrupts

Procedure

Follow the directions in the lab manual starting on page 4-1.

On the lab PC…

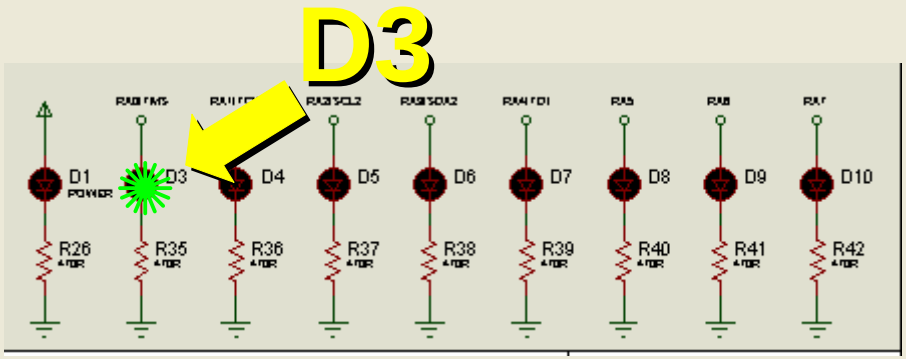
If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

Open the lab workspace by selecting from the menu:
File ▶ Open Workspace…
and select the file:
C:\MTT\TLS2130\Lab4\Lab4.mcw

Lab Exercise 4
Interrupts

Results

```
void __attribute__((interrupt)) _T1Interrupt(void)
{
    LATAbits.LATA0 ^= 1;   //Toggle LED
    IFS0bits.T1IF = 0;     //Clear interrupt flag
}
```



Lab Exercise 4
Interrupts

Results

Timing Calculations

F_{OSC} = F_{RC} / CLKDIV = 8 MHz / 2 = 4 MHz
$T1_{CLK}$ = Fosc / 2 = 4 MHz / 2 = 2 MHz
$T1_{TICK}$ = (1 / T1CLK) * PRESCALE = (1 / 2 MHz) * 8 = 4 μs
$T1_{Period}$ = $T1_{TICK}$ * 2^{16} = 4 μs * 65536 = 0.26s

Blink Period = 2 * $T1_{Period}$ = 2 * 0.26 s = **0.52 s**

Lab Exercise 4
Interrupts

Conclusions

- Interrupt functions need to be tagged with the interrupt attribute: `__attribute__((interrupt))`
- Interrupts should use the names Microchip provides in the linker script:

  C:\Program Files\Microchip\MPLAB C30\Support\PIC24F\p24FJ128GA010.gld

- You must clear the interrupt flag

Working with Libraries

Working With Libraries

- **MPLAB® C30 includes several libraries:**
  - **libc.a = Standard C Library**
  - **libm.a = Math Library**
  - **libdsp.a = DSP Library**
  - **libq.a = Fixed Point Math Library**
  - **libq-dsp.a = FP Math Library (DSP Engine)**
  - **lib*DEVICENUM*.a = Peripheral Libraries**

libpPIC24Fxxx.a

Working With Libraries

- To use the peripheral libraries:
  - #include the appropriate header file (timer.h)
  - Call the functions as documented in the library user's guide
- Only the object files you use from a library will be compiled into the final hex file
- Libraries shipped with MPLAB C will be automatically linked into your project when they are used

Working With Libraries

- **Library Header Files**

adc.h	A/D Converter
comparator.h	Analog Comparator
crc.h	Cyclic Redundancy Check
Generic.h	Generally useful stuff
i2c.h	I2C Interface
incap.h	Input Capture
outcompare.h	Output Compare
PIC24_periph_features.h	Peripheral Pin Select
pmp.h	Parallel Master Port
ports.h	I/O Ports
PwrMgnt.h	Power Management
rtcc.h	Real-Time Clock Calendar
spi.h	SPI Interface
timer.h	Timers
uart.h	UART
wdt.h	Watchdog Timer

OpenTimer1()
Example Library Function

Function Prototype

```
void OpenTimer1(unsigned int config, unsigned int period);
```

config **Parameter (OR these values together for desired configuration)**

Timer Module On/Off
- T1_ON
- T1_OFF

Timer Module Idle mode On/Off
- T1_IDLE_CON
- T1_IDLE_STOP

Timer Gate time accumulation enable
- T1_GATE_ON
- T1_GATE_OFF

Timer prescaler
- T1_PS_1_1
- T1_PS_1_8
- T1_PS_1_64
- T1_PS_1_128

Timer Synchronous clock enable
- T1_SYNC_EXT_ON
- T1_SYNC_EXT_OFF

Timer clock source
- T1_SOURCE_EXT
- T1_SOURCE_INT

Example:

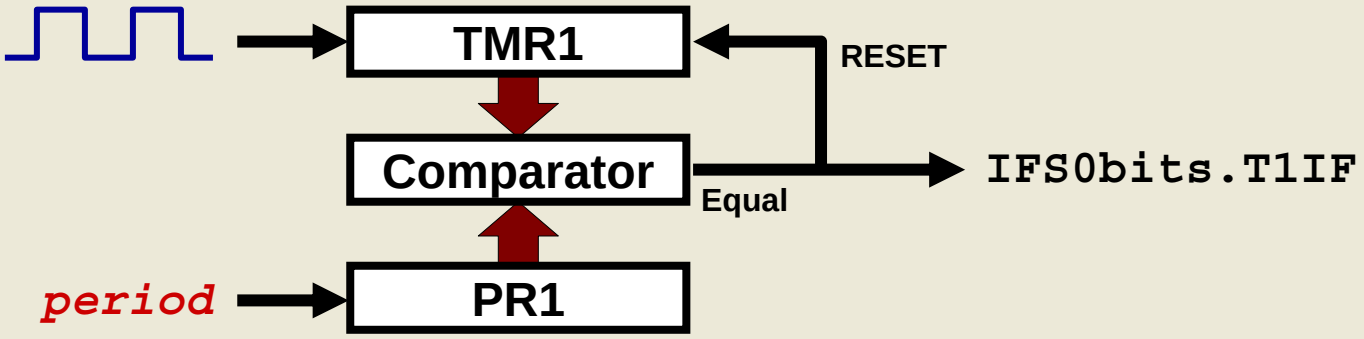
```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_GATE_OFF & T1_PS_1_8 &
           T1_SYNC_EXT_OFF & T1_SOURCE_INT, 0xFFFF);
```

OpenTimer1()
Example Library Function

Function Prototype

```
void OpenTimer1(unsigned int config, unsigned int period);
```

period Parameter



Example:

```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_GATE_OFF & T1_PS_1_8 &
           T1_SYNC_EXT_OFF & T1_SOURCE_INT, 0xFFFF);
```

ConfigIntTimer1()

Example Library Function

Function Prototype

```
void ConfigIntTimer1(unsigned int config);
```

config Parameter (OR these values together for desired configuration)

Interrupt Enable	Interrupt Priority
T1_INT_ON	T1_INT_PRIOR_7
T1_INT_OFF	T1_INT_PRIOR_6
	T1_INT_PRIOR_5
	T1_INT_PRIOR_4
	T1_INT_PRIOR_3
	T1_INT_PRIOR_2
	T1_INT_PRIOR_1
	T1_INT_PRIOR_0

Example:

```
ConfigIntTimer1(T1_INT_ON & T1_INT_PRIOR_7);
```

Lab Exercise 5

Working with Peripheral Libraries

Lab Exercise 5
Working with Peripheral Libraries

Objective

Using the Timer 1 interrupt, blink LED D3 (RA0) at a rate determined by the internal RC oscillator's default frequency and Timer 1's period with a 1:8 prescaler and Fosc/2 as its clock source.

- **The objective is the same as lab 4…**
  - **This time, the Timer 1 interrupt is already written for you**
  - **You need to initialize Timer 1 using the PIC24F peripheral libraries using the parameters given in the lab manual**

Lab Exercise 5

Working with Peripheral Libraries

Procedure

Follow the directions in the lab manual starting on page 5-1.

On the lab PC…

If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

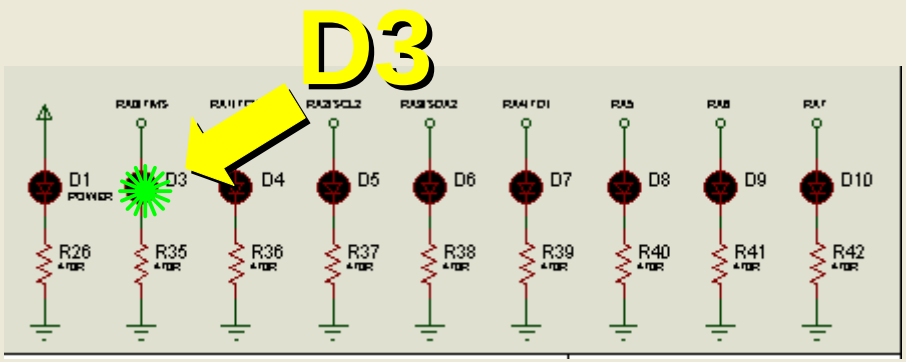
Open the lab workspace by selecting from the menu: File ▶ Open Workspace…
and select the file:

C:\MTT\TLS2130\Lab5\Lab5.mcw

Lab Exercise 5
Working with Peripheral Libraries

Results

```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_GATE_OFF &
           T1_PS_1_8 & T1_SYNC_EXT_OFF &
           T1_SOURCE_INT, 0xFFFF);

ConfigIntTimer1(T1_INT_PRIOR_7 & T1_INT_ON);
```



Lab Exercise 5
Working with Peripheral Libraries

Conclusions

- **Peripheral libraries can simplify the task of configuring on chip peripherals**
- **To use the peripheral libraries, you must**
  - **Include the appropriate header file in your source**
  - **And together choices from all options when you call the functions – do not rely on defaults**

Lab Exercise 6

Creating Custom Libraries

Lab Exercise 6
Creating Custom Libraries

Objective

Build a library (archive) file from a simple C source file that contains a single function. Then, use that function in a separate project that includes the newly created library file in its project tree.

- **Follow along with the instructor to create a custom library and a project to test it**
- **The library will consist of two source files, each containing a single function**
- **The code is already written for you**

Lab Exercise 6
Creating Custom Libraries

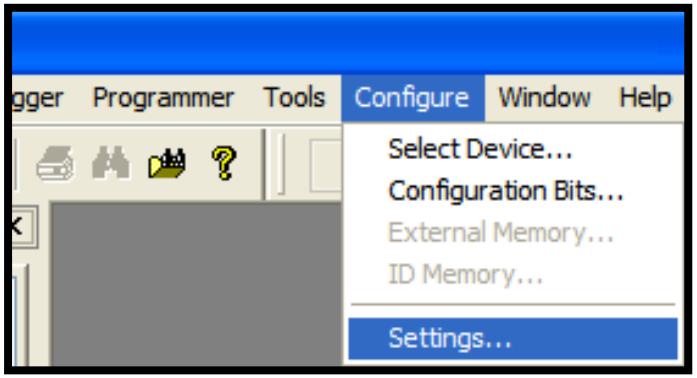
Procedure

Follow the directions in the lab manual starting on page 6-1, or follow along with the instructor and the slides as we walk through this project together.

If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

1 Open the MPLAB Settings

From the menu bar, select:
Configure ▶ Settings...

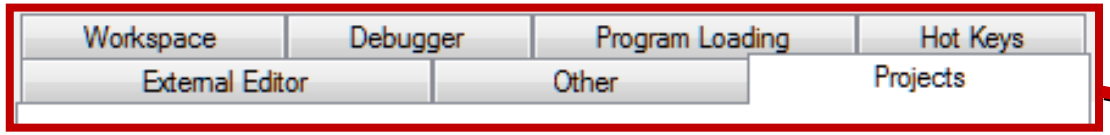


Lab Exercise 6
Creating Custom Libraries

Configure MPLAB so that multiple projects may be opened within one workspace

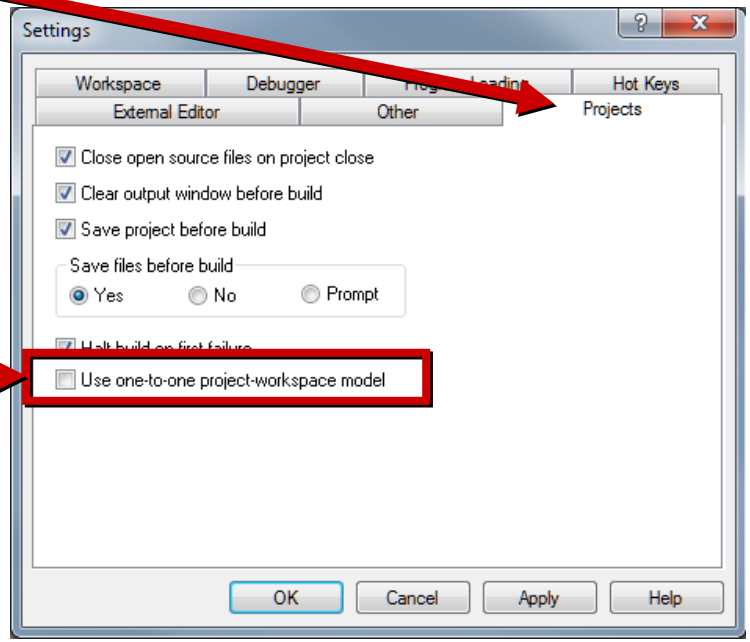
2 Select the Project Settings

On the tabs, select: Projects



3 Deselect one-to-one project-workspace model

Uncheck the last check box:



Click OK **when done**

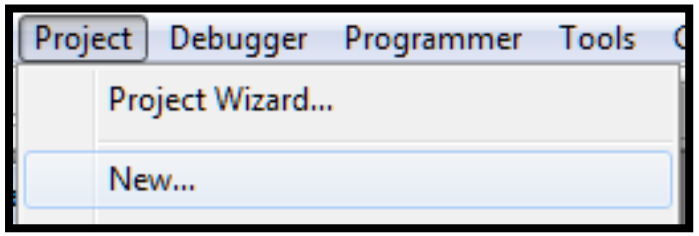
Lab Exercise 6
Creating Custom Libraries

Create the Library Source Project

4 Create New Project

From the menu bar, select:

Project ▸ New...



5 Name the Project

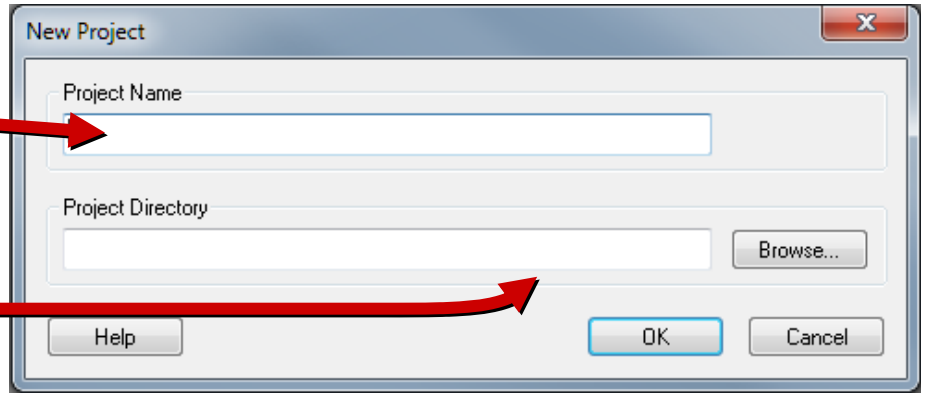
In the Project Name box, enter:

MyLib

6 Select Project Directory

Click on Browse... and select:

C:\MTT\TLS2130\Lab6



Click OK when done

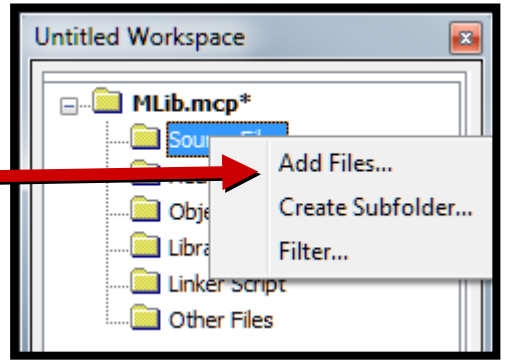
The name of this project will be the name of the library file (e.g. MyLib.a)

Lab Exercise 6
Creating Custom Libraries

Add the library source code to the project

7 Add Source File to Project

In the project tree, right click on Source Files and select: Add Files…



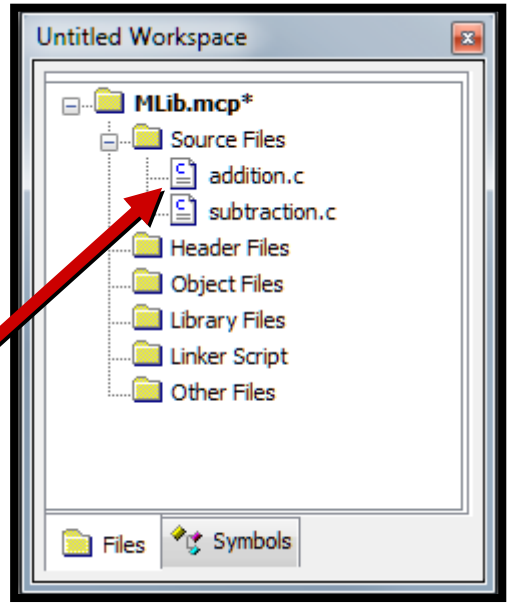
8 Select Library Source Files

In the add file dialog box, select the library source files from the lab 6 directory (the library code has already been written for you):

C:\MTT\TLS2130\Lab6\addition.c

C:\MTT\TLS2130\Lab6\subtraction.c

addition.c and subtraction.c should appear in the project tree under Source Files.



Lab Exercise 6

Creating Custom Libraries

A look at the library source code and its associated header file

Library source code contains functions written in the usual way. Nothing special needs to be done just because this will be included in a library.

The header file contains function prototypes for all the functions in the library file. This will be required to use the library in other projects.

addition.c

```
int add(int a, int b)
{
        return (a + b);
}
```

subtraction.c

```
int sub(int a, int b)
{
        return (a - b);
}
```

MyLib.h

```
int add(int a, int b);
int sub(int a, int b);
```

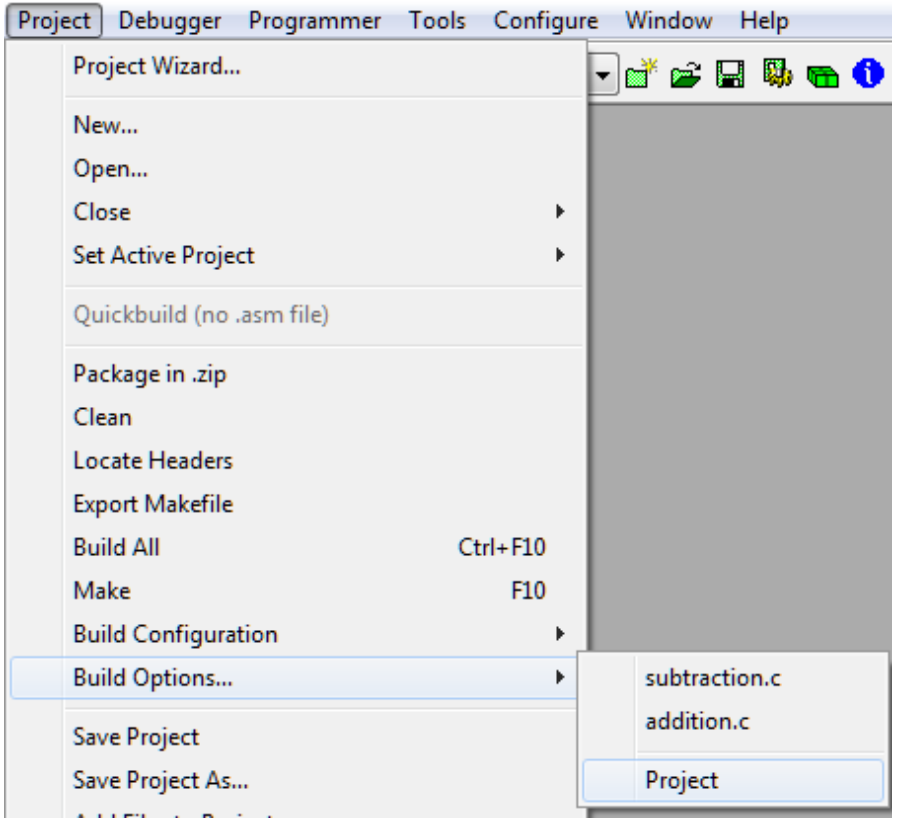
Lab Exercise 6
Creating Custom Libraries

Configure the project build options

9 Open the Project Build Options Dialog Box

From the menu bar, select:

Project ▸ Build Options... ▸ Project



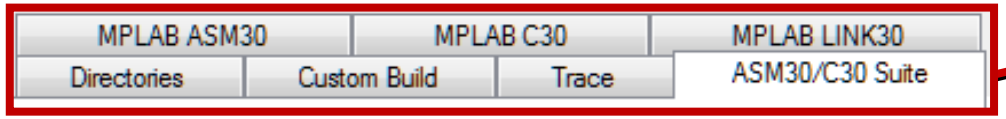
Lab Exercise 6
Creating Custom Libraries

Configure the project build options

10 Setup Tool Suite Options

From the tabs, select:

ASM30/C30 Suite



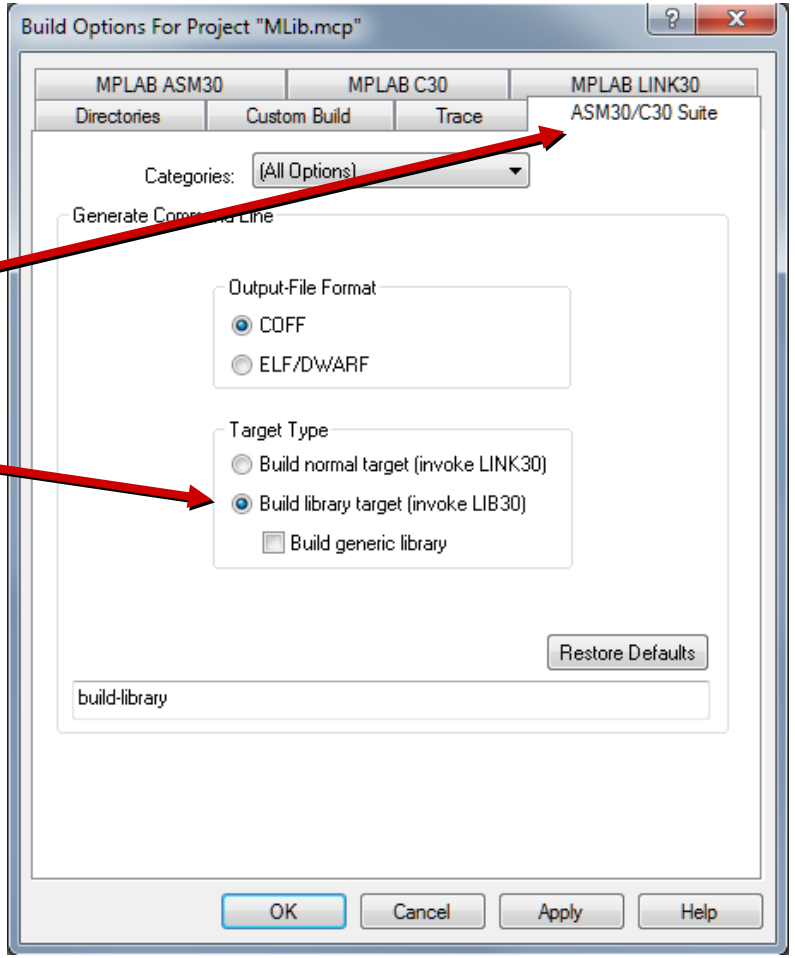
11 Set the Target Type

In the Target Type box, select:

Build library target (invoke LIB30)

and check:

Build generic library

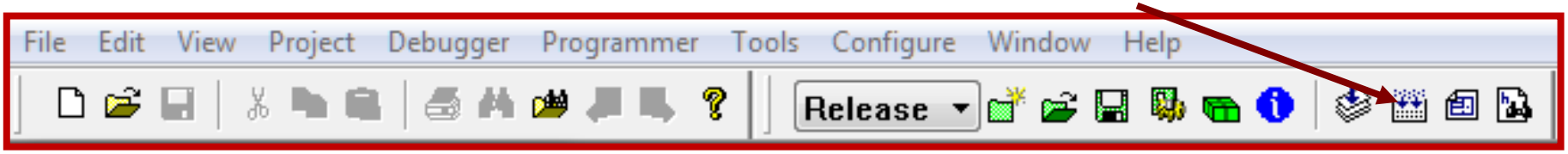


Click OK **when done**

Lab Exercise 6
Creating Custom Libraries

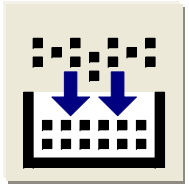
Build the library

12 Compile (Build All)



12 Build the Libarary

From the tool bar, click on the Build All button.



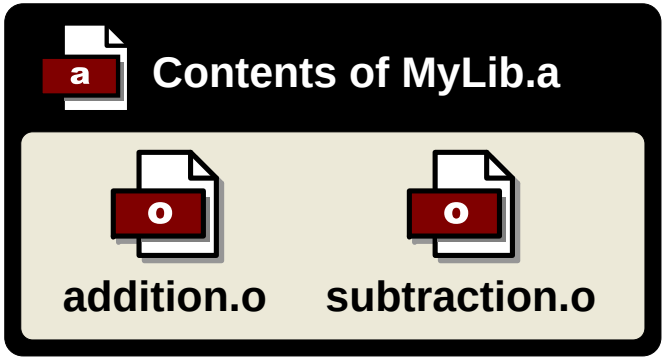
This step will generate a library file called MyLib.a that may now be used in any project for a 16-bit PIC®

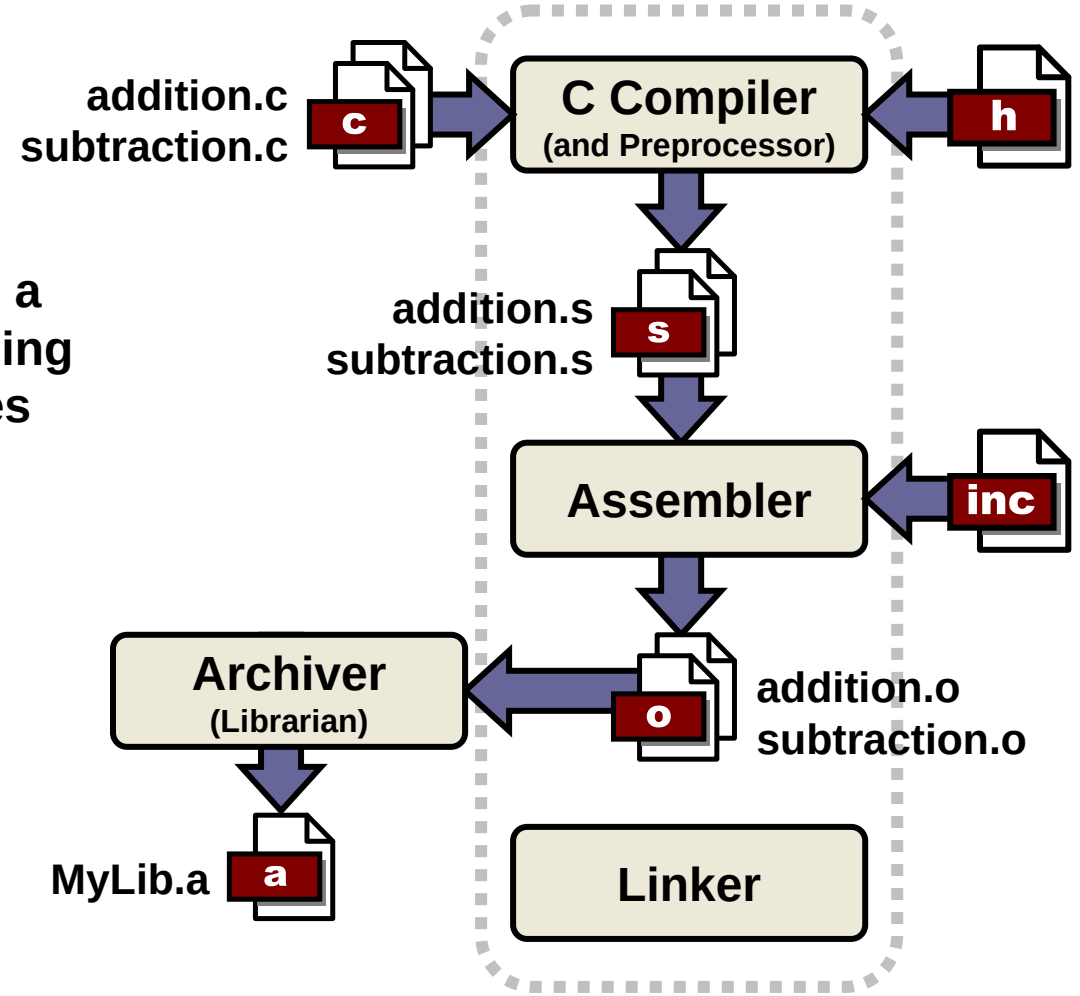
When building a library, no linker script is required because the code being generated will not be placed in a device's memory map until it is part of a normal project.

Lab Exercise 6
Creating Custom Libraries

Library Build Process

Including an archive (*.a) into a project, is the same as including all of the individual object files (*.o) that it contains...





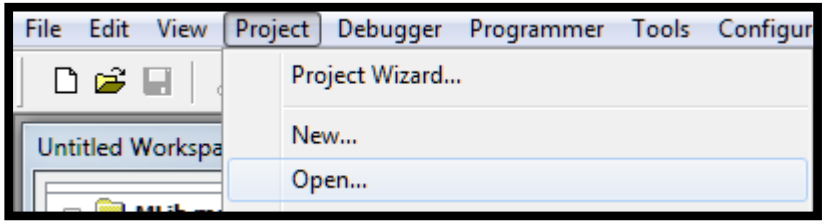
Lab Exercise 6
Creating Custom Libraries

Open (or create) a test project

13 Open Test Project

From the menu bar, select:

Project ▸ Open...

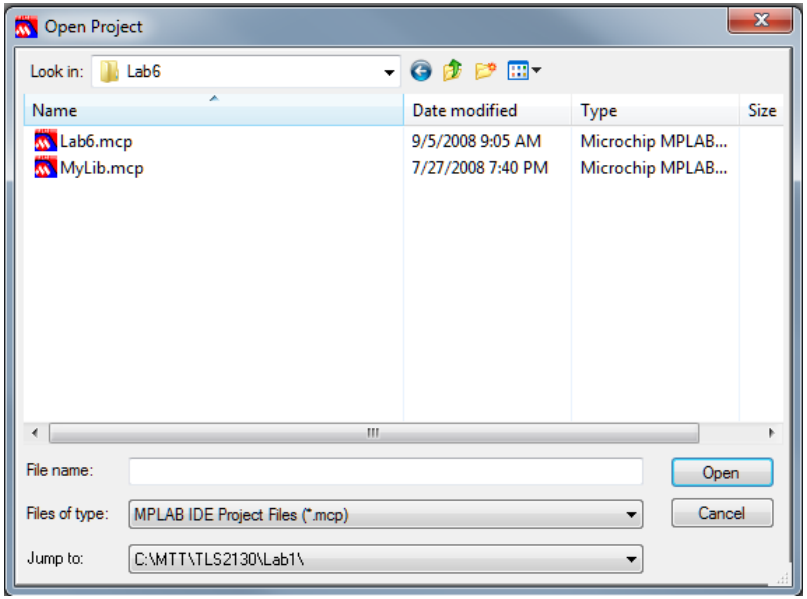


If doing this on your own, at this point you may create a new C30 based project following the steps from Lab 1. Include the library file just created into the new project.

14 Select the Lab6 Project

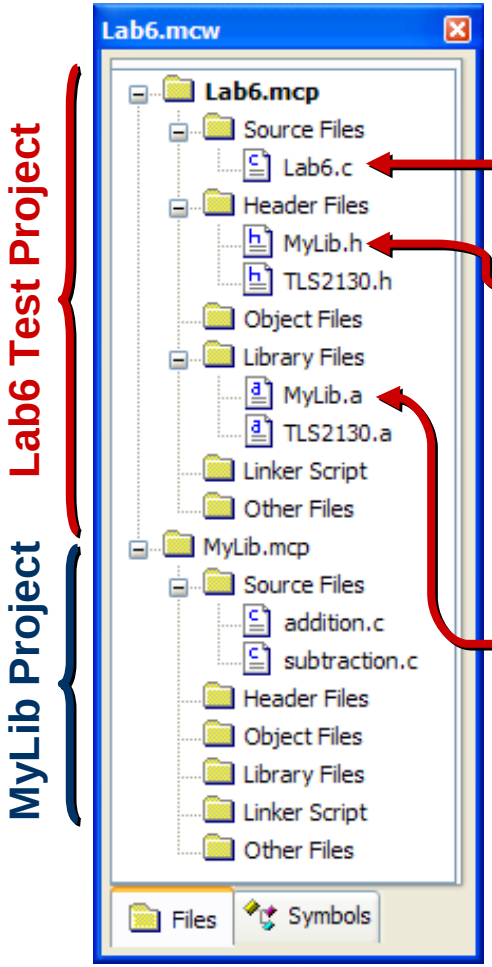
From the open project dialog box select:

C:\MTT\TLS2130\Lab6\Lab6.mcp



Lab Exercise 6
Creating Custom Libraries

A look at the Lab6 test project…



Lab6.c

MyLib.h

Created by you, to provide access to functions in library

MyLib.a

Generated in Step 12

MyLib.c

```
#include "TLS2130.h"
#include "MyLib.h"
int sum, difference;

int main(void)
{
    lcdInit();
    sum = add(5, 2);
    difference = sub(7, 2);

    lcdPutInt(c, DEC);
```

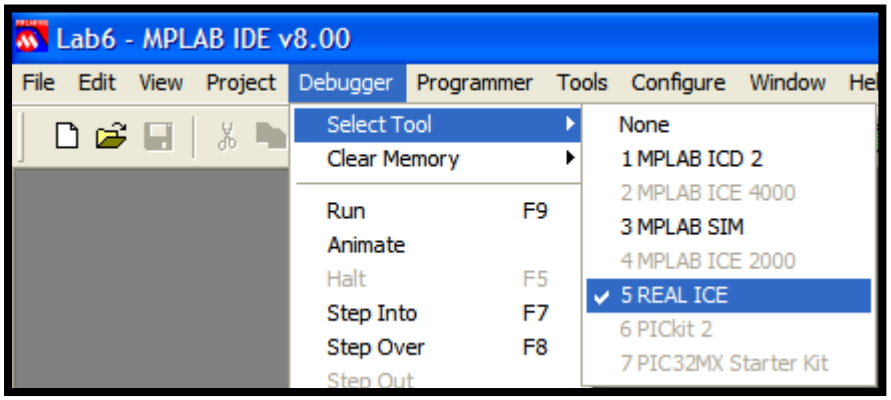
Lab Exercise 6
Creating Custom Libraries

Enable your debug tool

15 Enable Proteus™

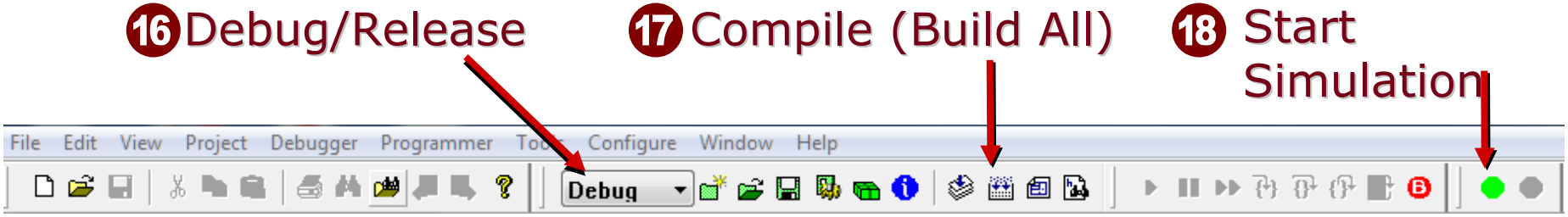
If not already enabled, from the tool bar select:

Debugger ▸ Select Tool ▸ Proteus VSM

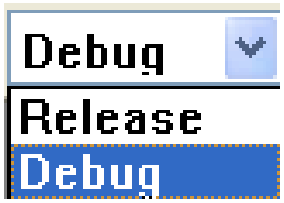


Lab Exercise 6
Creating Custom Libraries

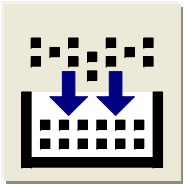
Build and run the program



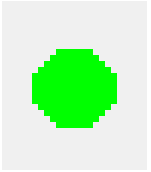
16 **Select Debug mode.**



17 **Click on the Build All button.**



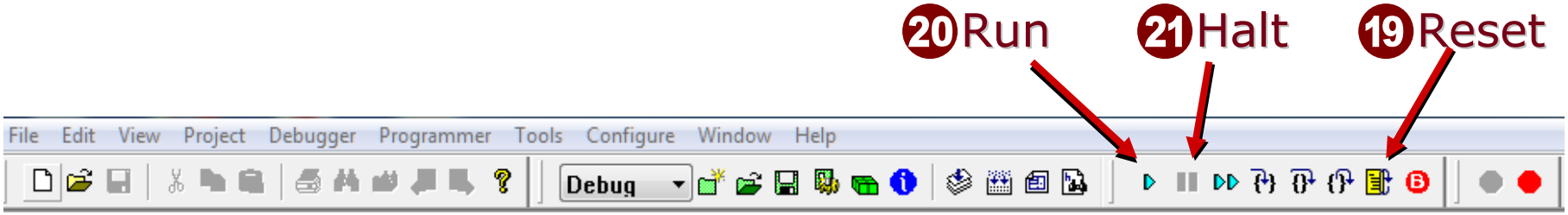
18 **If no errors are reported,**
Click on the Start Simulation button.



Lab Exercise 6

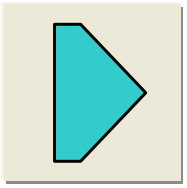
Creating Custom Libraries

Build and run the program

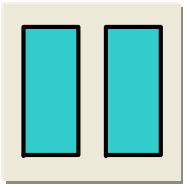


19 **When programming completes, Click on the Reset button.**

20 **Click on the Run button.**



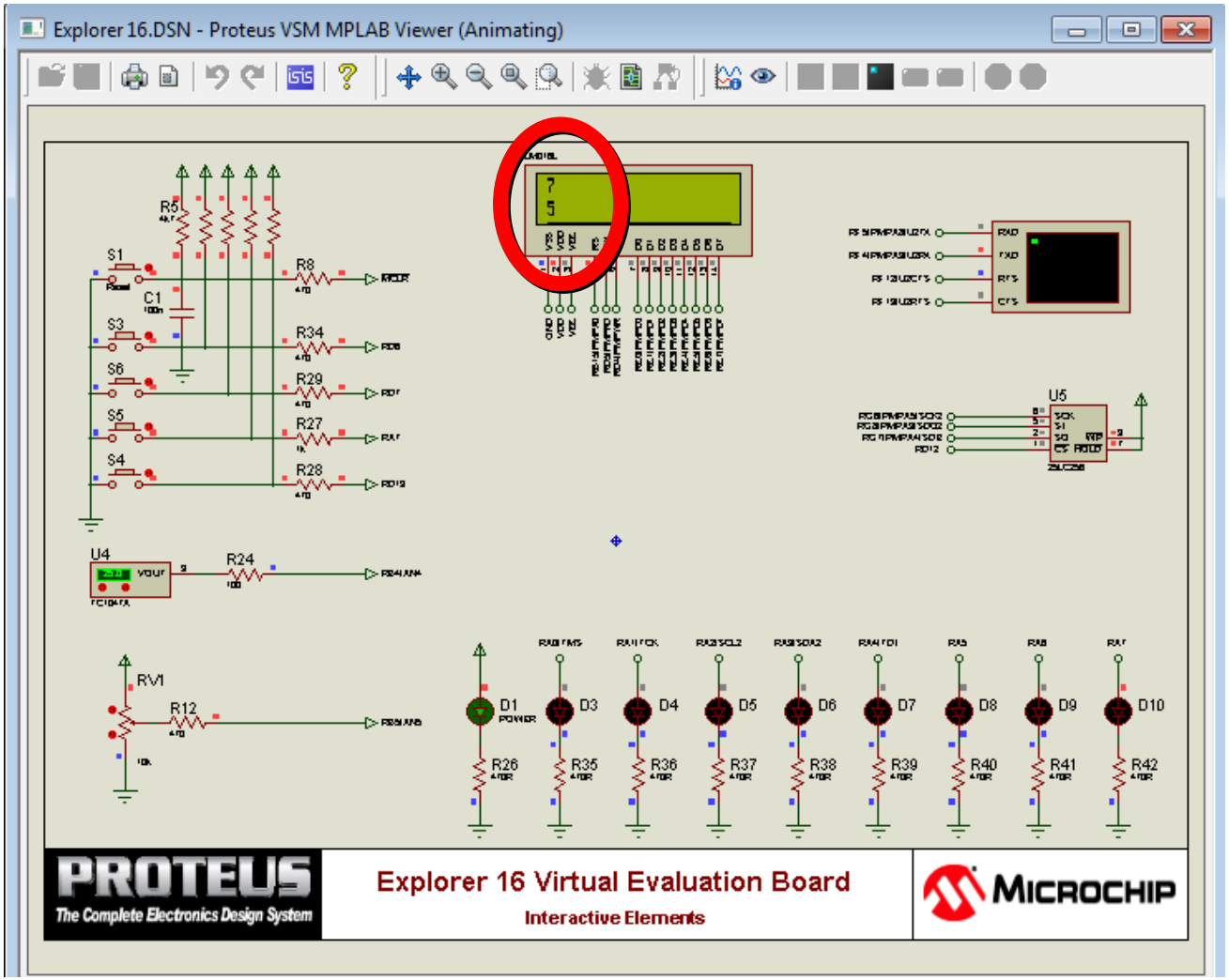
21 **Click on the Halt button.**



Lab Exercise 6

Creating Custom Libraries

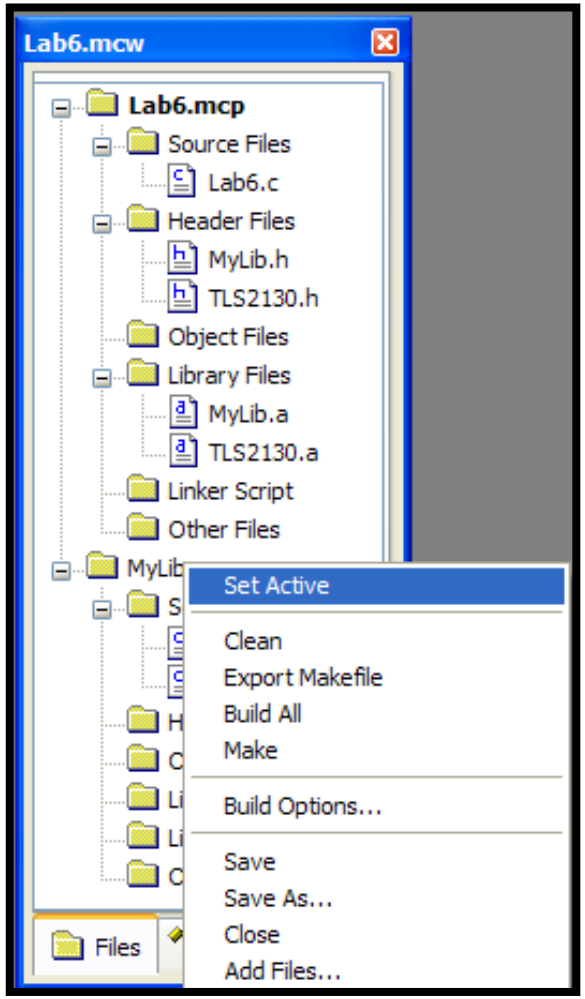
Results



Lab Exercise 6

Creating Custom Libraries

OPTIONAL – Continue Library Development



1. **Make Library Project Active**
   Right click on the MyLib project and select:
   Set Active
2. **Add to or modify library source code (and its associated header)**
3. **Rebuild Library as in Step 12**
4. **Make Test Project Active**
   Right click on the Lab6 project and select:
   Set Active
5. **Rebuild and Run Lab6 as in Step 17 through 21**

Mixing C and Assembly

Inline Assembly Solutions

Inline Assembly
Simple Form –Single Line

Syntax

```
asm ("instruction")
```

- **Only a single string can be passed**
- ***Generally* used for instructions that take no operands or take immediate operands**
- **For ANSI compliance use `__asm__` instead of `asm`**

Examples

```
asm ("nop");              // One Cycle Delay
asm ("clrwdt");           // Clear Watchdog Timer
asm ("pwrsav #0");        // Sleep mode
asm ("pwrsav #1");        // Idle mode
```

Inline Assembly
Simple Form – Multiple Lines

Syntax

```
asm ("instruction_1"
     "instruction_2"
     …
     "instruction_n")
```

- Only one "asm" keyword is required
- Include each instruction within double quotes
- Put each instruction on a separate line for better readability
- One set of parentheses encloses the entire list of instructions within the asm statement

Inline Assembly
Extended Form

Syntax

```
asm ( "template"
    [:["constraint"(output_operand) [, … ] ]
     :["constraint"(input_operand)  [, … ] ]
     :["clobbers" [, … ] ] ]
     );
```

- **Works with optimizer better**
- **More specific about which resources are used**
- **Simplifies interaction between C and assembly**

Example

```
asm ("mov %0, w0" : : "g" (myVar) : "W0");
```

Inline Assembly
Passing C Variables

Example

```
int x = 5, y = 2;
int foo(void)
{
    int result;
    asm("add %1, %2, %0"
        : "=r" (result)
        : "r" (x), "r" (y)
        );

    return result;
}
```

Before Operation:

result: 0000

After Operation:

result: 0007

Inline Assembly
Constraint Letters

Constraint Letters Supported by MPLAB® C30

a	Claims WREG
b	Divide support register W1
c	Multiply support register W2
d	General purpose data registers W1-W14
e	Non-divide support registers W2-W14
g	Any register, memory or immediate integer operand is allowed (except non-general registers)
i	Any immediate integer operand (constant value) is allowed. Includes symbolic constants.
r	A register operand is allowed provided that it is in a general register.
v	AWB register W13
w	Accumulator register A - B
x	x pre-fetch registers W8-W9
y	y pre-fetch registers W10-W11
z	MAC pre-fetch registers W4-W7
0,1,…9	An operand that matches the specified operand number is allowed.
T	A near or far data operand.
U	A near data operand.

Inline Assembly Macros

Inline Assembly Macro Definitions

```
#define Nop()     {__asm__ volatile ("nop");}
#define ClrWdt()  {__asm__ volatile ("clrwdt");}
#define Sleep()   {__asm__ volatile ("pwrsav #0");}
#define Idle()    {__asm__ volatile ("pwrsav #1");}
```

- **Provide an easy way to execute specific assembly language instructions that have no C equivalent (builtins are even better…)**

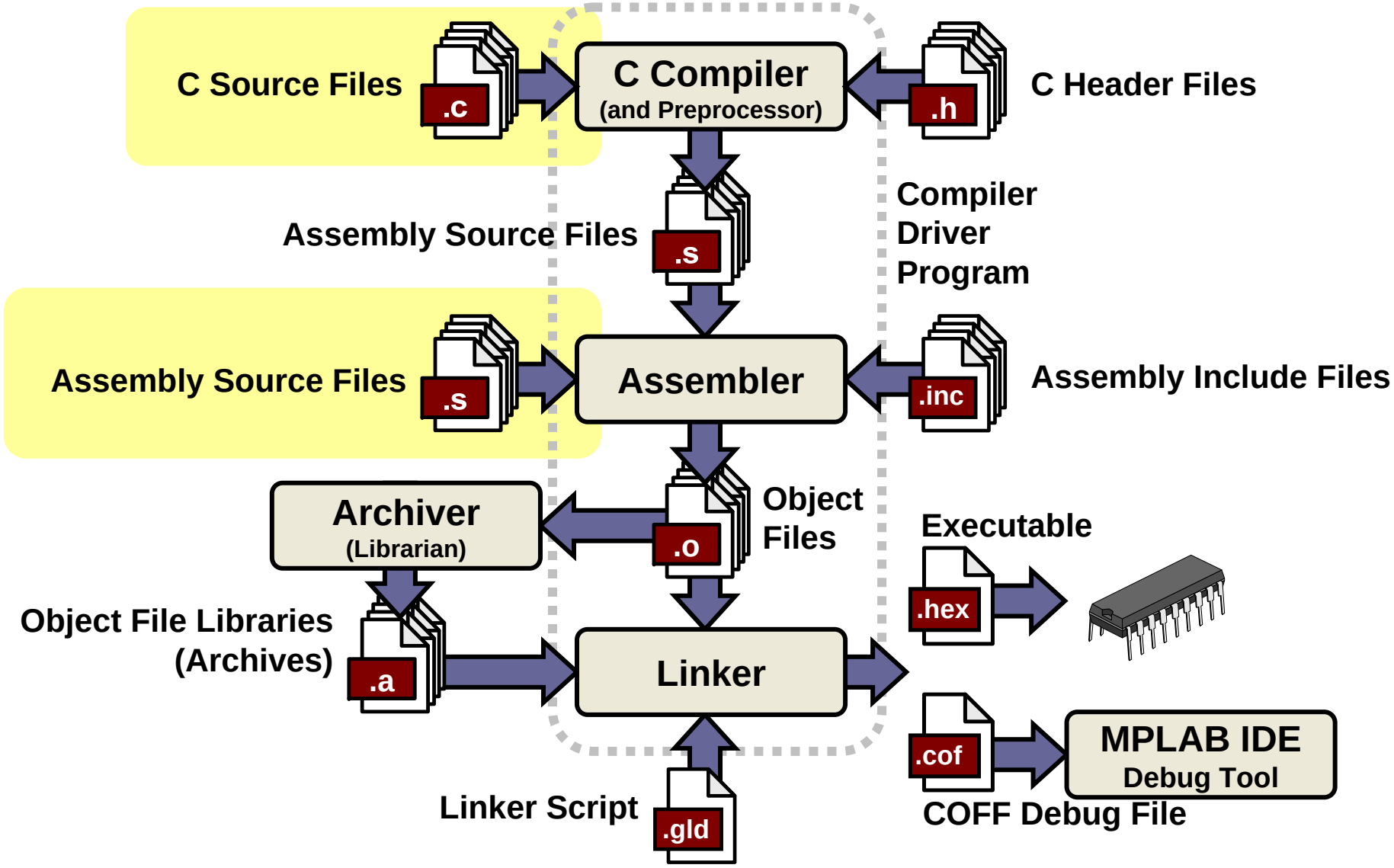
Examples

```
Nop();          // Insert nop instruction
ClrWdt();       // Clear the watchdog timer
Sleep();        // Enter SLEEP mode
Idle();         // Enter IDLE mode
```

Mixing C and Assembly

Multi-File Project Solutions

Development Tools Data Flow



Using a C Variable in Assembly

In the C file…

- **Declare a global or static variable in the usual way (This won't work with non-static variables)**

C_File.c

```
#include "p24fj128ga010.h"

unsigned int x = 0;
unsigned int foo();

int main(void)
{
        foo();
        while(1);
}
```

A global variable has a permanent address in RAM and may be accessed as a register (or sequential series of registers) in assembly language.

Using a C Variable in Assembly

In the Assembly file…

- Declare the variable name as `.extern`
- Add an underscore in front of the variable name

ASM_File.s

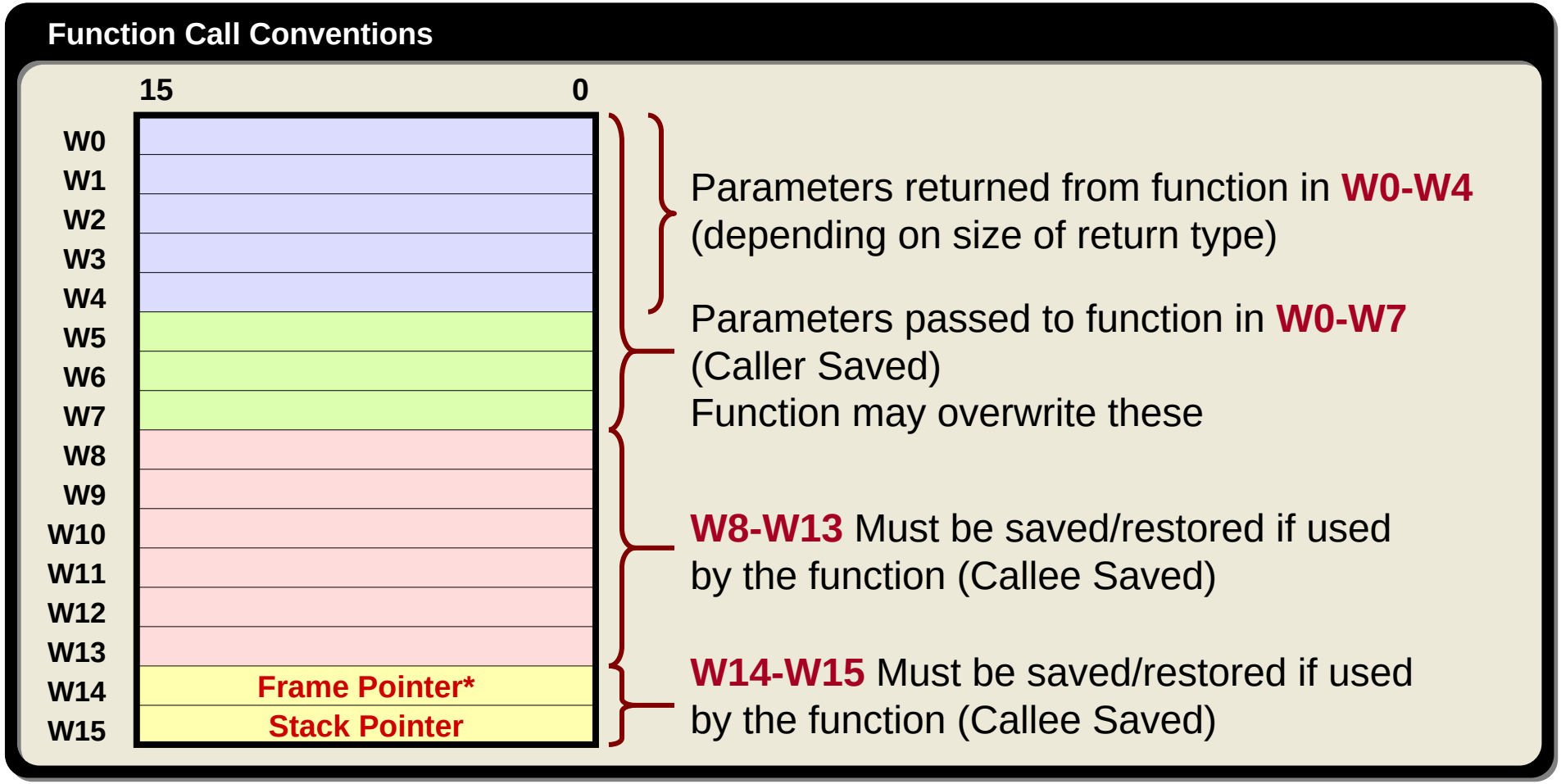
```
        .include "p24fj128ga010.inc"
        .global _foo
        .extern _x

_foo:
        inc _x
        return

        .end
```

Performs same function as `extern` keyword in C. Note that the C variable name must be preceded by an underscore here in the assembly file.

Any identifier that will be used in both C and assembly must have an underscore in front of it in the assembly code but not in the C code.

Calling an Assembly Function
Function Call Conventions

- **For non-interrupt functions**



Calling an Assembly Function
In the C File…

- Include function prototype for assembly function
- Function name is same as ASM subroutine name
- Function is called like an ordinary C function

C_File.c

```
#include "p24fj128ga010.h"

unsigned int x = 0;
extern unsigned int AddNumbers(unsigned int a,
                               unsigned int b);
int main(void)
{
        x = AddNumbers(5, 3);
        while(1);
}
```

Calling an Assembly Function
In the Assembly File…

- Declare the subroutine name as *.global*
- Add an underscore '_' to the front of the name
- Parameters *usually* passed in W registers…

ASM_File.s

```
        .include "p24fj128ga010.inc"
        .global _AddNumbers

_AddNumbers:
        add W0, W1, W0
        return

        .end
```

Calling an Assembly Function
Parameter Passing

- Parameters are processed from left to right
- Parameters are passed in the first available W register with the proper alignment

C_File.c

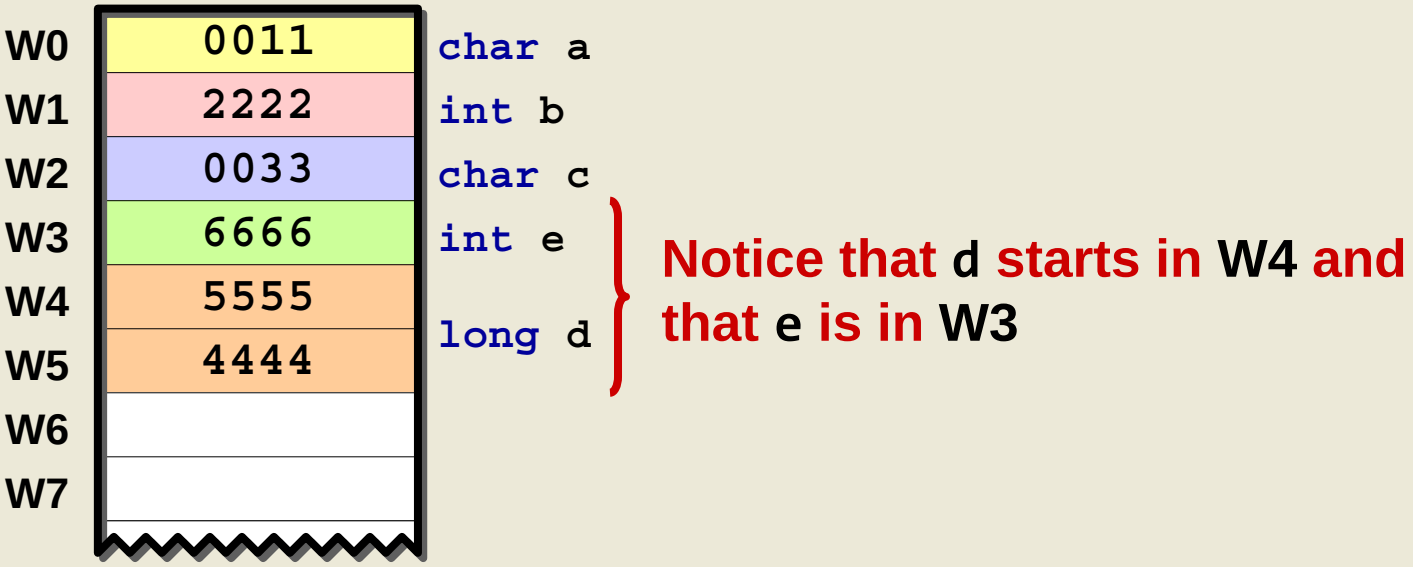
```
#include "p24fj128ga010.h"

void foo(char a, int b, char c, long d, int e);

int main(void)
{
      foo(0x11, 0x2222, 0x33, 0x44445555, 0x6666);
      while(1);
}
```

Calling an Assembly Function
Parameter Passing

Parameter Passing Example

Function Prototype: `void foo(char a, int b, char c, long d, int e);`

Function Call: `foo(0x11, 0x2222, 0x33, 0x44445555, 0x6666);`



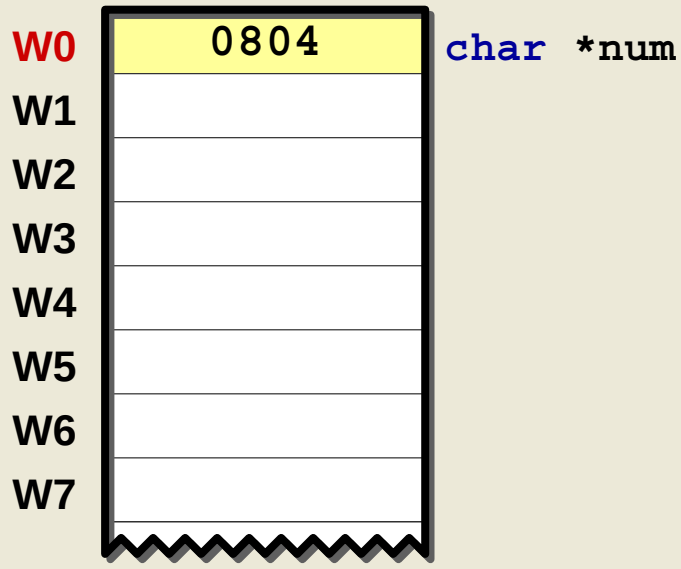
Two word (32-bit) variables must start in an even numbered W register
Four word (64-bit) variables must start in W0 or W4
If all variables don't fit, then the stack is used for the overflow.

Calling an Assembly Function
Passing Non-Scalar Parameters

- **Non-Scalar parameters such as arrays and structures will usually be passed as a pointer in one of the W registers (data pointers are 16-bits)**

Example

```
#include "p24fj128ga010.h"

void foo(char a[4]);
char num[4] = {0, 1, 2, 3};

int main(void)
{
      foo(num);
      while(1);
}
```

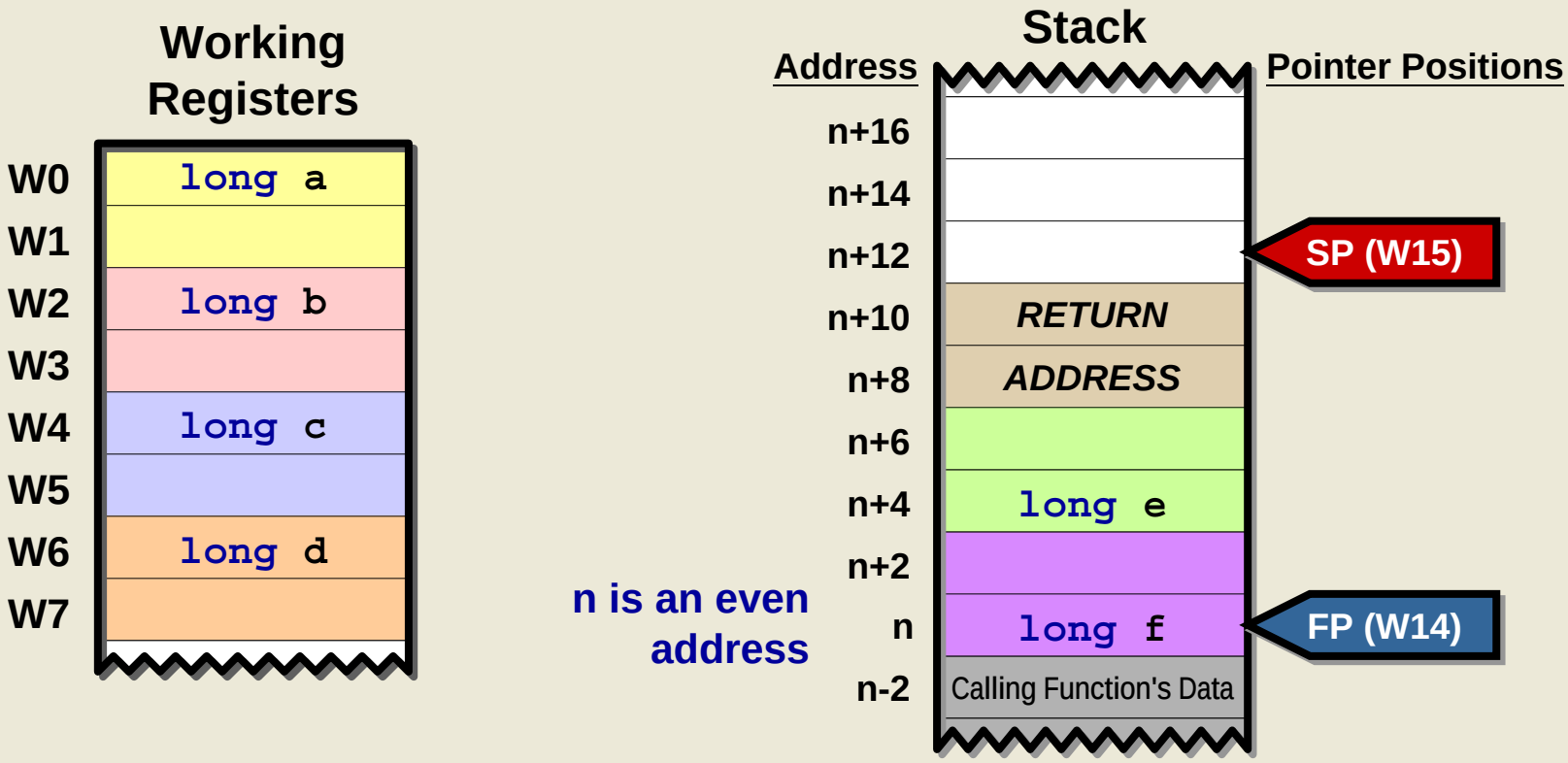


Calling an Assembly Function
Additional Parameters Passed via Stack

Example: Additional Parameters Passed via Software Stack

Function Prototype:

```
void foo(long a, long b, long c,
         long d, long e, long f);
```



Lab Exercise 7

Mixing C and Assembly

Lab Exercise 7
Mixing C and Assembly

Objective

From the file Lab7.c, call the functions:

```
int AddFunction(int, int);
int MostSignificant1(int);
```

which are written in assembly language in the file Lab7_asm.s. Display the output of the functions on the Explorer 16's LCD.

- **Assembly code is already written for you**
- **Add in the necessary hooks in the C source file so that you can call the assembly functions**

Lab Exercise 7
Mixing C and Assembly

Procedure

Follow the directions in the lab manual starting on page 7-1.

On the lab PC…

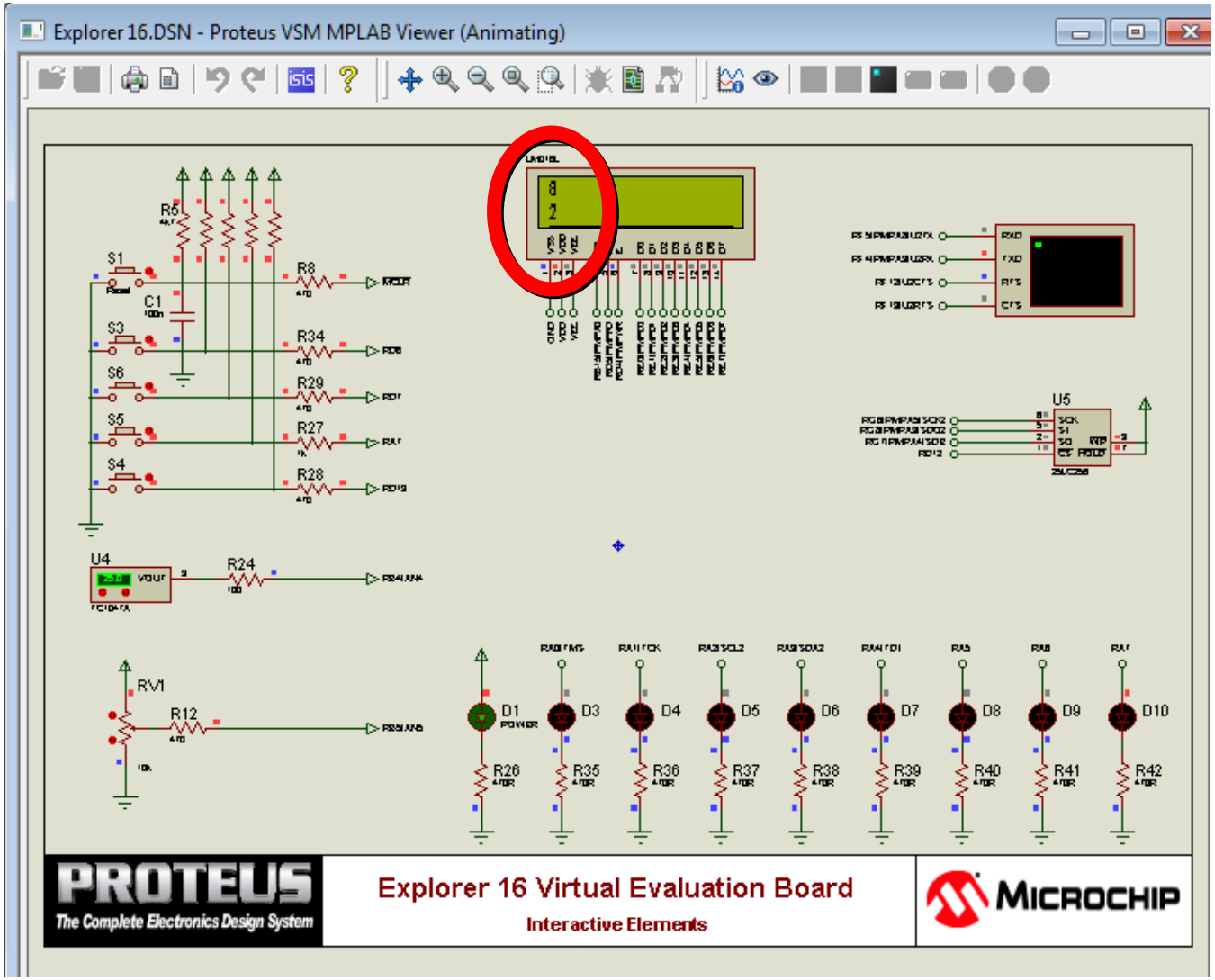
If you currently have a project or workspace open, close it now by selecting from the menu:
File ▶ Close Workspace

Open the lab workspace by selecting from the menu: File ▶ Open Workspace…
and select the file:

C:\MTT\TLS2130\Lab7\Lab7.mcw

Lab Exercise 7
Mixing C and Assembly

Results



Lab Exercise 7
Mixing C and Assembly

Lab7_asm.s

```
        .include "p24fj128ga010.inc"
        .global _AddFunction
        .global _MostSignificant1

        .text
_AddFunction:
        add w0,w1,w0            ; Add two integers
        return                  ; Return result in W0

_MostSignificant1:
        ff1l [w0], w0           ; Find first 1 from left
                                ;(result counted from left)
        subr w0, #16, w0        ; Adjust to give traditional bit
                                ; position number from right
        return                  ; Return result in W0

        .end
```

1. Make assembly subroutine names visible as functions in C
2. Parameters passed via W0-W7 (Only W0 and W1 used here)
3. Return value passed in W0

Lab Exercise 7
Mixing C and Assembly

Lab7.c – One Possible Solution…

```
int a, b, c;

int AddFunction(int x, int y);
int MostSignificant1(int *x);

int main(void)
{
    lcdInit();
    a = 5;
    b = 3;

    c = AddFunction(a, b);        // c = a + b (in assembly file)

    lcdPutInt(c, DEC);            // Display value of c on first line of LCD
    lcdPutCur(1,0);               // Move cursor to second line of LCD

    c = MostSignificant1(&a);     // Find bit position of most significant 1

    lcdPutInt(c, DEC);            // Display value of c on second line of LCD
    while (1);
}
```

Lab Exercise 7
Mixing C and Assembly

Conclusions

- **Functions written in assembly can easily be integrated into a C based project**
- **From C's perspective, assembly function is no different from a C function**
- **From assembly perspective, you must use caution so as to not break the C code**

Optimization Techniques

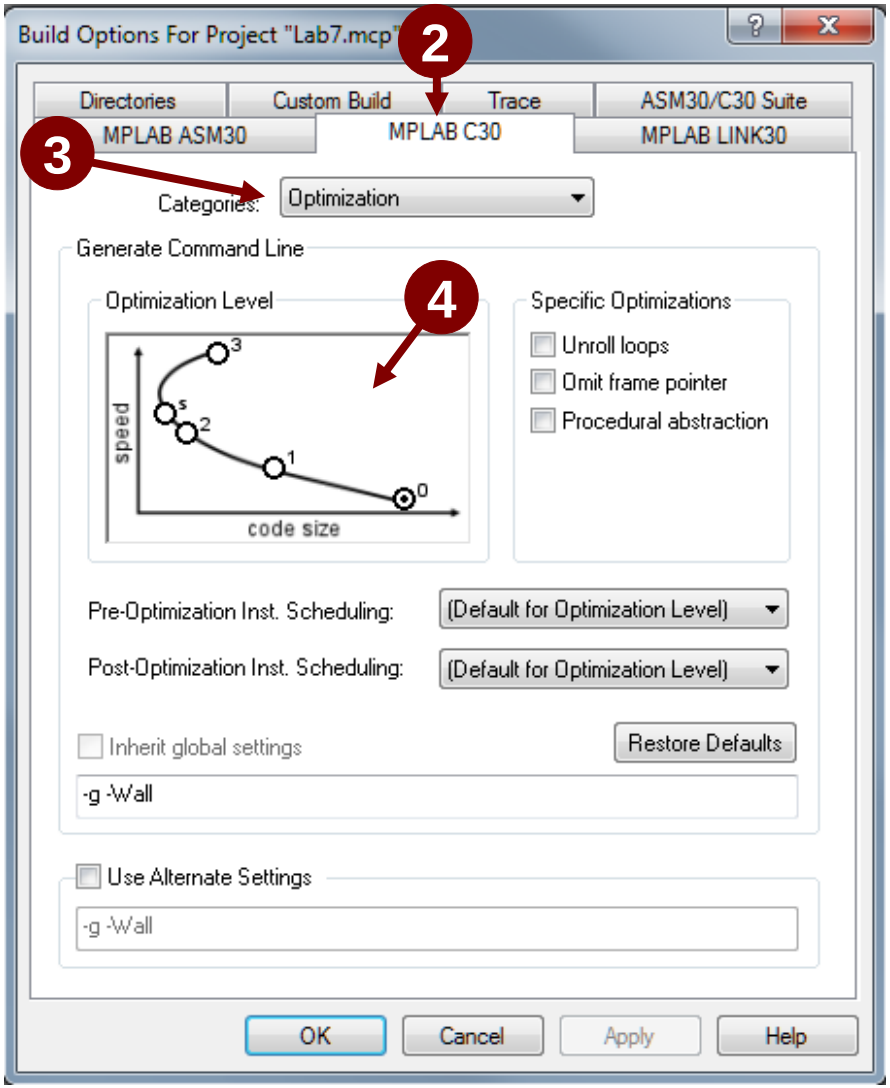
Generating More Efficient Code

Optimization Techniques

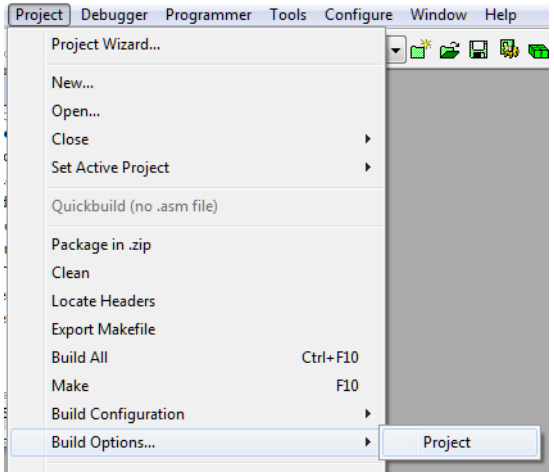
- Compilers perform two kinds of optimization:
  - Speed – Get more performance out of your code
  - Size – Get more code into your microcontroller
- Several techniques are used behind the scenes to create optimized code
- Usually, there is a tradeoff between size and speed (occasionally you get both)
- Additional optimization may be achieved manually, by using built-in functions to take advantage of architectural features that are not easily accessed with ordinary C code

Compiler Optimizations

How to Enable MPLAB® C30 Compiler Optimizations



1. From the menu, select:
   Project ▸ Build Options... ▸ Project



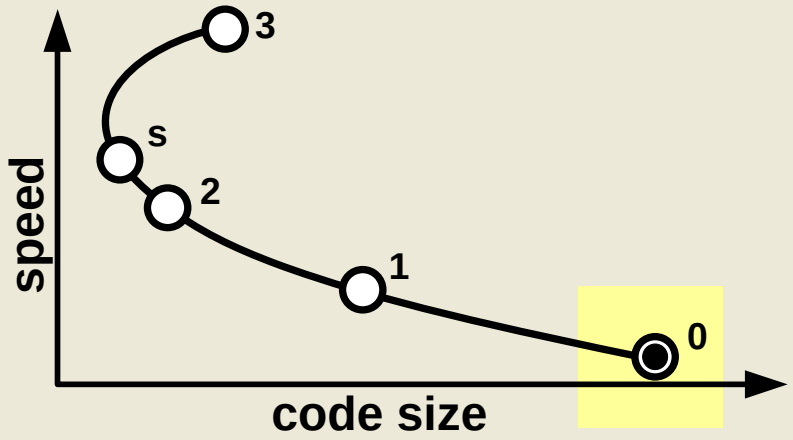
2. Select the MPLAB C30 Tab

3. From the Categories combo box, select Optimization

4. Choose an Optimization Level above 0

Compiler Optimizations
-O0 (Level 0) – Do Not Optimize

Optimization Level 0



Enabled Optimizations:

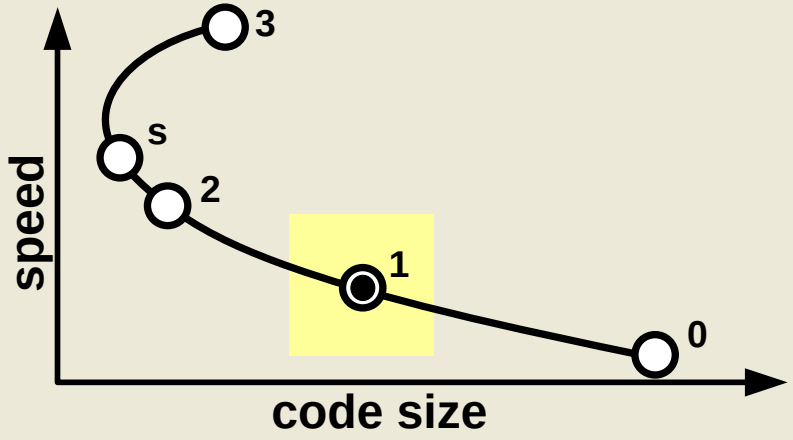
- NONE

- **Fastest Compilation Time**
- **Easiest for Debugging**
- **Code not rearranged**
- **Code behaves as expected when variable values changed at break point**

Compiler Optimizations

`-O` or `-O1` (Level 1) – Optimize

Optimization Level 1



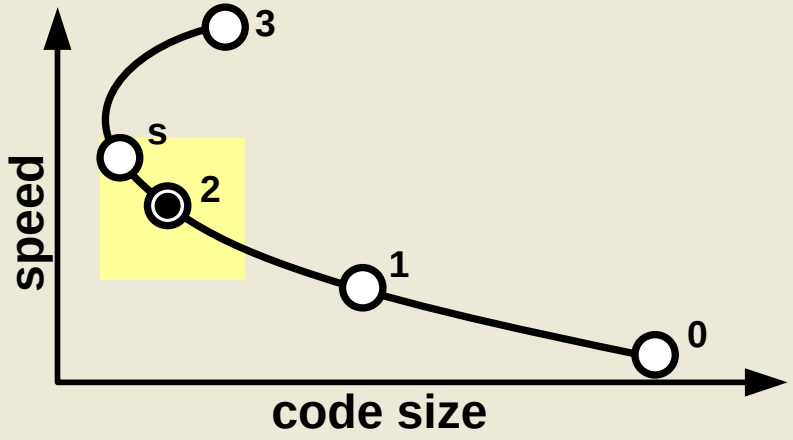
Enabled Optimizations:

- `-fthread-jumps`
- `-fdefer-pop`
- `-fomit-frame-pointer`

- **Compilation takes a bit longer**
- **Compilation requires much more memory on PC**
- **Compiler tries to reduce both code size and execution time**

Compiler Optimizations
-O2 (Level 2) – Optimize Even More

Optimization Level 2



Enabled Optimizations:

All optimizations EXCEPT:

- `-funroll-loops`
- `-finline-functions`

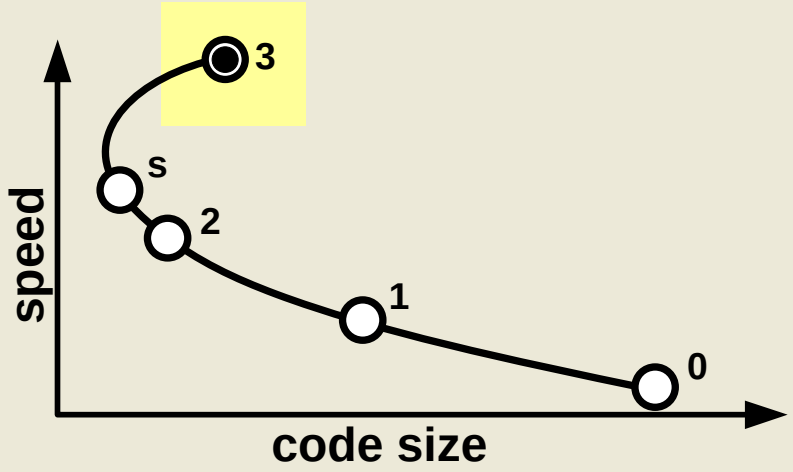
Also enables:

- `-fforce-mem`
- `-fomit-frame-pointer`

- **Performs nearly all optimizations that do not involve a space versus speed trade-off**

Compiler Optimizations
-O3 (Level 3) – Optimize Yet More

Optimization Level 2



Enabled Optimizations:

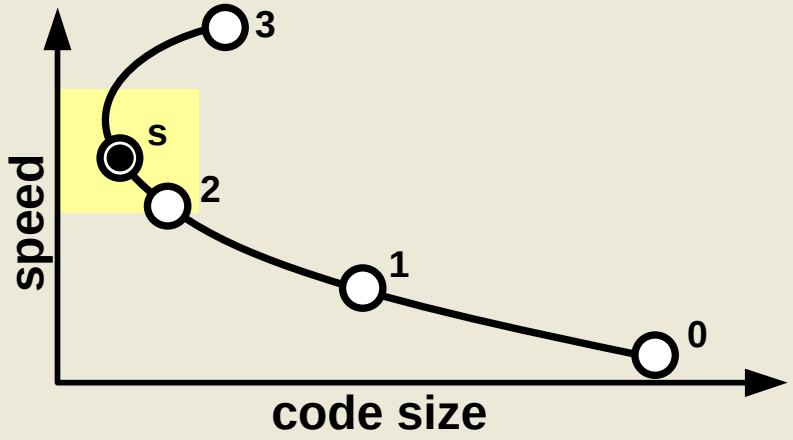
All optimizations plus:

- `-finline-functions`

- **Turns on all optimizations in level 2 (`-O2`)**

Compiler Optimizations
-Os (Level S) – Optimize For Size

Optimization Level 2



Enabled Optimizations:

All Level 2 optimizations that do not increase code size

- **Turns on all optimizations in level 2 (-O2) that do not typically increase code size**
- **Performs further optimizations to reduce code size**

MPLAB® C30's Built-In Functions

- Functions built into the compiler itself to implement architecture specific tasks that cannot be implemented directly in ANSI standard C
  - No libraries need to be added
  - Just call like an ordinary function

MPLAB® C30's Built-In Functions

Built-In Function Prototypes (See MPLAB® C30 Help File for Details) – Part 1

```
int __builtin_addab(void);

int __builtin_add(int value, const int shift);

void __builtin_btg(unsigned int *, unsigned int 0xn);

int __builtin_clr(void);

int __builtin_clr_prefetch(int **xptr, int *xval, int xincr,
                           int **yptr, int *yval, int yincr, int *AWB);

signed int __builtin_divmodsd(signed long dividend, signed int divisor,
                              signed int *remainder);

unsigned int __builtin_divmodud(unsigned long dividend,
                                unsigned int divisor,
                                unsigned int *remainder);

int __builtin_divsd(const long num, const int den);

unsigned int __builtin_divud(const unsigned
                             long num, const unsigned int den);
```

MPLAB® C30's Built-In Functions

Built-In Function Prototypes (See MPLAB® C30 Help File for Details) – Part 2

```
unsigned int __builtin_dmaoffset(const void *p);

int __builtin_ed(int sqr, int **xptr, int xincr,
                 int **yptr, int yincr, int *distance);

int __builtin_edac(int sqr, int **xptr, int xincr,
                   int **yptr, int yincr, int *distance);

int __builtin_fbcl(int value);

int __builtin_lac(int value, int shift);

int __builtin_mac(int a, int b,
                  int **xptr, int *xval, int xincr,
                  int **yptr, int *yval, int yincr, int *AWB);

signed int __builtin_modsd(signed long dividend,
                           signed int divisor);

int __builtin_movsac(int **xptr, int *xval, int xincr,
                     int **yptr, int *yval, int yincr, int *AWB);
```

MPLAB® C30's Built-In Functions

Built-In Function Prototypes (See MPLAB® C30 Help File for Details) – Part 3

```
int __builtin_mpy(int a, int b,
                  int **xptr, int *xval, int xincr,
                  int **yptr, int *yval, int yincr);
int __builtin_mpyn(int a, int b,
                   int **xptr, int *xval, int xincr,
                   int **yptr, int *yval, int yincr);
int __builtin_msc(int a, int b,
                  int **xptr, int *xval, int xincr,
                  int **yptr, int *yval, int yincr, int *AWB);
signed long __builtin_mulss(const signed int p0, const signed int p1);
signed long __builtin_mulsu(const signed int p0,
                            const unsigned int p1);
signed long __builtin_mulus(const unsigned int p0,
                            const signed int p1);
unsigned long __builtin_muluu(const unsigned int p0,
                              const unsigned int p1);
void __builtin_nop(void);
```

MPLAB® C30's Built-In Functions

Built-In Function Prototypes (See MPLAB® C30 Help File for Details) – Part 4

```
unsigned int __builtin_psvpage(const void *p);
unsigned int __builtin_psvoffset(const void *p);
unsigned int __builtin_readsfr(const void *p);
int __builtin_return_address (const int level);
int __builtin_sac(int value, int shift);
int __builtin_sacr(int value, int shift);
int __builtin_sftac(int shift);
int __builtin_subab(void);
unsigned int __builtin_tblpage(const void *p);
unsigned int __builtin_tbloffset(const void *p);
unsigned int __builtin_tblrdh(unsigned int offset);
unsigned int __builtin_tblrdl(unsigned int offset);
void __builtin_tblwth(unsigned int offset unsigned int data);
```

MPLAB® C30's Built-In Functions

Built-In Function Prototypes (See MPLAB® C30 Help File for Details) – Part 5

```
void __builtin_tblwtl(unsigned int offset unsigned int data);
void __builtin_write_NVM(void);
void __builtin_write_OSCCONL(unsigned char value);
void __builtin_write_OSCCONH(unsigned char value);
```

Built-In Function Prototypes (Undocumented as of Revision F of MPLAB® C30 User's Guide)

```
void __builtin_disi(unsigned int cycles);
```

Multiplication Code
Using C's Multiply Operator – No Typecast

Assembly output when using C's multiply operator *

```
#include <p24fj128ga010.h>

int a = 250, b = 250;
long c;

int main(void)
{
    c = a * b;

    while(1);
}
```

This code will not yield the correct results if the value of the result requires more than 16-bits

```
mov.w    _b,W1
mov.w    _a,W0
mul.ss   W1,W0,W0
asr      W0,#15,W1
mov.w    W0,_c
mov.w    W1,_c+2
```

6 Instructions / 6 Cycles

Multiplication Code
Using C's Multiply Operator

Assembly output when using C's multiply operator *

```
#include <p24fj128ga010.h>

int a = 250, b = 250;
long c;

int main(void)
{
    c = (long)a * b;

    while(1);
}
```

13 Instructions / 13 Cycles

```
mov.w    _a,W0
mul.su   W0,#1,W8
mov.w    _b,W0
mul.su   W0,#1,W2
mul.uu   W8,W2,W6
mul.ss   W8,W3,W0
mov.w    W7,W4
add.w    W4,W0,W4
mul.ss   W2,W9,W0
add.w    W4,W0,W4
mov.w    W4,W7
mov.w    W6,_c
mov.w    W7,_c+2
```

Multiplication Code
Using __builtin_mulss()

Assembly output when using C30's built-in mulss function

```
#include <p24fj128ga010.h>

int a = 250, b = 250;
long c;

int main(void)
{
    c = __builtin_mulss(a, b);

    while(1);
}
```

```
mov.w    _b,W1
mov.w    _a,W0
mul.ss   W0,W1,W0
mov.w    W0,_c
mov.w    W1,_c+2
```

5 Instructions / 5 Cycles

Multiplication Code
Why Doesn't the Compiler Do This Automatically?

- **Because it wouldn't be a *C* compiler**
  - **The ANSI C language requires that multiplication be handled a certain way**
  - **Changing that way creates a new C-like language that will behave in a dramatically different way from ANSI C**
  - **C code would not behave the way an experienced C programmer would expect**
  - **It would create non-portable code from standard C syntax**
- **Blame K&R – not Microchip ☺**

Toggle Code
Using C's ^= Operator with Bit Field Member

Assembly output when using C's ^= operator to toggle a bit field

```
#include <p24fj128ga010.h>

int main(void)
{
    _LATA0 ^= 1;
}
```

```
mov.b   _LATA,W0
and.b   W0,#1,W0
btg     W0,#0
and.b   W0,#1,W0
and.b   W0,#1,W2
mov.w   #0x2c4,W1
mov.b   [W1],W1
mov.b   #0xfe,W0
and.b   W1,W0,W0
ior.b   W0,W2,W0
mov.b   W0,0x02c4
```

11 Instructions / 11 Cycles

Toggle Code
Using __builtin_btg()

Assembly output when using __builtin_btg()

```
#include <p24fj128ga010.h>

int main(void)
{
    __builtin_btg(&LATA, 0);
}
```

```
mov.w    #_LATA,W0
btg      [W0],#0
```

2 Instructions / 2 Cycles

Toggle Code
Using Inline Assembly

Assembly output when using inline assembly

```
#include <p24fj128ga010.h>

int main(void)
{
    asm("btg LATA, #0");
}
```

This code will only work with registers in `near` data space. All SFRs are in `near` space.

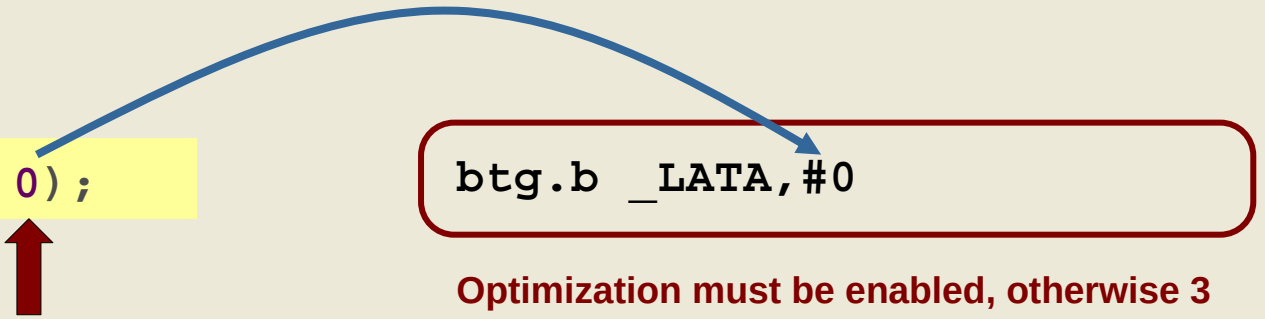
```
btg.b _LATA,#0
```

1 Instruction / 1 Cycle

Toggle Code
Using C's ^= Operator with `int` Variable

Assembly output when using xor with a shifted 1

```
#include <p24fj128ga010.h>

int main(void)
{
    LATA ^= (1 << 0);
}
```



Bit to Toggle

Optimization must be enabled, otherwise 3 instructions will be generated.

1 Instruction / 1 Cycle

Optimization Techniques
Data Types

- Use `int` for array index variables
- Use `int` for local `auto` variables
- If you aren't trying to save memory, use an `int` because not all instructions support byte access (`.b` instruction suffix in assembly)

Optimization Techniques
Bit Fields

- More readable but can be less efficient
- Good:
  - Assigning literals to bit fields (`REGbits.bit = 1`)
  - Testing bit fields (`REGbits.bit == 1`)
- Bad:
  - Toggling bits (`REGbits.bit ^= 1`)
  - Assigning a variable value to a bit field
  - Arithmetic on a bit field

Optimization Techniques
Modifying SFRs

- With `volatile` variables / registers, a sequence of bit field modifications cannot be optimized to a single write:

```
T1CONbits.TCKPS = 2;
T1CONbits.TGATE = 0;
T1CONbits.TSIDL = 1;
T1CONbits.TON = 1;
```

```
mov.b   T1CON,W0
bclr    W0,#TCKPS0
bset    W0,#TCKPS1
mov.b   W0,T1CON
bclr    T1CON,#TGATE
bset    T1CON,#TSIDL
bset    T1CON,#TON
```

- Write a full integer to the entire register instead:

```
T1CON = 0xA020;
```

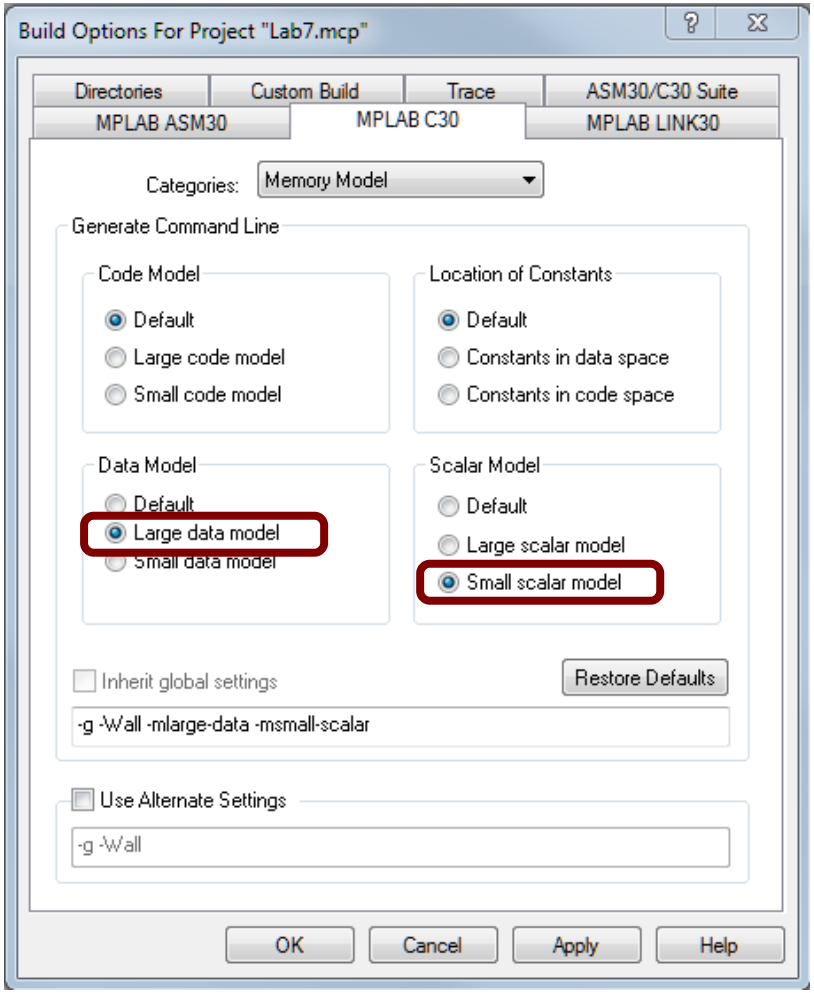
```
mov.w   #0xa020,W0
mov.w   W0,T1CON
```

Optimization Techniques
Miscellaneous

- Ensure that function prototypes match arguments (especially signed-ness)
- Use local/auto variables in preference to global/static variables – the compiler can put them into W registers
- Do not use `char` for auto variables

Optimization Techniques
Data Memory Models

- **Use Large Data and Small Scalar Models**
  - **Entire data memory map is available**
  - **Puts scalar variables in near memory**
  - **Puts non-scalars in far memory (arrays, structures, unions)**



Inline Functions

Syntax

```
inline returnType identifier(parameterList)
{
     //Function code here
}
```

- Integrates function's code into code for its callers
- Usually faster due to lower overhead
- Code size may be smaller or larger
- May only be used if function definition is visible in file where used (not just the function's prototype)
- inline Functions may be placed in header files

Inline Functions

- Best used with short functions where overhead is an issue
- Must enable optimizations or use the `-finline` command line switch
- Similar to macros but without the preprocessor
  - Aware of C syntax and constructs
  - May be further optimized by compiler

Example

```
inline int square(int a)
{
        return __builtin_mulss(a, a);
}
```

Memory Models

X and Y Data Space (dsPIC® Only)

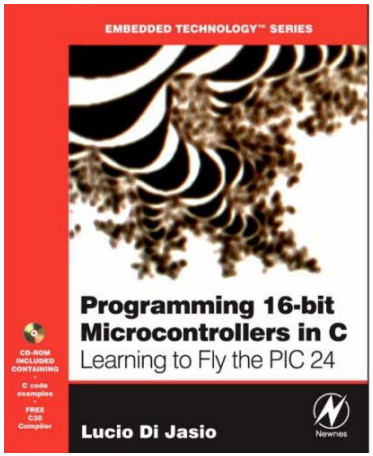
- Compiler does not directly support separating variables into X and Y data
- Section attribute may be used to explicitly locate variables in X and Y spaces

Example

```
float buffer[32] __attribute__((__section__(".ydata")));

float coeff[16] __attribute__((__section__(".xdata")));
```

References

Suggested Reading



Programming 16-bit Microcontrollers in C

by Lucio Di Jasio

ISBN-10: 0750682922

ISBN-13: 978-0750682923

http://www.flyingpic24.com

Thank You!