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

Timer 1 Counting Modes

In this article we will continue to look at Timer 1, specifically:

- counting modes
- repetition counter
- update/overflow events

This article will assume some knowledge from the following two posts published previously:

- Generating a Regular Pulse Using Timer 2
- Generating PWM Signals on the STM8S

Unlike previous posts we will not be resetting the system clock but will instead leave this running using the default 2 MHz internal HSI oscillator.

Test Environment

In order to demonstrate the features of the timer I will be using my Saleae Logic Analyser as we will be observing events which occur many times a second. I have the following connections set up:

Saleae	STM8S Pin	Description
Ground	N/A	Ground
Black	PD5	Indicate the state of the Timer 2 overflow interrupt
Brown	Timer 1, Channel 4	PWM signal from Timer 1
Red	PD4	Indicate state of Timer 1 (running or halted)

This set up will result in a series of charts from the Logic software which look like this:

Logic Analyser Output

The top portion of the display (labelled TIM2 Overflow) indicates when an overflow event has occurred on Timer 2. This timer is used to control the application. A change from low to high starts Timer 1 and the timer window for the observation runs to the next transition from high back to low. There is then a pause and then the whole cycle starts again.

One thing that you can do to make the capture of these traces easier is to set a sample trigger on one of the traces. If you look at the trace labelled *TIM2 Overflow* you will see that one of the four square buttons is highlighted. This is showing that capture of data will begin when the logic analyser detects a signal which changes from low to high. You can make use of this by deploying and starting the application as follows:

1. Compile and deploy the application
2. Start data capture on the logic analyser
3. Start the application

When you click the *Start* button on the logic analyser (step 2) a window will appear which indicates that the software is monitoring the data looking for a trigger (in this case rising edge on PD5) before it will start to capture data.

The Registers

The example code we will use shows how we can change the properties of the output and also the frequency of the interrupts generated by using the following registers:

- TIM1_CR1_DIR – Counter Direction
- TIM1_RCR – Repetition Counter

TIM1_CR1_DIR – Counter Direction

This register determines the direction of the counter, either up from 0 to the auto-reload values (set to 0) or down from the auto-reload value (set to 1).

TIM1_RCR – Repetition Counter

This counter can be used to determine the frequency of the overflow interrupts. Normally (i.e. by setting this register to 0) an overflow/underflow interrupt is generated every time the counter is reloaded from the auto-reload registers. By setting this register to any number other than zero (i.e. n), the overflow/underflow interrupt will only be generated after *n + 1* overflow/underflows have been detected.

Software

We will use the registers described above to generate a series of 5 pulses every 50 mS. The code to do this is as follows:

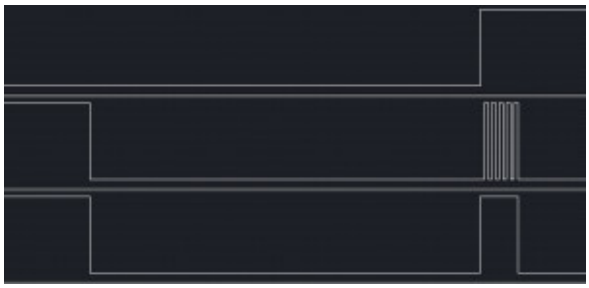
```
1  #if defined DISCOVERY
2      #include <iostm8S105c6.h>
3  #elif defined PROTOMODULE
4      #include <iostm8s103k3.h>
5  #else
6      #include <iostm8s103f3.h>
7  #endif
8  #include <intrinsics.h>
9
10 //-------------------------------------------------------------------------
11 //
12 //  Timer 2 Overflow handler.
13 //
14 #pragma vector = TIM2_OVR_UIF_vector
15 __interrupt void TIM2_UPD_OVF_IRQHandler(void)
```

```
21          PD_ODR_ODR4 = 0;
22      }
23      else
24      {
25          //
26          //  Force Timer 1 to update without generating an interrupt.
27          //  This is necessary to makes sure we start off with the correct
28          //  number of PWM pulses for the first instance only.
29          //
30          TIM1_CR1_URS = 1;
31          TIM1_EGR_UG = 1;
32          //
33          //  Reset the indicators.
34          //
35          PD_ODR_ODR5 = 1;
36          PD_ODR_ODR4 = 1;
37          //
38          //  Enable Timer 1
39          //
40          TIM1_CR1_CEN = 1;           //  Start Timer 1.
41      }
42      TIM2_SR1_UIF = 0;               //  Reset the interrupt otherwise it wil
43  }
44
45  //-------------------------------------------------------------------------
46  //
47  //  Timer 1 Overflow handler.
48  //
49  #pragma vector = TIM1_OVR_UIF_vector
50  __interrupt void TIM1_UPD_OVF_IRQHandler(void)
51  {
52      PD_ODR_ODR4 = !PD_ODR_ODR4; //0;                //  Signal to the user t
53      TIM1_CR1_CEN = 0;               //  Stop Timer 1.
54      TIM1_SR1_UIF = 0;               //  Reset the interrupt otherwise it wil
55  }
56
57  //-------------------------------------------------------------------------
58  //
59  //  Set up Timer 1, channel 3 to output a single pulse lasting 240 uS.
60  //
61  void SetupTimer1()
62  {
63      TIM1_ARRH = 0x03;       //  Reload counter = 960
64      TIM1_ARRL = 0xc0;
65      TIM1_PSCRH = 0;         //  Prescalar = 0 (i.e. 1)
66      TIM1_PSCRL = 0;
67      //
68      //  Select 0 for up counting or 1 for down counting.
69      //
70      TIM1_CR1_DIR = 0;       //  Up counter.
71      //
72      //  Select 0 for edge aligned, 1 for mode 1 centre aligned,
73      //  2 for mode 2 centre aligned or 3 for mode 3 centre aligned.
74      //
75      TIM1_CR1_CMS = 0;       //  Edge aligned counter.
76      //
```

```
82      //  Now configure Timer 1, channel 4.
83      //
84      TIM1_CCMR4_OC4M = 7;    //  Set up to use PWM mode 2.
85      TIM1_CCER2_CC4E = 1;    //  Output is enabled.
86      TIM1_CCER2_CC4P = 0;    //  Active is defined as high.
87      TIM1_CCR4H = 0x01;      //  480 = 50% duty cycle (based on TIM1_ARR).
88      TIM1_CCR4L = 0xe0;
89      TIM1_BKR_MOE = 1;       //  Enable the main output.
90      TIM1_IER_UIE = 1;       //  Turn interrupts on.
91  }
92
93  //---------------------------------------------------------------------
94  //
95  //  Setup Timer 2 to generate a 40 Hz interrupt based upon a 2 MHz timer.  T
96  //  will result in a signal with a frequency of 20Hz.
97  //
98  void SetupTimer2()
99  {
100     TIM2_PSCR = 0x00;       //  Prescaler = 1.
101     TIM2_ARRH = 0xc3;       //  High byte of 50,000.
102     TIM2_ARRL = 0x50;       //  Low byte of 50,000.
103     TIM2_IER_UIE = 1;       //  Enable the update interrupts.
104     TIM2_CR1_CEN = 1;       //  Finally enable the timer.
105 }
106
107 //---------------------------------------------------------------------
108 //
109 //  Now set up the output ports.
110 //
111 //  Setup the port used to signal to the outside world that a timer event ha
112 //  been generated.
113 //
114 void SetupOutputPorts()
115 {
116     PD_ODR = 0;             //  All pins are turned off.
117     //
118     //  PD5 is used to indicate the firing of the update/overflow event for
119     //
120     PD_DDR_DDR5 = 1;
121     PD_CR1_C15 = 1;
122     PD_CR2_C25 = 1;
123     //
124     //  PD4 is used to indicate the firing of the update/overflow event for
125     //
126     PD_DDR_DDR4 = 1;
127     PD_CR1_C14 = 1;
128     PD_CR2_C24 = 1;
129 }
130
131 //---------------------------------------------------------------------
132 //
133 //  Main program loop.
134 //
135 void main()
136 {
137     __disable_interrupt();
```

```
142         while (1)
143         {
144             __wait_for_interrupt();
145         }
146     }
```

If we now have a look at the output on the Logic Analyser we can see how the various signals indicate what is happening with the Timers.



Five Pulses

The top trace of the output shows us when Timer 2 has overflowed (i.e. every 25mS). The code above goes through a cycle of enabling Timer 1 and 25mS later disabling Timer 1. You can see the output from Timer 1, Channel 3 in the middle of the three traces. The final (bottom) trace shows when Timer 1 overflows.

We have used much of the code presented above in previous examples in the series. We are setting up the two timers (Timer 1 and Timer 2, Channel 3) and adding interrupt handlers for the counter overflows. We are also setting up Port D as an output port to give us some diagnostic traces.

One difference from previous examples is that we are running this code at a slower clock frequency. By not setting up the system clock we are running at the default clock rate (2 MHz clock from the internal clock source).

Timer 1 Overflow Handler

This handler is really simple and does nothing more than use PD4 to indicate it has been called and then turns the timer off.

Timer 2 Overflow Handler

This handler is a little more complex. The first things to note is that it uses PD5 and works out if we are starting timers or pausing (PD5 = 1).

The next thing to note is the use of two more registers *TIM1_CR1_URS* and *TIM1_EGR_UG*. These two registers are used together to force the counter register to be reloaded without generating an interrupt. This is necessary to ensure that we start from a known value.

Conclusion

We have seen how we can use the logic analyser and some output pins (PD4 and PD5) to give us information about the sequence of events (interrupts). This is a very useful diagnostic tool and often the only one which is available to you when operating in this sort of environment.

I would also suggest trying some of the following and viewing the output on a logic analyser:

- Changing the counting mode

Source Code Compatibility

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

Tags: Electronics, STM8

Friday, September 14th, 2012 at 1:36 pm • Electronics, 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