Using hardware and software to make new stuff

Search

type, hit enter

Categories

- 3D Printing (1)
- Affinity Photo (1)
- Aide-memoir (1)
- Analog (1)
- Ansible (3)
- Dates to Remember (3)
- Docker (1)
- Electronics (144)
- ESP8266 (12)
- FPGA (4)
- Garden (1)
- General (8)
- Home Built CPU (6)
- Informatica (1)
- Internet of Things (8)
- iOS (2)
- KiCad (4)
- MSP430 (2)
- Netduino (49)
- NuttX (9)
- Photography (3)
- Pico (10)
- Raspberry Pi (17)
- Silverlight (6)
- Software Development (88)
- STM32 (5)
- STM8 (54)
- Tips (5)

Archives

- July 2024
- June 2024
- May 2024
- April 2024
- March 2024
- February 2024
- January 2024
- December 2023
- November 2023
- October 2023

- April 2023
- February 2020
- January 2020
- July 2019
- February 2018
- July 2017
- June 2017
- April 2017
- March 2017
- February 2017
- October 2016
- September 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- January 2016
- December 2015
- November 2015
- October 2015
- August 2015
- June 2015
- May 2015
- April 2015
- March 2015
- February 2015
- January 2015
- December 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- October 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- March 2013
- January 2013
- November 2012
- October 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012

- October 2011
- September 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- April 2010

Generating PWM Signals using the STM8S

In a recent post we looked at the generation of a square wave signal using a timer and the update/overflow interrupt. There we generated a 20 Hz signal from the STM8S by toggling an output port using direct access to a port configured as an output port. In this post we will go one step further and use the capabilities of the timer to generate the pulse directly. We will also look at how we can manipulate the registers to allow the generation of a PWM pulse from the STM8S by simply changing the register values used to configure the port.

It is important that you read and understand the previous post Using Timers on the STM8S before continuing further as we will be using much of the knowledge in that post here.

So the project definition here is simple to start with, we will generate a square wave without using GPIO ports. We will then follow up on this by changing the values in the registers to generate a PWM signal with a duty cycle which can be defined by the programmer.

The Registers

The application will use most of the registers described in the previous post as well as the following:

1. TIM2_CCR1H & TIM2_CCR1L – Capture/Compare Register 1 High/Low
2. TIM2_CCER1 – Capture/Compare Enable register 1
3. TIM2_CCMR1 – Capture/Compare Mode Register 1

TIM2_CCR1H & TIM2_CCR1L – Capture/Compare Register 1 High/Low

These two registers are analogous to the TIM2_ARRH and TIM2_ARRL registers. TIM2_ARRH/L are used to determine the period of the signal whilst TIM2_CCR1H/L are used to determine the duty cycle of the signal. Let us assume that we are using the value of 50,000 for TIM2_ARRH/L as in the last post then by setting TIM2_CCR1H/L to 25,000 will give a duty cycle of 50%. Similarly, setting TIM2_CCR1H/L to 12,500 with give a duty cycle of 25% (or 75%) depending upon the register settings for active high/low – see TIM2_CCER1.

TIM2_CCER1 – Capture/Compare Enable register 1

We will be using two bits in this register, Capture/Compare 1 Output Polarity (CC1P) and Capture/Compare output Enable (CC1E).

So let's start with the easy one, CC1E. This simply enables or disables the capture/compare for channel 1 of Timer 2. Setting this to 1 enables the mode, setting this to 0 disables the mode.

as consider a simple example, namely a PWM signal with a duty cycle of 50%. So, for 50% of the time the signal is logic 1 (high) and for 50% of the time the signal is logic 0 (low). Or another way of looking at it is that if we define high to be active and low to be inactive then for 50% of the time the signal is active and 50% of the time the signal is inactive.

CC1P allows us to define what we mean by active and inactive. Once we have the application written we can change this value and see the effect on the output.

TIM2_CCMR1 – Capture/Compare Mode Register 1

This register allows the application to change the way in which the channel is configured. In this case we will only be concerned with setting this to one of two values, namely 6 or 7.

Value	Mode	Description
110 – 6	PWM Mode 1	In up-counting mode, the channel is active if the counter is less than CCR1, otherwise it is inactive. In down-counting mode the channel is inactive when the counter is greater than CCR1, otherwise the channel is inactive.
111 – 7	PWM Mode 2	In up-counting mode the channel is inactive as long as the counter is less than CCR1.

Software

The first two things we will do is steal some code from previous posts, namely the Configuring the System Clock and Using Timers on the STM8S. We will use the *InitialiseSystemClock* and *InitialiseTimer2* methods respectively.

The next thing we need to consider is how we set up the timer. We will continue to use Timer 2 so we can again use some of the code from previous posts. However, we need to make a few modifications to *SetupTimer2* method.

So let's start by having a 25% duty cycle (25% high, 75% low). At the moment we are not too worried about the frequency of the signal so let's work with TIM2_ARRH/L set to 50,000 as in the previous post. This means that we want the output low for 75% of the time (37,500 counts) and high for 25% of the time (12,500 counts). Time for the first decision, let's use PWM mode 1 (TIM2_CCMR1_OC1M = 6).

Given the default mode (down-counting) and looking at the register definition for TIM2_CCMR1_OC1M we want to define active as a logic 0 and inactive as a logic 1. So this means we need to set TIM_CCMR1_OC1M = 0.

If we put all of this together we end up with the following method:

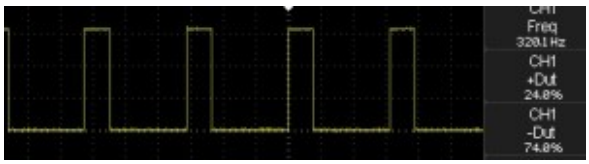
```
    TIM2_ARRL = 0x50;       //  Low byte of 50,000.
    TIM2_CCR1H = 0x30;      //  High byte of 12,500
    TIM2_CCR1L = 0xd4;      //  Low byte of 12,500
    TIM2_CCER1_CC1P = 0;    //  Active high.
    TIM2_CCER1_CC1E = 1;    //  Enable compare mode for channel 1
    TIM2_CCMR1_OC1M = 6;    //  PWM Mode 1 - active if counter < CCR1, inacti
    TIM2_CR1_CEN = 1;       //  Finally enable the timer.
}
```

And as we are using a timer to do all of the work, our main method becomes:

```
void main()
{
    //
    //  Initialise the system.
    //
    __disable_interrupt();
    InitialiseSystemClock();
    InitialiseTimer2();
    SetupTimer2();
    __enable_interrupt();
    while (1)
    {
        __wait_for_interrupt();
    }
}
```

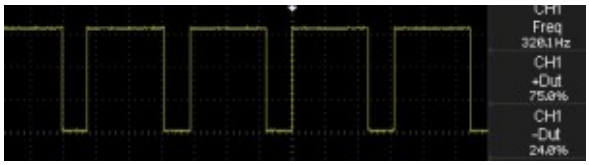
Let's Look at the Output

If we look at the clock settings (16MHz) and the value for TIM2_ARR (50,000) we should be looking for a signal with a frequency of around 320 Hz (16 MHz / 50,000). So compiling the above we are expecting a 320Hz signal with a duty cycle of 25%. Here is the output on the oscilloscope:



PWM with a 25% high signal

Changing the value of TIM2_CCER1_CC1P to 1 gives the following:



PWM with a 75% high signal

As you can see, the polarity allows us to change what we mean by active.

Some ideas for you:

Source Code Compatibility

System	Compatible?
STM8S103F3 (Breadboard)	✓
Variable Lab Protomodule	✓
STM8S Discovery	✓

Tags: Electronics, Software Development, STM8

Thursday, August 30th, 2012 at 10:36 pm • Electronics, Software Development, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

Pages

- About
- Making a Netduino GO! Module
- The Way of the Register
- Making an IR Remote Control
- Weather Station Project
- NuttX and Raspberry Pi PicoW

Support This Site

Currently Reading