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

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

Generating a Regular Pulse Using Timer 2

In previous posts you may have seen an example program where we generate a 20Hz signal using the overflow interrupt on Timer 2. Here we will translate the post to use direct register access rather than use the STD Peripheral Library.

So the project definition is simple, output a regular signal (20Hz with a 50% duty cycle) on Port D, Pin 4 (i.e. pin 2 on the STM8S103F3P3).

Algorithm

To make this project a low on processor power we will use interrupts to generate the pulse. To do this we will make use of one of the STM8S timers, namely Timer 2 (T2). The algorithm becomes:

1. Turn off the timer
2. Setup the timer to generate an interrupt every 1 / 40th of a second
3. Set up the output port to generate the signal.
4. Wait for interrupts indefinitely

The Interrupt Service Routine (ISR) then has one very simple task, toggle the output port and wait for the next interrupt.

The Registers

This application is simple and really only uses a fraction of the power of the STM8S timers. In fact we can set up the chip using relatively few registers. With the exception of resetting the timer to a known state we will be using only six registers in this exercise:

1. TIM2_PSCR
2. TIM2_ARRH and TIM2_ARRL
3. TIM2_IER
4. TIM2_CR1
5. TIM2_SR1

TIM2_PSCR – Timer 2 Prescalar

The 16-bit counter in Timer 2 receives a clock signal from the prescalar. This in turn receives a clock from the internal clock of the STM8S (i.e. f_{master}). The prescalar divides the f_{master} clock by the prescalar set in the TIM2_PSCR register. This allows the timer to receive a slower clock signal than that running the STM8S. The prescalar is a power of 2 and the effective frequency of the clock running Timer 2 is given by the following formula:

TIM2_PSCR is a 4 bit number and this restricts the value of the prescalar to 1 to 32,768.

We will come back to this formula when we write the software in order to calculate the prescalar we will need to generate the 20Hz clock signal.

TIM2_ARRH and TIM2_ARRL – Counter Auto-Reload Registers

We will be using the counter as a simple up/down counter. We will be loading this register with a counter value which the timer will count up to / down from. An interrupt will be generated when the counter value has been reached (for up) or zero is reached (for down). The counter will then be reset using the values in these two registers.

The only important thing to note about these two registers is that TIM2_ARRH must be loaded with a value before TIM2_ARRL.

TIM2_IER – Interrupt Enable Register

This register determines which interrupts Timer2 can generate. In our case we only need one, namely the update interrupt. This is generated when the counter value has been reached.

The interrupt is enabled by setting TIM2_IER_UIE to 1.

TIM2_CR1 – Timer 2 Control Register 1

The only bit we will be interested here is the Counter ENable bit (CEN). This will be used to start the counter.

TIM2_SR1 – Timer 2 Status Register 1

This register gives us status information about the timer. There is only one bit we are interested in for this exercise and that is the Update Interrupt Flag (UIF). This bit determines if an update interrupt is pending. The bit is set by hardware but crucially it must be reset by software.

When we enter the ISR, this bit will have been set by the hardware controlling the timer. On existing the ISR the hardware will check the status of the bit. If it is set then the interrupt will be generated once more. This means that if we are not careful then we can end up in a cycle of generating an interrupt, processing the interrupt in the ISR and then generating the interrupt again ad infinitum. It is therefore crucial that this bit is cleared before the ISR is exited.

Software

One of the first things to note is that as with all of the examples we will discuss in this series, we will assume a clock running using the internal oscillator and set to 16MHz.

The code which will deal with the interrupt has a very simple job to do, namely toggle the pin we are using to generate the output pulse. One thing to note it that as we are toggling the pin in this method we will effectively be halving the output frequency of the signal which has been generated. Lets look at what is happening.

1. ISR 1 – output is low we will make the output high.
2. ISR 2 – output is high we will make the signal low
3. ISR 3 – output is low we will make the signal high
4. etc.

The frequency of the output for a regular signal is determined by the amount of time between the two rising edges of the output. So in our case, the time is double the frequency of the calls to the ISR as we toggle the

post.

```
//
//  Timer 2 Overflow handler.
//
#pragma vector = TIM2_OVR_UIF_vector
__interrupt void TIM2_UPD_OVF_IRQHandler(void)
{
    PD_ODR_ODR4 = !PD_ODR_ODR4;     //  Toggle Port D, pin 4.
    TIM2_SR1_UIF = 0;               //  Reset the interrupt otherwise it will
}
```

If you have been following the series, the next piece of code should also be familiar (see the Simple GPIO example). We will be setting up Port D, pin 4 to be an output port. This is the pin which will output the signal we will be generating.

```
//
//  Setup the port used to signal to the outside world that a timer even has
//  been generated.
//
void SetupOutputPorts()
{
    PD_ODR = 0;             //  All pins are turned off.
    PD_DDR_DDR4 = 1;        //  Port D, pin 4 is used as a signal.
    PD_CR1_C14 = 1;         //  Port D, pin 4 is Push-Pull
    PD_CR2_C24 = 1;         //  Port D, Pin 4 is generating a pulse under 2 M
}
```

The next method resets Timer 2 and put it into a known state. This simply requires resetting all of the Timer 2 registers to 0.

```
//
//  Reset Timer 2 to a known state.
//
void InitialiseTimer2()
{
    TIM2_CR1 = 0;               // Turn everything TIM2 related off.
    TIM2_IER = 0;
    TIM2_SR2 = 0;
    TIM2_CCER1 = 0;
    TIM2_CCER2 = 0;
    TIM2_CCER1 = 0;
    TIM2_CCER2 = 0;
    TIM2_CCMR1 = 0;
    TIM2_CCMR2 = 0;
    TIM2_CCMR3 = 0;
    TIM2_CNTRH = 0;
    TIM2_CNTRL = 0;
    TIM2_PSCR = 0;
    TIM2_ARRH  = 0;
    TIM2_ARRL  = 0;
    TIM2_CCR1H = 0;
    TIM2_CCR1L = 0;
```

```
    }
```

The next thing we need is a method which sets the Timer 2 to generate the interrupt. This is where we need to start doing some calculations.

So let's start with the frequency of the clock going into the counter for Timer 2. As we have seen earlier, this is given by the following:

$f_{counter} = f_{master} / 2^{TIM2\_PSCR}$

Now we also know that the interrupts will be generated every time the counter value is reached. So the frequency of the interrupt is given by the following:

$f_{interrupt} = f_{counter} / counter$

Putting the two together we get the following:

$f_{interrupt} = f_{master} / (2^{TIM2\_PSCR} * counter)$

A little rearranging gives:

$(2^{TIM2\_PSCR} * counter) = f_{master} / f_{interrupt}$

If we plug in the numbers we know, f_{master} = 16MHz and $f_{interrupt}$ = 40 (remember that the frequency of the signal we are generating is half the frequency of the interrupts) then we find:

$(2^{TIM2\_PSCR} * counter) = 400{,}000$

So, if we take 400,000 and divide by 50,000 (for simplicity) then we have a factor of 8. So, given that the counter is a 16-bit counter then the counter should be 50,000 and the prescalar should be 3 ($2^3 = 8$).

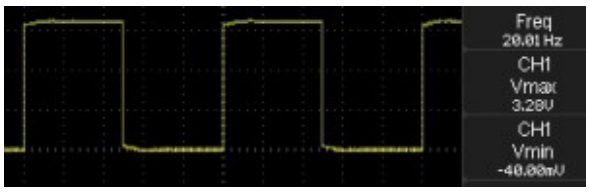
```
//
//  Setup Timer 2 to generate a 20 Hz interrupt based upon a 16 MHz timer.
//
void SetupTimer2()
{
    TIM2_PSCR = 0x03;       //  Prescaler = 8.
    TIM2_ARRH = 0xc3;       //  High byte of 50,000.
    TIM2_ARRL = 0x50;       //  Low byte of 50,000.
    TIM2_IER_UIE = 1;       //  Enable the update interrupts.
    TIM2_CR1_CEN = 1;       //  Finally enable the timer.
}
```

Now we have all of the component parts we simply need to call the methods to set everything up and then wait for the interrupts to fire. So our main method looks like this:

```
//
//  Main program loop.
//
int main( void )
{
    __disable_interrupt();
    SetupOutputPorts();
```

```
        {
            __wait_for_interrupt();
        }
    }
```

Running this application results in the following trace on the oscilloscope:



20 Hz Square Wave Output on an Oscilloscope

A quick look at the measurements shows that this application is generating a 20Hz signal. If you don't have a scope then you can hook a LED (use the LED circuit from the previous article on external interrupts) through a transistor. You should be able to see the LED flicker as it is turned on and off. You could also slow down the rate of the pulses by looking at changing the prescalar or adding a counter.

As always, the source code is available for download.

Source Code Compatibility

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

Tags: Electronics, STM8, The Way of the Register

Wednesday, August 29th, 2012 at 7:32 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