

# Using GPIO with MSPWare

---

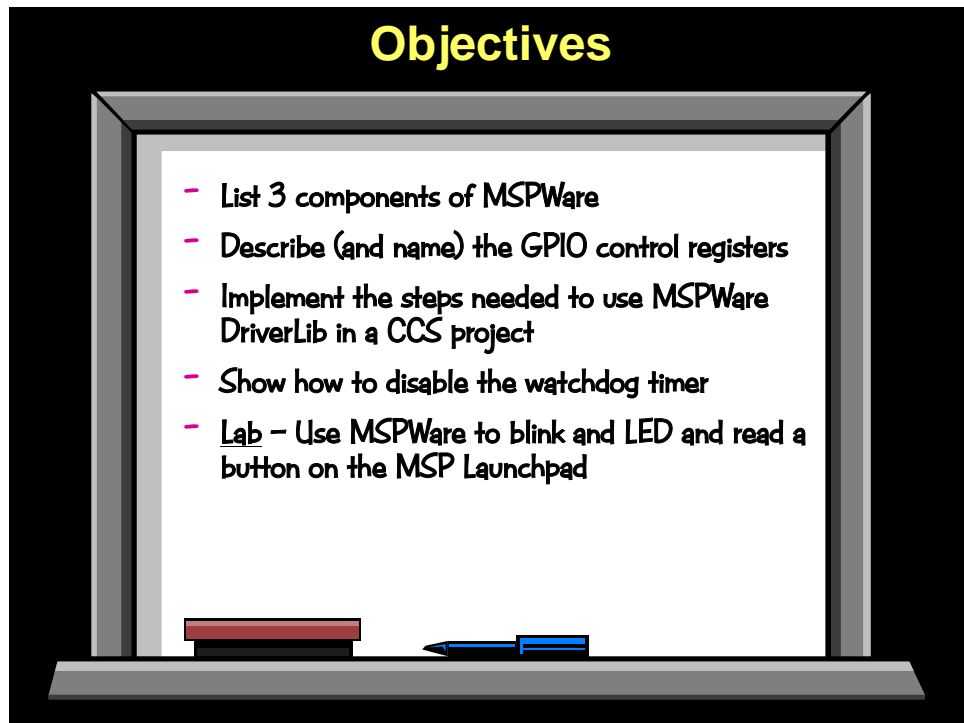
## Introduction

In the previous lab exercise, we blinked an LED on the MSP LaunchPad, but we didn't write the code – we were able to import a generic 'blink' template that ships with CCSstudio.

This chapter explores the GPIO (general purpose bit input/output) features of the MSP family. By examining the hardware operation of the I/O pins, as well as the registers that control them, we gain insight into the many ways we can utilize these features.

To make programming easier, we'll use the driver library (DriverLib) component of MSPWare. While not actually a set of "drivers" in the traditional sense, this library provides us the software tools to quickly build and deploy our own driver code for MSP devices.

## Learning Objectives

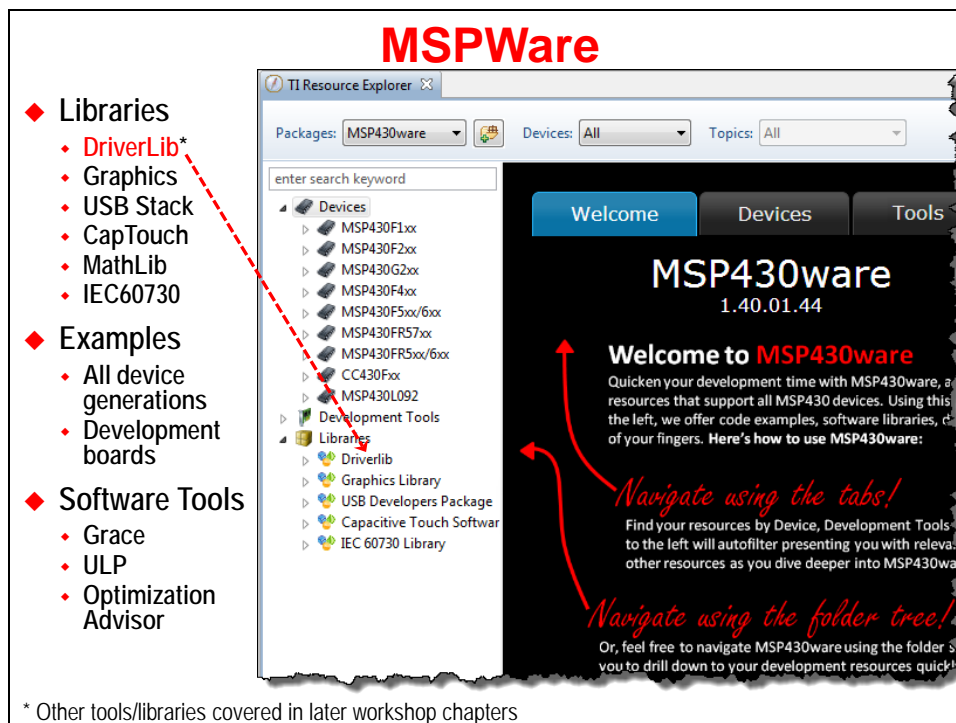


## Chapter Topics

<b>Using GPIO with MSPWare .....</b>	<b>3-1</b>
<i>MSPWare (DriverLib)</i> .....	3-3
Installing MSPWare.....	3-3
DriverLib.....	3-4
DriverLib Modules .....	3-5
Programming Methods – Summary .....	3-5
<i>MSP GPIO</i> .....	3-6
GPIO Basics.....	3-6
Input or Output .....	3-7
GPIO Output .....	3-8
GPIO Input .....	3-9
Drive Strength .....	3-10
Flexible Pin Usage (Muxing) .....	3-11
Pin Selection .....	3-12
Devices with Multiple Pin Selection Registers .....	3-13
Port Mapping .....	3-14
Summary .....	3-15
<i>Before We Get Started Coding</i> .....	3-17
1. #Include Files .....	3-17
2. Disable Watchdog Timer.....	3-18
3. Pin Unlocking (only required for FRAM devices) .....	3-19
<i>Lab 3</i> .....	3-21
Lab 3a – Blinking an LED.....	3-22
Abstract .....	3-22
GPIO Output Worksheet .....	3-23
MSPWare DriverLib .....	3-23
GPIO Output.....	3-23
Procedure.....	3-25
Add MSPWare DriverLib.....	3-27
Add ‘Blink’ Code to <code>main.c</code> .....	3-29
Debug.....	3-30
Lab 3b – Reading a Push Button .....	3-32
Abstract .....	3-32
GPIO Input Worksheet .....	3-32
Procedure.....	3-35
File Management .....	3-35
Copy code from the previous project .....	3-36
Add Setup Code... to reference the push button.....	3-37
Modify Loop.....	3-37
Verify Code.....	3-39
Optional Exercises .....	3-39
<i>Chapter 3 Appendix</i> .....	3-40
Manually Adding DriverLib to a MSP430 Project.....	3-40
Worksheet Answers .....	3-43

## MSPWare (DriverLib)

MSPWare is a bundle of Libraries, Examples and Tools supporting the MSP family of microcontrollers. To simplify the installation of all these elements, they have been bundled together into a single (.exe) file.



## Installing MSPWare

When you install MSPWare as part of CCSv6 – or from the stand-alone MSPWare installer downloaded directly from the TI website – it is, by default, installed to,

C:\ti\msp\MSPWare\_2\_21\_00\_39\

When MSPWare is updated, they increase the revisions numbers – for example, from 1\_60\_02\_09 to 1\_80\_01\_03. Note that it's possible that our lab exercises may show a slightly older version of the MSPWare libraries, if there was a release made after the workshop materials were updated.

To update MSPWare, you can use the auto-update feature of CCS. Alternatively, you can download the stand-alone installer from the [MSPWare](#) webpage.

## DriverLib

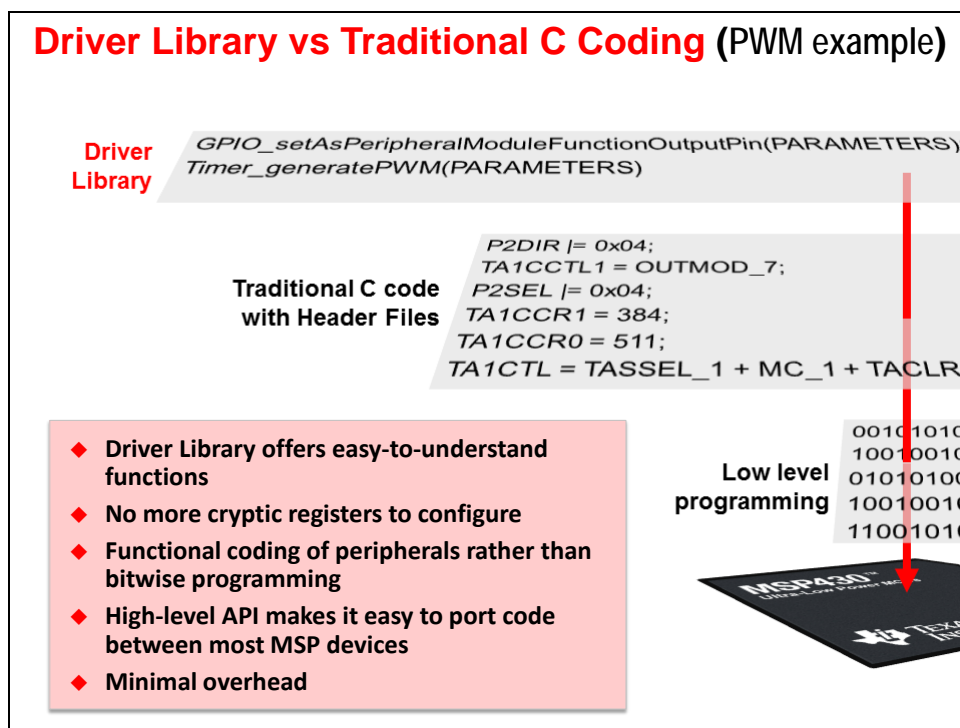
The MSPWare library used most often in this workshop will be the Driver Library – often called DriverLib.

To quote the DriverLib documentation (we couldn't have said this better ourselves):

*The Texas Instruments® MSP® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the MSP430F5xx/6xx family of microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.*

While we recommend that you read the entire “Introduction” in the DriverLib users guide (look in the “doc” folder within the DriverLib folder), but this statement does a good job stating the intent of the driver library.

In the following graphic, you can see that the Driver Library provides a convenient way to program the MSP peripherals in an easy-to-read (hence easy-to-maintain).



In the previous chapter, we showed you the method of “Traditional Register-Level C code with Header Files”. In a few rare cases, you might still want to use the Register symbols; in fact, DriverLib itself utilizes some of these symbols, so they are both compatible with each other.

This said, the convenience of DriverLib's API easily makes it the most desirable method of programming MSP peripherals.

*On a side note, you might remember a similar diagram (to that above) from the previous chapter. One big difference is that the previous diagram had an additional, higher-level layer called Energia. Energia (the Arduino port for TI microcontrollers) provides a convenient, portable means of programming MSP peripherals – in fact, it's even easier to use than DriverLib. Once again, you can even mix the three programming paradigms. For some, this is a godsend, for others, it's one abstraction layer too much; therefore, most of the chapters in this workshop will focus on DriverLib. Please check out the “Energia” chapter, though, if you're interested in using the Arduino port for rapid prototyping (or production coding).*

## DriverLib Modules

For the most part, DriverLib is organized by peripheral modules. Each peripheral has its own API specification; this provides good modularity and makes it easy to reuse peripheral code across devices whose peripherals are in common. There are cases where one module may rely on another, but in most cases they are independent API sets.

MSPWare DriverLib Modules						
Basics & Clocking	Memory	Analog	Power	Timing	Accelerators	I/O
CS	FLASH	ADC10	PMM	TIMER_A	AES	GPIO
USC	FRAM	ADC12	BATT	TIMER_B	CRC	PM
SFR	RAM	SD24	LDO	TIMER_D	DMA	SPI
SYS		COMP		WDT	MPY32	I2C
TLV		REF		RTC		UART
		DAC		TEC		

Used in this chapter

- Software modules tend to match 1-to-1 with hardware peripherals
- Some of the module names above have been abbreviated
- Not all devices have all hardware (and thus, software) modules
- DriverLib is not currently available for MSP F1x, F2x/4x, or G2x devices

## Programming Methods – Summary

Over the past two chapters we have introduced five ways to program the MSP. They are listed below along with the chapters (and courses) they are discussed in.

Name 5 ways to program GPIO:

1. **Register Layer (device specific) header files (.h/.cmd)** Ch2
2. **MSPWare DriverLib** Ch3
3. **Energia** Ch11
4. **Grace graphical driverlib tool** (Value-line and FR58/59xx devices) \*  
\*see Chapter 8 in the "G2553 Value-Line Launchpad Workshop"
5. **TI-RTOS GPIO Driver**

\* [http://processors.wiki.ti.com/index.php/Getting\\_Started\\_with\\_the\\_MSP430G2553\\_Value-Line\\_LaunchPad\\_Workshop](http://processors.wiki.ti.com/index.php/Getting_Started_with_the_MSP430G2553_Value-Line_LaunchPad_Workshop)

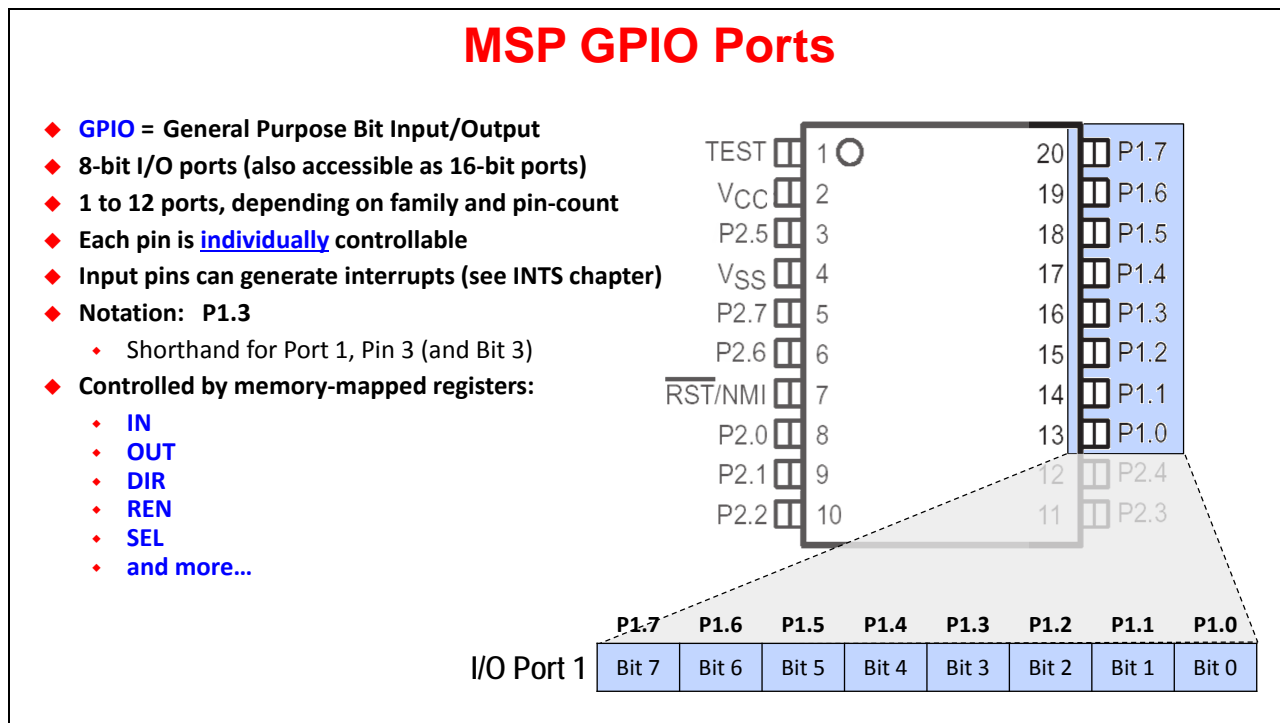
\*\* The [TI Cloud](#) tools now include a "PinMux" GUI tool which allows you to generate a GPIO initialization function. At the current time, this tool only supports the MSP432, with plans in 2016 to expand it to the full MSP line of MCU's.

# MSP GPIO

## GPIO Basics

General Purpose Bit Input/Output (GPIO) provides a means of controlling – or observing – pin values on the microcontroller. This is the most basic service provided by processors.

The MSP provides one or more 8-bit I/O ports. The number of ports is often correlated to the number of pins on the device – more pins, more I/O. The I/O port bits (and their related pins) are enumerated with a Port number, along with the bit/pin number; for example, the first pin of Port 1 is called: P1.0.



Why did we say pin/bit number? Each I/O pin is individually controllable via CPU registers. For example, if we want to read the input value on P1.0, we can look in bit 0 of the register called P1IN. There are a number of registers used to view and control the I/O pins. In this chapter we'll examine most of them; though, a few – such as those related to interrupts – will be explored in a later chapter.

**Note:** As mentioned in the previous paragraph, many GPIO pins can be used to trigger interrupts to the CPU. The number of pins that support interrupts depends upon which device you're using. Most devices support interrupts with Ports 1 and 2, but make sure you reference your device's datasheet.

## Input or Output

Each GPIO pin is individually controllable – that is, you can configure each pin to be either an input or an output. This is managed via the DIR register; for example, to set P1.7 to be an output you would need to set P1DIR.7 = 1 (as shown below).

### Pin Direction (PxDIR): Input or Output

**Direction Register (DIR)**

- ◆ Each bit in (DIR) specifies whether associated pin is an input or output
- ◆ Notation: **PxDIR.y**  
Shorthand for Port “x” and Pin “y”
- ◆ Bit Values: **0 = input**  
**1 = output**

**Code Example**  
For Port 1, set Pin 0 and 7 as outputs

- ◆ Register-Level:

```
P1DIR |= 0x81;
```
- ◆ MSPware DriverLib :

```
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 + GPIO_PIN7 );
```
- ◆ Energia

```
pinMode( P1_0, OUTPUT ); // Like Arduino, all pins are enumerated 1 thru XX
pinMode( P1_7, OUTPUT ); // which could be used in place of Px_y syntax
```

	7	6	5	4	3	2	1	0
P1IN								
P1OUT								
P1DIR	1	x	x	x	x	x	x	1

Remember that we had multiple programming methodologies? Our graphic above shows us three of them.

- You'll see the "Register Level" example above uses C code to set the P1DIR register to a given hex value.
- On the other hand, the "MSPWare DriverLib" function allows you to set one or more pins of a given port as an output. (By the way, to set up the pin as an input, you would use one of the `GPIO_setAsInputPin()` functions.)
- An Energia example is also shown. This library provides configures GPIO using a `pinMode()` function. (This function, as you might expect, matches the Arduino pin initialization function.)

These three methods end up setting the same registers to the same bit values. The advantage of DriverLib (or Energia) is that you won't have to re-lookup the register names if you should ever have to read or tweak the code in the future.

---

**Note:** As stated earlier in the chapter, the GRACE tool is discussed in its own chapter – found in the [G2553 LaunchPad Workshop](#).

---

With the direction configured you will either use the respective IN or OUT register to view or set the pin value (as we'll see on the next couple pages). We'll look at these next.

## GPIO Output

Once you've configured a pin as an output with the PxDIR register, you can set the pin's value using the PxOUT register. For P1.7, this would be the P1OUT register.

### Output Register (OUT)

- ◆ When DIR.bit is 1, the pin reflects the same bit value in the OUT register
- ◆ Notation: **PxOUT.y**  
Shorthand for Port "x" and Pin "y"
- ◆ Bit Values:   0 = low  
                  1 = high

### Code Example

Output a "1" (high) on P1.7

- ◆ Register-Level:
 

`P1OUT |= 0x80;`
- ◆ MSPware DriverLib :
 

`GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN7 );`
- ◆ Energia
 

`digitalWrite( P1_7, HIGH );`

## GPIO Output

	7	6	5	4	3	2	1	0
P1IN	x							
P1OUT	1							
P1DIR	1							

Once again, the DriverLib `GPIO_setOutputHighOnPin()` or `GPIO_SetOutputLowOnPin()` functions are an easy way to write to the PxOUT registers. You can set multiple pins/bits by or'ing (+) them together (similar to the P1DIR example on the previous page).



## GPIO Input

Reading a pin's value is done by reading the PxIN register. The `GPIO_getInputValue()` DriverLib function returns this value to a variable in your program.

### Background

- ◆ Input pins are held in high-impedance (Hi-Z) state, so they can react to an incoming 0 or 1 signal
- ◆ When not driven, Hi-Z inputs may float up/down ... prevent this with pullup or pulldown resistors
- ◆ Resistor ENable (REN) register turns on resistors; OUT selects up/down

### Bit Values

REN	OUT	If Input Pin (DIR = 0)
1	0	Pull-down resistor
1	1	Pull-up resistor
0	x	Resistors disabled

### Code Example

Use P1.7 as input (with pull-up resistor)

◆ **Register-Level:**

```
P1DIR |= 0x80;
P1REN |= 0x80;
P1OUT &= 0x80;
value = P1IN & 0x80;
```

## GPIO Input (Resistors)

◆ **Register-Level:**

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P1IN	x							
P1OUT	1							
P1DIR	0							
P1REN	1							

◆ **Energia:**

```
pinMode(buttonPin, INPUT_PULLUP);
value = digitalRead(buttonPin);
```

◆ **MSPware DriverLib :**

```
unsigned short usiButton = 0;
GPIO_setAsInputPinWithPullUpResistor( GPIO_PORT_P1, GPIO_PIN7);
usiButton = GPIO_getInputPinValue( GPIO_PORT_P1, GPIO_PIN7);
```

Input pins are slightly more complicated than output pins. While the PxDIR function selects whether a pin is used for an input or output, your input pin may need further configuration.

When using a pin as an input, what value does the pin have when it is not being driven by external hardware? Unfortunately, when not being driven, an input pin 'floats' – that is, it can change state arbitrarily. Not only is this undesirable from a logical point of view, but even worse, power is consumed every time the pin changes state. The common solution is to tie the pin high (or low) through a resistor. When driven, the external signal can override the weak pull-up (or pull-down); otherwise the resistor holds the input to a given value.

To minimize system cost and power, most MSP I/O ports provide internal pull-up and pull-down resistors. You can enable these resistors via the PxREN (Resistor ENable) register bits. When PxREN is used to enable these resistors, the associated PxOUT bit lets you choose whether the pull-up or pull-down resistor is enabled.

Of course, unless you are using Energia, we recommend configuring the pull-up or pull-down resistor using one of the following GPIO DriverLib functions:

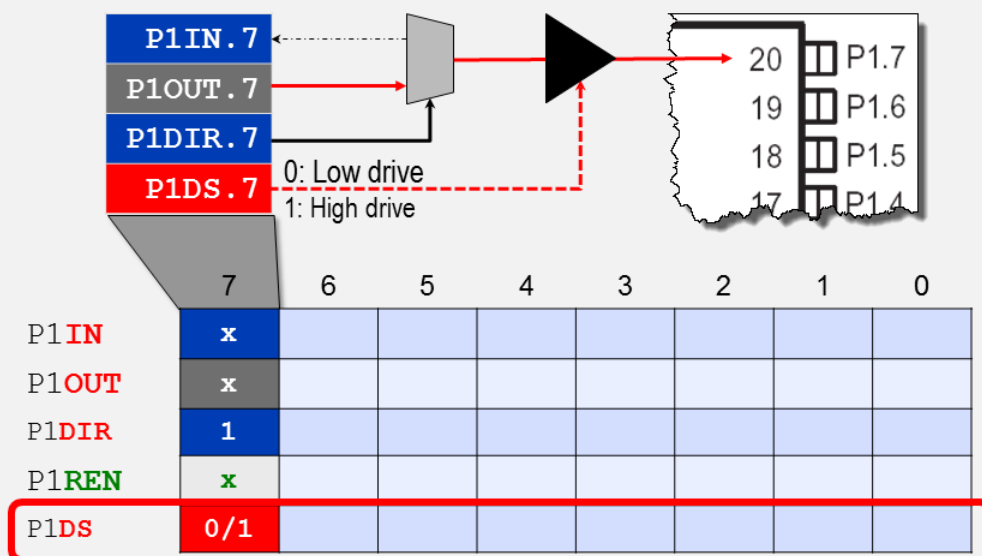
```
GPIO_setAsInputPin()
GPIO_setAsInputPinWithPullUpResistor()
GPIO_setAsInputPinWithPullDownResistor()
```

**Note:** Another feature of input pins is their ability to generate CPU interrupts. We won't cover those details in this chapter; rather, we'll save that discussion for the *Interrupts* chapter.

## Drive Strength

The F5xx/6xx series of MSP devices allow the designer to select whether they want outputs to be driven with lower or higher drive strength. The benefit of this extra feature is that it allows you to tune or power dissipation of your system. You can minimize the extra power usage of outputs when and where it is not needed.

### Output Drive Strength (F5xx/6xx only)



- ◆ F5xx (e.g. 'F5529) devices have individually programmable drive strength
- ◆ MSP430ware example:

```
GPIO_setDriveStrength( GPIO_PORT_P1, GPIO_PIN7 );
```

## Flexible Pin Usage (Muxing)

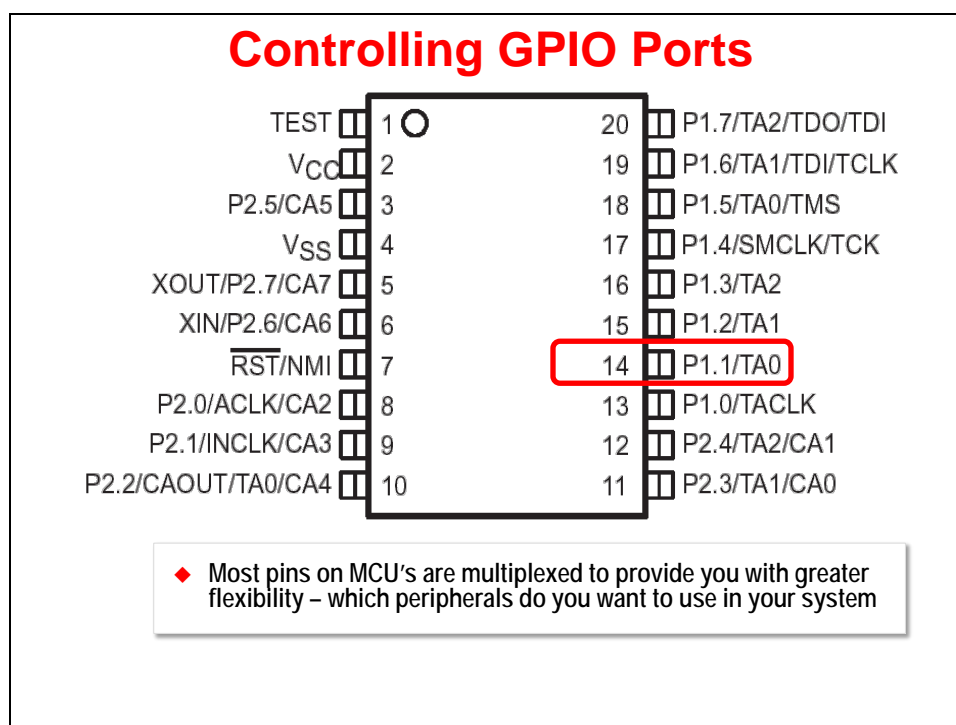
Microcontroller designers have to deal with two conflicting user needs:

*More Capability vs Lower Cost*

Users want as many features as possible on their processors; the more peripheral options, the better. For example, some users may want 2 serial ports, where others might need 4 I/O pins.

The more pins you add to a device, the greater the cost. (Not only does this make the device more expensive, but it adds to the overall board/system cost.) Therefore, if we added pins for every feature stuffed into our microcontrollers, the cost quickly becomes untenable.

The way this is managed is by 'muxing' different functions onto each pin. In other words, you can select which function you want to use for any given pin on the device. For example, looking at pin 14 in the following diagram, it can be used as either GPIO pin P1.1 or for Timer A0.



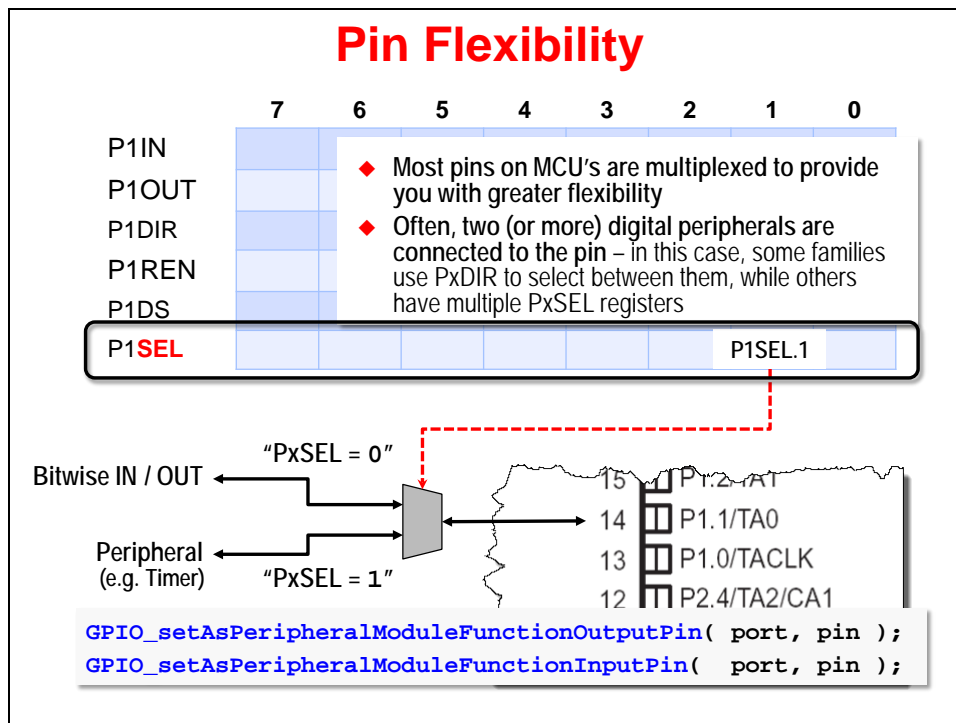
While these pin configurations can be changed at runtime, this isn't very common. The primary reason for this flexibility is so you can choose which features are needed for your specific system.

**Note:** Please do not select your specific device – or layout your board's hardware – before deciding which features are needed for your system.

If you have done microcontroller system design in the past, this is probably an obvious statement, but it's a mistake we've seen a number of times in the past.

## Pin Selection

The PxSEL register lets you choose whether to use the “peripheral” or “GPIO” functionality assigned to each pin. As you can see in the diagram below, DriverLib provides functions to specify this functionality.



## Devices with Multiple Pin Selection Registers

Some MSP devices actually have two *pin select* registers, as this lets them support a greater number of pin mux options.

**P1SEL0 & P2SEL1: FR5969 Example**

**Table 50. Port P1 (P1.0 to P1.2) Pin Functions (From the 'FR5969 datasheet')**

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS/SIGNALS <sup>(1)</sup>		
			P1DIR.x	P1SEL1.x	P1SEL0.x
P1.0/TA0.1/DMAE0/RTC CLK/A0/C0/VREF-/VREF-	0	P1.0 (I/O)	I: 0; O: 1	0	0
		TA0.CCI1A	0	0	1
		TA0.1	1	0	1
		DMAE0	0	1	0
		RTCCLK	1	1	0

**Table 8-2. I/O Function Selection (From the 'FR5969 User's Guide')**

PxSEL1	PxSEL0	I/O Function
0	0	General purpose I/O is selected
0	1	Primary module function is selected
1	0	Secondary module function is selected
1	1	Tertiary module function is selected

♦ The User's Guide tells us how to read the datasheet  
 ♦ DriverLib uses **I/O Function** name

```

// Set pin P1.0 to output TA0.1 (which is the CCR1 output signal for TIMER0_A)
GPIO_setAsPeripheralModuleFunctionOutputPin(
    GPIO_PORT_P1,           // I/O Port number
    GPIO_PIN0,              // Pin Number
    GPIO_PRIMARY_MODULE_FUNCTION); // Which peripheral function
                                // (primary, secondary, ternary)
  
```

**User Guide** (points to Table 8-2)

**Datasheet** (points to Table 50)

In the device *User's Guide*, they generically name the different peripheral I/O selections (first, second, and third) with the names:

- Primary
- Secondary
- Tertiary

Because the specific peripheral selections can vary from device-to-device, the detailed options are not described in the *User's Guide*, but rather in each device's *datasheet*. Unfortunately, though, the datasheets do not use the actual Primary/Secondary/Tertiary terminology. That said, you can match the PSEL bit values to figure this out. For example, on the 'FR5969 (in above diagram):

If P1DIR = 1, then TA0.1 is the **Primary** selection since P1SEL1.0:P1SEL2.0 = 01

Another way to say this is that because the *datasheet* shows that the TA0.1 PSEL values are "01", we know from the *User's Guide* that this correlates to the *Primary* function.

The DriverLib functions let you set both "Select" registers with one function call. This is done by adding a third argument in which to specify which I/O function you want to enable:

- GPIO\_PRIMARY\_MODULE\_FUNCTION
- GPIO\_SECONDARY\_MODULE\_FUNCTION
- GPIO\_TERNARY\_MODULE\_FUNCTION

You can see an example of this function in the above graphic.

To summarize – you can figure out which enumeration to use by comparing the selections from both the datasheet and user's guide. (In fact, 'FR5969 users will do this for the Timer chapter lab exercise.)

## Port Mapping

Many MSP devices support a “Port Mapping” feature (for example, the MSP430F5529 and MSP432P401R devices). This feature provides additional flexibility for mapping peripheral functions to pins. The signals that support port mapping are indicated with the `PM_` prefix.

Port Map (PM) Feature

- ◆ Pins which support Port Mapping can be connected to any of the “PM\_” signals
- ◆ Details:
  - `PM_`xxx = port-map capable signal
  - Datasheet specifies mappable ports
  - By default, single configuration per reset (PUC); though, runtime re-configuration is allowed
  - PM configuration is password protected
- ◆ Pin 16 Example: **P2.0 / PM\_UCA1STE**
  - P2.0 is muxed with Port-Mapped signal
  - Any PM signal can be mapped to this pin
  - UCA1STE (uart) is the default primary “peripheral” signal for Pin 16

**MSP432P401R LaunchPad**

P2.0 → Red LED  
P2.1 → Blue LED  
P2.2 → Green LED

16	<input type="checkbox"/> P2.0/PM_UCA1STE
17	<input type="checkbox"/> P2.1/PMUCA1CLK
18	<input type="checkbox"/> P2.2/PM_UCA1RXD
19	<input type="checkbox"/> P2.3/PM_UCA1TXD
20	<input type="checkbox"/> P2.4/PM_TA0.1
21	<input type="checkbox"/> P2.5/PM_TA0.2
22	<input type="checkbox"/> P2.6/PM_TA0.3
23	<input type="checkbox"/> P2.7/PM_TA0.4
24	<input type="checkbox"/> P10.4/TA3.0/CO.7
25	<input type="checkbox"/> P10.5/TA3.1/CO.6

**Example: Drive LED with Timer/PWM**

- LaunchPad uses PM pins to drive LED
- Config `PM_TA0.x` (Timer\_A) pins to the LED, which allows you to drive the LED with timer or PWM
- MSP432 DriverLib Example (see lab example “lab\_06c\_timerDirectDriveLed\_portmap”):

```

// The array will map TimerA0 output to Blue LED
uint8_t myP2map[] =
{
    PM_NONE,           // P2.0 (Red LED)
    PM_NONE,           // P2.1 (Green LED)
    PM_TA0CCR2A,        // P2.2 (Blue LED)
    PM_NONE, PM_NONE,
    PM_NONE, PM_NONE,
    PM_NONE
};

// Call port map fcn to remap pin P2.2 to TACCR0
MAP_PMAP_configurePorts (
    (uint8_t *) myP2map, // Pins to map
    P2MAP,               // Which port to remap
    1,                   // # of ports to cfg
    PMAP_DISABLE_RECONFIGURATION
);

```

The datasheet pin layout diagram, as shown above, provides the default pin mapping. In most cases, though, any signal with a `PM_` prefix can be mapped to any pin that supports a `PM_` signal. (I think of the PM peripheral as a built-in crossbar switch for pin signals.)

The above example shows a Timer\_A pin (TA0.2) being port mapped to pin 18. (Otherwise, by default, it only shows up on pin 21.)

During the Timer lab, MSP432 users can use the PM feature to route the timer signal directly to an LED on the LaunchPad – without needing a jumper wire.

## Summary

The following graphic summarizes the GPIO features (and nomenclature) across the MSP devices discussed throughout this workshop:

# GPIO Summary

	PA		PB		PC		PD		PE		PJ*	Reset Value (PUC)
	P1†	P2	P3	P4	P5	P6	P7	P8	P9	P10		
PxIN	<b>All LaunchPad Devices fully support Ports 1 and 2</b>		All Except 'G2553		F5529 (P8 x3-bits) FR4133 (P8 x12-bits) FR6989 MSP432				FR6989 MSP432		All Except 'G2553	undef
PxOUT												unchg
PxDIR												0x00
PxREN												0x00
PxDS												0x00
PxSEL												0x00
PxIV			FR5969, FR6989 and MSP432	MSP432P401R (only)						0x00		
PxIES										unchg		
PxIE	0x00											
PxIFG											0x00	

FR6989/P401R (100-pin)

F5529/FR4133 only (80-pin)

FR5969 only (48-pin)

G2553 only (20-pin)

FR6989 has 83 I/O; P401 has 78 I/O

◆ Each numbered port has 8 bits, unless noted otherwise

◆ At reset, all I/O pins are set as ... inputs

◆ You should initialize all pins (to prevent floating inputs)

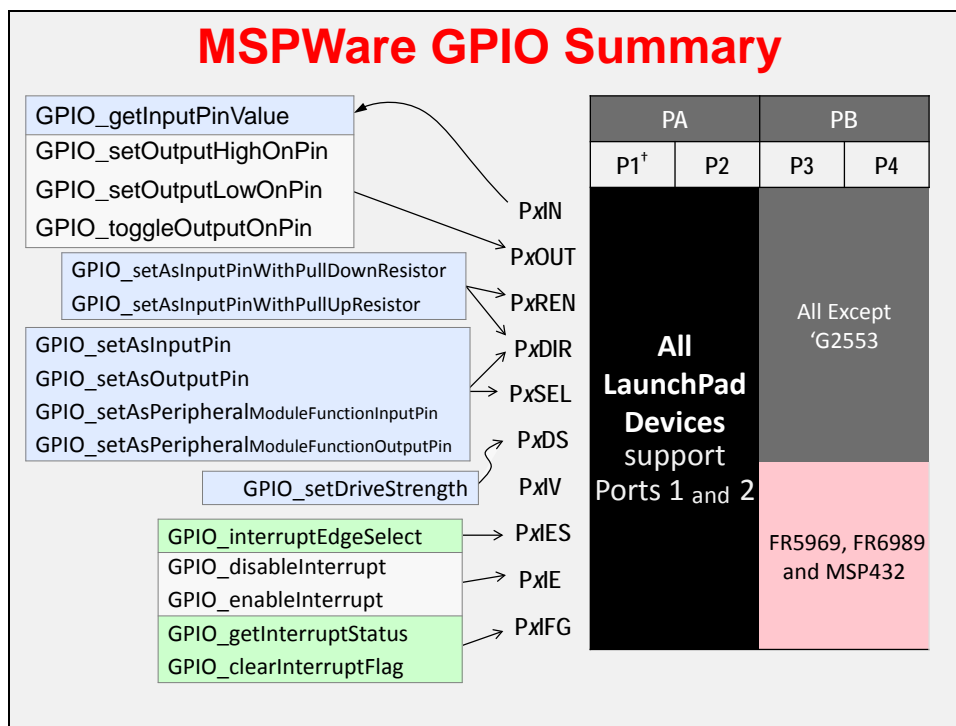
◆ Analog/LCD functions can 'preempt' digital pin selection

◆ PJ is shared with emulation pins (#pins is device dependent)

What can we deduce from this table?

- The various GPIO memory-mapped registers are listed down the first column. Most of these were described in the preceding discussion. (Interrupt registers are discussed in later. chapter.)
- All these devices (in fact, most all MSP devices) contain at least two 8-bit I/O ports (P1, P2) which provide the GPIO functionality – including interrupt inputs. We demonstrated this above by using the 'black' fill under ports P1 and P2; notice it covers every register's row.
- Alternatively, you can program ports 1 and 2 simultaneously by writing to port "PA". This means by writing to PAOUT, you can concurrently configure the outputs of all 16-pins.
- The 'G2553 Value-Line device only includes P1 and P2. (There just aren't enough pins on this device to support more I/O ports.)
- The 'FR5xx/6xx FRAM devices added interrupt support for PB (i.e. ports P3 & P4). MSP432 can support interrupts on all ports, though the 'P401R currently supports Ports 1 thru 6.
- Five (of our six) devices have enough pins to support ports P5 – P8. While two of them even support Ports P9 and P10. Note, though, that not every port may fully support all 8 pins; for example, on the 'F5529, the P8 port only contains 3-pins.
- Port PJ is unique. For many devices, it's only 4-bits wide. Four pins on this port support emulation functionality for programming or debugging the processor. Most MSP devices support both 4-wire (JTAG) and 2-wire ([Spy-Bi-Wire](#)) debugging modes. As with other ports, these pins can also be reconfigured for GPIO.

The following diagram summarizes the GPIO API found in MSPWare DriverLib. Not only have we listed the various functions, but we've indicated which GPIO registers they write to (or read from).





# Before We Get Started Coding

## Getting Your Program Started

We cover system initialization details in Chapter 4, but here are a few items needed for Lab 3:

1. Include required `#include` files
2. Turn off the Watchdog timer
3. Unlock pins (FRAM devices)

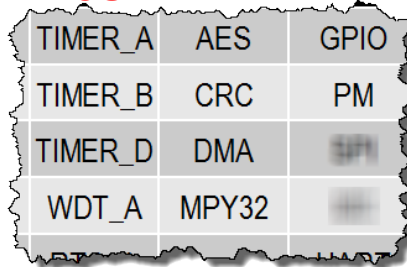
## 1. #Include Files

If you are an experienced C programmer, you have probably become accustomed to using *include* files. As described in the last chapter, every MSP device has a device-specific header (.h) file which defines its various registers and symbols. When using the “Register” layer model of programming, you would need to include this header file.

MSPWare Driver Library builds upon the device-specific header file, creating a header file for each DriverLib peripheral module. To make programming easier, though, they combined all these header files into a single “driverlib.h” file.

### Include Files

- ◆ Like most C programs, you need to include the required header files
  - ◆ Every MSP device has its own .h file to define various symbols and registers
  - ◆ Each DriverLib module has a header file to define its API
  - ◆ DriverLib defines all peripherals available for each given device – include `hw_memmap.h` (from /inc folder)
- ◆ For easy-to-use, DriverLib created a **single header file** for you to include: `driverlib.h`



TIMER_A	AES	GPIO
TIMER_B	CRC	PM
TIMER_D	DMA	UART
WDT_A	MPY32	...

```
#include <driverlib.h>

GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN7);
```

## 2. Disable Watchdog Timer

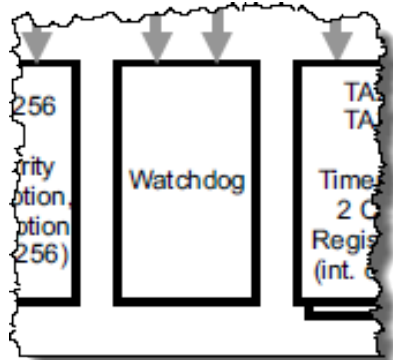
The MSP watchdog timer is always enabled. If you're just trying to get your first program to run, you won't need this feature, thus you can stop this timer with the DriverLib function shown below.

### Disable WatchDog Timer

- ◆ MSP watchdog timer is **always enabled** at reset
- ◆ Watchdog timer requires modification password (0x5A)
- ◆ Easiest **solution**:  
Begin your program with the DriverLib watchdog hold function

```
#include <driverlib.h>

WDT_A_hold(WDT_A_BASE); //Stop watchdog timer
```



The diagram illustrates the internal structure of the Watchdog Timer (WDT) module. It features three main vertical blocks: a left block labeled '256' and 'nity', a central block labeled 'Watchdog', and a right block labeled 'TA' and 'Time'. Arrows point from the top of each block to a common horizontal line. The right block also contains the text '2 C' and 'Regis (int.'.

---

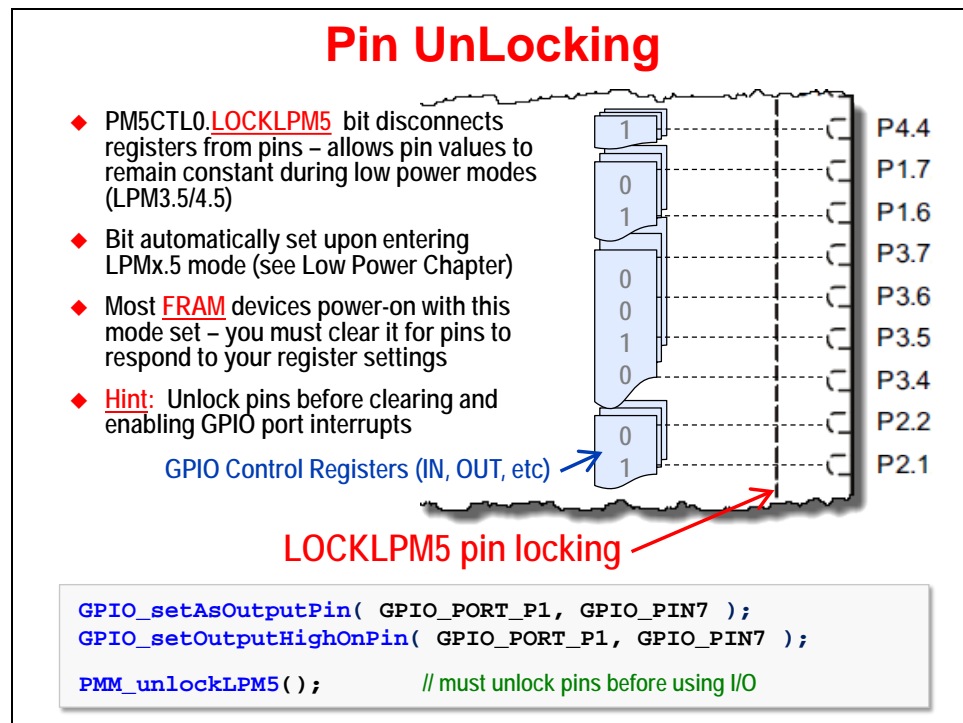
**Note:** We discuss the watchdog timer in more detail during the next chapter.

---

### 3. Pin Unlocking (only required for FRAM devices)

Pin locking is a feature that holds the last state of all GPIO pins when a device is put into its lowest power modes – that is, when power is removed from the memory and registers. Without this ‘locking’ feature, the pins would lose their values when entering these power modes.

The pin-locking feature freezes the state of each pin, effectively disconnecting them from their associated register bits (i.e. PxOUT) – you can think of like there is a switch along the vertical dashed line in the following diagram.



Many earlier devices, such as the 'F5529, provide the pin-locking feature – although, it was not enabled by default until the second generation FRAM devices; these devices always wake from reset with their pins locked.

Whenever this feature is enabled (by default or manually), there is an additional ‘unlocking’ step required in order for your I/O to respond to the values written to the GPIO control registers.

As shown above, it is suggested that you set your GPIO registers and then unlock the registers using the `PMM_unlockLPM5()` function.

**Note:** When using GPIO port interrupts, it's recommended that you unlock the pins before clearing and enabling the port interrupts. This will prevent spurious interrupts from being generated during pin unlocking. (There is an example of this in the *Interrupts* chapter.)

# Notes:

## Lab 3

We begin with a short Worksheet in preparation for coding GPIO using the MSPWare DriverLib. Following that, you'll implement the blinking LED example using DriverLib, finally adding a test of the push button in the final part of the lab exercise.

### Lab 3 – Blink with MSPWare

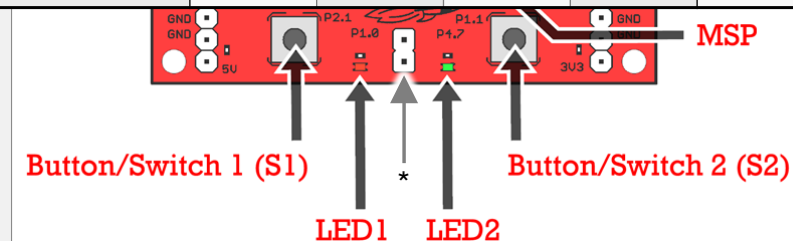
- ◆ **Lab Worksheet... a Quiz, of sorts on:**
  - ◆ GPIO
  - ◆ DriverLib
  - ◆ Path Variables
- ◆ **Lab 3a – Embedded 'Hello World'**
  - ◆ Create a MSPWare DriverLib GPIO project
  - ◆ Use IDE path variables to make your project portable
  - ◆ Write code to enable LED
  - ◆ Use simple (inefficient) delay function to create blinking LED
  - ◆ Use CCS debugging windows to view registers and memory
- ◆ **Lab 3b – Read Launchpad Push Button**
  - ◆ Test the state of the push button
  - ◆ Only blink LED when button is pushed (again, inefficient, but we'll fix that in Chapter 5)



Here's a helpful Port/Pin summary for the LaunchPads LEDs and Buttons.

### Launchpad Pins for LEDs/Switches

Launchpad →	F5529	FR4133	FR5969	FR6989	MSP432
LED1 (Red)	P1.0*	P1.0*	P4.6*	P1.0*	P1.0*
LED2 (Green)	P4.7	P4.0	P1.0	P9.7*	R: P2.0* G: P2.1* B: P2.2*
Button 1	P2.1	P1.2	P4.5	P1.1	P1.1
Button 2	P1.1	P2.6	P1.1	P1.2	P1.4



\* LED connected via jumper; lets you gain access to port/pin or LED

## Lab 3a – Blinking an LED

### Abstract

This lab creates what is often called, the *embedded* version of the "Hello World" program.

Your code should blink the LaunchPads LED using the MSPWare DriverLib library. While this is a simple exercise, it's perfect for learning the mechanics of integrating DriverLib into your programs.

Part of learning to use a library involves adding it to our project and adding its location the compiler's search path.

Finally, along with single-stepping our program, we will explore the "Registers" window in CCS. This lets us view the CPU registers, watching how they change as we step thru our code.

---

**Note:** From an efficiency standpoint, this code example is a BAD way to implement a blinking LED. The `_delay_cycles()` function is VERY INEFFICIENT. A timer, which we'll learn about in a later chapter, would be a better, lower-power way to implement a delay. For our purposes in this chapter, though, this is an easy way to get started with.

---

**Hint:** The MSP DriverLib Users Guide is a good resource to help you answer the questions on the next page. It can be found in the MSPWare "doc" folders:

```
\MSPWare_2_21_00_39\driverlib\doc\MSP430F5xx_6xx\  
\MSPWare_2_21_00_39\driverlib\doc\MSP430FR2xx_4xx\  
\MSPWare_2_21_00_39\driverlib\doc\MSP430FR5xx_6xx\  
\MSPWare_2_21_00_39\driverlib\doc\MSP432P4xx\
```

## GPIO Output Worksheet

### MSPWare DriverLib

1. Where is your MSPWare folder located? *(You should have written this down in the Installation Guide)*  
\_\_\_\_\_
2. To use the MSPWare GPIO and Watchdog API, which header file needs to be included in your source file? *(Hint: We discussed this during the presentation in the "Before We Get Started" section.)*

```
#include < _____ >
```

3. Which DriverLib function stops the Watchdog timer?  
*(Hint: Look in DriverLib User's Guide or the "Before We Get Started" section of this chapter.)*

```
_____ ;
```

### GPIO Output

4. Which I/O pin on Port 1 is connected to an LED (on your LaunchPad)?  
\_\_\_\_\_

What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?

*(Hint: Look at the chapter slides titled: "PxDIR (Pin Direction)" and "GPIO Output".)*

```
_____ ;
```

```
_____ ;
```


**FRAM**

For FRAM devices, what additional function is needed to make the I/O work (i.e. to connect the GPIO registers to the pin)?

```
_____ ;
```

5. Using the `__delay_cycles()` intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, and then low?  
(Hint: What two GPIO functions set an I/O Pin high and low?)

```
#define ONE_SECOND 800000

while (1) {
    //Set pin to "1" (hint, see question 4)
    _____ ;

    __delay_cycles( ONE_SECOND );
    // Set pin to "0"
    _____ ;

    __delay_cycles( ONE_SECOND );
}
```

**Double-check that your answers are correct**

**... using the solutions found in the Chapter 3 Appendix (pg 3-43).**



## Procedure

### 1. Close any open project and file.

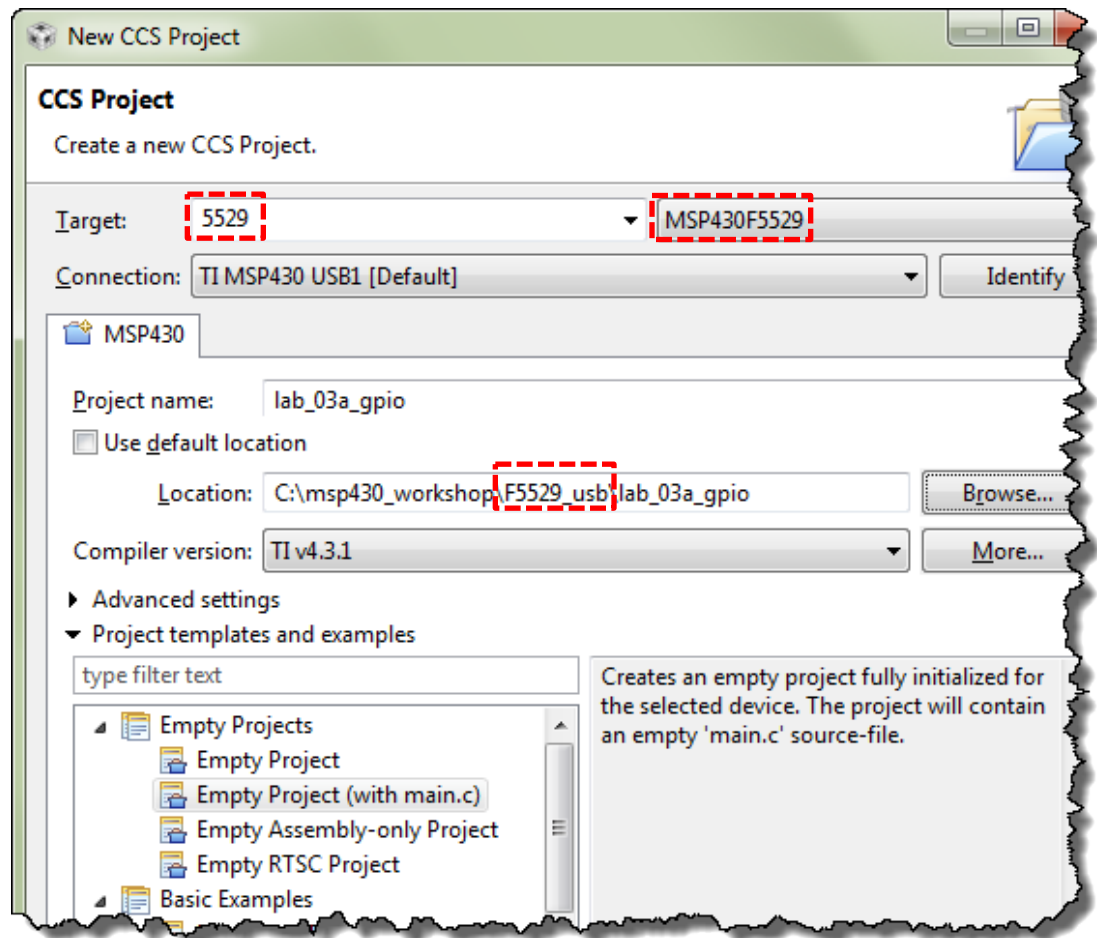
This helps to prevent us from accidentally working on the wrong file, which is easy to do when we have multiple lab exercises that use "main.c". If a previous project is open:

Right-click on the project and select "Close Project"

### 2. Create a new project.

Name the new project: lab\_03a\_gpio

Fill in the new project dialog as shown below, and then click Finish.



If you have questions about creating CCS projects, you can refer back to Lab 2b.

**Note:** If you're working with the 'FR5969', 'FR4133', or 'MSP432P401R', please replace the 'F5529' references shown above with those required for your LaunchPad.

Also, your compiler version may be more recent than the one shown in the screen capture.

**3. Notice that the main() function already turns off the watchdog timer.**

Although this is not required, you can replace this “register-based” code with the DriverLib function. Either way works fine. *If you want to use DriverLib, please reference your Worksheet answer #3 (on page 3-23).*

**4. Add required header file(s).**

Add the #include header required by MSPWare DriverLib. (See Worksheet question #2).

**Hint:** The default `main.c` created by the new project wizard already has `#include <msp430.h>`. You can replace this with the DriverLib #include. It's OK to have both of them, but the DriverLib header file references `msp430.h` for you.

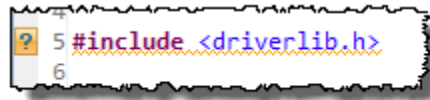
**5. Build your program.**

Even though we haven't added any code yet, try building the program.

???

**6. Why the build error?**

Depending upon which version of CCS you have, you might have seen a question mark (?) in front of the #include before you built the program.



When building your program, you should have received a build error. What caused this error?

---

## Add MSPWare DriverLib

Hopefully you answered the last question by saying that we need to add the DriverLib library to our project. The question marks told us that CCS couldn't find the header file.

Adding the DriverLib library is a two-step process:

- Import a copy of the library
- Include the location in the CCS build search path

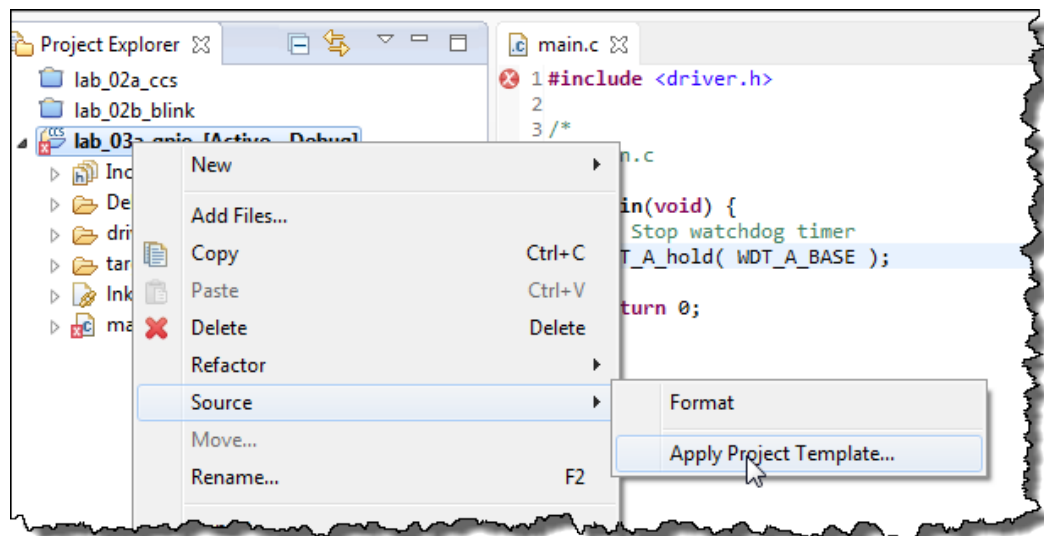
It may seem difficult to believe, but adding a library is often a difficult task. While the steps are straightforward, the directory path locations must be exactly correct or your program will not build. *(A troublesome problem for users of all devices, architectures and compilers.)*

The MSP tools include an automated method for adding the Driver Library to your project. This helps to minimize the error and frustration with getting every reference exactly right.

### 7. Add MSPWare DriverLib library to your project using the “Apply Project Template”.

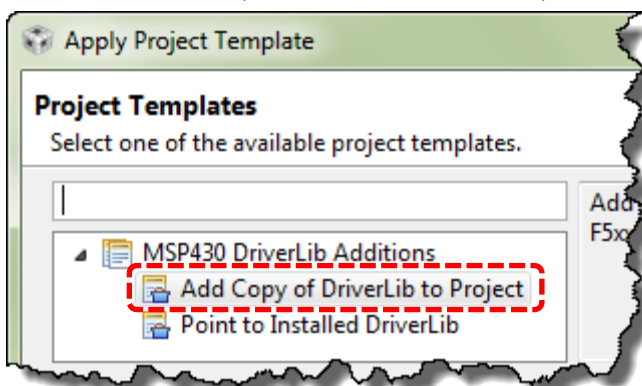
CCS uses the target specified for your project to select the correct version of DriverLib when you apply the DriverLib project template. Add this to your project with the following:

Right-Click → Source → Apply Project Template...

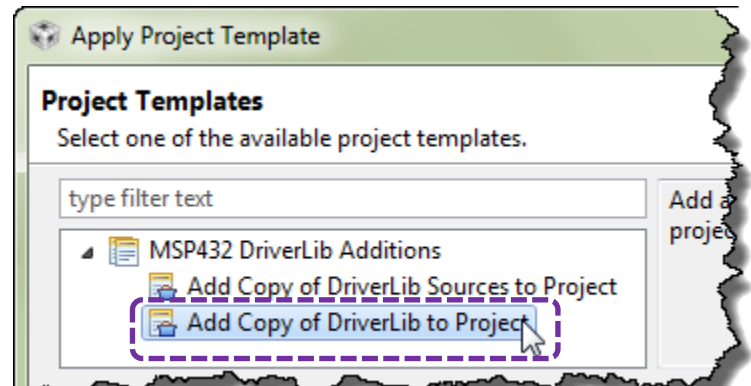


**Note:** 'FR6989 apply template is broken in MSPWare\_2\_21\_00\_39. See appendix (pg 3-40) for manual instructions.

### MSP430 (F5529, FR5969, FR4133)




### MSP432



After clicking *Finish*, you should notice the library added to your project.


- **MSP430** DriverLib is provided in source code; therefore your project should now include the “driverlib” source folder. Also, the compiler search path will include a pointer to the header files inside this folder.

▷  driverlib

- **MSP432** DriverLib is provided in both source and binary. The recommended template will add the driverlib binary file (.lib) as well as the source code. In this case, though, the source folder is flagged as “Exclude from Build”. Additionally, a “rom” folder is added (and ‘excluded’) which provides a copy of the DriverLib source that has been hardcoded into ROM.

▷  driverlib

▷  rom

 msp432p4xx\_driverlib.lib



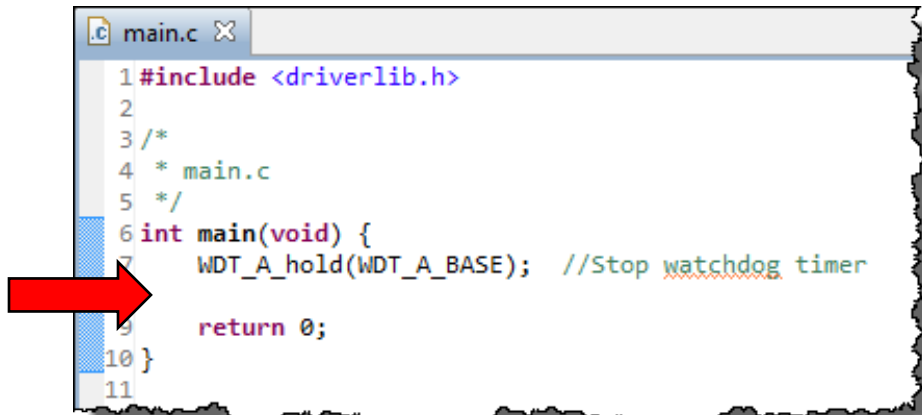
8. Click the build toolbar button to verify that your edits, thus far, are correct.

## Add 'Blink' Code to main.c

### 9. Add the code to set up P1.0 as an output pin.

Reference Worksheet question #4 (page 3-23).

Begin writing your code after the code that disables the watchdog timer as shown:



```

1 #include <driverlib.h>
2
3 /*
4  * main.c
5  */
6 int main(void) {
7     WDT_A_hold(WDT_A_BASE); //Stop watchdog timer
8
9     return 0;
10 }
11

```

FR4133

FR5969  
FR6989

**Hint:** If you're using the 'FR5969', 'FR6989' or 'FR4133 LaunchPad, don't forget to add the line of code which unlocks the pins. (Reference Worksheet question 4b (page 3-23).

### 10. Create a while{} loop that turns the P1.0 LED off/on with a 1 second delay.

Reference Worksheet question #5 (page 3-24). Begin the while{} loop after the code you wrote in the previous step (to set up the output pin).

Also, don't forget to add the #define for "ONE\_SECOND" towards the top of the file.



### 11. Build your program with the Hammer icon.

Make sure your program builds correctly, fixing any syntax mistakes found by the compiler. For now, you can ignore any remarks or advice recommendations, we'll explore those later.



### 12. Load and Run your program.

Click the *Debug* button to start the debugger and download your program. Then click the *Resume* button to run the code.

Does your LED flash? \_\_\_\_\_

If it doesn't, let's hope following debug steps help you to track down your error.

If it does, hooray! We still think you should perform the following debug steps, if only to better understand some of the debugging features built into CCS.



### 13. Suspend the debugger.

Alt-F8

## Debug



14. Restart your program.

15. Open the Registers window and view P1DIR and P1OUT. Then single-step past the GPIO DriverLib functions.

View → Registers



Expand Port\_1\_2, P1OUT and P1DIR as shown and then single-step ( i.e. Step Over – F6 ) until you execute this line:

```
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
```

Your register view should now look similar to this:

Name		Value
Port_A		
Port_1_2		
P1IN		0xFF
P1OUT		0x01
P1OUT7		0
P1OUT6		0
P1OUT5		0
P1OUT4		0
P1OUT3		0
P1OUT2		0
P1OUT1		0
P1OUT0		1
P1DIR		0x01
P1DIR7		0
P1DIR6		0
P1DIR5		0
P1DIR4		0
P1DIR3		0
P1DIR2		0
P1DIR1		0
P1DIR0		1
P1REN		0x00

16. Single-step until you reach the `_delay_cycles()` function.

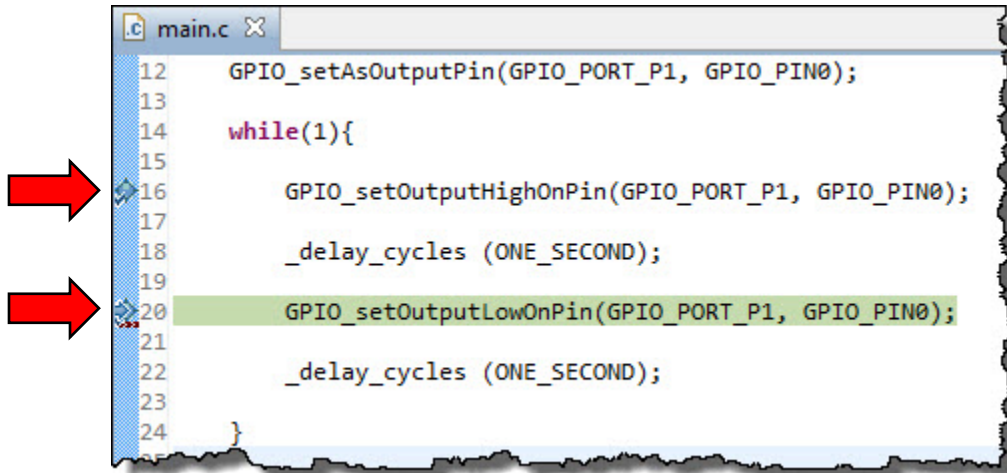
You should see the P1OUT register change as you step over the appropriate function.

Unfortunately, the “Step Over” command doesn’t step over `_delay_cycles()`. So let’s try using breakpoints..

**17. Set breakpoints on both `GPIO_setAs...` functions, then *Run/Resume* and check values in *Registers* window.**

Since it's difficult to step over `_delay_cycles()`, we'll just run past them. With breakpoints set on both lines where the GPIO pin value changes, you should see the LED toggle each time you run.

Set breakpoints as shown below:



Then click Run/Resume several times while keeping your eye on the LED.

**Oops!** Following these debugging steps, we found a problem in our original code. A cut and paste error left us with two lines of code in the loop that both turned off the LED.

While these are basic debugging techniques, these simple steps are powerful techniques for finding and fixing errors in your code.

**18. If you're using an FRAM LaunchPad, you may want to examine the PM5CTL0 register.**

**FR4133**

**FR5969  
FR6989**

If you've already run your code, the `PM5CTL0.LOCKLPM5` bit should already have been cleared by your program. It requires power-cycle to reset to set this to its initial condition. Follow these steps to see your code "unlock" the pins on the device.

- a) If running, suspend your program.

Alt-F8

- b) Open the register window and display the `LOCKLPM5` bit.

- c) Perform a *Hard Reset*.

Run → Reset → Hard Reset

- d) Then, restart the program.



- e) Finally, single-step your program until you see the `LOCKLPM5` value change to 0.

Name	Value	Description
▶ 1010 0101 PMMIFG	0x0200	PMM Interrupt Flag [Mem
▲ 1010 0101 PM5CTL0	0x0001	PMM Power Mode 5 Cont
1010 0101 LOCKLPM5	1	Lock I/O pin configuration
▶ 0101 Port_A		

▶ 1010 0101 PMMIFG	0x0200
▲ 1010 0101 PM5CTL0	0x0000
1010 0101 LOCKLPM5	0

## Lab 3b – Reading a Push Button

### Abstract

The goal of this lab is to light the LED when the SW1 button is pushed.

After setting up the two pins we need (one input, one output), the code enters an endless while loop where it checks the state of the push button and lights the LED if the button is pushed down.

Basic Steps:

- Cut/Paste previous project
- Delete/replace previous while loop
- Single-step code to observe behavior
- Run, to watch it work!

---

**Note:** "Polling" the button is very inefficient!

We'll improve on this in both the Interrupts and Timers chapters and exercises.

---

### GPIO Input Worksheet

1. What three different DriverLib functions can set up a GPIO pin for input?

*Hint: One place to look would be the MSP DriverLib Users Guide found in the MSPWare folder:  
\\MSPWare\_2\_21\_00\_39\\driverlib\\doc\\<target>*

---

---

2. What can happen to an input pin that isn't tied high or low? (*Hint: See "GPIO Input" topic on pg 3-9.*)

---

---

3. Which I/O pin on Port 1 is connected to a Switch (on your LaunchPad)?

---

Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup this pin as an input: (*Hint: See "GPIO Input" topic on pg 3-9.*)

---



4. Complete the following code to read pin P1.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read the pin for push-button 2

    usiButton1 = _____;

    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
    else {
        // Otherwise, if button is up, turn off LED
        GPIO_setOutputLowOnPin(  GPIO_PORT_P1, GPIO_PIN0 );
    }
}
```

5. In embedded systems, what is the name given to the way in which we are reading the button?  
(Hint – it is not an interrupt.)
- 

**Check your answers against ours ... see the Chapter 3 Appendix (pg 3-43)**

# Notes:

## Procedure

### File Management

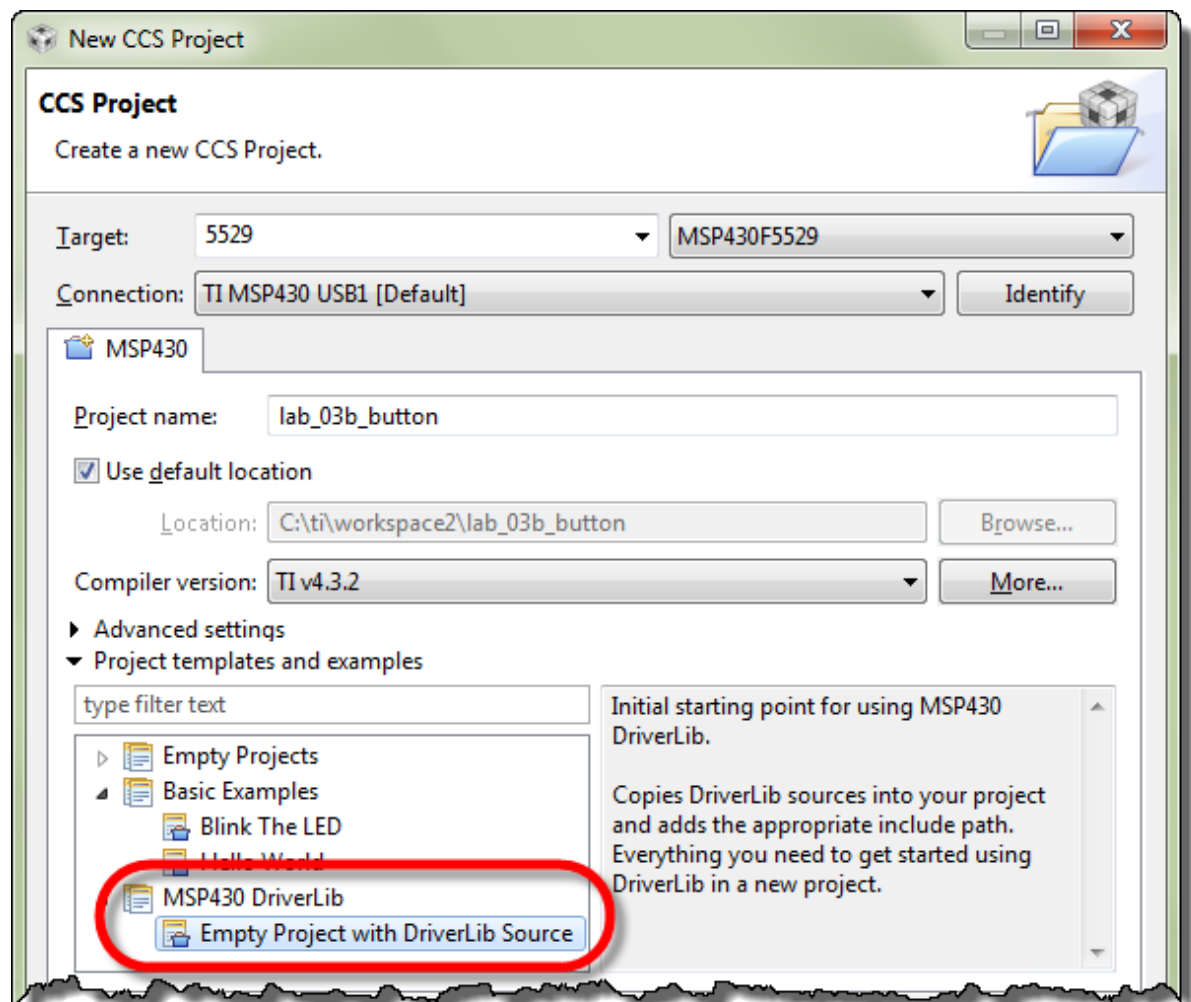
We're going to try another – easier – method of creating a new DriverLib project from scratch.

1. **Terminate the debugger (if it's still running).**
2. **Create a new driverlib project using the DriverLib project template.**

There are a couple different ways to import the example projects, but we've picked the easiest method, using the DriverLib project template.

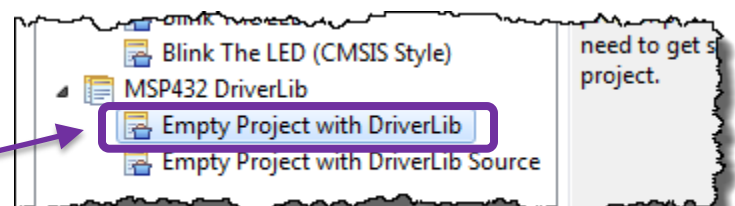
Create a new project – as you have done previously – but in this case you should select the template, as shown below:

Empty Project with DriverLib Source



**MSP432**

For the MSP432, choose the "Empty Project with Driverlib" template.



**3. Quickly examine the new lab\_03b\_button Project.**

Looking at this project, you'll see that it already has the DriverLib library imported into the project. Also, the required #include search path entry has also been added to the project.

**Copy code from the previous project****4. Replace main.c with your previous one.**

- a) Delete the 'empty' main.c from the new project.
- b) Copy/Paste main.c from lab\_03a\_gpio to lab\_03b\_button.

You can easily copy and paste files right inside the CCS Project Explorer. Simply right-click on the file (main.c) from the previous project and select "Copy"; then right-click on the new project and select "Paste".

*(Alternatively, we could have just copied and pasted the main() function from our previous lab project, but we found it easier to copy the whole file.)*

**5. Close the previous project: lab\_03a\_gpio**

As we've learned, this should close and open .c source files associated with the closed project, which can help us from accidentally editing the wrong file. *(Believe us, this happens a lot.)*

Right-click on the project and select "Close Project"

**6. Make sure the new project is active and then build it.**

Even though we haven't added anything new to our previous code, it's a good idea to try building it now, so that we can catch any errors introduced by creating/copying the new project.

## Add Setup Code... to reference the push button

7. Open `main.c` for editing.

8. Before the `main()` function, add the global variable: `usiButton1`

```
volatile unsigned short usiButton1 = 0;
```

Let's explain some of our choices:

**Global variable:** We chose to use a *global* variable because it's in scope all the time. Since it exists all the time (as opposed to a *local* variable), it can be a little bit easier to debug our code. Otherwise, local variables are probably a better choice: better programming style, less prone to naming conflicts and more memory efficient.

**Volatile:** We'll use this variable to hold the state of the switch, after reading it with our `DriverLib` function. The 'volatile' keyword effectively tells the compiler not to optimize away this variable.

**unsigned short** ... You tell us, why did we pick that? \_\_\_\_\_

**usiButton1:** The 'usi' is Hungarian notation for *unsigned short integer*. We added the '1' to 'Button', just in case we want to add a variable for the other button later on. (We could have also used the names 'S1' and 'S2' as they're labeled on the LaunchPad, but we liked 'Button' better.)

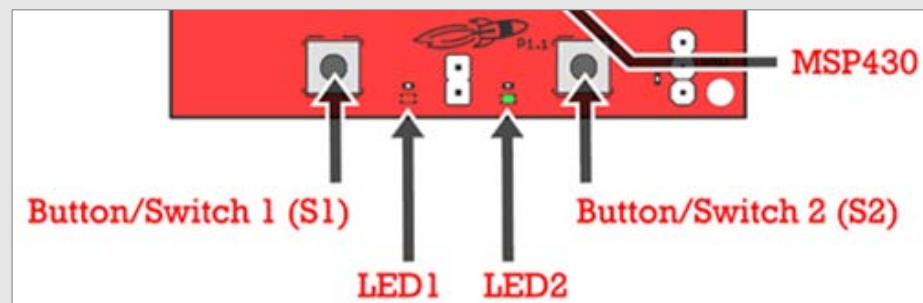
**=0** ... well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

9. In `main()`, add code to set push button as an input with pull-up resistor.

This setup code should go before the `while{} loop`.

Don't forget, this code was the answer to Worksheet question #3 (page 3-32).

**Hint:** We should have recommended bringing a magnifying glass to read the silk screen on the LaunchPad board. It can be difficult to read the button (and LED) labels. It may easier to reference the Quick Start sheet that came with your LaunchPad.



## Modify Loop

10. Modify the `while{} loop` to light the LED when the S2 push button is pressed.

Comment out (or delete) LED blinking code and replace it with the code we created in the Worksheet question #4 (page 3-33).

At this point, your `main.c` file should look similar to following code. *Note that the 'FR4133 code uses a different pin on Port 1 (P1.2).*

```
// -----
// main.c (for lab_03b_button project) ('FR5969 LaunchPad)
// -----

//***** Header Files *****
#include <driverlib.h>

//***** Global Variables *****
volatile unsigned short usiButton1 = 0;

//***** Functions *****
void main (void)
{
    // Stop watchdog timer
    WDT_A_hold( WDT_A_BASE );

    // Set pin P1.0 to output direction and initialize low
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );

    // Set switch 2 (S2) as input button (connected to P1.1)
    GPIO_setAsInputPinWithPullUpResistor( GPIO_PORT_P1, GPIO_PIN1 );

    // Unlock pins (required for most FRAM devices)
    PMM_unlockLPM5();

    while(1) {
        // Read P1.1 pin connected to push button 2
        usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 );

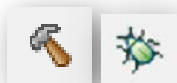
        if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
            // If button is down, turn on LED
            GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
        else {
            // If button is up, turn off LED
            GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
    }
}
```

**Hint:** If you want to minimize typing errors, you can copy/paste the code from the listing above. Better yet, we also placed a copy of this code into the lab's readme file (in the lab folder); just in case the copy/paste doesn't work well from the PDF file.

Copying from PDF will usually mess up the code's indentation. You can fix this by selecting the code inside CCS and telling it to clean-up indentation:

Right-click → Source → Correct Indentation (Ctrl+I)

## Verify Code



### 11. Build & Load program.

### 12. Add the `usiButton1` variable to the Watch Expression window.

Hint: Select the variable name before you right-click on it and add it to the *Watch* window.



### 13. Single-step project. Watch the LED and variable.

Loop thru `while{}` multiple times with the button pressed (and not pressed), watching the variable (and LED) change value.



### 14. Run the program.

Go ahead and click the Run/Resume toolbar button and watch the LED light whenever you push the button.

---

**Note:** Once again, this is NOT efficient code. It would be much better to use the push-button input pin as an interrupt ... which we'll do in Chapter 5.

---

## Optional Exercises

- Try this lab without pull-up (or pull-down) resistor.

Without the resistor, is the pushbutton's value always consistent? (yes / no) \_\_\_\_\_

- Try using the other LED on the board ...
- ... or the other pushbutton.



## Chapter 3 Appendix

### Manually Adding DriverLib to a MSP430 Project

In the lab procedure, we utilized the “automated” method for adding DriverLib to a project. This was a quick, handy technique for adding the library – which also greatly minimizes any potential errors. *(One of the biggest errors most users make involves getting the wrong location for libraries and search paths.)*

Hopefully you won’t ever have to add DriverLib manually, though reading through the procedure can be instructive. The trick is that it’s not enough to add the library; you must also add the appropriate search references

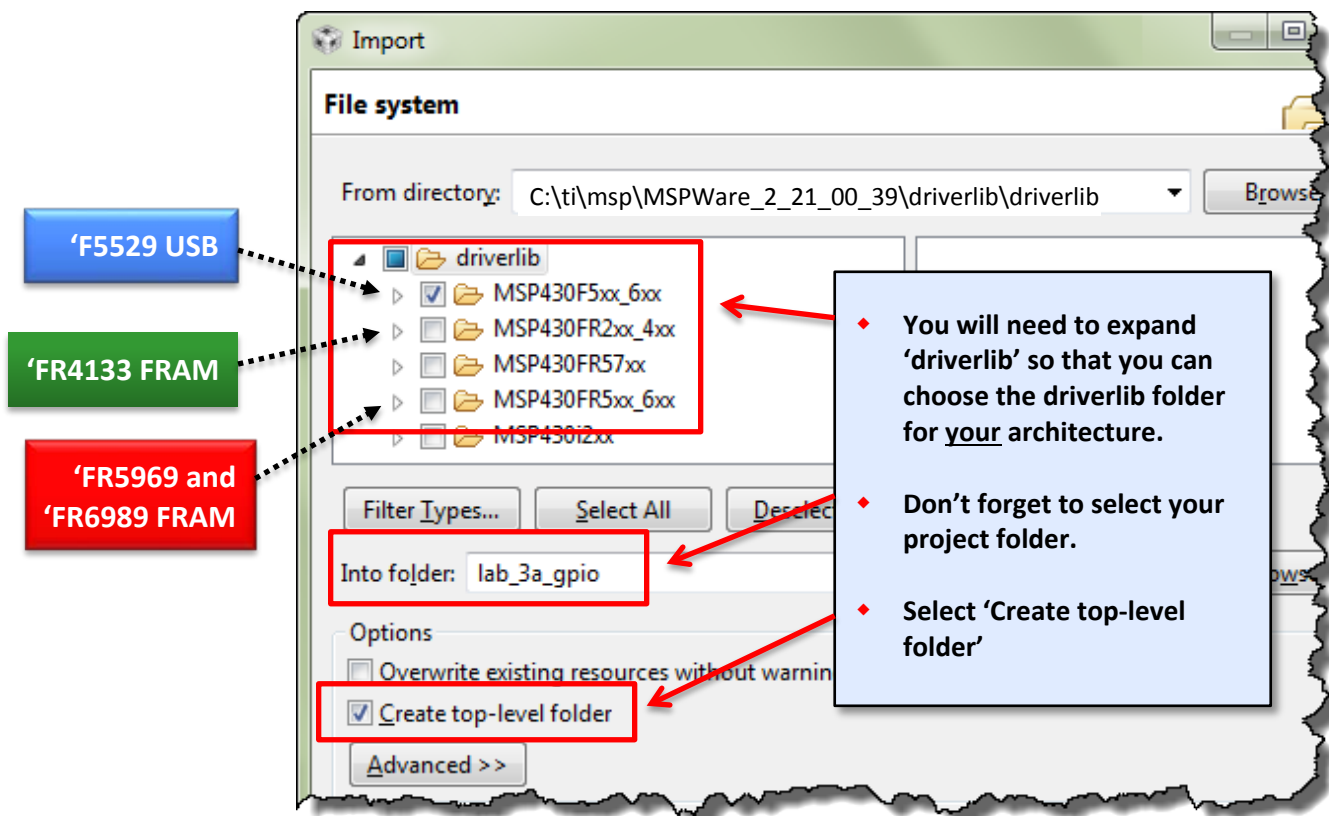
Adding the DriverLib library is a two-step process:

- Import a copy of the library (*source code library for MSP430; .lib or source for MSP432*)
- Include the location in the CCS build search path

#### 1. Import MSPWare DriverLib library to your project.

File → Import → Import... → General → File System

Then select the version and path of MSPWare you are using. Note: Your path may be slightly different than what is shown below, based on the location where you installed MSPWare.



After clicking *Finish*, you should notice the library folder was added to your project:

▷ driverlib/MSP430F5xx\_6xx



or one of these, depending on which LaunchPad you're using:

```
driverlib/MSP430FR5xx_6xx
```

```
driverlib/MSP430FR2xx_4xx
```

## 2. Update your project's search path with the location of DriverLib header files.

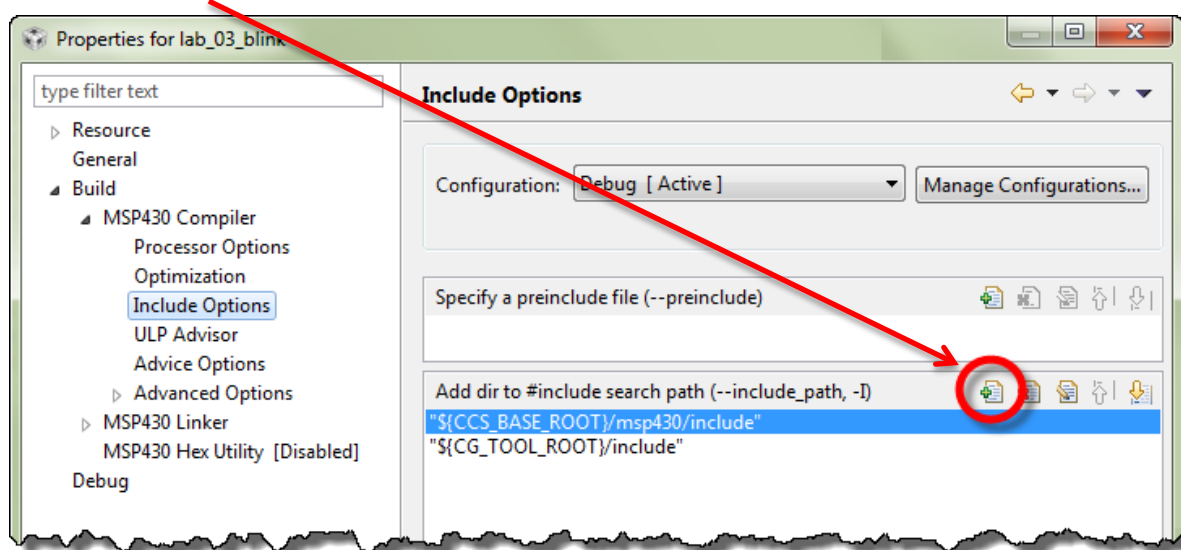
Along with adding the library, we also need to tell the compiler where to find it.

Open the Include Options and add the directory to #include search path:

Right-click project → Properties

Then select: Build → Compiler → Include Options

And click the “Add” search path button.



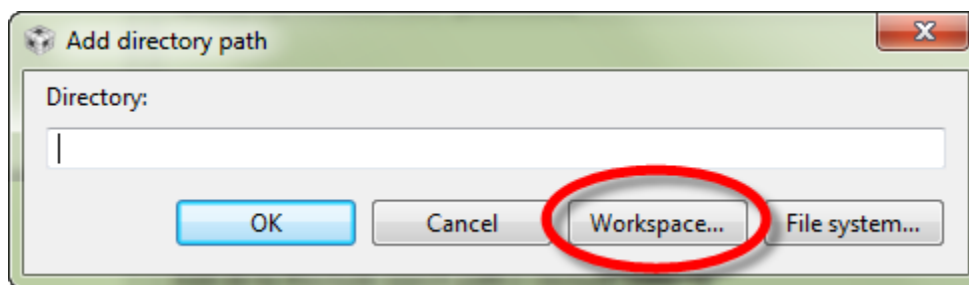
When the “Add directory path” dialog appears, you can add one of these paths manually:

```
${PROJECT_ROOT}\driverlib\MSP430F5xx_6xx
```

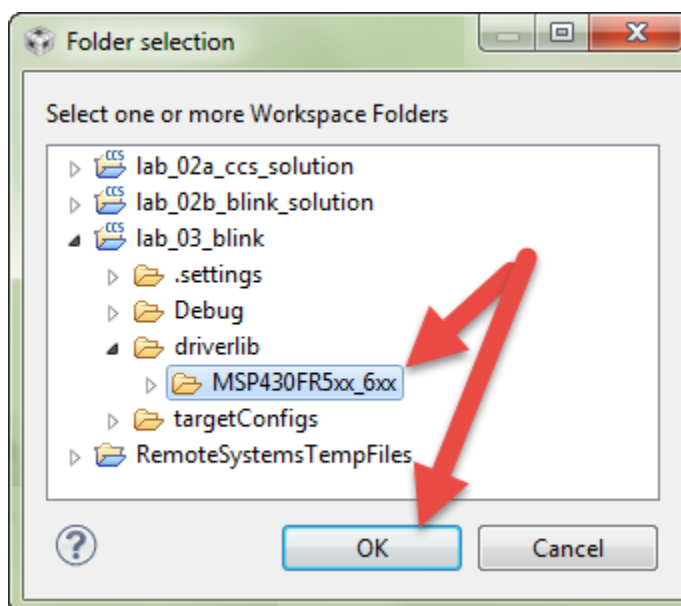
```
${PROJECT_ROOT}\driverlib\MSP430FR2xx_4xx
```

```
${PROJECT_ROOT}\driverlib\MSP430FR5xx_6xx
```

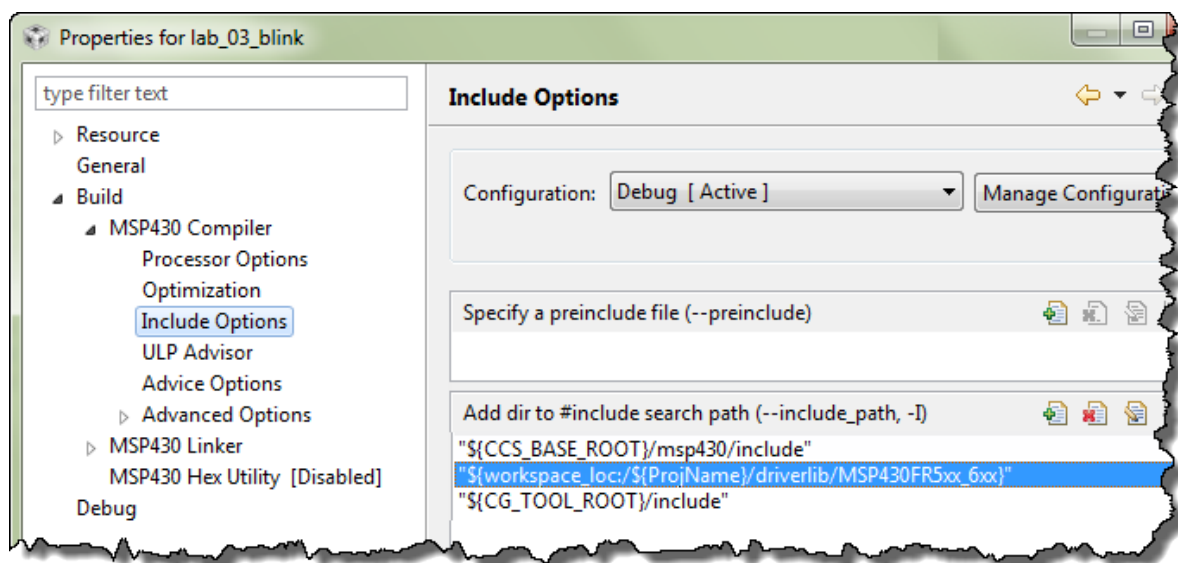
or minimize typing errors by selecting it from the **Workspace** (as shown below).



Select the driverlib folder and click OK.



Clicking OK once more returns us to the project's properties. Notice that the driverlib directory – found inside the workspace & project directory – has now been added to the project #include search path.



After inspecting the new search path, you can close the project properties dialog.

## Worksheet Answers

### Lab3a – Worksheet

1. Where is your MSPWare folder located?  
**Most likely:** C:\ti\MSP\MSPWare\_2\_21\_00\_39\
2. To use the MSPWare GPIO and Watchdog API, which header file needs to be included in your source file?  

```
#include < driverlib.h >
```
3. Which DriverLib function stops the Watchdog timer?  
WDT\_A\_hold( WDT\_A\_BASE ) ;
- 4a. Which I/O pin on Port 1 is connected to an LED (on your Launchpad)?  
All LaunchPads have P1.0
- 4b. What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?  
GPIO\_setAsOutputPin( GPIO\_PORT\_P1, GPIO\_PIN0 ) ;  
GPIO\_setOutputHighOnPin( GPIO\_PORT\_P1, GPIO\_PIN0 ) ;
- 4c. For FRAM devices, what additional function is needed to make GPIO work (i.e. to connect the I/O registers to their pins)?  
PMM\_unlockLPM5( ) ;

### Lab3a – Worksheet

5. Using the `__delay_cycles()` intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, then low?  

```
#define ONE_SECOND 800000

while (1) {
    //Set an LED pin to "1"
    GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
    __delay_cycles( ONE_SECOND ) ;
    // Set the pin to "0"
    GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
    __delay_cycles( ONE_SECOND ) ;
}
```

## Lab3b – Worksheet

1. What 3 functions can be used to set a pin for GPIO input?  
Hint, one place to look would be the MSP Driverlib Users Guide found here:  
 \MSPWare\_2\_21\_00\_39\driverlib\doc\<target>\
   
GPIO\_setAsInputPin()  
GPIO\_setAsInputPinWithPullDownResistor()  
GPIO\_setAsInputPinWithPullUpResistor()
2. What can happen to an input pin that isn't tied high or low?  
The input pin could end up floating up or down. This uses more power ... and can give you erroneous results.
- 3a. Which I/O pin on Port 1 is connected to a Switch (on your Launchpad)?  
All except 'FR4133: P1.1                  FR4133: P1.2
- 3b. Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup this pin as an input:  
GPIO\_setAsInputPinWithPullUpResistor ( GPIO\_PORT\_P1, GPIO\_PIN1 ) ;  
or GPIO\_setAsInputPinWithPullUpResistor ( GPIO\_PORT\_P1, GPIO\_PIN2 ) ;

## Lab3b – Worksheet

4. Complete the following code to read pin P1.1: (use P1.2 for FR4133)
 

```
volatile unsigned short usiButton1 = 0;
while(1) {
    // Read the pin for push-button S2
    usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 ) ;
    if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
        // If button is down, turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
    else {
        // Otherwise, if button is up, turn off LED
        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
    }
}
```

*For 'FR4133 use  
GPIO\_PIN2*
5. In embedded systems, what is the name given to the way in which we are reading the button? (Hint, it's not an interrupt)  
"Polling"