**Getting Started with the MSP430 Launchpad: The digital stream, Part 1**

Adrian Fernandez and Dung Dang - October 08, 2014

**Editor's Note:** For individuals looking to explore MCUs and MCU-based design principles, low-cost development kits offer an easy, low-risk entry into the intricacies of these complex ICs. In this detailed guide to the TI LaunchPad development kit, the authors break down hardware circuits and walk through MCU-based projects designed to help users work with the LaunchPad - the little red board that do so many big things, not the least of which is helping individuals gain knowledge and experience in MCU-based design. In this installment excerpted from the book, the authors review the basics of digital I/O and walk through examples of I/O design using the TI MSP430 MCU.

Elsevier is offering this and other engineering books at a 25% discount. To use this discount, click on the coupon to the right and use code PBTY14 (valid through March 31, 2015) during checkout.

Adapted from *Getting Started with the MSP430 Launchpad, 1st Edition* by Adrian Fernandez and Dung Dang (Newnes)

---

**Chapter 9. 1s and 0s Revisited: The Digital Stream**

**9.1 The Flexible Digital World**

In the previous chapter, we learned about how the LaunchPad can interact with the vast and colorful world of analog. When you compare analog’s world of infinite colors to the black and white world of digital, one might come to the conclusion that analog is greatly superior. Well, let’s put that conclusion away because we have a few clarifications to make.

It is true that 1s and 0s are quite limited in terms of the information they can carry individually. It is also quite unfortunate that the microcontroller’s brain can only understand and interpret these 0s and 1s. But don’t forget the fact that the LaunchPad brain can digest and process those 0s and 1s very quickly. Millions or even up to billions of computations can happen every second depending on the microcontroller. Our LaunchPad can do up to 16 million computations made up of these 0s and 1s every second.

Thanks to this speed, our LaunchPad can actually send out a stream of 1s and 0s in various patterns to create more information beyond just ON and OFF, HIGH and LOW, or black and white. The first method that we’ll introduce is PWM or Pulse-Width Modulation.
9.2 Pulse-Width Modulation

Define Pulse-Width Modulation

Pulse-width modulation is a digital technique to control a signal by repeatedly toggling a signal between a HIGH and a LOW state in a consistent pattern. We can portray new information by changing how long the signal is HIGH versus LOW. The PWM signal has two key parameters—frequency and duty cycle.

The period of the PWM signal (measured in seconds) indicates the amount of time it takes for the signal to complete one cycle. Refer to the image above. We can see that a period is a complete cycle of the signal, changing from LOW signal to a HIGH signal and back to a LOW signal. The signal pattern of each period is repeated over and over again.

PWM frequency

The frequency of the PWM signal, measured in Hertz (Hz), indicates the number of complete periods that occur in 1 second. In our LaunchPad, the PWM signal typically has a frequency of 490 Hz. This means that our PWM signal completes 490 periods every second. The period and frequency of a signal are closely related, in that the faster the frequency, the smaller the period. The slower the frequency, the larger the period.

The relationship between period and frequency is realized with this equation:

\[ \text{Period(s)} = \frac{1}{\text{Frequency(Hz)}} \]

For example, the period of a 1-kHz (1000 Hz) signal is \(\frac{1}{1000 \text{ Hz}} = 0.001 \text{ s or } 1 \text{ ms}\).

The Duty Cycle of the PWM signal is the percentage of time that our PWM signal is in a HIGH state versus a LOW state. For example, a simple square wave has a duty cycle of 50%. However, we can change the duty cycle of our PWM signal, anywhere from 0% (for always OFF) to 100% (for always ON) and anywhere in between, depending on what type of information we want to portray.
And that’s it! PWM is a simple digital signal and is a great example of how we can start to extract more information out of the digital brain of our LaunchPad.

PWM is primarily useful because of the ability to choose the duty cycle from a fixed frequency digital signal. Once fixed on a specific frequency, 490 Hz in our example, the LaunchPad can convey different information to other “intelligent” devices by varying the duty cycle from 0% to 100%. The PWM signal receiver can decode the duty cycle into different available actions and behave accordingly.

Actually, PWM can also be used to control the power that is fed into certain electrical devices. By changing the duty cycle of a PWM signal, we can actually simulate an average voltage (or current) output. The longer the duty cycle is, relative to the period, means that, on average, the signal is more HIGH than LOW. This means on average more voltage (or current) is seen at the digital output pin. If we start at a duty cycle of 0%, we should see an effective average voltage of 0 V. Increasing duty cycle to 50%, we’ll start to see an effective average voltage of ~1.8 V (VCC/2). And lastly, at a duty cycle of 100%, we should see an effective voltage of ~3.6 V (VCC).

As a matter of fact, this is exactly how we created the analog output signal in our previous chapter using our `analogWrite()` function.

### 9.3 WHAT? `ANALOGWRITE()` IS A LIE

Sadly so. The MSP430 device on the MSP430 LaunchPad does not actually have a true analog output module, which is often called a digital-to-analog converter (or DAC). Now, some other MSP430 devices do have the actual DAC modules, but the MSP430G2553 device on our LaunchPad does not.

#### PWM signal components

If we take a close up look at our PWM signal, the ON and OFF components inside of a single period look very discrete and separated. However, the faster the frequency of our PWM signal, the harder it is to discern between the ON and OFF state and the signal appears to be averaged out. To explain this concept, think of an animated flipbook. If we flipped this animation one time per second, our animation would look jerky and very unconvincing. However, if we increase the frequency of our flipbook to change frames 60 times per second, suddenly the animation is smooth and realistic.
Similarly, we can simulate an average voltage output at a pin by varying the duty cycle at a pin when frequency is high enough. Fortunately for us, the default 490 Hz frequency of our LaunchPad is quick enough to fool most external devices into thinking a stable voltage is present.

A component can perceive the PWM signal as an average voltage signal that is somewhat between VCC and 0 V. The value of this averaged voltage signal can be determined by the relative ON duration of the signal in each period or cycle. For example, if the duty cycle of the signal is 75%, the averaged out voltage level can appear to be ~75% of VCC.

\[
\text{Average signal (V)} = \text{Duty cycle (\%)} \times \text{VCC (V)}
\]

So just to reiterate, the `analogWrite()` function that we played with last chapter actually isn’t analog at all! It’s truly a digital function that simulates an analog output by varying the duty cycle of a PWM output signal.

The parameter that we passed into our `analogWrite()` function was simply telling the PWM signal what duty cycle it should toggle at. Because our `analogWrite()` function can accept an input parameter between (0-255), we can tell our PWM signal to generate up to 256 unique duty cycles. An input of 255 causes a duty cycle of 100%, while an input of 0 creates a duty cycle of 0%. This input maps linearly and we can get a good idea of what average voltage we can expect to see at the digital output pin for `analogWrite(analogWriteInput)` using the equation below:

\[
\text{Average voltage} = \text{VCC} \times (\text{analogWriteInput} = 255)
\]

So we were using PWM signals without even knowing it! In Chapter 8, we used this digital trick to change the intensity of LEDs. We even used three PWM signals to change the intensity of three LEDs to create a wide array of colors.

As we have actually spent quite some time with our PWM signal between this chapter and the last, let’s spend the rest of the chapter learning more about a special type of PWM signal—the square
wave.

9.4 Project 17: Square Wave And A Buzzer

A square wave is a unique PWM signal which has a duty cycle of 50%—this means that the signal is ON just as much as it is OFF. In this project, we will use a square wave to make some noise using a piezo buzzer.

Ingredients:

1. LaunchPad
2. Educational BoosterPack
   OR
   Breadboard, piezo buzzer (2.048 kHz), 33 Ω resistor

Piezo buzzers

Piezo buzzers are used for making bleeps, bloops, and tones. We can generate these types of sounds by providing a square wave to a piezo buzzer. The frequency of the square wave (Hz) determines the pitch of the noise the buzzer generates. The slower the frequency, the lower the pitch. The higher the frequency, the higher the pitch.

The buzzer that comes with the Educational BoosterPack is pretty small, but it does pack quite a punch. If you have the Educational BoosterPack, simply plug it into the LaunchPad. As always, be sure to have the LED jumpers removed (P1.0 and P1.6) and position the TXD and RXD jumpers to enable the LaunchPad to send diagnostic information to the computer for simple debugging.

If you are not using the Educational BoosterPack, you can wire up an external buzzer as shown in the following diagram. Connect one pin to ground and the other pin to a PWM-capable pin on the microcontroller (the piezo is bi-directional so either pin can be connected to GND or the MSP430). In this example, we are going to connect it to the XIN pin (pin 19). This is exactly the same way the buzzer on the Educational BoosterPack is connected to our LaunchPad.
Once your hardware is all set, open up a new sketch inside of Energia and type in the following lines of code.

```cpp
const int buzzerPin = 19;
void setup() {
}

void loop() {
  tone(buzzerPin, 698, 500); // 698Hz = Note F5. Hold note for 500ms
delay(500); // delay between tones
}
```

Once done typing, click Verify and Download. Once it’s loaded up, the sketch should play the note F5 for half a second, pause for half a second, and repeat the tone again.

There is a new function that is introduced in this sketch. The `tone(buzzerPin, 698, 500)` function call instructs the LaunchPad to generate a 698-Hz tone for 500 ms at the buzzerPin. This `tone()` function is very similar to our `analogWrite()` function. The main difference, however, is that the parameter in `analogWrite()` changed the duty cycle while frequency stayed constant. With `tone()`, our parameter changes the frequency, while duty cycle stays constant at 50%. We also get one more parameter to play with, which tells the `tone()` function how many milliseconds it should hold the note for. Unlike most, our LaunchPad can hold a pitch quite well!

The time parameter is an optional one. If the time parameter isn’t provided, then the specified pitch will be held indefinitely.

If a time parameter isn’t provided, we need to use another function to stop the square wave. The `noTone(buzzerPin)` can be used to tell the LaunchPad to stop with all the noises that might be sourced by the buzzerPin.

Stay tuned for Part 2 with more examples of I/O design using the TI LaunchPad.

About the authors: Adrian Fernandez and Dung Dang

Copyright © 2014 Elsevier, Inc. All rights reserved.
Printed with permission from Newnes, a division of Elsevier. Copyright 2014. For more information on this title and other similar books, please visit www.newnespress.com.