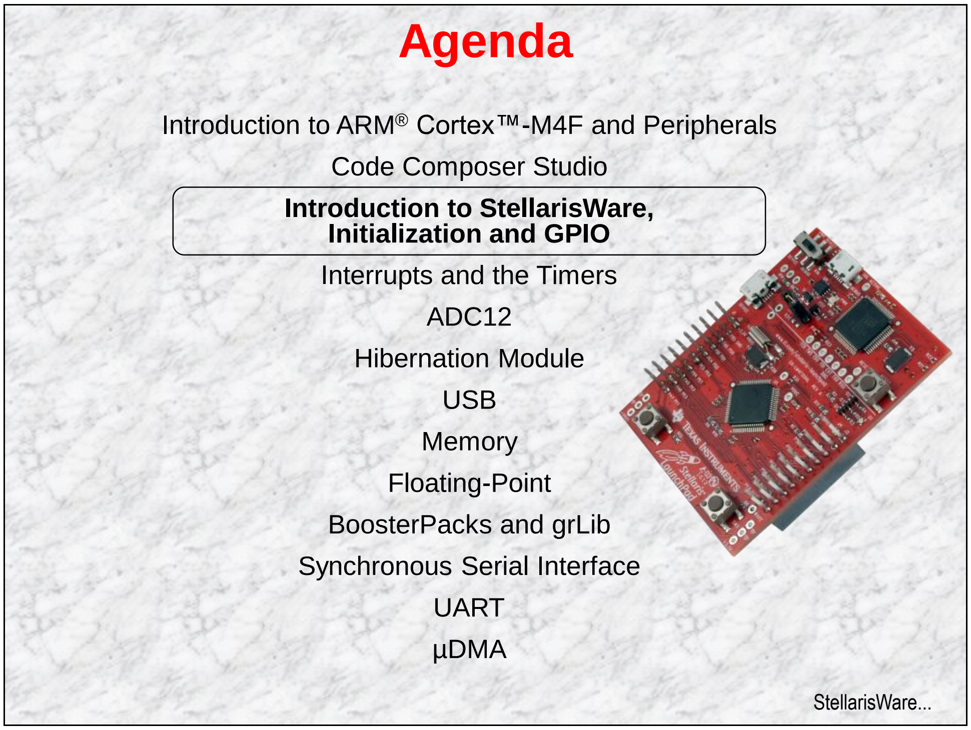
StellarisWare, Initialization and GPIO

Introduction

This chapter will introduce you to StellarisWare. The lab exercise uses StellarisWare API functions to set up the clock, and to configure and write to the GPIO port.



Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to StellarisWare, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory

Floating-Point

BoosterPacks and grLib

Synchronous Serial Interface

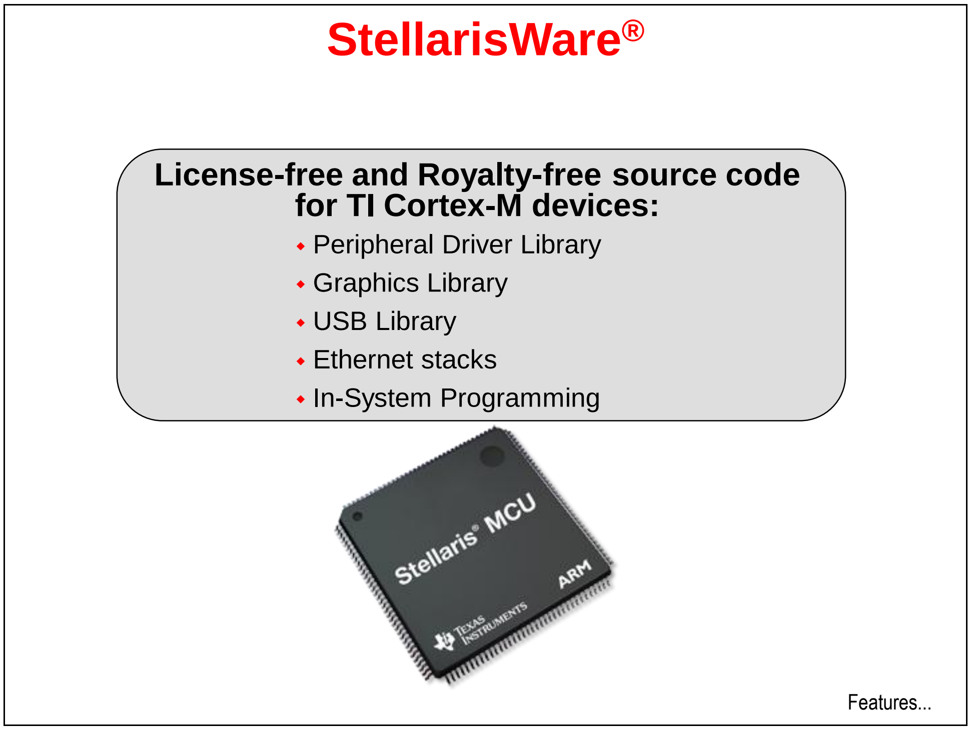
UART

µDMA

StellarisWare...

Chapter Topics

StellarisWare



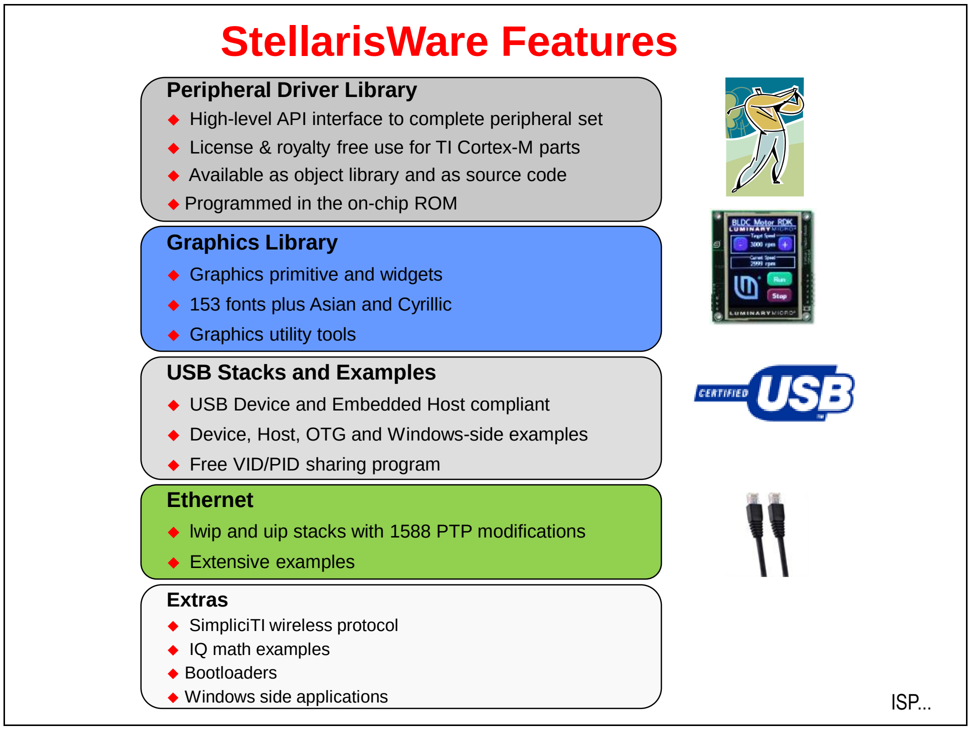
StellarisWare®

License-free and Royalty-free source code for TI Cortex-M devices:

- Peripheral Driver Library
- Graphics Library
- USB Library
- Ethernet stacks
- In-System Programming

Stellaris® MCU
ARM
TEXAS INSTRUMENTS

Features...



StellarisWare 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 in the on-chip ROM

Graphics Library

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

USB Stacks and Examples

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

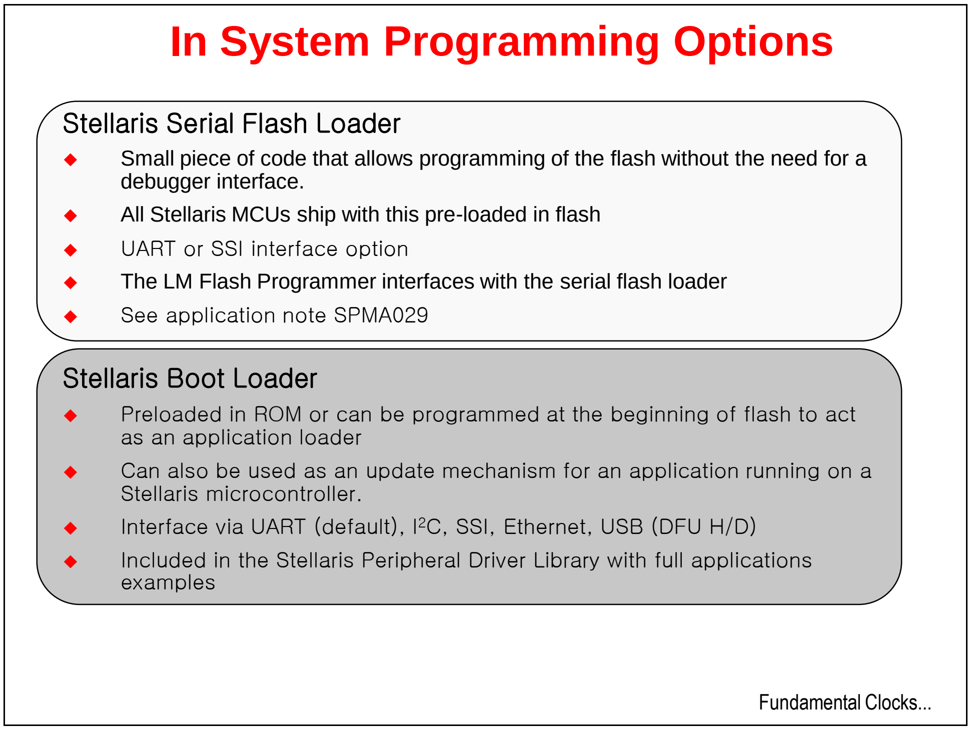
Ethernet

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

Extras

- SimpliciTI wireless protocol
- IQ math examples
- Bootloaders
- Windows side applications

ISP...



In System Programming Options

Stellaris Serial Flash Loader

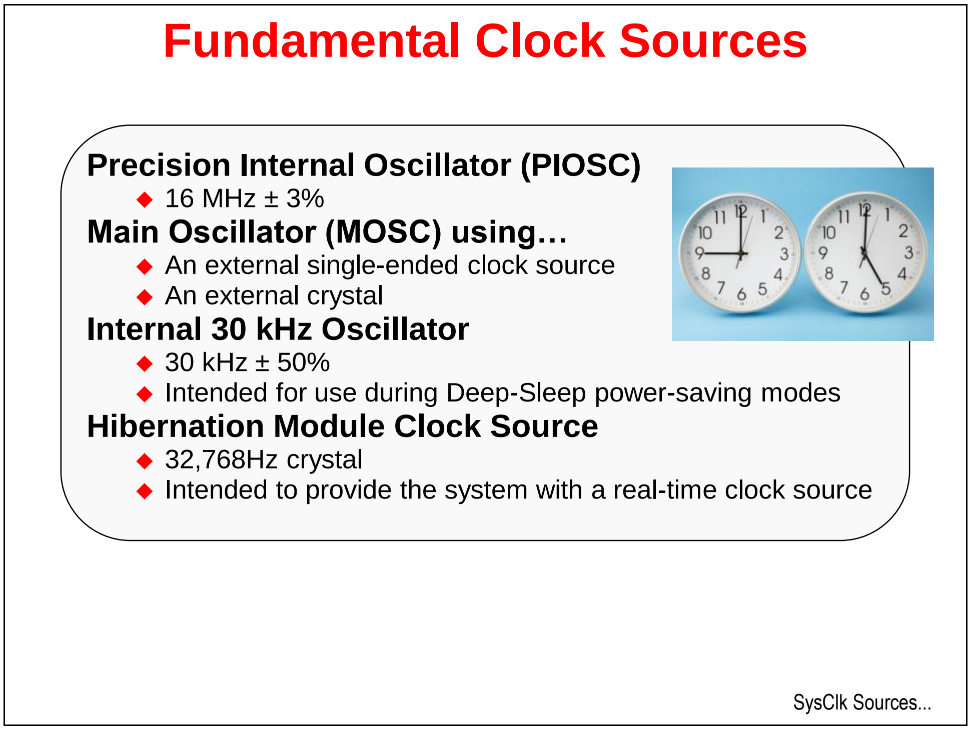
- Small piece of code that allows programming of the flash without the need for a debugger interface.
- All Stellaris MCUs ship with this pre-loaded in flash
- UART or SSI interface option
- The LM Flash Programmer interfaces with the serial flash loader
- See application note SPMA029

Stellaris 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 Stellaris microcontroller.
- Interface via UART (default), I^2C, SSI, Ethernet, USB (DFU H/D)
- Included in the Stellaris 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

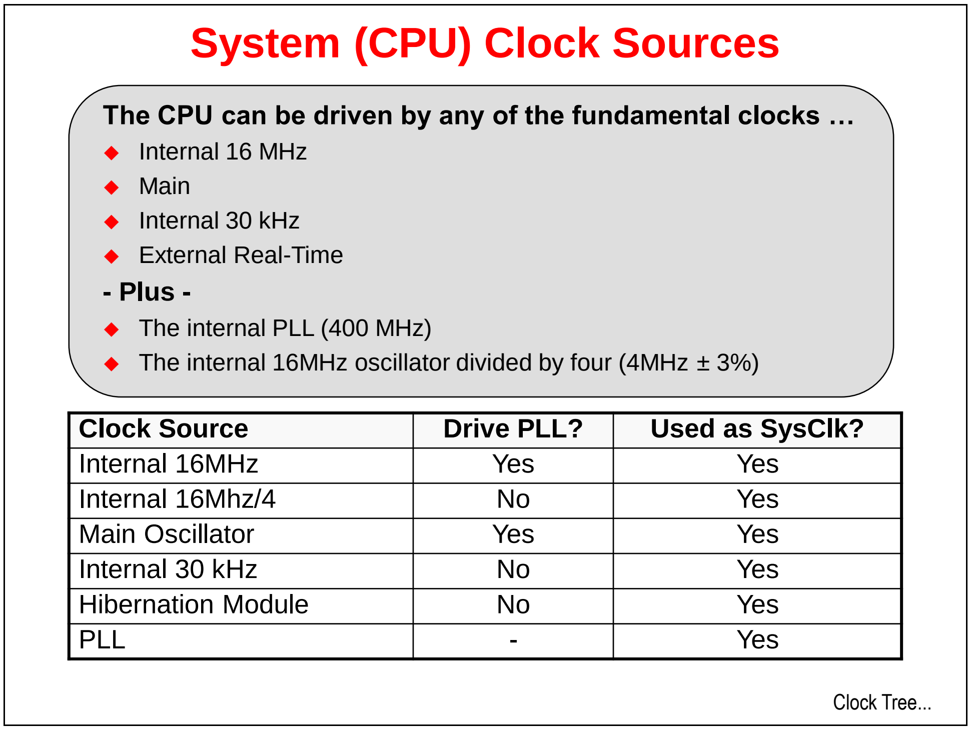
Internal 30 kHz Oscillator

- 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

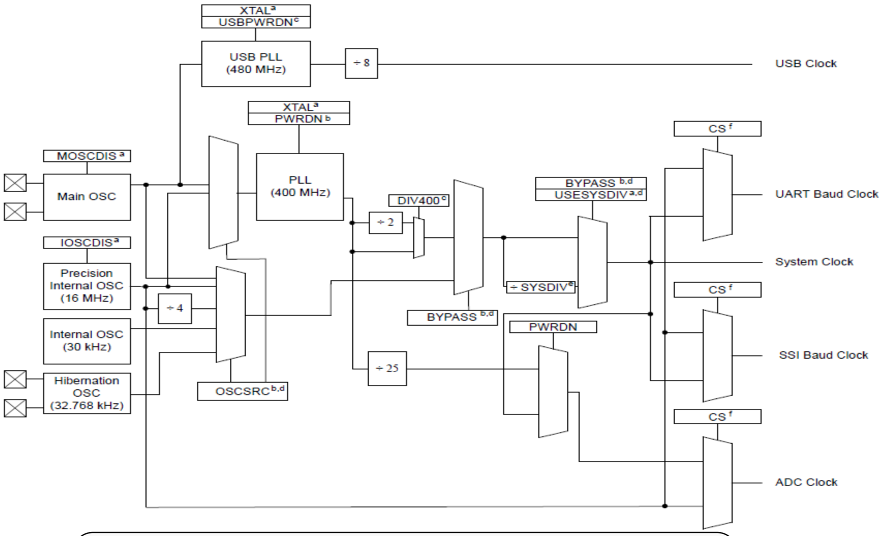
- Plus -

- 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...

Stellaris Clock Tree

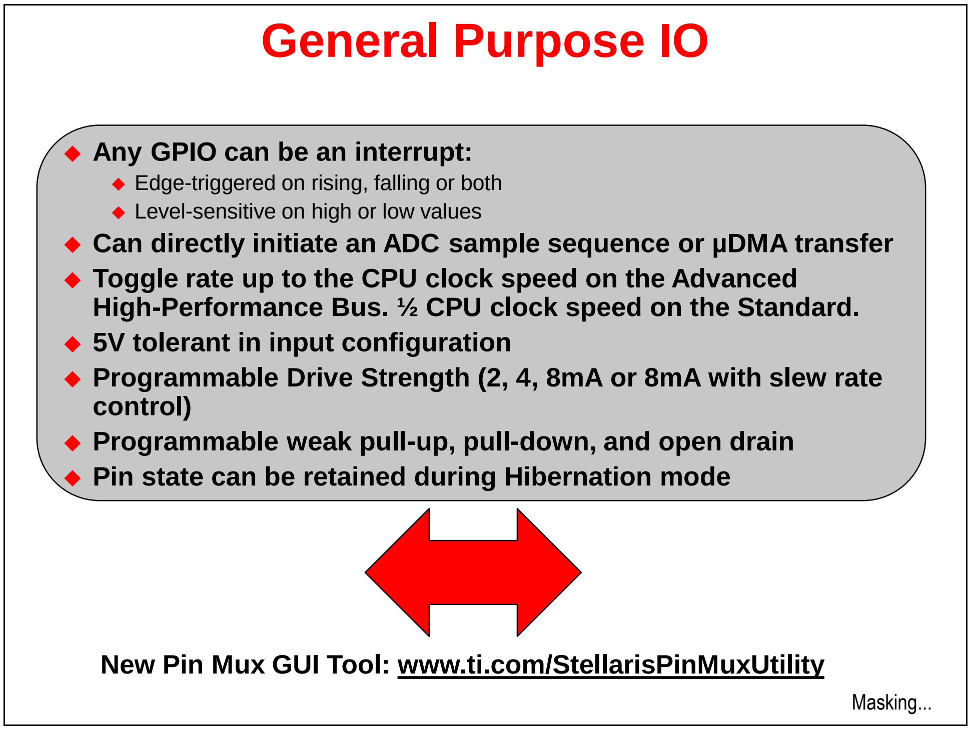


driverLib API SysCtlClockSet() selects:

- SYSDIV divider setting
- OSC or PLL
- Main or Internal oscillator
- Crystal frequency

GPIO...

GPIO



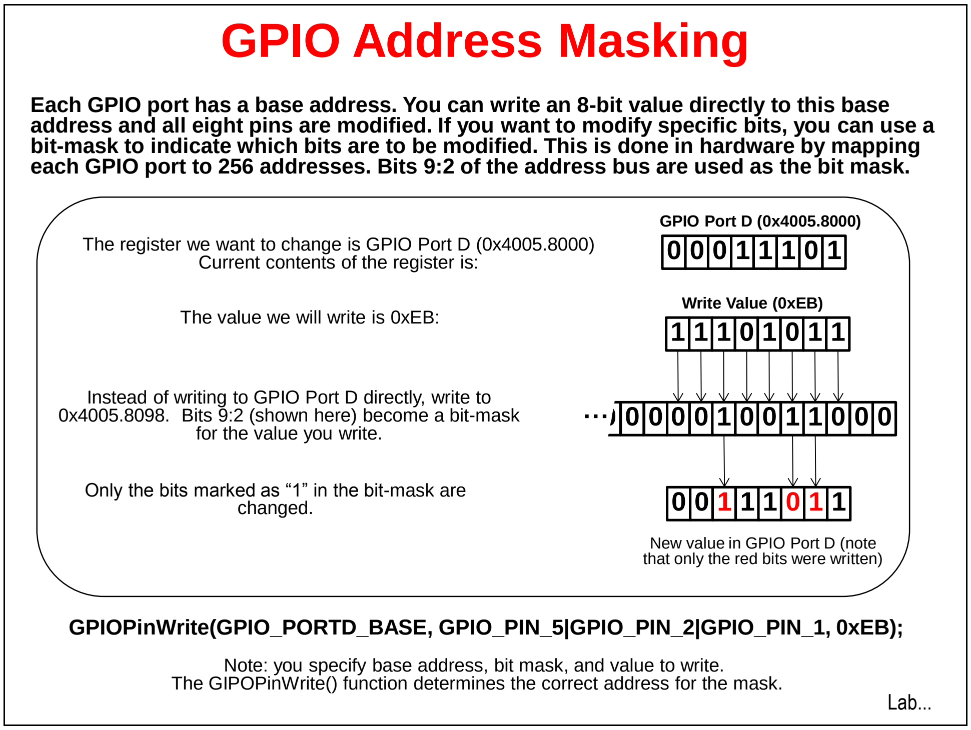
General Purpose IO

- 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

New Pin Mux GUI Tool: www.ti.com/StellarisPinMuxUtility

Masking...

www.ti.com/StellarisPinMuxUtility



GPIO Address Masking

Each GPIO port has a base address. You can write an 8-bit value directly to this base address and all eight pins are modified. If you want to modify specific bits, you can use a bit-mask to indicate which bits are to be modified. This is done in hardware by mapping each GPIO port to 256 addresses. Bits 9:2 of the address bus are used as the bit mask.

The register we want to change is GPIO Port D (0x4005.8000)
Current contents of the register is:

GPIO Port D (0x4005.8000)

0 0 0 1 1 1 0 1

The value we will write is 0xEB:

Write Value (0xEB)

1 1 1 0 1 0 1 1

Instead of writing to GPIO Port D directly, write to 0x4005.8098. Bits 9:2 (shown here) become a bit-mask for the value you write.

…0 0 0 0 1 0 0 1 1 0 0 0

Only the bits marked as "1" in the bit-mask are changed.

0 0 1 1 1 0 1 1

New value in GPIO Port D (note that only the red bits were written)

GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_5|GPIO_PIN_2|GPIO_PIN_1, 0xEB);

Note: you specify base address, bit mask, and value to write.
The GIPOPinWrite() function determines the correct address for the mask.

Lab...

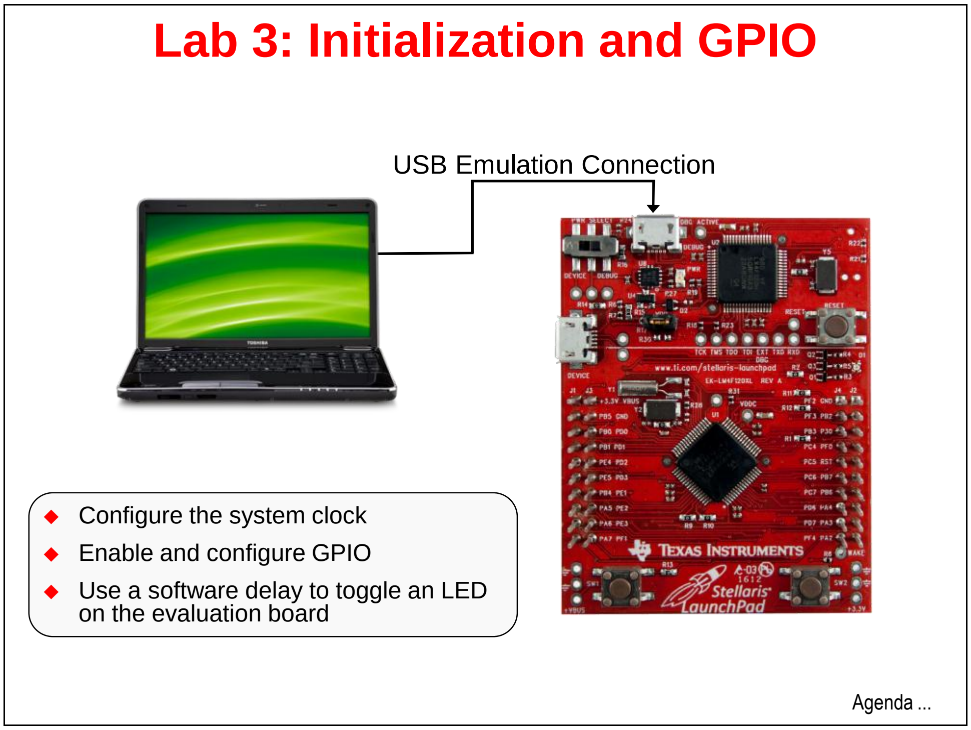
The masking technique used on ARM Cortex-M GPIO is similar to the "bit-banding" technique used in memory. To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the **GPIO Data (GPIODATA)** register by using bits [9:2] of the address bus as a mask. In this 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.

.

Lab 3: Initialization and GPIO

Objective

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



Lab 3: Initialization and GPIO

USB Emulation Connection

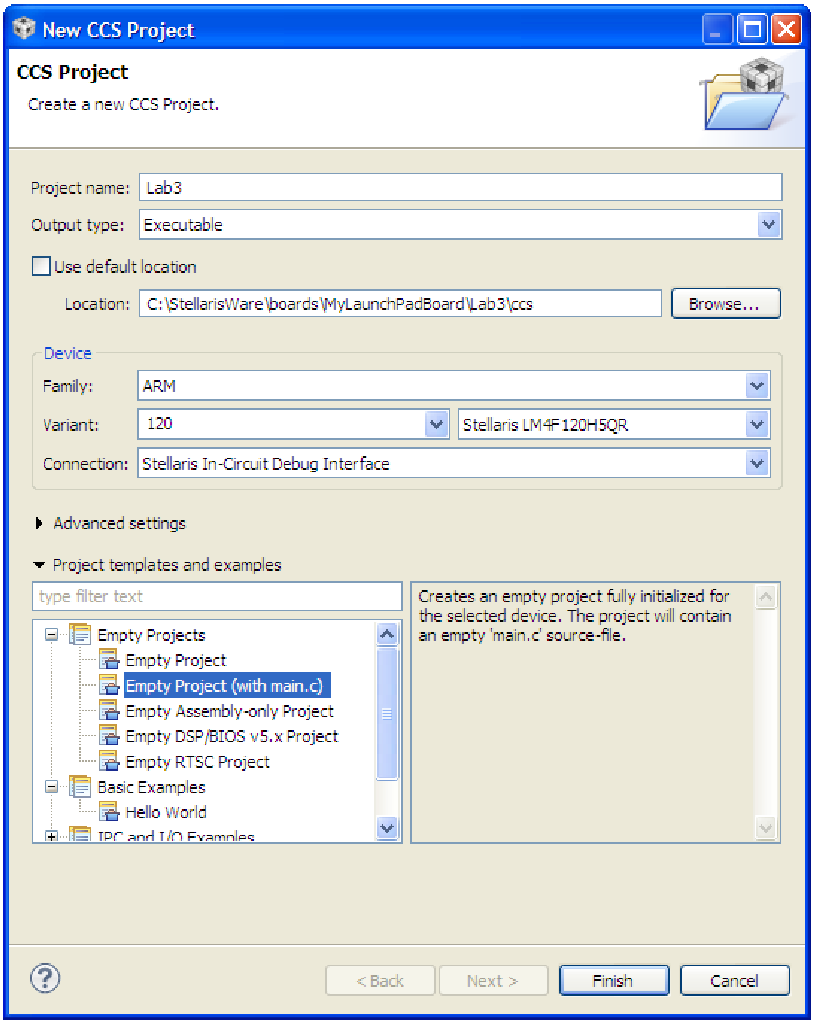
- Configure the system clock
- Enable and configure GPIO
- Use a software delay to toggle an LED on the evaluation board

Agenda ...

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 “ccs” folder you created. **This step is important to make your project portable and in order for the include paths to work correctly.** In the variant box, just type “120” to narrow the results in the right-hand box. In the Project templates and examples window, select Empty Project (with main.c). Click Finish.



New CCS Project

CCS Project

Create a new CCS Project.

Project name: Lab3

Output type: Executable

☐ Use default location

Location: C:\StellarisWare\boards\MyLaunchPadBoard\Lab3\ccs

Device

Family: ARM

Variant: 120 | Stellaris LM4F120H5QR

Connection: Stellaris In-Circuit Debug Interface

▸ Advanced settings

▾ Project templates and examples

type filter text

- Empty Projects
  - Empty Project
  - Empty Project (with main.c)
  - Empty Assembly-only Project
  - Empty DSP/BIOS v5.x Project
  - Empty RTSC Project
- Basic Examples
  - Hello World
- IPC and I/O Examples

Creates an empty project fully initialized for the selected device. The project will contain an empty 'main.c' source-file.

When the wizard completes, close the Grace tab if it appears, then click the + or ▷ next to Lab3 in the Project Explorer pane to expand the project. Note that Code Composer has automatically added `main.c` file to your project. We placed `startup_ccs.c` in the folder beforehand, so it was automatically added to the project. We also placed a file called `main.txt` in the 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`. Type (or cut/paste from the pdf file) the following lines into `main.c` to include the header files needed to access the StellarisWare APIs as well as a variable definition:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

int PinData=2;
```

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

`hw_types.h` : Defines common types and macros such as tBoolean and HWREG(x).

`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.

`int PinData=2`; : Creates an integer variable called PinData and initializes it to 2. This will be used to cycle through the three LEDs, lighting them one at a time.

You will see question marks to the left of the include lines in `main.c` displayed in Code Composer. We have not yet defined the path to the include folders, so Code Composer can’t find them. We’ll fix this later.

Main() Function

3. Next, we’ll drop in a template for our main function. Leave a line for spacing and add this code after the previous declarations:

```
int main(void)

{

}
```

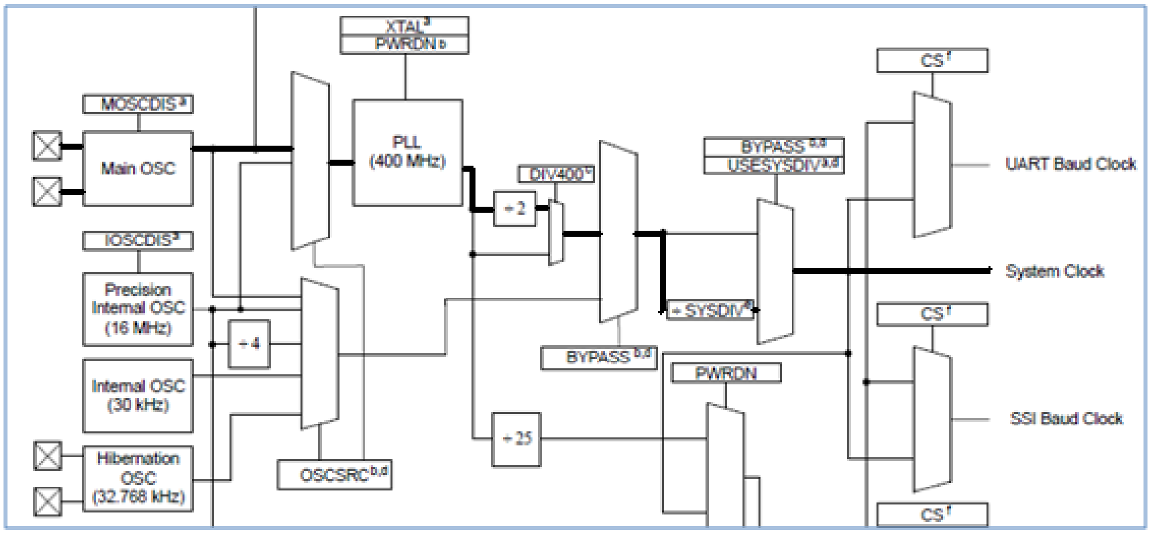
If you type this in, notice that the editor will add the closing brace when you add the opening one. Why wasn’t this thought of sooner?

Clock Setup

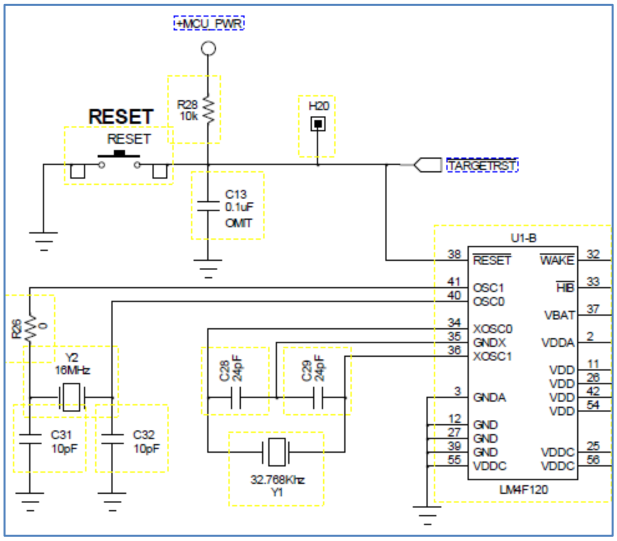
4. Configure the system clock to run using a 16MHz crystal on the main oscillator, driving the 400MHz PLL. The 400MHz PLL oscillates at only that frequency, but can be driven by crystals or oscillators running between 5 and 25MHz. There is a default /2 divider in the clock path and we are specifying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

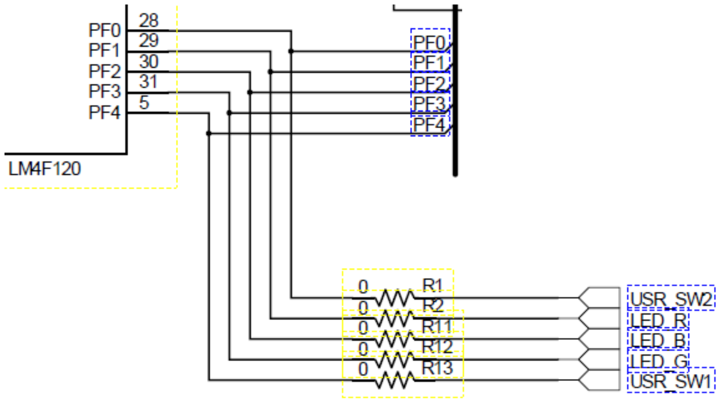


The diagram below is an excerpt from the LaunchPad board schematic. Note that the crystal attached to the main oscillator inputs is 16MHz, while the crystal attached to the real-time clock (RTC) inputs is 32,768Hz.



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 Stellaris 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 StellarisWare. The count parameter is the loop count, not the actual delay in clock 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 PinData 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 PinData 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 Stellaris 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, PinData);

       SysCtlDelay(2000000);

       GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);

       SysCtlDelay(2000000);

       if(PinData==8) {PinData=2;} else {PinData=PinData*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**. Also 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 "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
int PinData=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, PinData);
                SysCtlDelay(2000000);
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
                SysCtlDelay(2000000);
                if(PinData==8) {PinData=2;} else {PinData=PinData*2;}
        }
}
```

Sorry about the small font here, but any larger font made the `SysCtlClockSet()` instruction look strange. 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 Lab3/ccs folder.

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 add the start up code and set our build options.

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. We included this file in your folder.

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

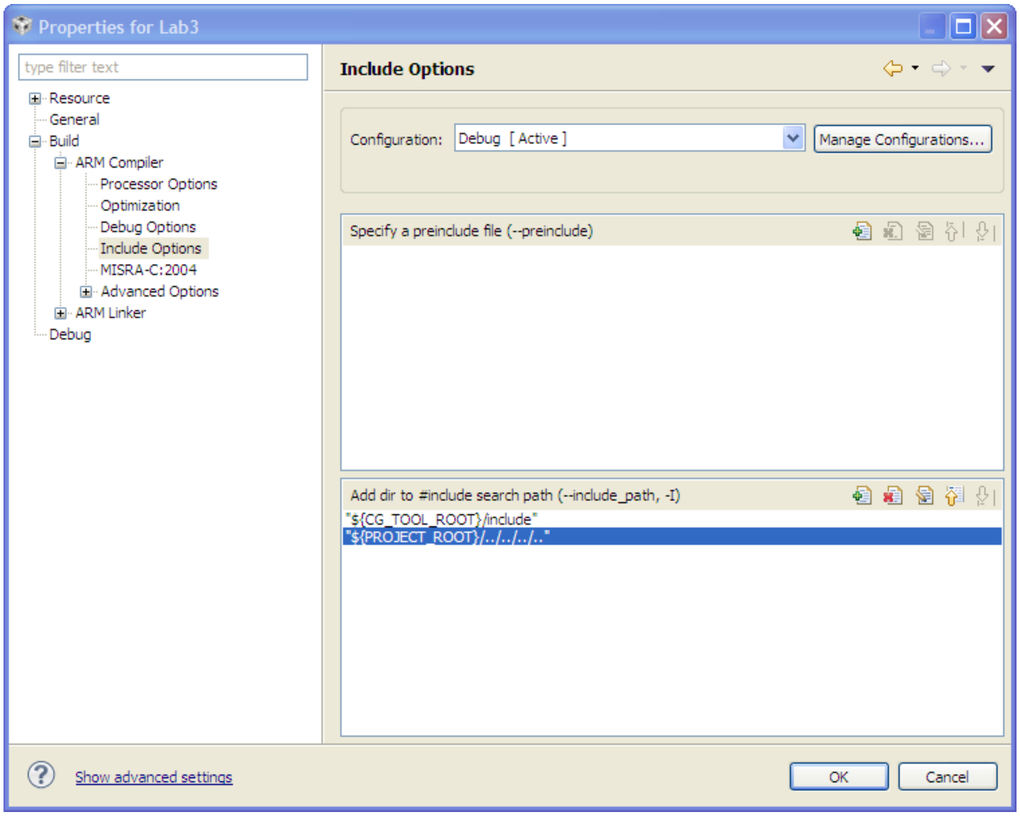
Set the Build Options

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:

   ${PROJECT_ROOT}/../../../..

   If you followed the instructions when you created the Lab3 project, this path, 4 levels above the project folder, will give your project access to the `inc` and `driverlib` folders. Otherwise you will have to adjust the path. You can check it for yourself using Windows Explorer.

   Avoid typing errors and copy/paste from the workbook pdf for this and the next step.



Include Options

Configuration: Debug [Active]

Specify a preinclude file (--preinclude)

Add dir to #include search path (--include_path, -I)

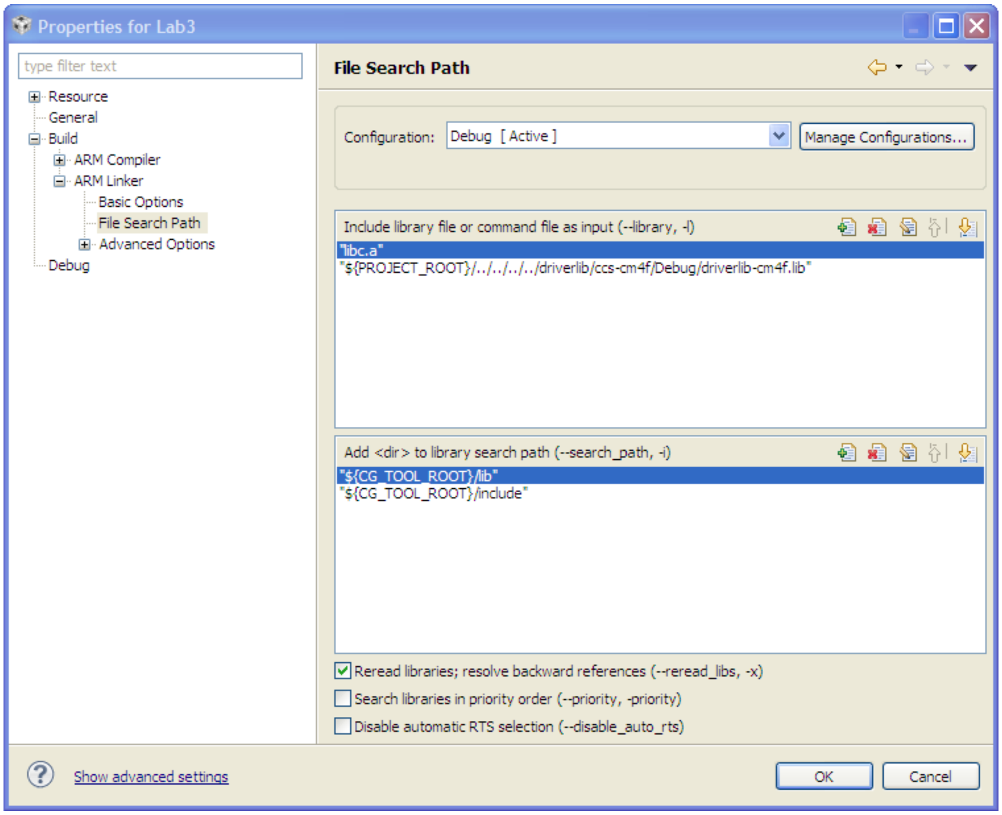
"${CG_TOOL_ROOT}/include"
"${PROJECT_ROOT}/../../../.."

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. Right-click on Lab3 again in the Project Explorer pane and select Properties. Under **ARM Linker,** click **File Search Path**. We need to provide the project with the path to the M4F libraries. Add the following include library file to the top window:

${PROJECT_ROOT}/../../../../driverlib/ccs-cm4f/Debug/driverlib-cm4f.lib

Of course, if you did not follow the directions when creating the Lab3 project, this path will have to be adjusted like the previous one.



Configuration: Debug [Active]

Include library file or command file as input (--library, -l)

"libc.a"
"${PROJECT_ROOT}/../../../../driverlib/ccs-cm4f/Debug/driverlib-cm4f.lib"

Add <dir> to library search path (--search_path, -i)

"${CG_TOOL_ROOT}/lib"
"${CG_TOOL_ROOT}/include"

☑ Reread libraries; resolve backward references (--reread_libs, -x)

□ Search libraries in priority order (--priority, -priority)

□ Disable automatic RTS selection (--disable_auto_rts)

Click OK to save your changes.

Compile, Download and Run the Code

11. 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 are playing around with the code and get the message “*No source available for … ”,* 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.

Examine the Stellaris Pin Masking Feature

Note that the following steps differ slightly from the workshop video.

12. Let's change the code so that all three LEDs are on all the time. Make the following changes:

    Find the line containing `int PinData=2;` and change it to `int PinData=14;`

    Find the line containing `if(PinData` … and comment it out by adding `//` to the start of the line.

    Save your changes.

13. Compile and download your application by clicking the Debug button on the menu bar. Click the Resume button to run the code. With all three LEDs being lit at the same time, you should see them flashing an almost white color.

14. Now let's use the pin masking feature to light the LEDs one at the time. We don't have to go back to the CCS Edit perspective to edit the code. We can do it right here. In the code window, look at the first line containing `GPIOPinWrite().` The pin mask here is `GPIO_PIN_1| GPIO_PIN_2| GPIO_PIN_3`, meaning that all three of these bit positions, corresponding to the positions of the LED will be sent to the GPIO port. Change the bit mask to `GPIO_PIN_1`. The line should look like this:

    ```
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, PinData);
    ```

15. Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click OK. When you are asked if you want to terminate the debug sessions, click Yes.

    Before clicking the Resume button, predict which LED you expect to flash: _________

    Click the Resume button . If you predicted red, you were correct.

16. In the code window, change the first GPIOPinWrite() line to:

    ```
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, PinData);
    ```

17. Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click OK. When you are asked if you want to terminate the debug sessions, click Yes.

    Before clicking the Resume button, predict which LED you expect to flash: _________

    Click the Resume button . If you predicted blue, you were correct.

18. In the code window, change the first GPIOPinWrite() line to:

    ```
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, PinData);
    ```

19. Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click OK. When you are asked if you want to terminate the debug sessions, click Yes.

    Before clicking the Resume button, predict which LED you expect to flash: _________

    Click the Resume button . If you predicted green, you were correct.

20. Change the code back to the original set up: Make the following changes:

    Find the line containing `int PinData14;` and change it to `int PinData=2;`

    Find the line containing `if(PinData` … and uncomment it

    Find the line containing the first GPIOPinWrite() and change it back to:

    ```
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1| GPIO_PIN_2| GPIO_PIN_3, PinData);
    ```

21. Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click OK. When you are asked if you want to terminate the debug sessions, click Yes. Click the Resume button and verify that the code works like it did before.

22. **Homework idea:** Look at the use of the `ButtonsPoll()` API call in the `qs-rgb.c` file in the quickstart application (qs-rgb) folder. Write code to use that API function to turn the LEDs on and off using the pushbuttons.

You're done.