

Using Energia (Arduino) with the MSP430

Introduction



This chapter of the MSP430 workshop explores Energia, the Arduino port for the Texas Instruments Launchpad kits.

After a quick definition and history of Arduino and Energia, we provide a quick introduction to Wiring – the language/library used by Arduino & Energia.

Most of the learning comes from using the Launchpad board along with the Energia IDE to light LED's, read switches and communicate with your PC via the serial connection.

Learning Objectives, Requirements, Prereq's

Prerequisites & Objectives

◆ Prerequisites

- ◆ Basic knowledge of C language
- ◆ Basic understanding of using a C library and header files
- ◆ This chapter doesn't explain clock, interrupt, and GPIO features in detail, this is left to the other chapters in the MSP430 workshop

◆ Requirements - Tools and Software

◆ Hardware

- ◆ Windows (XP, 7, 8) PC with available USB port
- ◆ MSP430 Launchpad (v1.5)

◆ Software

- ◆ Energia Download
- ◆ Launchpad drivers
- ◆ (Optional) MSP430ware / Driverlib

Already installed, if you have installed CCSv5.x

◆ Objectives

- ◆ Define 'Arduino' and describe what it was created for
- ◆ Define 'Energia' and explain what it is 'forked' from
- ◆ Install Energia, open and run included example sketches
- ◆ Use serial communication between the board & PC
- ◆ Add an external interrupt to an Energia sketch
- ◆ Modify CPU registers from an Energia sketch
- ◆ Change the default system clock rate of the Launchpad from Energia

Chapter Topics

Using Energia (Arduino) with the MSP430	11-1
<i>What is Arduino</i>	11-3
<i>Energia</i>	11-4
<i>Programming Energia (and Arduino)</i>	11-7
Programming with 'Wiring'	11-7
Wiring Language/Library Reference	11-8
How Does 'Wiring' Compare?.....	11-9
Hardware pinout.....	11-10
<i>Energia IDE</i>	11-12
Examples, Lots of Examples.....	11-13
<i>Energia/Arduino References</i>	11-14
 <i>Lab 11</i>	 11-15
Installing Energia.....	11-16
Installing the LaunchPad drivers	11-16
Installing Energia.....	11-16
Starting and Configuring Energia	11-17
Lab 11a – Blink	11-20
Your First Sketch.....	11-20
Modifying Blink.....	11-23
Lab 11b – Pushing Your button	11-24
Examine the code.....	11-24
Reverse button/LED action.....	11-25
Lab 11c – Serial Communication (and Debugging)	11-26
What if the Serial Monitor is blank? (Launchpad Configuration).....	11-27
Blink with Serial Communication.....	11-28
Another Pushbutton/Serial Example.....	11-28
Lab 11d – Using Interrupts	11-29
Adding an Interrupt	11-29
TIMER_A	11-31
Lab 11e – Blink Fast/Slow by Changing Clocks	11-32
Blink (with default clock rate)	11-32
Where, oh where, is Main.....	11-32
Two ways to change the MSP430 clock source	11-35
Clocking CPU with VLO	11-36
Sidebar – initClocks().....	11-38
Sidebar Cont'd - Where is <u>F_CPU</u> defined?.....	11-39
<i>Lab Debrief</i>	11-40

What is Arduino

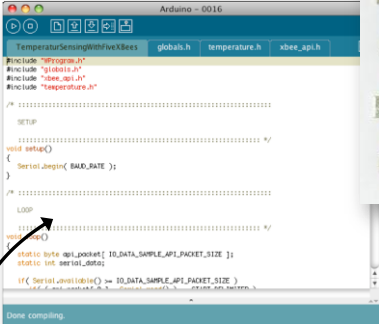
Physical Computing ... Hardware Hacking ... a couple of the names given to Arduino.

- *Our home computers are great at communicating with other computers and (sometimes) with us, but they have no idea what is going on in the world around them. Arduino, on the other hand, is made to be hooked up to sensors which feed it physical information.*¹ These can be as simple as pressing a button, or as complex as using ultrasound to detect distance, or maybe having your garage door tweet every time it's opened.
- *So the Arduino is essentially a simple computer with eyes and ears. Why is it so popular? Because the hardware is cheap, it's easy to program and there is a huge web community, which means that beginners can find help and download myriad programs.*¹

What is Arduino?

Tools

IDE: write, compile, upload



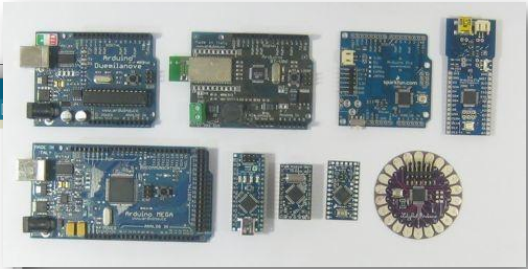
Code

'Wiring' Language includes:

- ◆ C/C++ software
- ◆ Arduino library of functions

Hardware

Open source µC boards with pins and I/O



- ◆ **Physical Computing**
Software that interacts with the real world
- ◆ **Open-source ecosystem**
Tools, Software, Hardware (Creative Commons)
- ◆ **Popular solution for...**
Open-source programmers, hobbyists, rapid prototyping

- *The idea is to write a few lines of code, connect a few electronic components to the Wiring hardware and observe how a light turns on when person approaches it, write a few more lines, add another sensor, and see how this light changes when the illumination level in a room decreases. This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.*²

In the end, Arduino is basically an ecosystem for easy, hardware-oriented, real-world programming. It combines the Tools, Software and Hardware for talking to the world.

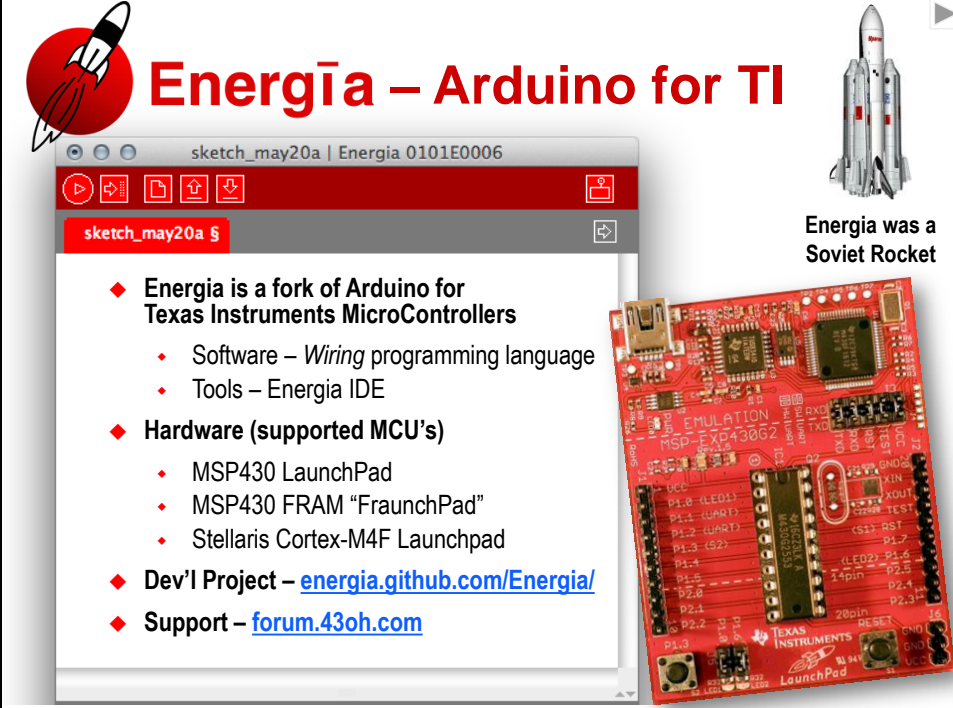
¹ <http://www.wired.com/gadgetlab/2008/04/just-what-is-an/>

² http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia

/ener'gi·a/ ; e·ner·gi·a

Energia (Russian: Энергия, Energiya, "Energy") was a Soviet rocket that was designed by NPO Energia to serve as a heavy-lift expendable launch system as well as a booster for the Buran spacecraft.³



Energia – Arduino for TI

sketch_may20a | Energia 0101E0006

sketch_may20a §

- ◆ **Energia is a fork of Arduino for Texas Instruments MicroControllers**
 - ◆ Software – *Wiring* programming language
 - ◆ Tools – Energia IDE
- ◆ **Hardware (supported MCU's)**
 - ◆ MSP430 LaunchPad
 - ◆ MSP430 FRAM "FraunchPad"
 - ◆ Stellaris Cortex-M4F Launchpad
- ◆ **Dev'l Project – [energia.github.com/Energia/](https://github.com/Energia)**
- ◆ **Support – forum.43oh.com**

Energia was a Soviet Rocket

EMULATION MSP-EXP430G2

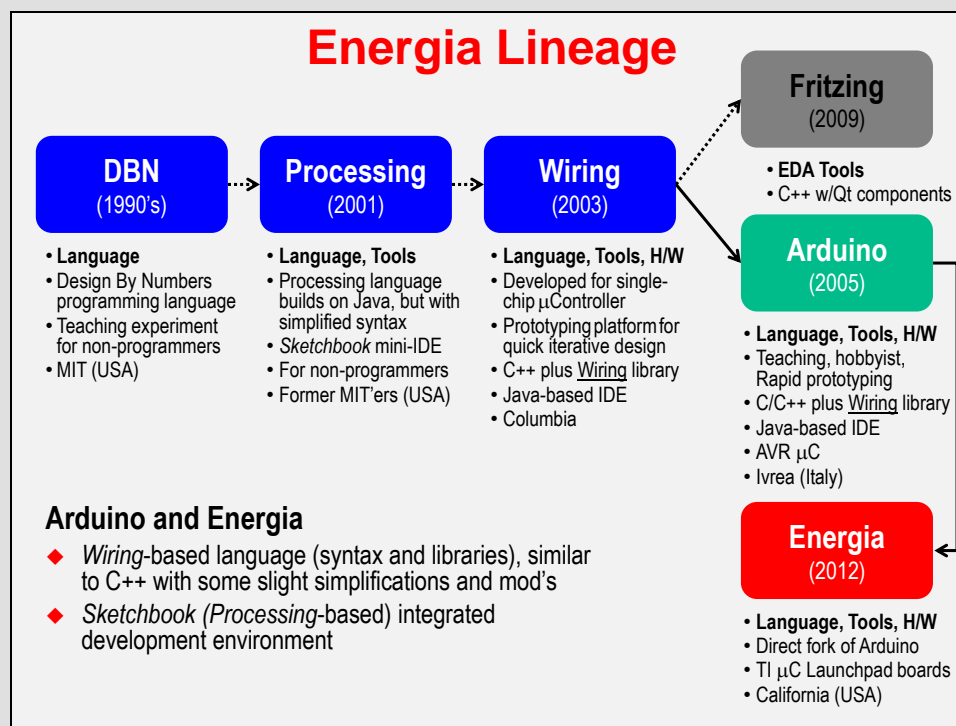
TEXAS INSTRUMENTS LaunchPad

Energia is a rapid electronics prototyping platform for the Texas Instruments msp430 LaunchPad. Energia is based on Wiring and Arduino and uses the Processing IDE. It is a fork of the Arduino ecosystem, but centered around the popular TI microcontrollers: MSP430 and ARM Cortex-M4F.

Similar to it's predecessor, it an open-sourced project. It's development is community supported, being hosted on github.com.

³ <http://en.wikipedia.org/wiki/Energia>

Sidebar – Energia Lineage



Design By Numbers (or DBN programming language) was an influential experiment in teaching programming initiated at the MIT Media Lab during the 1990s. Led by John Maeda and his students they created software aimed at allowing designers, artists and other non-programmers to easily start computer programming. The software itself could be run in a browser and published alongside the software was a book and courseware.⁴

Processing (2001) - One of the stated aims of Processing is to act as a tool to get non-programmers started with programming, through the instant gratification of visual feedback.⁵

This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.

Wiring (2003)⁶ - The Wiring IDE is a cross-platform application written in Java which is derived from the IDE made for the Processing programming language. It is designed to introduce programming and sketching with electronics to artists and designers. It includes a code editor ... capable of compiling and uploading programs to the board with a single click.

The Wiring IDE comes with a C /C++ library called "Wiring", which makes common input/output operations much easier. Wiring programs are written in C/C++, although users only need to define two functions to make a runnable program: `setup()` and `loop()`.

When the user clicks the "Upload to Wiring hardware" button in the IDE, a copy of the code is written to a temporary file with an extra include header at the top and a very simple `main()` function at the bottom, to make it a valid C++ program.

⁴ http://en.wikipedia.org/wiki/Design_By_Numbers_%28programming_language%29

⁵ [http://en.wikipedia.org/wiki/Processing_\(programming_language\)](http://en.wikipedia.org/wiki/Processing_(programming_language))

⁶ http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia Lineage (cont'd)

Arduino⁷ - In 2005, in Ivrea, Italy, a project was initiated to make a device for controlling student-built interaction design projects with less expense than with other prototyping systems available at the time. Founders Massimo Banzi and David Cuartielles named the project after Arduin of Ivrea, the main historical character of the town.

The Arduino project is a fork of the open source Wiring platform and is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a Processing-based integrated development environment.

Energia (2012) – As explained in the previous section of this chapter, Energia is a fork of Arduino which utilizes the Texas Instruments microcontroller Launchpad development boards.

Fritzing (2009)⁸ - An open-source initiative to support designers, artists, researchers and hobbyists to take the step from physical prototyping to actual product.

It's essentially an Electronic Design Automation software with a low entry barrier, suited for the needs of designers and artists. It uses the metaphor of the breadboard, so that it is easy to transfer your hardware sketch to the software. From there it is possible to create PCB layouts for turning it into a robust PCB yourself or by help of a manufacturer.

⁷ <http://en.wikipedia.org/wiki/Arduino>

⁸ [http:// Fritzing.org](http://Fritzing.org)

Programming Energia (and Arduino)

Programming with ‘Wiring’

Energia / Arduino Programming

- ◆ **Arduino programs are called sketches**
From the idea that we're...
Sketching with hardware
- ◆ **Sketches require only two functions to run cyclically:**
 - `setup()`
 - `loop()`
- ◆ **Are C/C++ programs that can use Arduino's *Wiring* library**
Library included with IDE
- ◆ **If necessary, you can access H/W specific features of μ C, but that hurts portability**
- ◆ **Blink is μ C's 'Hello World' ex.**
 - 'Wiring' makes this simple
 - Like most first examples, it is not optimized

```

// Most boards have LED and resistor connected
// between pin 14 and ground (pinout on later slide)
#define LED_PIN 14

void setup () {
    // enable pin 14 for digital output
    pinMode(LED_PIN, OUTPUT);
}

void loop () {
    digitalWrite(LED_PIN, HIGH); // turn on LED
    delay(1000); // wait one second (1000ms)
    digitalWrite(LED_PIN, LOW); // turn off LED
    delay(1000); // wait one second
}

```

Programming in Arduino is relatively easy. Essentially, it is C/C++ programming, but the *Wiring* library simplifies many tasks. As an example, we use the *Blink* sketch (i.e. program) that is one of examples that is included with Arduino (and Energia). In fact, this example is so ubiquitous that most engineers think of it as “*Hello World*” of embedded programming.

How does the ‘Wiring’ library help to make things easier? Let’s examine the Blink code above:

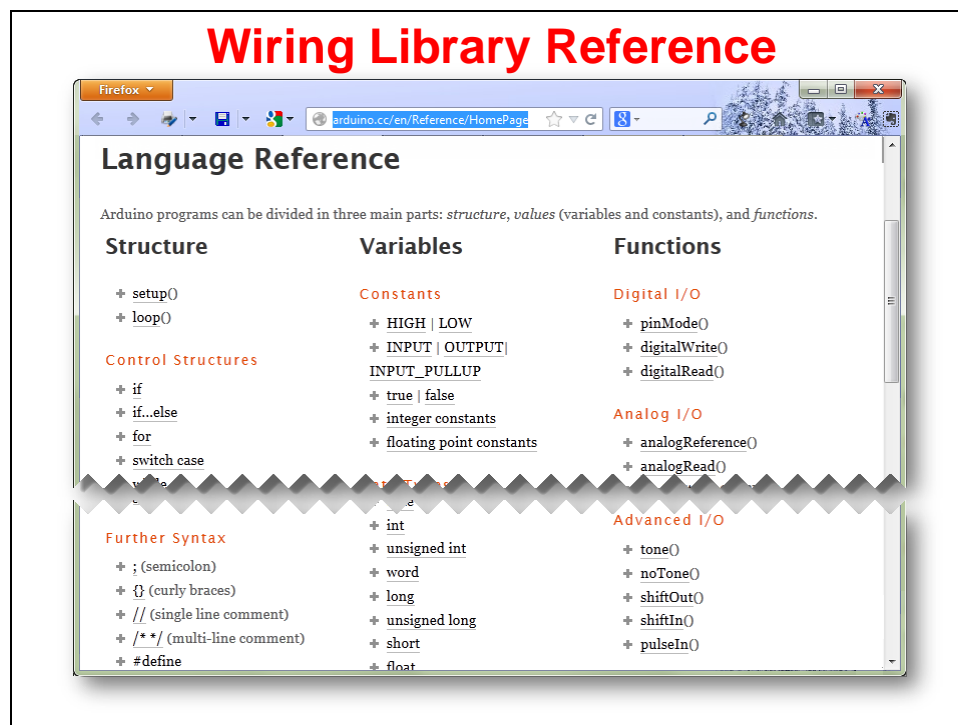
- A sketch only requires two functions:
 - **setup()** – a function run once at the start of a program which can be used to define initial environment settings
 - **loop()** – a function called repeatedly until the board is powered off
- Reading and Writing pins (i.e. General Purpose Input Output – GPIO) is encapsulated in three simple functions: one function defines the I/O pin, the other two let you read or write the pin. In the example above, this allows us to turn on/off the LED connected to a pin on our microcontroller.
- The **delay()** function makes it simple to pause program execution for a given number of microseconds. In fact, in the Energia implementation, the `delay()` function even utilizes a timer which allows the processor to go into low power mode while waiting.
- Finally, which not shown here, Arduino/Energia makes using the serial port as easy as using `printf()` in standard C programs.

About the only difference between Arduino and Energia programming is that you might see some hardware specific commands in the sketch. For example, in one of the later lab exercises, you will see how you can change the clock source for the TI MSP430 microcontroller. Changing clocks is often done on the MSP430 so that you can balance processing speed against long battery life.

Wiring Language/Library Reference

What commands are available when programming with 'Wiring' in Arduino and Energia?

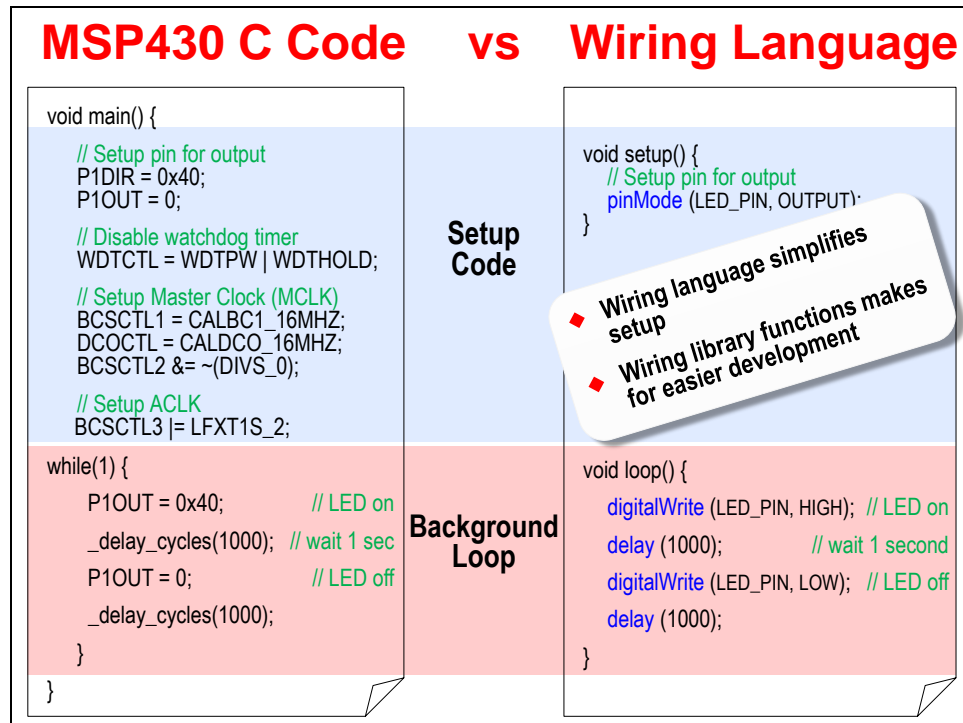
Arduino provides a language reference on their website. This defines the operators, controls, and functions needed for programming in Arduino (and Energia).⁹ You will also find a similar HTML reference available in the Energia installation zip file.



⁹ <http://arduino.cc/en/Reference/HomePage>

How Does 'Wiring' Compare?

How does the 'Wiring' language compare to standard C code?



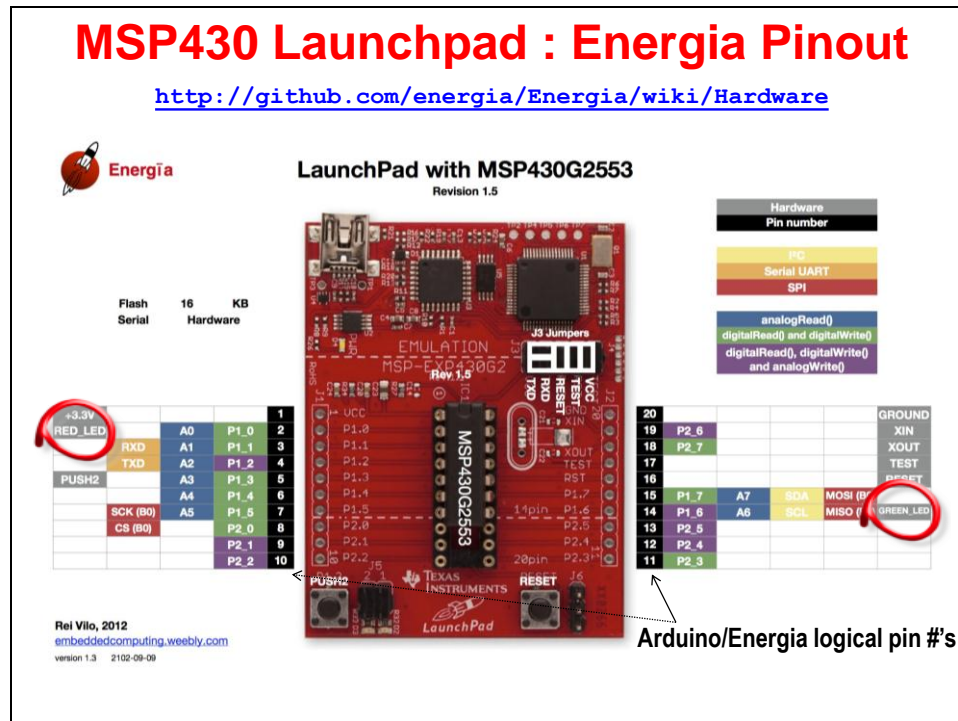
This comparison helps to demonstrate the simplicity of programming with Energia. As stated before, this can make for very effective rapid prototyping.

Later, during one of the lab exercises, we will examine some of the underpinnings of *Wiring*. Although the language makes programming easier, the same actual code is required for both sides of this diagram. In the case of *Wiring*, this is encapsulated by the language/library. You will see later on where this is done; armed with this knowledge, you can change the default values defined by the folks who ported Arduino over to Energia for the TI microcontrollers.

Hardware pinout

Arduino programming refers to Arduino “pins” throughout the language and examples. In the original implementation, these refer directly to the original hardware platform.

When adapting the Arduino library/language over to other processors, such as the TI microcontrollers, these pins must be mapped to the available hardware. The following screen capture from the Energia wiki shows the mapping for the MSP430 (v1.5 'G2553) Launchpad development board. There are similar diagrams for the other supported TI boards; please find these at wiki page: <https://github.com/energia/Energia/wiki/Hardware>.



Color Coded Pin Mapping

The wiki authors have color coded the pins to try and make things easier. The **Black** numbers represent the *Arduino Pin Numbers*. Thus, you can write to the pins using the pin numbers:

```
pinMode(2, OUTPUT);
digitalWrite(2, HIGH);
```

The **Grey** values show the hardware elements that are being mapped, such as the LED's or PushButton. You can use these alternative names: RED_LED; GREEN_LED; PUSH2; and TEMPESENSOR. Thus, to turn on the red LED, you could use:

```
pinMode(RED_LED, OUTPUT);
digitalWrite(RED_LED, HIGH);
```

Pins can also be address by there alternative names, such as **P1_0**. These correlate to the GPIO port (P1) and pin (0) names (P1.0) as defined by the MSP430. (In fact, the Launchpads conveniently show which I/O pins are mapped to the Boosterpack header connectors.) Using these symbols, we can write to pins using the following:

```
pinMode(P1_0, OUTPUT);
digitalWrite(P1_0, HIGH);
```

The remaining colored items show how various pins are used for digital, analog or communications purposes. The color legend on the right side of the diagram demonstrates the meaning of the various colors.

- **Green** indicates that you can use the associated pins with the *digitalRead()* and *digitalWrite()* functions.
- **Purple** is similar to Green, though you can also use the *analogWrite()* function with these pins.
- **Yellow**, **Orange**, and **Red** specify these pins are used for serial communication: UART, I2C, and SPI protocols, respectively.
- Finally, **Blue** demonstrates which pins are connected to the MSP430's ADC (analog to digital converter).

Serial Port Jumpers

For the MSP430G2553, the jumpers J3 for serial port depend on the revision of the LaunchPad board. Be sure to examine the pin-mapping diagram for your board to assure the serial port jumpers are correctly positioned.

Should you do Pullups or Not?

To reduce power consumption, MSP430 Value-Line Launchpads (version V1.5 and later) are shipped without pull-up resistors on PUSH2 (S2 or P1_3 or pin 5). This saves (77uA) if port P1_3 is driven LOW. (On your LaunchPad just below the "M" in the text "MSP-EXP430G2" see if R34 is missing.) For these newer launchpads, sketches using PUSH2 should enable the internal pull-up resistor in the MSP430. This is a simple change; for example:

```
pinMode(PUSH2, INPUT); now looks like pinMode(PUSH2, INPUT_PULLUP);
```

Hardware Pin References

As stated above, the Energia wiki (<https://github.com/energia/Energia/wiki/Hardware>) shows these pin mapping diagrams for each of the Energia supported boards. You can also refer to the source code which defines this pin mapping; look for `Energia/hardware/msp430/variants/launchpad/pins_energia.h`. This header file can be found on [github](https://github.com), or in the files installed with Energia.

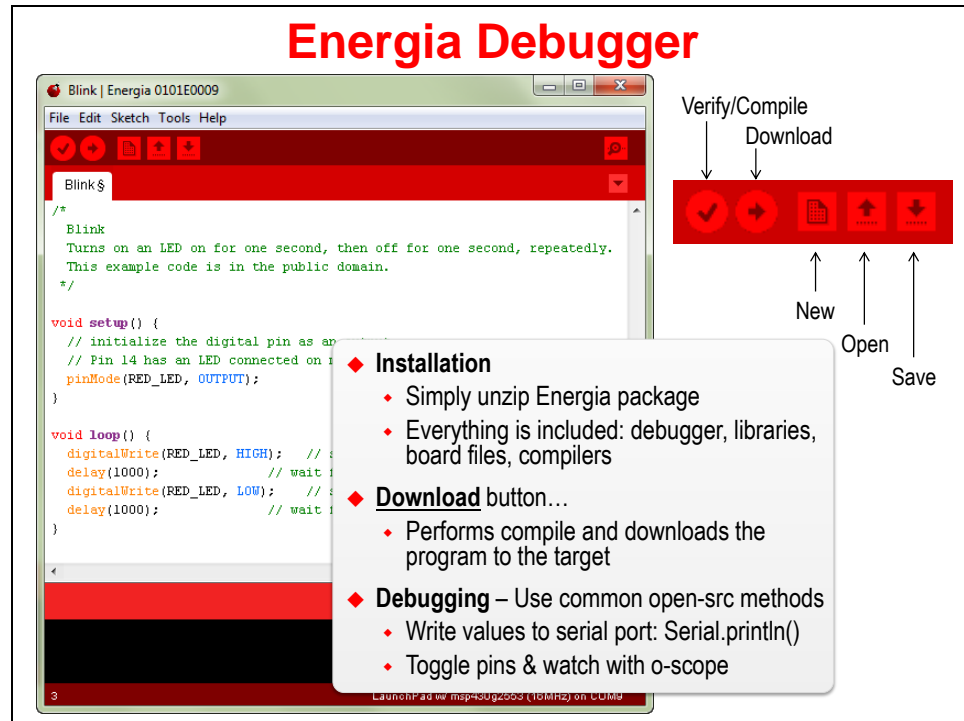
Sidebar

How can some 'pins' be connected to various pieces of hardware? (For example, PUSH2 and A3 (analog input 3) are both mapped to pin 5.)

Well, most processors today have *multiplexed* pins; i.e. each pin can have multiple functionality. While a given 'pin' can only be used for one function at a time, the chip designers give users many options to choose from. In an ideal world, we could just put as many pins as we want on a device; but unfortunately this costs too much, therefore multiplexing is a common cost/functionality tradeoff.

Energia IDE

The Energia IDE (integrated debugger and editor; integrated development environment) has been written in Java. This is how they can provide versions of the tools for multiple host platforms (Windows, Mac, Linux).



Installation of the tools couldn't be much simpler – unzip the package ... that's it. (Though, if you have not already installed TI's Code Composer Studio IDE, you may have to install drivers so that the Energia debugger can talk to the TI Launchpad board.)

Editing code is straightforward. Syntax highlighting, as well as brace matching help to minimize errors.

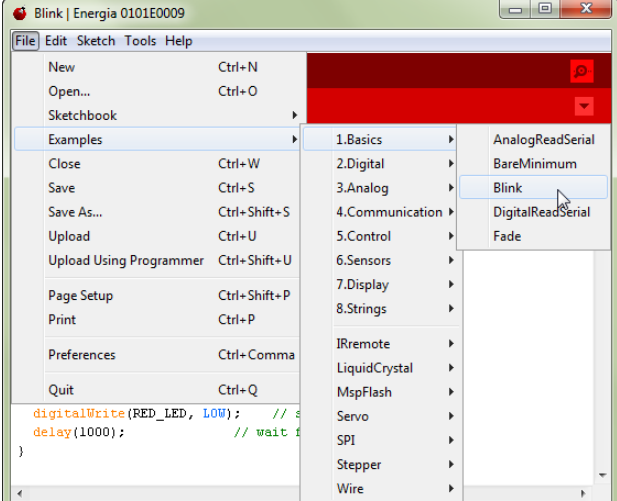
Compiling and **downloading** the program is as simple as clicking the *Download* button.

Debugging code is handled in the common, open-source fashion: printf() style. Although, rather than using printf(), you can use the Serial print functions to keep track of what is going on with your programs. Similarly, we often use LED's to help indicate status of program execution. And, if you have an oscilloscope or logic analyzer, you can also toggle other GPIO pins to evaluate the runtime state of your program sketches. (*We explore using LED's and serial communications in the upcoming lab exercises.*)

Examples, Lots of Examples

Energia ships with many examples. These are great for getting started with programming – or when trying to learn a new functionality. Our upcoming lab exercises will follow with this tradition of starting from these simple examples.

Energia Sketches (Examples)



The screenshot shows the Energia IDE interface. The 'File' menu is open, and the 'Examples' option is selected, which has opened a submenu. In this submenu, the '1.Basics' category is expanded, showing several examples: 'AnalogReadSerial', 'BareMinimum', 'Blink' (which is highlighted by the mouse), 'DigitalReadSerial', and 'Fade'. The main code editor in the background shows a snippet of C code: `digitalWrite(RED_LED, LOW); // s` and `delay(1000); // wait t`.

- ◆ **Basic Sketches**
 - ◆ Blink is the 'hello world' of micro's
 - ◆ BareMinimum is just `setup()` and `loop()`
- ◆ **Selecting example...**
 - ◆ Opens sketch in debugger window
 - ◆ Click download to compile, download and run

Energia/Arduino References

There are many more Arduino references that could possibly be listed here, but this should help get you started.

Where To Go For More Information

◆ Energia

- Home: <http://energia.nu/>
- Download: <http://energia.nu/download/>
- Wiki: <https://github.com/energia/Energia/wiki>
- Supported Boards: <https://github.com/energia/Energia/wiki/Hardware>
(H/W pin mapping)
- Getting Started: <https://github.com/energia/Energia/wiki/Getting-Started>
- Support Forum: <http://forum.43oh.com/forum/28-energia/>

◆ Launchpad Boards

- MSP430: <http://www.ti.com/tool/msp-exp430g2> (wiki) (eStore)
- ARM Cortex-M4F: [Launchpad](#) [Wiki](#) [eStore](#)

◆ Arduino:

- Site: <http://www.arduino.cc/>
- Reference: <http://arduino.cc/en/Reference/HomePage>
- Comic book: <http://www.jodyculkin.com/.../arduino-comic-latest3.pdf>

Energia

- Home: <http://energia.nu/>
- Download: <http://energia.nu/download/>
- Wiki: <https://github.com/energia/Energia/wiki>
- Supported Boards: <https://github.com/energia/Energia/wiki/Hardware>
(H/W pin mapping)
- Getting Started: <https://github.com/energia/Energia/wiki/Getting-Started>
- Support Forum: <http://forum.43oh.com/forum/28-energia/>

Launchpad Boards

- MSP430: <http://www.ti.com/tool/msp-exp430g2> (wiki) (eStore)
- ARM Cortex-M4F: [Launchpad](#) [Wiki](#) [eStore](#)

Arduino

- Site: <http://www.arduino.cc/>
- Reference: <http://arduino.cc/en/Reference/HomePage>
- Comic book: <http://www.jodyculkin.com/.../arduino-comic-latest3.pdf>

Lab 11

This set of lab exercises will give you the chance to start exploring Energia: the included examples, the 'Wiring' language, as well as how Arduino has been adapted for the TI Launchpad boards.

The lab exercises begin with the installation of Energia, then give you the opportunity to try out the basic 'Blink' example included with the Energia package. Then we'll follow this by trying a few more examples – including trying some of our own.

Lab Exercises

Installing Energia

- A. Blinking the LED**
- B. Pushing the Button**
- C. Serial Communication & Debugging**
- D. Interrupts**
- E. Blink Fast ... Blink Slow ... Blink Very Slow**

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to [Lab 11a – Blink](#).

These installation instructions were adapted from the Energia Getting Started wiki page. See this site for notes on *Mac OSX* and *Linux* installations.

<https://github.com/energia/Energia/wiki/Getting-Started>

Note: If you are attending a workshop, the following files should have been downloaded as part of the workshop's pre-work. If you need them and do not have network access, please check with your instructor.

Installing the LaunchPad drivers

1. To use Energia you will need to have the LaunchPad drivers installed.

For Windows Users

If TI's Code Composer Studio 5.x with MSP430 support is already installed on your computer then the drivers are already installed. Skip to the next step.

- a) Download the LaunchPad drivers for Windows:
[LaunchPad CDC drivers zip file for Windows 32 and 64 bit](#)
- b) Unzip and double click DPinst.exe for Windows 32bit or DPinst64.exe for Windows 64 bit.
- c) Follow the installer instructions.

Installing Energia

2. Download Energia, if you haven't done so already.

The most recent release of Energia can be downloaded from the [download](#) page.

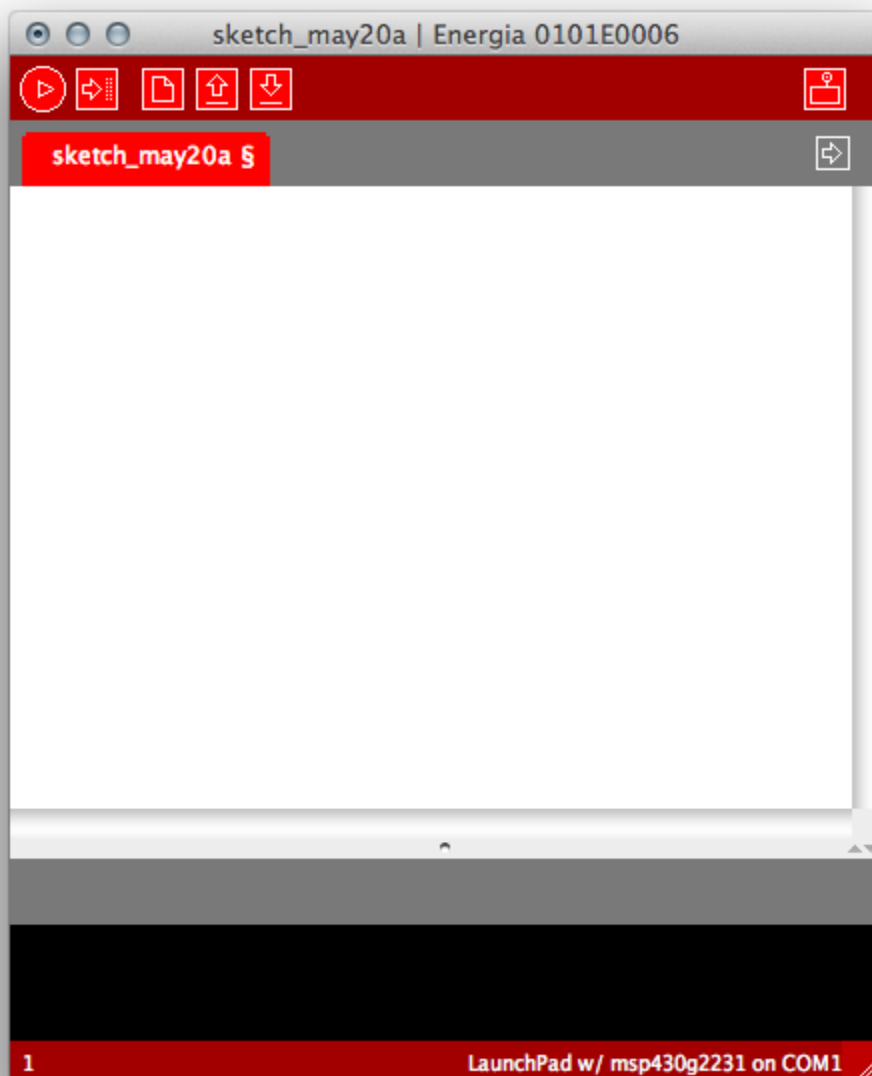
Windows Users

Double click and extract the energia-0101EXXXX-windows.zip file to a desired location.

Starting and Configuring Energia

3. Double click Energia.exe (Windows users).

Energia will start and an empty Sketch window will appear.



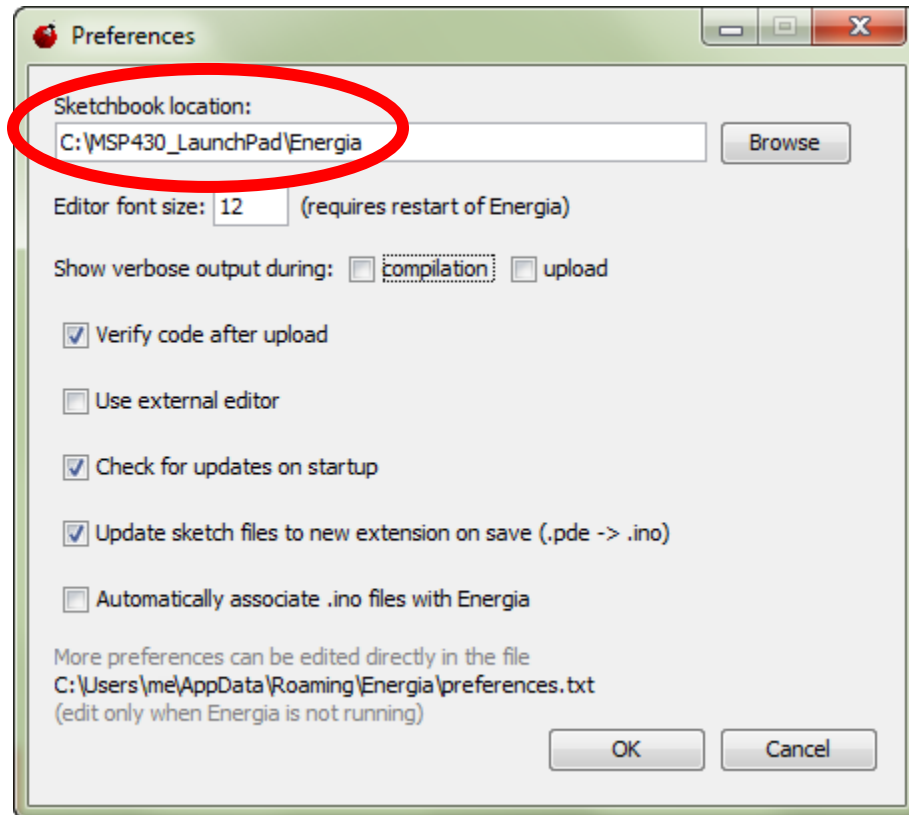
4. Set your *working folder* in Energia.

It makes it easier to save and open files if Energia defaults to the folder where you want to put your sketches.

The easiest way to set this locations is via Energia's preferences dialog:

File → Preferences

Which opens:



5. Selecting the Serial Port

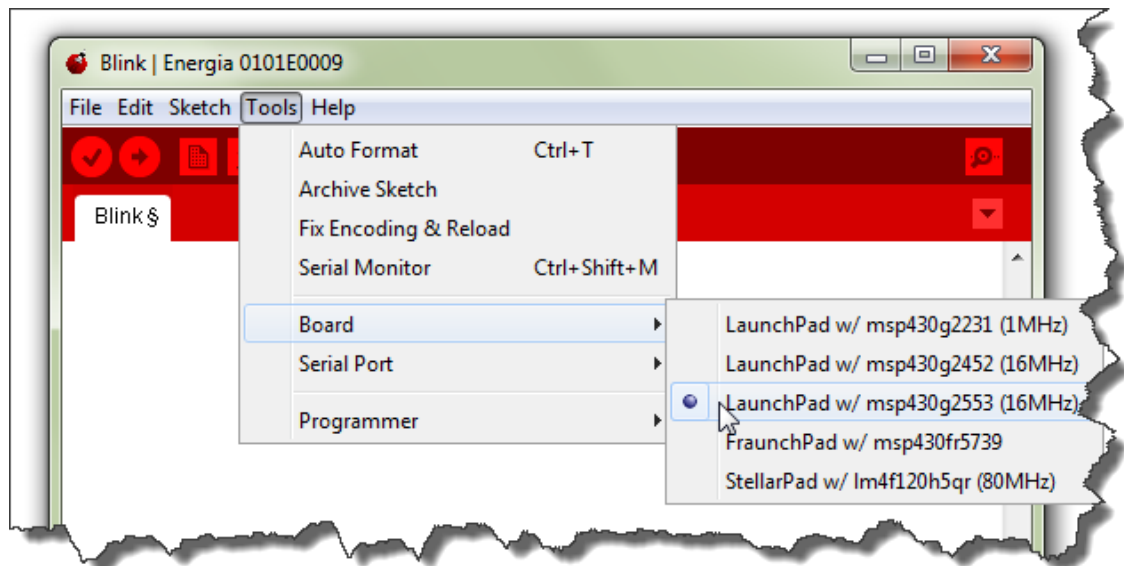
Select **Serial Port** from the **Tools** menu to view the available serial ports.

For Windows, they will be listed as COMXXX port and usually a higher number is the LaunchPad com port. On Mac OS X they will be listed as /dev/cu.uart-XXXX.

Hint: For more on configuring your Launchpad for serial communication, see [Serial-Communication](#).

6. Select the board you are using – most likely the msp430g2553 (16MHz).

To select the board or rather the msp430 in your LaunchPad, select **Board** from the **Tools** menu and choose the board that matched the msp430 in the LaunchPad.



Lab 11a – Blink

Don't blink, or this lab will go by without you seeing it. It's a very simple lab exercise – that happens to be one of the many examples included with the Energia package.

As simple as this example is, it's a great way to begin. In fact, if you have followed the flow of this workshop, you may recognize the *Blink* example essentially replicates the lab exercise we created in *Chapter 3* of the workshop.

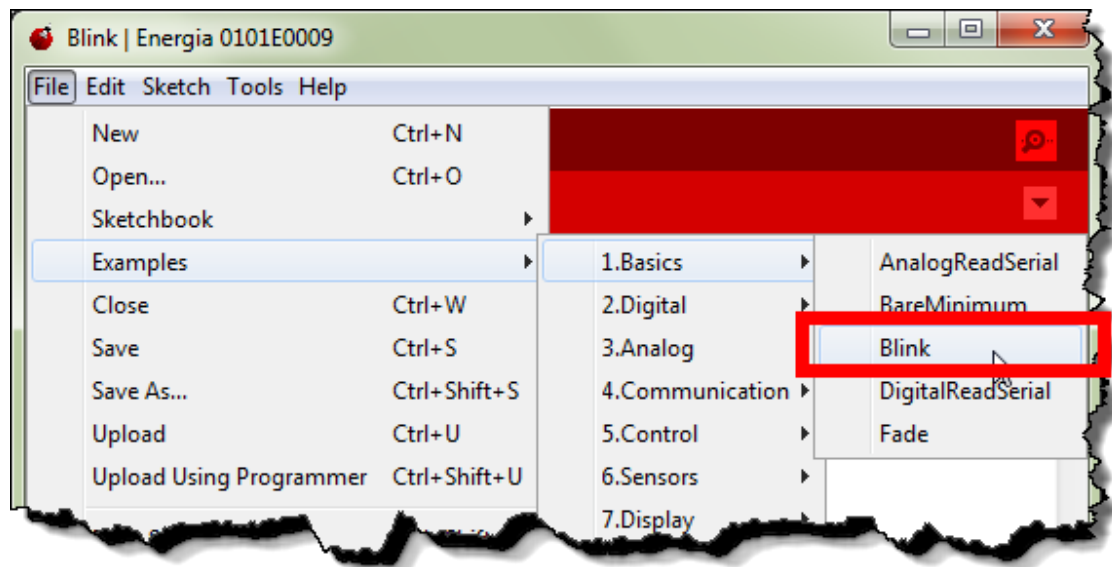
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit. We will explore how this is accomplished – i.e. where these differences/simplifications come from in *Lab11e – Changing the CPU Clocks (Blink Fast, Blink Slow, Blink Really Slow)*.

Your First Sketch

1. Open the *Blink* sketch (i.e. program).

Load the *Blinky* example into the editor; select ***Blink*** from the *Examples* menu.

File → Examples → 1.Basics → Blink



2. Examine the code.

Looking at the Blink sketch, we see the code we quickly examined during our chapter discussion. This code looks very much like standard C code. (In Lab11d we examine some of the specific differences between this sketch and C code.)

At this point, due to their similarity to standard C language code, we will assume that you recognize most of the elements of this code. By that, we mean you should recognize and understand the following items:

- **#define** – to declare symbols
- **Functions** – what a function is, including: void, () and {}
- **Comments** – declared here using // characters

What we do want to comment on is the names of the two functions defined here:

- **setup()**: happens one time when program starts to run
- **loop()**: repeats over and over again

This is the basic structure of an Energia/Arduino sketch. Every sketch should have – at the very least – these two functions. Of course, if you don't need to setup anything, for example, you can leave it empty.

```


/*
  Blink
  Turns on an LED on for one second, then off for one second,
  repeatedly. This example code is in the public domain.
*/

void setup () {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
    pinMode (RED_LED, OUTPUT);
}

void loop () {
    digitalWrite (RED_LED, HIGH);    // turn on LED
    delay (1000);                    // wait one second (1000ms)
    digitalWrite (RED_LED, LOW);     // turn off LED
    delay (1000);                    // wait one second
}

```

3. Compile and upload your program to the board.

To compile and upload the Sketch to the LaunchPad click the  button.



Do you see the LED blinking? What color LED is blinking? _____

What pin is this LED connected to? _____

(Be aware, in the current release of Energia, this could be a trick question.)

Hint: We recommend you check out the Hardware Pin Mapping to answer this last question. There's a copy of it in the presentation. Of course, the original is on the Energia [wiki](#).

Modifying Blink

4. Copy sketch to new file before modification.

We recommend saving the original Blink sketch to a new file before modifying the code.

File → Save As...

Save it to:

C:\MSP430_LaunchPad\Energia\Blink_Green

Hint: This will actually save the file to:

C:\MSP430_LaunchPad\Energia\Blink_Green\Blink_Green.ino

Energia requires the sketch file (.ino) to be in a folder named for the project.

5. How can you change which color LED blinks?

Examine the H/W pin mapping for your board to determine what needs to change.

Please describe it here: _____

6. Make the other LED blink.

Change the code, to make the other LED blink.

When you've changed the code, click the **Upload** button to: compile the sketch; upload the program to the processor's Flash memory; and, run the program sketch.

Did it work? _____

(We hope so. Please ask for help if you cannot get it to work.)

Lab 11b – Pushing Your button

Next, let's figure out how to use the button on the Launchpad. It's not very difficult, but since there's already a sketch for that, we'll go ahead and use it.

1. Open the *Button* sketch (i.e. program).

Load the *Button* example into the editor.

File → Examples → 2.Digital → Button

2. Try out the sketch.

Before we even examine the code, let's try it out. *(We know you were just like us and going to do it anyway.)*

When you push the button the (GREEN or RED) LED goes (ON or OFF)? _____

By the way, you probably know this already from earlier in the workshop, but which button are we using? Remember, the “user” button is called PUSH2 (the board silkscreen says P1.3) as shown here:



Examine the code

3. The author of this sketch used the LED in a slightly different fashion.

How is the LED defined differently in the Button Sketch versus the Blink sketch?

4. Looking at the pushbutton...

How is the pushbutton created/used differently from the LED? _____

What “Energia” pin is the button connected to? _____

What is the difference between INPUT and INPUT_PULLUP? _____

5. A couple more items to notice...

Just like standard C code, we can create variables. What is the global variable used for in this example?

Finally, this is a very simple way to read and respond to a button. What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

(Note, we will look at this 'more efficient' method in a later part of the lab.)

Reverse button/LED action

Do you find this example to be the reverse of what you expected? Would you prefer the LED to go ON when the button is pushed, rather than the reverse. Let's give that a try.

6. Save the example to sketch new file before modification.

Once again, we recommend saving the original sketch before modification. Save it to:

```
C:\MSP430_LaunchPad\Energia\Button_reversed
```

7. Make the LED light only when the button is pressed.

Change the code as needed.

Hint: The changes required are similar to what you would do in C, they are not unique to *Energia/Arduino*.

8. When your changes are finished, upload it to your Launchpad.

Did it work? _____

Lab 11c – Serial Communication (and Debugging)

This lab uses the serial port (UART) to send data back and forth to the PC from the Launchpad.

In and of itself, this is a useful and common thing we do in embedded processing. It's the most common way to talk with other hardware. Beyond that, this is also the most common debugging method in Arduino programming. *Think of this as the “**printf**” for the embedded world of microcontrollers.*

1. Open the *DigitalReadSerial* example.

Once again, we find there's a (very) simple example to get us started.

File → Examples → 1.Basics → DigitalReadSerial

2. Examine the code.

This is a very simple program, but that's good since it's very easy to see what Energia/Arduino needs to get the serial port working.

```
/* DigitalReadSerial

   Reads a digital input on pin 2, prints the result to the
   serial monitor (This example code is in the public domain) */

void setup() {
  Serial.begin(9600);           // msp430g2231 must use 4800
  pinMode(PUSH2, INPUT_PULLUP);
}

void loop() {
  int sensorValue = digitalRead(PUSH2);
  Serial.println(sensorValue);
}
```

As you can see, serial communication is very simple. Only one function call is needed to setup the serial port: **Serial.begin()**. Then you can start writing to it, as we see here in the **loop()** function.

Note: Why are we limited to 9600 baud (roughly, 9600 bits per second)?

The Launchpad onboard emulation's USB to serial bridge is limited to 9600 baud. It is not a hardware limitation of the MSP430 device. Please refer to the wiki for more info:
<https://github.com/energia/Energia/wiki/Serial-Communication>.

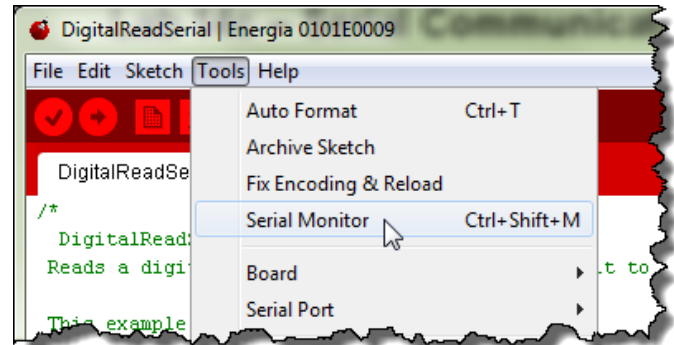
3. Download and run the sketch.

With the code downloaded and (automatically) running on the Launchpad, go ahead and push the button.

But, how do we *know* it is running? It doesn't change the LED, it only sends back the current pushbutton value over the serial port.

4. Open the serial monitor.

Energia includes a simple serial terminal program. It makes it easy to view (and send) serial streams via your computer.

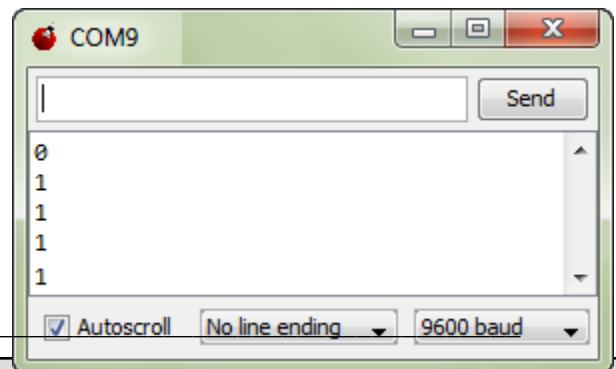


With the Serial Monitor open, and the sketch running, you should see something like this:

You should see either a "1" or "0" depending upon whether the button is up or down.

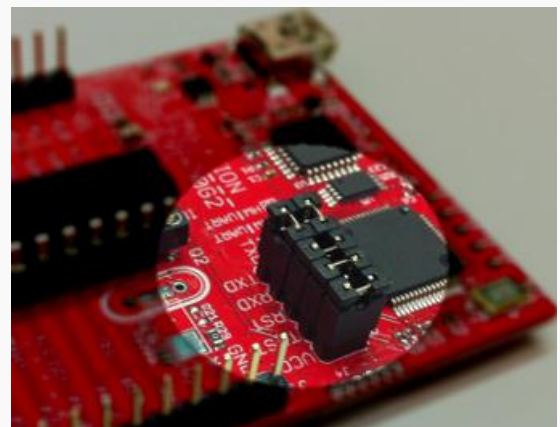
Also, notice that the value is updated continuously, since the sketch reads the button and writes it to port in the **loop()** function.

Do you see numbers in the serial monitor?



What if the Serial Monitor is blank? (Launchpad Configuration)

If this is the case, your Launchpad is most likely configured incorrectly. For serial communications to work correctly, the J3 jumpers need to be configured differently than how the board is configured out-of-the-box. (This fooled us, too.) Refer to these diagrams for correct operation.



Blink with Serial Communication

Let's try combining a couple of our previous sketches: *Blink* and *DigitalReadSerial*.

5. Open the *Button* sketch.

Load the *Button* from the *Examples* menu.

```
File → Examples → 2.Digital → Button
```

6. Save it to a new file before modification.

Once again, we recommend saving the original sketch before modification. Save it to:

```
C:\MSP430_LaunchPad\Energia\Serial_Button
```

7. Add 'serial' code to your *Serial_Button* sketch.

Take the serial communications code from our previous example and add it to your new *Serial_Button* sketch. (Hint, it should only require two lines of code.)

8. Download and test the example.

Did you see the Serial Monitor and LED changing when you push the button?

9. Considerations for debugging...

How you can use both of these items for debugging?

Serial Port; LED (And, what if you didn't have an LED available on your board?):

Another Pushbutton/Serial Example

Before finishing Lab 11C, let's look at one more example.

10. Open the *StateChangeDetection* sketch.

Load the *sketch* from the *Examples* menu.

```
File → Examples → 2.Digital → StateChangeDetection
```

11. Examine the sketch, download and run it.

How is this sketch different? What makes it more efficient? _____

How is this (and all our sketches, up to this point) inefficient? _____

Lab 11d – Using Interrupts

Interrupts are a key part of embedded systems. It is responding to external events and peripherals that allow our programs to ‘talk’ to the real world.

Thusfar, we have actually worked with a couple different interrupts without having to know anything about them. Our serial communications involved interrupts, although the Wiring language insulates us from needing to know the details. Also, there is a timer involved in the `delay()` function; thankfully, it is also managed automatically for us.

In this part of the lab exercise, you will setup two different interrupts. The first one will be triggered by the pushbutton; the second, by one of the MSP430 timers.

1. Once again, let’s start with the *Blink* code.

File → Examples → 1.Basics → Blink

2. Save the sketch to a new file.

File → Save As...

Save it to:

C:\MSP430_LaunchPad\Energia\Interrupt_PushButton

3. Before we modify the file, run the sketch to make sure it works properly.

4. To `setup()`, configure the `GREEN_LED` and then initialize it to `LOW`.

This requires two lines of code which we have used many times already.

Adding an Interrupt

Adding an interrupt to our Energia sketch requires 3 things:

- An interrupt source – what will trigger our interrupt. (We will use the pushbutton.)
- An ISR (interrupt service routine) – what to do when the interrupt is triggered.
- The interruptAttach() function – this function hooks a trigger to an ISR. In our case, we will tell Energia to run our ISR when the button is pushed.

5. Interrupt Step 1 - Configure the PushButton for input.

Look back to an earlier lab if you don’t remember how to do this.

6. Interrupt Step 2 – Create an ISR.

Add the following function to your sketch; it will be your interrupt service routine. This is about as simple as we could make it.

```
void myISR()
{
    digitalWrite(GREEN_LED, HIGH);
}
```

In our function, all we are going to do is light the `GREEN_LED`. If you push the button and the Green LED turns on, you will know that successfully reached the ISR.

7. Interrupts Step 3 – Connect the pushbutton to our ISR.

You just need to add one more line of code to your *setup()* routine, the *attachInterrupt()* function. But what arguments are needed for this function? Let's look at the Arduino reference to figure it out.

[Help](#) → [Reference](#)

Look up the *attachInterrupt()* function. What three parameters are required?

1. _____
2. _____
3. _____

Once you have figured out the parameters, **add the function** to your *setup()* function.

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

When you push reset, the code should start over again. This should turn off the GREEN_LED, which you can then turn on again by pushing PUSH2.

Note: Did the GREEN_LED fail to light up? If so, that means you are not getting an interrupt.

First, check to make sure you have all three items – button is configured; *attachInterrupt()* function called from *setup()*; ISR routine that lights the GREEN_LED

The most common error involves setting up the push button incorrectly. The button needs to be configured with INPUT_PULLUP. In this way, the button is held high which lets the system detect when the value falls as the button is pressed.

Missing the INPUT_PULLUP is especially common since most Arduino examples – like the one shown on the *attachInterrupt()* reference page only show INPUT. This is because many boards include an external pullup resistor. Since the MSP430 contains an internal pullup, you can save money by using it instead.

TIMER_A

9. Create a new sketch and call it Interrupt_TimerA

File → New

File → Save As...

C:\MSP430_LaunchPad\Energia\Interrupt_TimerA

10. Add the following code to your new sketch.

```
#include <inttypes.h>

uint8_t timerCount = 0;

void setup()
{
    pinMode(RED_LED, OUTPUT);

    CCTLO = CCIE;
    TACTL = TASSEL_2 + MC_2;
}

void loop()
{
    // Nothing to do.
}

__attribute__((interrupt(TIMERO_A0_VECTOR)))
void myTimer_A(void)
{
    timerCount = (timerCount + 1) % 800;
    if(timerCount == 0)
        P1OUT ^= 1;
}
```

In this case, we are not using the `attachInterrupt()` function to setup the interrupt. If you double-check the Arduino reference, it states the function is used for 'external' interrupts. In our case, the MSP430's Timer_A is an internal interrupt.

In essence, though, the same three steps are required:

- a) The interrupt source must be setup. In our example, this means setting up the timers CCTLO (capture/compare control) and TACTL (TimerA control) registers.
- b) An ISR function – which, in this case, is named “myTimer_A”.
- c) A means to hook the interrupt source (trigger from TimerA) to our function. In this case, we need to plug the Interrupt Vector Table ourselves. The GCC compiler uses the `__attribute__((interrupt(TIMERO_A0_VECTOR)))` line to plug the Timer_A0 vector.

Note: You might remember that we introduced *Interrupts and Timers* in *Chapter 5*. In that lab, the syntax for the interrupt vector was slightly different than it is here. This is because we were using the TI compiler in that lab. Energia uses the open-source GCC compiler, which has a slightly different syntax.

Lab 11e – Blink Fast/Slow by Changing Clocks

We are going to create three different lab sketches in Lab 11d. All of them will essentially be our first 'Blink' sketch, but this time we're going to vary the system clock – which will affect the rate of blinking. We will help you with the required C code to change the clocks, but if you want to study this further, please refer to *Chapter 3 – Initialization and GPIO*.

Blink (with default clock rate)

1. Open the *Blink* example sketch.

File → Examples → 1.Basics → Blink

2. Save the example to a new sketch.

The default Master Clock used by Energia on the MSP430 Launchpad is the DCO running at 16MHz. With that in mind, let's rename our sketch:

C:\MSP430_LaunchPad\Energia\Blink_DCO_16

3. Speed up the initial blink rate by changing the arguments for delay().

When we slow down the clock, it will be much easier to visualize the different clock rates if we speed up the blink rate. Do this by changing the arguments to the delay() functions.

```
digitalWrite(RED_LED, HIGH);  
delay(10); //changed from 1000  
digitalWrite(RED_LED, LOW);  
delay(50); //changed from 1000
```

4. Download and observe sketch.

Do you notice it blinking faster?

Where, oh where, is Main

5. Where is the system clock set?

Before jumping into how to change the MSP430 system clock rate, let's explore how Energia sets up the clock in the first place. Thinking about this, my first question was...

What is the first function in every C program? (This is not meant to be a trick question)

If Energia/Arduino is built around the C language, where is the main() function? Once we answer this question, then we will see how the system clock is initialized.

6. Open main.cpp ... this is where it all starts.

`C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\main.cpp`


The "C:\TI\energia-0101E0009" may be different if you unzipped the Energia to a different location.

When you click the *Download* button, the tools combine your *setup()* and *loop()* functions into the `main.cpp` file included with Energia for your specific hardware.

Main should look like this:

Lab 11e.6: main.cpp

C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\



Clicking download combines sketch with main.cpp to create a valid c++ program

```

// main.cpp
#include < Energia.h >
int main(void)
{
    init();
    setup();
    for (;;) {
        loop();
        if (serialEventRun) {
            serialEventRun();
        }
    }
    return 0;
}

```

Energia.h contains the #defines, enums, prototypes, etc.

System initialization is done in **wiring.c** (see next slide)

We have already seen **setup()** and **loop()**. This is how Energia uses them.

Where do you think the MSP430 clocks are initialized? _____

7. Follow the trail. Open `wiring.c` to find how `init()` is implemented.

`C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\wiring.c`

The `init()` function implements the essential code required to get the MSP430 up and running. If you have already completed *Chapter 3 – Initialization and GPIO*, then you should recognize most of these activities. At reset, you need to perform two essential activities:

- Initialize the clocks (choose which clock source you want use)
- Turn off the Watchdog timer (unless you want to use it, as a watchdog)

The Energia `init()` function takes this three steps further. They also:

- Setup the Watchdog timer as a standard (i.e. interval) timer
- Setup two GPIO pins
- Enable interrupts globally

Lab 11e.7: `init()` in `wiring.c`

`C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\`

```
// wiring.c
void init()
{
    disableWatchDog();
    initClocks();
    enableWatchDogIntervalMode();
    // Default to GPIO (P2.6, P2.7)
    P2SEL &= ~(BIT6|BIT7);
    __eint();
}
enableWatchDogIntervalMode()
initClocks()
disableWatchDog()
enableWatchDog()
delayMicroseconds()
delay()
watchdog_isr ()
```

- ◆ `wiring.c` provides the core files for device specific architectures
- ◆ `init()` is where the default initializations are handled
- ◆ As discussed in [Ch 3](#) (Init & GPIO)
 - ◆ Watchdog timer (WDT+) is disabled
 - ◆ Clocks are initialized (DCO 16MHz)
 - ◆ WDT+ set as interval timer

`initClocks() ...`

Two ways to change the MSP430 clock source

There are two ways you can change your MSP430 clock source:

- Modify the `initClocks()` function defined in `wiring.c`
- Add the necessary code to your `Setup()` function to modify the clock sources

In this lab exercise, we will do the latter method. This has advantages & disadvantages:

Advantages

- Do not need to re-modify `wiring.c` after updating to new revision of Energia
- Changes are explicitly shown in your own sketch
- Each sketch sets its own clocking, if it needs to be changed
- In our lab, it allows us to demonstrate that you can modify hardware registers – i.e. processor specific hardware – from within your sketch

Disdvantages

- Code portability – any time you add processor specific code, this is something that will need to be modified whenever you want to port your Arduino/Energia code to another target platform
- A little less efficient in that clocking gets set twice
- You have to change each sketch (if you always want a different clock source/rate)

8. Write the code to run the MSP430 using the DCO at 1MHz.

Add the following code to your `setup()` function. (We have provided comments for the code, but for a better explanation of this code, please refer back to *Lab 3*.)

```
if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    FaultRoutine();                // If calibration data is erased
                                   // run FaultRoutine()
BCSCTL1 = CALBC1_1MHZ;           // Set range
DCOCTL = CALDCO_1MHZ;            // Set DCO step + modulation

BCSCTL3 |= LFXT1S_2;             // LFXT1 = VLO
IFG1 &= ~OIFG;                  // Clear OSCFault flag
BCSCTL2 |= SELM_0 + DIVM_3;      // MCLK = DCO/8
```

9. Add the FaultRoutine() to the sketch.

It is highly unlikely that you have erased the clock calibration data, but it never hurts to check for it. This fault routine simply sets the Red LED as a warning, then traps the processor in an endless loop. In “real life”, you would want to handle this, but this works for our training.

```
void FaultRoutine(void)
{
    P1OUT = 0x01;                // red LED on
    while(1);                    // TRAP
}
```

10. Download and observe the blink rate of the LED.

Does the LED flash more slowly? _____

Clocking CPU with VLO

11. Save your file to a new sketch: Blink_VLO

File → Save As...

C:\MSP430_LaunchPad\Energia\Blink_VLO

12. Why use VLO?

As our final experiment, let's change the clock source to the VLO – very low frequency oscillator. This internal clock source runs about 12KHz, which is very slow compared to using the DCO.

Why would we want to use the VLO (do you remember this from Chapter 3)?

13. Replace “clock setting code” in *setup()* with the following code to use the VLO.

Sorry, but once again, we're going to defer the explanation of this code snippet back to *Lab 3*.

```
BCSCTL3 |= LFXT1S_2;                // clock system setup
IFG1 &= ~OFIFG;
__bis_status_register(SCG1 + SCG0);
BCSCTL2 |= SELM_3 + DIVM_3;
```

You can delete the Fault Routine, if you would like. We will not use it for the VLO. No calibration data is provided for the VLO. There is a white paper available that describes how to calibrate the VLO from the DCO, though, most users prefer to use an external crystal (called LFXT1) when a low-power, accurate clock is needed.

14. Run the modified sketch.

Can you see the LED blinking? _____

You would have to wait a long time to see it blink. Think of it this way, we saw it blinking when the clock was running at 1MHz. We then changed the clock to 12KHz. That is a very large difference. You might see it 'blink' if you wait long enough.

15. Can you really have an efficient *delay()*? Yes, you can.

What we haven't discussed, up to this point, is that *delay()* is actually quite efficient. It uses a timer, rather than a traditional delay loop. Using the timer, it can put the processor to sleep while it wait for the specified time. Very clever!

If you want to see how this is implemented, examine how it was done in *wiring.c*. You will notice that they change the watchdog timer (WDT) into a standard interval timer, then use that timer to implement *delay()*.

16. Change delay() to use the VLO clock.

So, in step xx we changed the CPU clock rate to be sourced from the slow VLO. But, we didn't change the clock rate used by `delay()`.

Add the following line to your `setup()` function:

```
WDTCTL = WDTPW | WDTTMSSEL | WDTCNTCL | WDTSSSEL | WDT_DIV_BITS;
```

We took this code to setup the watchdog timer from `wiring.c`. The only thing we changed was to set the **WDTSSSEL** bit, which causes the timer to be sourced from VLO.

17. Try to build your code ...

The compile should fail, as `WDT_DIV_BITS` was not defined.

Add the following line to the top of your sketch. Again, we 'stole' this from `wiring.c`.

```
#define WDT_DIV_BITS WDT_MDLY_0_5
```

18. Finally, you should be able to run – and view – the LED blinking slowly.

If it's still too slow for your taste, try changing the `delay()` functions to 1 and 5, respectively, as opposed to 10 and 50.

Sidebar – initClocks()

Here is a snippet of the *initClocks()* function found in `wiring.c`. I say snippet, since I cut out the other CPU speeds that are also available (8 & 12 MHz).

The beginning of this function starts out by setting the calibration constants (that are provided in Flash memory) to their associated clock configuration registers.

Lab 11e (Sidebar): initClocks() in wiring.c

```
void initClocks(void)
{
    #if (F_CPU >= 16000000L)
        BCCTL1 = CALBC1_16MHZ;
        DCOCTL = CALDCO_16MHZ;
    #elif (F_CPU >= 1000000L)
        BCCTL1 = CALBC1_1MHZ;
        DCOCTL = CALDCO_1MHZ;
    #endif

    BCCTL2 &= ~(DIVS_0);
    BCCTL3 |= LFXTLIS_2;

    CSCTL2 &= ~SELM_7;
    CSCTL2 |= SELM_DCOCLK;
    CSCTL3 &= ~(DIVM_3|DIVS_3);

    #if F_CPU >= 16000000L
        CSCTL1 = DCORSEL;
    #elif F_CPU >= 1000000L
        CSCTL1 = DCOFSEL0|DCOFSEL1;
        CSCTL3 |= DIVM_3;
    #endif
}
```

- ♦ F_CPU defined in `boards.txt`
- ♦ Select 'board' via: Tools→Boards

Select correct calibration constants based on chosen clock frequency

- ♦ Set SMCLK to F_CPU
- Set ACLK to VLO (12Khz)

- ♦ Clear main clock (MCLK)
- Use DCO for MCLK
- Clear divide clock bits

Set MCLK as per F_CPU

`delay() ...`

If you work your way through the second and third parts of the code, you can see the BCS (Basic Clock System) control registers being set to configure the clock sources and speeds. Once again, there are more details on this in *Chapter 3* and its lab exercise.

Sidebar Cont'd - Where is F_CPU defined?

We searched high & low and couldn't find it. Finally, after reviewing a number of threads in the Energia forum, we found that it is specified in `boards.txt`. This is the file used by the debugger to specify which board (i.e. target) you want to work with. You can see the list from the Tools→Board menu.

C:\TI\energia-0101E0009\hardware\msp430\boards.txt

```
#####
lpmsp430g2231.name=LaunchPad w/ msp430g2231 (1MHz)
lpmsp430g2231.upload.protocol=rf2500
lpmsp430g2231.upload.maximum_size=2048
lpmsp430g2231.build.mcu=msp430g2231
lpmsp430g2231.build.f_cpu=1000000L
lpmsp430g2231.build.core=msp430
lpmsp430g2231.build.variant=launchpad

#####
lpmsp430g2231f.name=LaunchPad w/ msp430g2231 (16MHz)
lpmsp430g2231f.upload.protocol=rf2500
lpmsp430g2231f.upload.maximum_size=2048
lpmsp430g2231f.build.mcu=msp430g2231
lpmsp430g2231f.build.f_cpu=16000000L
lpmsp430g2231f.build.core=msp430
lpmsp430g2231f.build.variant=launchpad

#####
lpmsp430g2553.name=LaunchPad w/ msp430g2553 (16MHz)
lpmsp430g2553.upload.protocol=rf2500
lpmsp430g2553.upload.maximum_size=16384
lpmsp430g2553.build.mcu=msp430g2553
lpmsp430g2553.build.f_cpu=16000000L
lpmsp430g2553.build.core=msp430
lpmsp430g2553.build.variant=launchpad

#####
lpmsp430fr5739.name=FraunchPad w/ msp430fr5739
lpmsp430fr5739.upload.protocol=rf2500
lpmsp430fr5739.upload.maximum_size=15872
lpmsp430fr5739.build.mcu=msp430fr5739
lpmsp430fr5739.build.f_cpu=16000000L
lpmsp430fr5739.build.core=msp430
lpmsp430fr5739.build.variant=fraunchpad

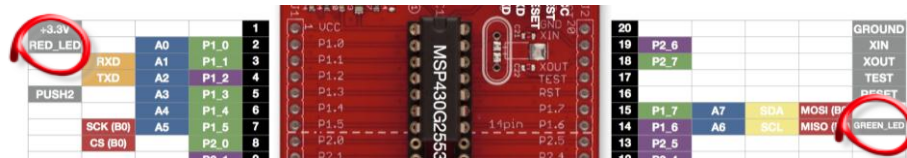
#####
```

Lab Debrief

Q&A: Lab11A (1)

Lab A

3. Do you see the LED blinking? What color LED is blinking? Red
 What pin is this LED connected to? Pin 2
 (Code says Pin14, it was RED that blinked)
 (Be aware, in the current release of Energia, this could be a trick question.)



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
  pinMode(RED_LED, OUTPUT);
}
```

Q&A: Lab11A (2)

5. How can you change which color LED blinks?
 Examine the H/W pin mapping for your board to determine what needs to change.
 Please describe it here: Change from Pin 2 to Pin 14, for the green to LED blink
 (Easy, Just use the pre-defined symbol: GREEN_LED)
 6. Make the other LED blink.
 Did it work? Yes

Q&A: Lab11B (1)

2. Try out the sketch.

When you push the button the (GREEN or RED) LED goes (ON or OFF)?

Green LED goes OFF

Examine the code

3. How is the LED defined differently in the 'Button' Sketch versus the 'Blink' sketch?

In 'Blink', the LED was #defined (as part of Energia);

in 'Button', it was defined as a const integer. Both work equally well.

4. How is the pushbutton created/used differently from the LED?

In Setup() it is configured as an 'input'; in loop() we use digitalRead()

What "Energia" pin is the button connected to? Pin 5

What is the difference between INPUT and INPUT_PULLUP?

INPUT config's the pin as a simple input – e.g. allowing you to read pushbutton.

Using INPUT_PULLUP config's the pin as an input with a series pullup resistor;

(many TI μ C provide these resistors as part of their hardware design).

Q&A: Lab11B (2)

5. Just like standard C code, we can create variables. What is the global variable used for in the 'Button' example?

'buttonState' global variable holds the value of the button returned by digitalRead().

We needed to store the button's value to perform the IF-THEN/ELSE command.

What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

It would be more efficient to let the button 'interrupt' the processor, as opposed to

reading the button over and over again. This is as the processor cannot SLEEP

while polling the pushbutton pin. If using an interrupt, the processor could sleep until

being woken up by a pushbutton interrupt.

(Note, we will look at this later.)

Reverse Button/LED action

6. Did it work? Yes (it should)

```
if (buttonState == HIGH) {
  // turn LED on:
  digitalWrite(ledPin, HIGH);
}
else {
  // turn LED off:
  digitalWrite(ledPin, LOW);
}
```

LOW (crossed out) **HIGH** (crossed out)

Q&A: Lab11C (1)

4. Did you see numbers in the serial monitor? Yes, if ...

If you didn't see anything in the Serial Monitor, what did you need to change?

Change the serial-port jumpers



Q&A: Lab11C (2)

Blink with Serial Communication (Serial_Button sketch)

8. Did you see the Serial Monitor and LED changing when you push the button?

You (we hope so)

```
void setup() {  
  Serial.begin(9600);  
  
  // initialize the LED pin as an output  
  pinMode(ledPin, OUTPUT);  
  
  void loop() {  
    // read the state of the pushbutton  
    buttonState = digitalRead(buttonPin);  
    Serial.println(buttonState);  
  }
```

9. Considerations for debugging... How you can use both of these items for debugging? (Serial Port and LED)

Use the serial port to send back info, just as you might use printf() in your C code.

An LED works well to indicate you reached a specific place in code. For example, later on we'll use this to indicate our program has jumped to an ISR (interrupt routine)

Similarly, many folks hook up an oscilloscope or logic analyzer to a pin, similar to using an LED. (Since our boards have more pins than LEDs.)

Q&A: Lab11C (3)

Another Pushbutton/Serial Example (StateChangeDetection sketch)

11. Examine the sketch, download and run it.

How is this sketch different? What makes it more efficient?

It only sends data over the UART whenever the button changes

How is this (and all our sketches, up to this point) inefficient?

Our pushbutton sketches – thusfar – have used polling to determine the state of the button. It would be more efficient to let the processor sleep; then be woken up by an interrupt generated when the pushbutton is depressed.

Q&A: Lab11D (1)

Interrupt Example (Interrupt_PushButton)

7. Look up the `attachInterrupt()` function. What three parameters are required?

1. Interrupt source – in our case, it's PUSH2 (or you could use 5, for the pin #)
2. ISR function to be called when int is triggered – for our ex, it's "myISR"
3. Mode – what state change to detect; the most common is "FALLING"

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

Notes:

- ◆ Use reset button to start program again and clear GREEN_LED
- ◆ Most common error, not configuring PUSH2 with `INPUT_PULLUP`.

Q&A: Lab11E (1)

5. What is the first function in every C program? (This is not meant to be a trick question)
_____ **main()** _____
6. Where do you think the MSP430 clocks are initialized? _____ **init() call inside main()** _____
10. Download and observe the blink rate of the LED. (Blink_DCO_1.ino)
Does the LED flash more slowly? _____ **yes (but it's hard to tell it's 16x slower)** _____
11. Why use VLO? _____ **very low power (but not very accurate)** _____