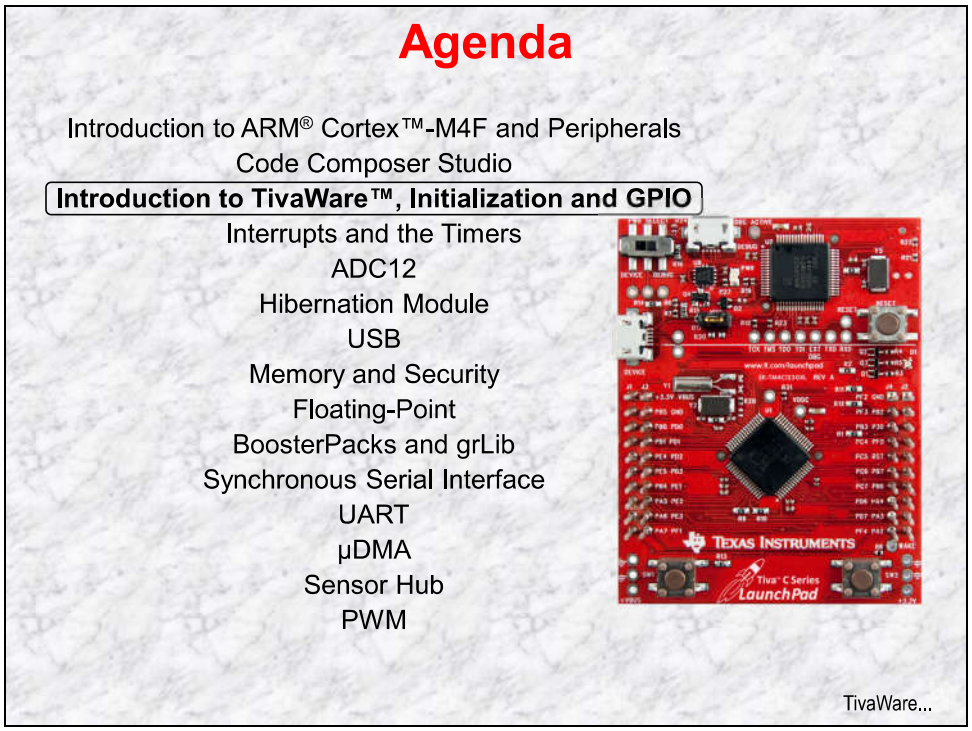
TivaWare™, Initialization and GPIO

Introduction

This chapter will introduce you to TivaWare, the initialization of the device and the operation of the GPIO. The lab exercise uses TivaWare API functions to set up the clock, and to configure and write to the GPIO port.



Chapter Topics

TivaWare

TivaWare™ for C Series Features

Peripheral Driver Library
- High-level API interface to complete peripheral set
- License & royalty free use for TI Cortex-M parts
- Available as object library and as source code
- **Programmed into the on-chip ROM**

USB Stacks and Examples
- USB Device and Embedded Host compliant
- Device, Host, OTG and Windows-side examples
- Free VID/PID sharing program

Extras
- Wireless protocols
- IQ math examples
- Bootloaders
- Windows side applications

Ethernet
- lwip and uip stacks with 1588 PTP modifications
- Extensive examples

Graphics Library
- Graphics primitive and widgets
- 153 fonts plus Asian and Cyrillic
- Graphics utility tools

Sensor Library
- An interrupt driven I²C master driver for handling I²C transfers
- A set of drivers for I²C connected sensors
- A set of routines for common sensor operations
- Three layers: Transport, Sensor and Processing

ISP Options...

In System Programming Options

Tiva Serial Flash Loader
- Small piece of code that allows programming of the flash without the need for a debugger interface.
- All Tiva C Series MCUs ship with the loader in flash
- UART or SSI interface option
- The LM Flash Programmer interfaces with the serial flash loader
- See application note SPMA029

Tiva Boot Loader
- Preloaded in ROM or can be programmed at the beginning of flash to act as an application loader
- Can also be used as an update mechanism for an application running on a Tiva microcontroller.
- Interface via UART (default), I²C, SSI, Ethernet, USB (DFU H/D)
- Included in the Tiva Peripheral Driver Library with full applications examples

Fundamental Clocks...

Clocking

Fundamental Clock Sources

Precision Internal Oscillator (PIOSC)
- 16 MHz ± 3%

Main Oscillator (MOSC) using…
- An external single-ended clock source
- An external crystal

Internal 30 kHz Oscillator
- 30 kHz ± 50%
- Intended for use during Deep-Sleep power-saving modes

Hibernation Module Clock Source
- 32,768Hz crystal
- Intended to provide the system with a real-time clock source

SysClk Sources...

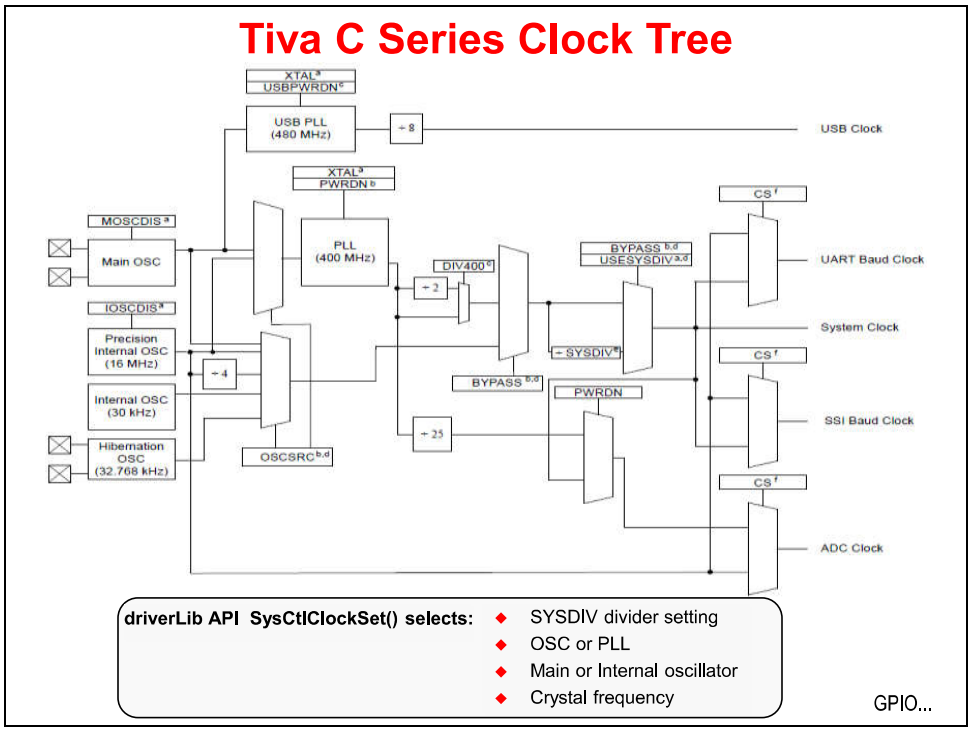
System (CPU) Clock Sources

The CPU can be driven by any of the fundamental clocks …
- Internal 16 MHz
- Main
- Internal 30 kHz
- External Real-Time

- Plus -
- The internal PLL (400 MHz)
- The internal 16MHz oscillator divided by four (4MHz ± 3%)

Clock Source	Drive PLL?	Used as SysClk?
Internal 16MHz	Yes	Yes
Internal 16Mhz/4	No	Yes
Main Oscillator	Yes	Yes
Internal 30 kHz	No	Yes
Hibernation Module	No	Yes
PLL	-	Yes

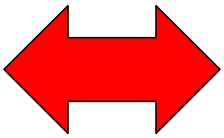
Clock Tree...


Tiva C Series Clock Tree
XTAL[a] USBPWRDN[c]
USB PLL (480 MHz)
÷ 8
USB Clock
XTAL[a] PWRDN[b]
PLL (400 MHz)
MOSCDIS[a]
Main OSC
IOSCDIS[a]
Precision Internal OSC (16 MHz)
÷ 4
Internal OSC (30 kHz)
Hibernation OSC (32.768 kHz)
OSCSRC[b,d]
DIV400[e]
÷ 2
÷ 25
BYPASS[b,d]
PWRDN
BYPASS[b,d] USESYSDIV[a,d]
÷ SYSDIV[e]
CS[f]
UART Baud Clock
System Clock
SSI Baud Clock
ADC Clock
driverLib API SysCtlClockSet() selects:
SYSDIV divider setting
OSC or PLL
Main or Internal oscillator
Crystal frequency
GPIO...

GPIO

General Purpose IO

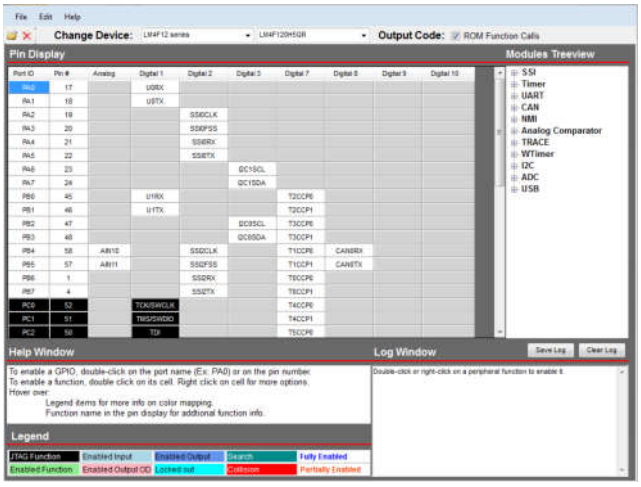
- **Any GPIO can be an interrupt:**
  - Edge-triggered on rising, falling or both
  - Level-sensitive on high or low values
- **Can directly initiate an ADC sample sequence or µDMA transfer**
- **Toggle rate up to the CPU clock speed on the Advanced High-Performance Bus. ½ CPU clock speed on the Standard.**
- **5V tolerant in input configuration**
- **Programmable Drive Strength (2, 4, 8mA or 8mA with slew rate control)**
- **Programmable weak pull-up, pull-down, and open drain**
- **Pin state can be retained during Hibernation mode**



Pin Mux Utility...

Pin Mux Utility

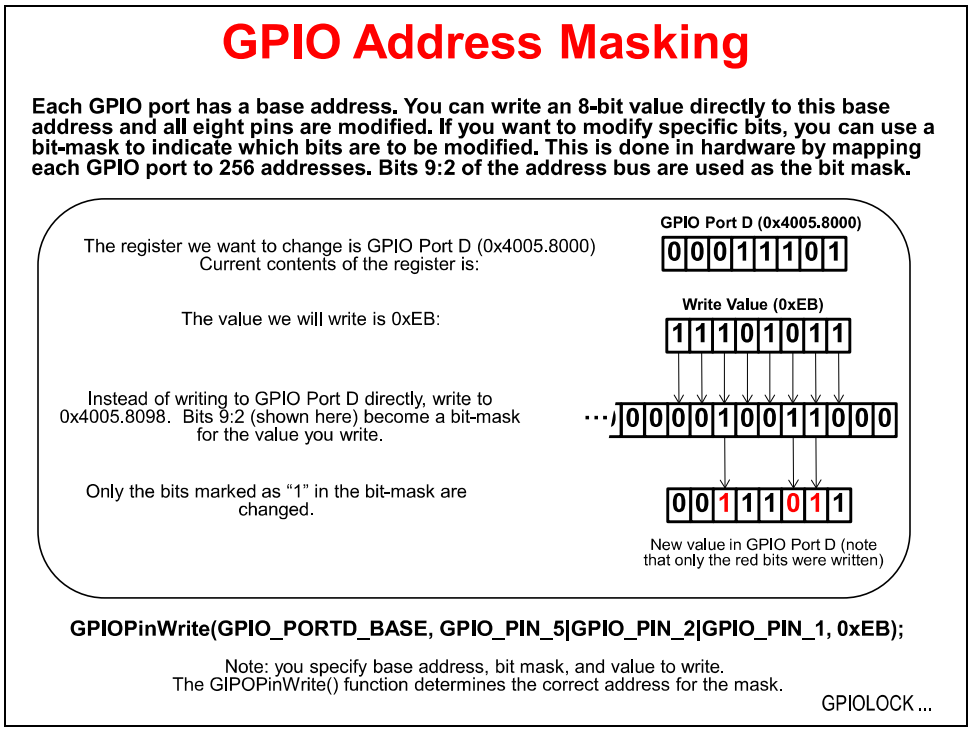
- Allows the user to graphically configure the device pin-out
- Generates source and header files for use with any of the supported IDE's



http://www.ti.com/tool/tm4c_pinmux

Masking...

http://www.ti.com/tool/tm4c_pinmux



The masking technique used on Tiva C Series GPIO is similar to the "bit-banding" technique used in memory. To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the **GPIO Data (GPIODATA)** register by using bits [9:2] of the address bus as a mask. In this manner, software can modify individual GPIO pins in a single, atomic read-modify-write (RMW) instruction without affecting the state of the other pins. This method is more efficient than the conventional method of performing a RMW operation to set or clear an individual GPIO pin. To implement this feature, the **GPIODATA** register covers 256 locations in the memory map.

.

Critical Function GPIO Protection

- **Six pins on the device are protected against accidental programming:**
  - PC3,2,1 & 0: JTAG/SWD
  - PD7 & PF0: NMI
- **Any write to the following registers for these pins will not be stored unless the GPIOLOCK register has been unlocked:**
  - GPIO Alternate Function Select register
  - GPIO Pull Up or Pull Down select registers
  - GPIO Digital Enable register
- **The following sequence will unlock the GPIOLOCK register for PF0 using direct register programming:**

```
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;
```

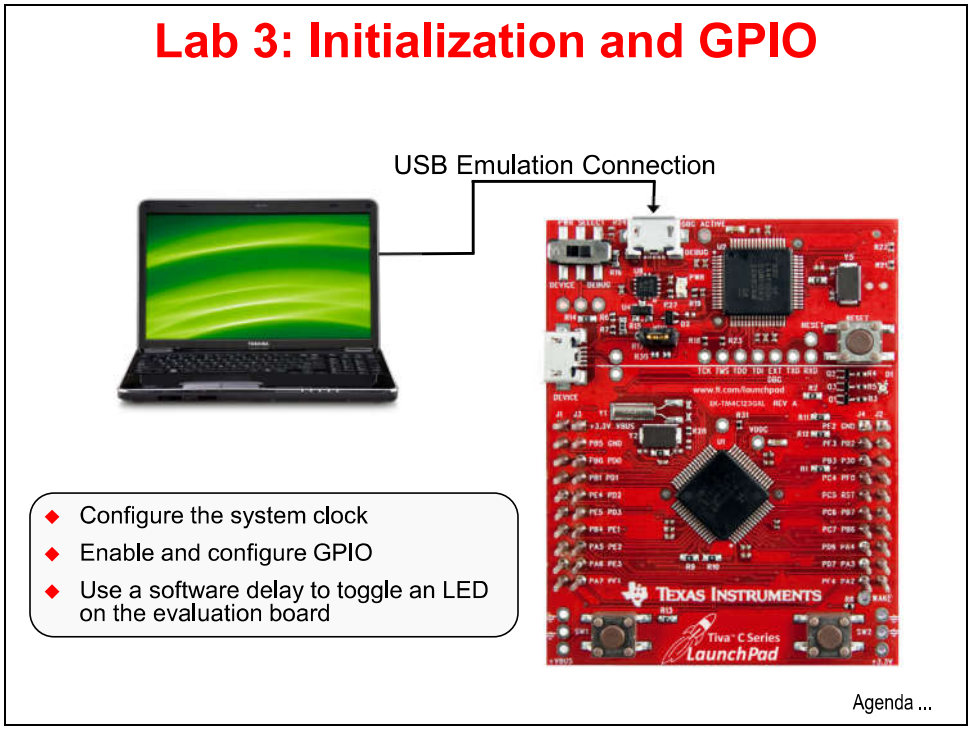
- **Reading the GPIOLOCK register returns it to lock status**

Lab...

Lab 3: Initialization and GPIO

Objective

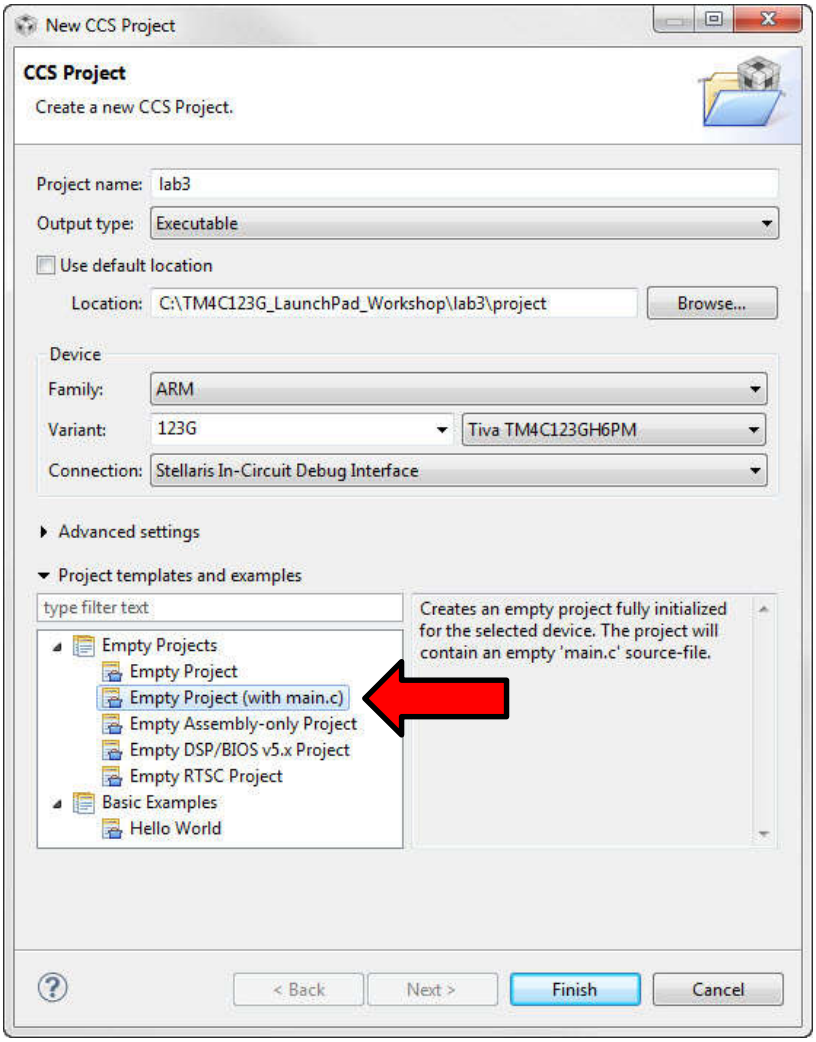
In this lab we'll learn how to initialize the clock system and the GPIO peripheral using TivaWare. We'll then use the GPIO output to blink an LED on the evaluation board.



Procedure

Create lab3 Project

1. ► Maximize Code Composer. On the CCS menu bar select File → New → CCS Project. Make the selections shown below. Make sure to **uncheck** the "Use default location" checkbox and select the correct path to the project folder as shown. In the variant box, just type "123G" to narrow the results in the right-hand box. In the Project templates and examples window, select **Empty Project (with main.c)**. Click Finish.



When the wizard completes, click the ▷ next to lab3 in the Project Explorer pane to expand the project. Note that Code Composer has automatically added a mostly empty `main.c` file to your project as well as the startup file.

Note: We placed a file called `main.txt` in the `lab3/project` folder which contains the final code for the lab. If you run into trouble, you can refer to this file.

Header Files

2. ► Delete the current contents of `main.c`.

TivaWare™ is written using the ISO/IEC 9899:1999 (or C99) C programming standards along with the Hungarian standard for naming variables. The C99 C programming conventions make better use of available hardware, including the IEE754 floating point unit. To keep everything looking the same, we're going to use those guidelines.

► Type (or cut/paste from the pdf file) the following lines into `main.c` to include the header files needed to access the TivaWare APIs as well as a variable definition:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"


uint8_t ui8PinData=2;
```

The use of the < > restricts the search path to only the specified path. Using the " " causes the search to start in the project directory. For includes like the two standard ones, you want to assure that you're accessing the original, standard files … not one's that may have been modified.

`stdint.h:` **V**ariable definitions for the C99 standard

`stdbool.int:` Boolean definitions for the C99 standard

`hw_memmap.h` : Macros defining the memory map of the Tiva C Series device. This includes defines such as peripheral base address locations such as GPIO_PORTF_BASE.

`hw_types.h` : Defines common types and macros

`sysctl.h` : Defines and macros for System Control API of DriverLib. This includes API functions such as `SysCtlClockSet` and `SysCtlClockGet`.

`gpio.h` : Defines and macros for GPIO API of DriverLib. This includes API functions such as `GPIOPinTypePWM` and `GPIOPinWrite`.

`uint8_t ui8PinData=2;` : Creates an integer variable called `ui8PinData` and initializes it to 2. This will be used to cycle through the three LEDs, lighting them one at a time. Note that the C99 type is an 8-bit unsigned integer and that the variable name reflects this.

You will see question marks to the left of the include lines in `main.c` displayed in the edit pane, telling us that the include files can't be found. We'll fix this later.

main() Function

3. Let's drop in a template for our main function.

   ► Leave a line for spacing and add this code after the previous declarations:

```
int main(void)

{

}
```

   If you type this in, notice that the editor will automatically add the closing brace when you add the opening one. Why wasn't this thought of sooner?

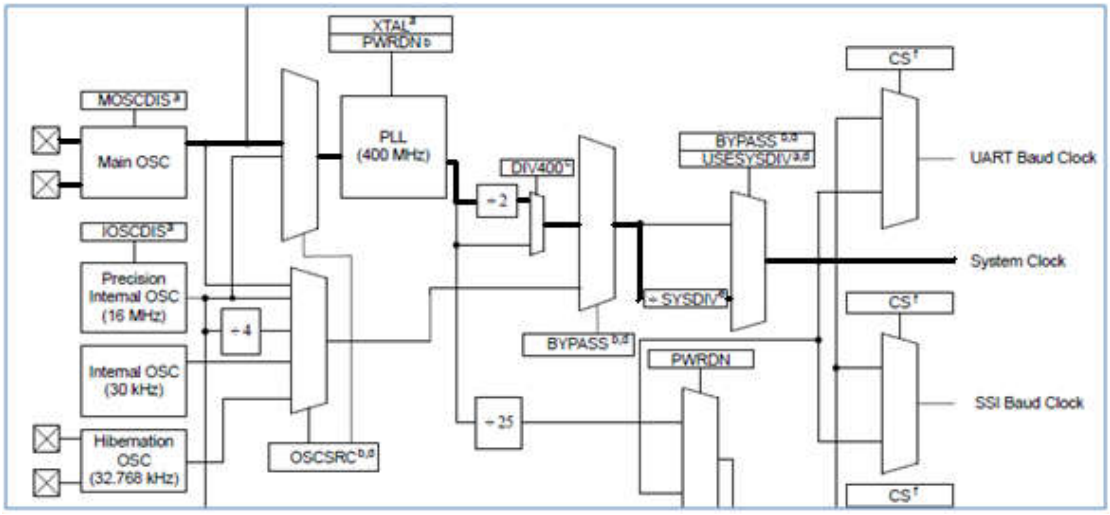
Clock Setup

4. Configure the system clock to run using a 16MHz crystal on the main oscillator, driving the 400MHz PLL. The 400MHz PLL oscillates at only that frequency, but can be driven by crystals or oscillators running between 5 and 25MHz. There is a default /2 divider in the clock path and we are specifying another /5, which totals 10. That means the System Clock will be 40MHz.

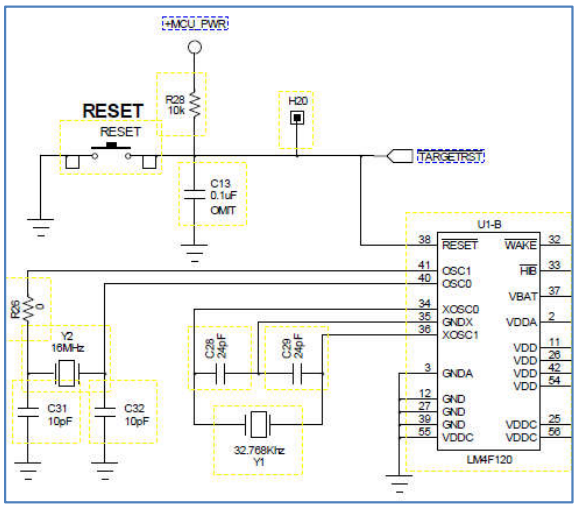
   ► Enter this single line of code inside `main()`:

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

   The diagram below is an abbreviated drawing of the clock tree to emphasize the System Clock path and choices. Note the darkened path.

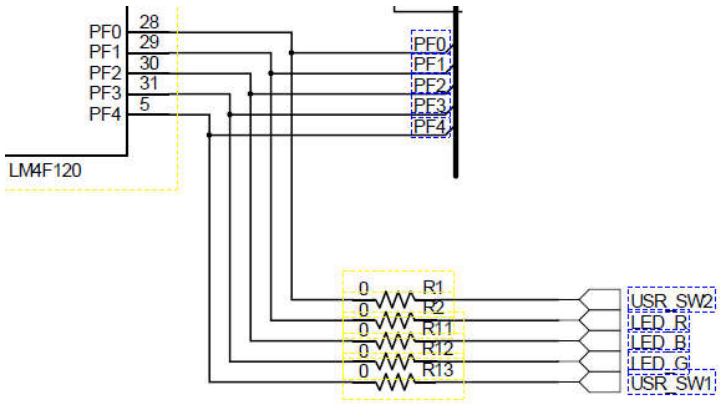


   The diagram below is an excerpt from the LaunchPad board schematic. Note that the crystal attached to the main oscillator inputs is 16MHz, while the crystal attached to the real-time clock (RTC) inputs is 32,768Hz.



GPIO Configuration

5. Before calling any peripheral specific `driverLib` function, we must enable the clock for that peripheral. If you fail to do this, it will result in a Fault ISR (address fault).This is a common mistake for new Tiva C Series users. The second statement below configures the three GPIO pins connected to the LEDs as outputs. The excerpt below of the LaunchPad board schematic shows GPIO pins PF1, PF2 and PF3 are connected to the LEDs.



► Leave a line for spacing, then enter these two lines of code inside `main()` after the line in the previous step.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

The base addresses of the GPIO ports listed in the User Guide are shown below. Note that they are all within the memory map's peripheral section shown in module 1. APB refers to the Advanced Peripheral Bus, while AHB refers to the Advanced High-Performance Bus. The AHB offers better back-to-back performance than the APB bus. GPIO ports accessed through the AHB can toggle every clock cycle vs. once every two cycles for ports on the APB. In power sensitive applications, the APB would be a better choice than the AHB. In our labs, `GPIO_PORTF_BASE` is `0x40025000`.

```
GPIO Port A (APB): 0x4000.4000
GPIO Port A (AHB): 0x4005.8000
GPIO Port B (APB): 0x4000.5000
GPIO Port B (AHB): 0x4005.9000
GPIO Port C (APB): 0x4000.6000
GPIO Port C (AHB): 0x4005.A000
GPIO Port D (APB): 0x4000.7000
GPIO Port D (AHB): 0x4005.B000
GPIO Port E (APB): 0x4002.4000
GPIO Port E (AHB): 0x4005.C000
GPIO Port F (APB): 0x4002.5000
GPIO Port F (AHB): 0x4005.D000
```

while() Loop

6. Finally, create a `while(1)` loop to send a "1" and "0" to the selected GPIO pin, with an equal delay between the two.

   `SysCtlDelay()` is a loop timer provided in TivaWare. The count parameter is the loop count, not the actual delay in clock cycles. Each loop is 3 CPU cycles.

   To write to the GPIO pin, use the GPIO API function call `GPIOPinWrite`. Make sure to read and understand how the `GPIOPinWrite` function is used in the datasheet. The third data argument is not simply a 1 or 0, but represents the entire 8-bit data port. The second argument is a bit-packed mask of the data being written.

   In our example below, we are writing the value in the `ui8PinData` variable to all three GPIO pins that are connected to the LEDs. Only those three pins will be written to based on the bit mask specified. The final instruction cycles through the LEDs by making ui8PinData equal to 2, 4, 8, 2, 4, 8 and so on. Note that the values sent to the pins match their positions; a "one" in the bit two position can only reach the bit two pin on the port.

   Now might be a good time to look at the Datasheet for your Tiva C Series device. Check out the GPIO chapter to understand the unique way the GPIO data register is designed and the advantages of this approach.

   ► Leave a line for spacing, and then add this code after the code in the previous step.

```
while(1)
{
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, ui8PinData);
SysCtlDelay(2000000);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
SysCtlDelay(2000000);
if(ui8PinData==8) {ui8PinData=2;} else {ui8PinData=ui8PinData*2;}
}
```

   If you find that the indentation of your code doesn't look quite right, ► select all of your code by clicking CTRL-A and then right-click on the selected code. Select **Source → Correct Indentation**. Notice the other great stuff under the **Source** and **Surround With** selections.

7. ► Click the Save button to save your work. Your code should look something like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

uint8_t ui8PinData=2;

int main(void)
{
        SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
        GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

        while(1)
        {
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1| GPIO_PIN_2| GPIO_PIN_3, ui8PinData);
                SysCtlDelay(2000000);
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
                SysCtlDelay(2000000);
                if(ui8PinData==8) {ui8PinData=2;} else {ui8PinData=ui8PinData*2;}
        }
}
```

If you're having problems, you can cut/paste this code into `main.c` or you can cut/paste from the `main.txt` file in your Project Explorer pane.

If you were to try building this code now (please don't), it would fail. Note the question marks next to the include statements … CCS has no idea where those files are located … we still need to set our build options.

NOTE: There is a delay of 3 to 6 clock cycles between enabling a peripheral and being able to use it. In most cases, the amount of time required by the API coding itself prevents any issues, but there are situations where you may be able to cause a system fault by attempting to access the peripheral before it becomes available.

A good programming habit is to interleave your peripheral enable statements as follows:

Enable ADC
Enable GPIO
Config ADC
Config GPIO

This interleaving will prevent any possible system faults without incorporating software delays.

Startup Code

8. In addition to the main file you have created, you will also need a startup file specific to the tool chain you are using. This file contains the vector table, startup routines to copy initialized data to RAM and clear the bss section, and default fault ISRs. The New Project wizard automatically added a copy of this file into the project for us.

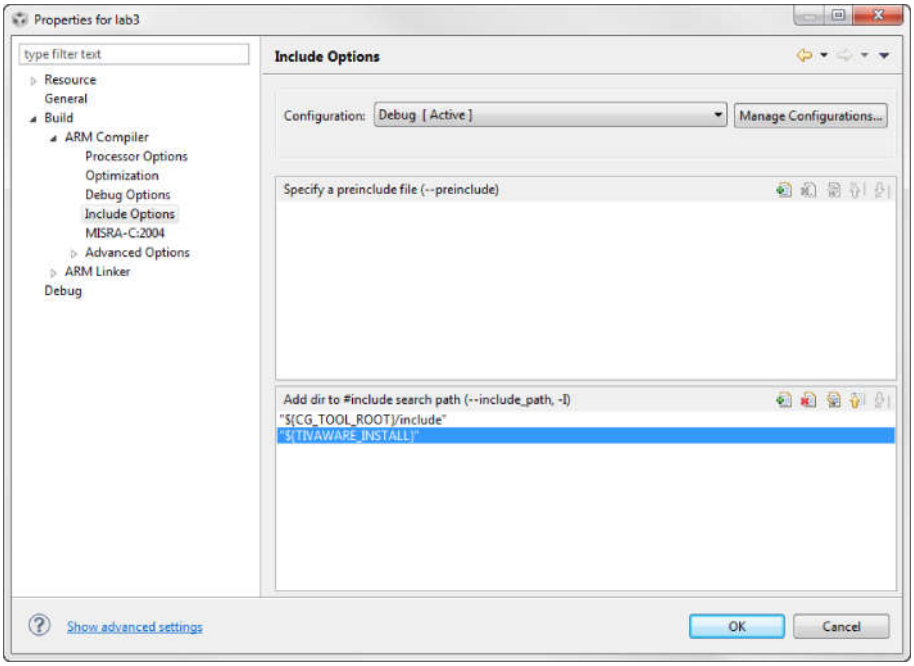
   ► Double-click on `tm4c123gh6pm_startup_ccs.c` in your Project Explorer pane and take a look around. Don't make any changes at this time. Close the file.

Set the Build Options

9. ► Right-click on Lab3 in the Project Explorer pane and select Properties. Click **Include Options** under **ARM Compiler**. In the bottom, **include search path** pane, click the Add button and add the following search path:

   ${TIVAWARE_INSTALL}

   Remember that those are braces, not parentheses. This is the path we created earlier by using the `vars.ini` file in the lab2 project. Since those paths are defined at the workspace level, we can simply use it again here.



► Click OK.

After a moment, CCS will refresh the project and you should see the question marks disappear from the include lines in `main.c`.

10. Add the Driver Library File

The driverlib.lib file needs to be in the `lab3` project. In `lab2` we added a link to this file. You can see it under your `lab2` project in the Project Explorer pane. Can it be as simple as dragging it over? Let's try it.

► Click and hold `driverlib.lib` under the `lab2` project in the Project Explorer pane. ► Drag it onto the `lab3` project and release7. You should now see the file under `lab3`.

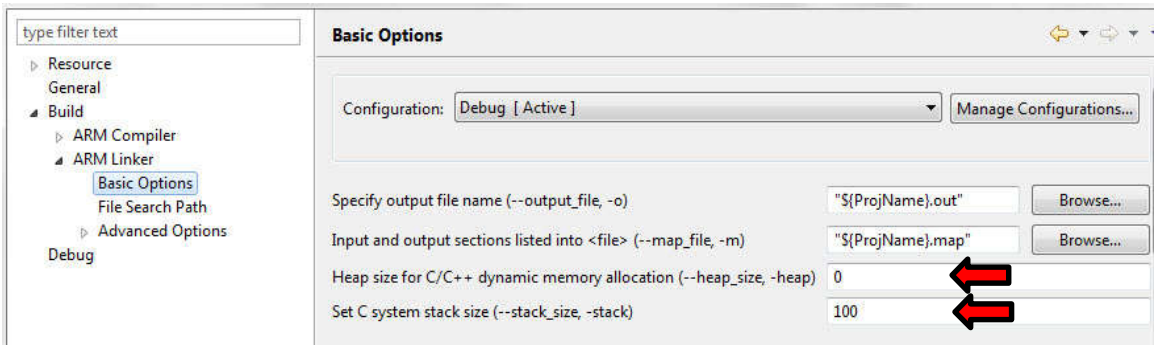
The file that was linked to `lab2` is now linked to `lab3`. That was even easier.

11. It can be easy to get confused and mistakenly build or work on the wrong project or file. To reduce that possibility, ► right-click on `lab2` and select *Close Project*. This will collapse the project and close any open files you have from the project. You can open it again at any time. ► Click on the lab3 project name to make sure the project is active. It will say **`lab3 [Active - Debug]`**. This tells you that the lab3 project is active and that the build configuration is debug.

12. Stack Considerations

► Test build the lab3 to check for errors by clicking the Build (Hammer) button. Note that a warning appears in the Problems pane in the lower right of CCS. This error; "creating .stack section with default size of 0x800…" tells us that the stack size was not specified. We can eliminate this warning by specifying the stack size(s).

► Right-click on the lab3 project in the Project Explorer pane and select Properties. Expand *Build → ARM Linker* and click on *Basic Options*. Find the *Heap size* and *Set C system stack size* boxes as shown below.



► Enter *0* for the *Heap size* and *100* for the *C system stack size* and click OK. We won't be using the heap in these labs and our need for a C stack is very limited. Failure to monitor the size of your stack(s) can result in significant amount of memory being wasted. Test build again to make sure the warning no longer appears.

These settings will be made for you in the rest of the labs.

Compile, Download and Run the Code

13. ► Compile and download your application by clicking the Debug button on the menu bar. If you are prompted to save changes, do so. If you have any issues, correct them, and then click the Debug button again (**see the hints page in section 2**). After a successful build, the CCS Debug perspective will appear.

► Click the Resume button to run the program that was downloaded to the flash memory of your device. You should see the LEDs flashing. If you want to edit the code to change the delay timing or which LEDs that are flashing, go ahead.

If you suspend the code and get the message "*No source available for ...",* simply close that editor tab. The source code for that function is not present in our project. It is only present as a library file.

► Click on the Terminate button to return to the CCS Edit perspective.