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

Single Conversion ADC on the STM8S

In this post we will have a look at the Analog to Digital Converter (ADC) on the STM8S microcontroller. The number of ADCs available will depend upon the STM8S you are using. We will be using ADC1 which should be present on all STM8S microcontrollers.

In order to show how the ADC works we will be using the STM8S as a dimmer switch for an LED. This simple example will demonstrate how we can read an analog value and use a PWM signal to control the brightness of an LED.

In order to do this we will need the following:

- Potentiometer to provide an analog signal varying from 3.3V to GND.
- LED circuit from the *External Interrupts in the STM8S* posting

We will also be using Timer 1, Channel 4 to generate a PWM signal to control the brightness of the LED (see *Generating PWM Signals using the STM8S*.

The algorithm we will be using is as follows:

1. Configure the system
2. Read the value from the ADC
3. Set the PWM output based upon the analog reading
4. Pause for 1/10th second
5. Repeat from step 2

We will achieve this by using interrupts from the following resources:

- Timer 3 – Generates the PWM signal which will be used to control the LED
- Timer 2 – 1/10th second interrupt which triggers the ADC process
- ADC – Conversion is completed, adjust the PWM output

ADC Features

The ADC has several modes of operations. We will be using the simplest, namely single conversion mode. As the name suggests, this mode performs a conversion on a specific channel. We will also instruct the microcontroller to generate an interrupt once the conversion is complete.

Amongst the other features and modes on the STM8S are the following:

- Single scan mode – Perform a single conversion on a number of channels.
- Continuous and Buffered Continuous – Perform continuous conversions. New conversions start as soon as the current conversion has completed.

- External Trigger – An external trigger is used to start a conversion.

The conversion takes 14uS after a stabilisation period. Once the stabilisation is complete, readings are available without any further pauses.

The Registers

So let's have a look at the registers we will be using in order to control the ADC:

- ADC_CR2_ALIGN – Data alignment
- ADC_CSR_CH – Channel selection
- ADC_DRH/L – Analog conversion result
- ADC_CR1_ADON – Turn ADC on / off, trigger conversion
- ADC_CR3_DBUF – Data buffer Availability
- ADC_CSR_EOCIE- Enable ADC interrupts
- ADC_CSR_EOC – End of Conversion

ADC_CR1_ADON – ADC On/Off

The ADON flag determines of the ADC is on or off. It also determines if a conversion has been triggered. Setting ADON to 0 turns the ADC off. Setting ADON to 1 the first time turns the ADC on. Setting this value a second (or subsequent time) starts a conversion.

ADC_CSR_CH – Channel Selection

The CH flag determines the channel which should be converted.

ADC_CR2_ALIGN – Data Alignment

The ALIGN flag determines the type of alignment in the result registers. We will be setting this to right align (set to 1) the data in the registers.

ADC_DRH/L – Conversion Result

This pair of registers holds the result of the conversion. The order the registers should be read is dependent upon the alignment of the data in the registers. For right aligned data we need to read the DRL before DRH.

ADC_CSR_EOCIE – Enable ADC Interrupts

Turn the ADC interrupts on / off.

ADC_CSR_EOC – End of Conversion

This bit is set by the hardware when the conversion has completed. It should be reset by the software in the Interrupt Service Routine (ISR).

Unused Registers

The application we are going to be writing is simple and only performs a conversion once every 1/10th second. This is plenty of time to perform a conversion and process the data before the next conversion starts. As such, we

This post requires some additional hardware to be added to the circuit containing the STM8S:

- LED which is being controller through a transistor configured as a switch
- Potentiometer to provide an analog signal for conversion

LED Output

Use the LED Output circuit in the post *External Interrupts on the STM8S*. Connect the base of the transistor to the output of Timer 1, Channel 3.

Potentiometer

Connect the potentiometer so that one pin is connected to ground and one to 3.3V. The output (the wiper) should be connected to AIN4 on the STM8S. I used a 10K potentiometer for this example.

Software

As we have already noted, we will be driving the application by using interrupts. We will also use a few techniques/methods from previous posts. So let's look at each of the elements we will be using.

Timer 2 – Start Conversions

Timer 2 is used to generate 10 interrupts per second. Each interrupt will trigger a new conversion. Setting up the timer should look familiar:

```
1  void SetupTimer2()
2  {
3      TIM2_PSCR = 0x05;       //  Prescaler = 32.
4      TIM2_ARRH = 0xc3;       //  High byte of 50,000.
5      TIM2_ARRL = 0x50;       //  Low byte of 50,000.
6      TIM2_IER_UIE = 1;       //  Enable the update interrupts.
7      TIM2_CR1_CEN = 1;       //  Finally enable the timer.
8  }
```

The ISR is a simple method, it has only one main function, namely to start the conversion.

```
1  #pragma vector = TIM2_OVR_UIF_vector
2  __interrupt void TIM2_UPD_OVF_IRQHandler(void)
3  {
4      PD_ODR_ODR5 = !PD_ODR_ODR5;        //  Indicate that the ADC has completed
5
6      ADC_CR1_ADON = 1;       //  Second write starts the conversion.
7
8      TIM2_SR1_UIF = 0;       //  Reset the interrupt otherwise it will fire aga
9  }
```

Note the comment on the ADC register *Second write starts the conversion*. This method assumes that we have set *ADC_CR1_ADON* at least once previously. As you will see later, we set this register in the setup method for the ADC.

The ADC generates a 10 bit value. We will therefore set up Timer 1 to generate a PWM signal which is 1024 (2^{10}) clock signals in width. We can therefore use the value from the conversion to directly drive the PWM duty cycle.

```
1  void SetupTimer1()
2  {
3      TIM1_ARRH = 0x03;       //  Reload counter = 1023 (10 bits)
4      TIM1_ARRL = 0xff;
5      TIM1_PSCRH = 0;         //  Prescalar = 0 (i.e. 1)
6      TIM1_PSCRL = 0;
7      TIM1_CR1_DIR = 1;       //  Down counter.
8      TIM1_CR1_CMS = 0;       //  Edge aligned counter.
9      TIM1_RCR = 0;           //  Repetition count.
10     TIM1_CCMR4_OC4M = 7;    //  Set up to use PWM mode 2.
11     TIM1_CCER2_CC4E = 1;    //  Output is enabled.
12     TIM1_CCER2_CC4P = 0;    //  Active is defined as high.
13     TIM1_CCR4H = 0x03;      //  Start with the PWM signal off.
14     TIM1_CCR4L = 0xff;
15     TIM1_BKR_MOE = 1;       //  Enable the main output.
16     TIM1_CR1_CEN = 1;
17 }
```

Note that the Auto-reload registers is set to 1023 (0x3ff). We also set the capture compare registers to 1023 at the start. This will turn the LED off when the program starts.

System Clock

The program will be generating a PWM signal with a reasonably high clock frequency. In order to do this we will set the clock to use the internal oscillator running at 16MHz.

```
1  void InitialiseSystemClock()
2  {
3      CLK_ICKR = 0;                       //  Reset the Internal Clock Register
4      CLK_ICKR_HSIEN = 1;                 //  Enable the HSI.
5      CLK_ECKR = 0;                       //  Disable the external clock.
6      while (CLK_ICKR_HSIRDY == 0);       //  Wait for the HSI to be ready for
7      CLK_CKDIVR = 0;                     //  Ensure the clocks are running at
8      CLK_PCKENR1 = 0xff;                 //  Enable all peripheral clocks.
9      CLK_PCKENR2 = 0xff;                 //  Ditto.
10     CLK_CCOR = 0;                       //  Turn off CCO.
11     CLK_HSITRIMR = 0;                   //  Turn off any HSIU trimming.
12     CLK_SWIMCCR = 0;                    //  Set SWIM to run at clock / 2.
13     CLK_SWR = 0xe1;                     //  Use HSI as the clock source.
14     CLK_SWCR = 0;                       //  Reset the clock switch control re
15     CLK_SWCR_SWEN = 1;                  //  Enable switching.
16     while (CLK_SWCR_SWBSY != 0);        //  Pause while the clock switch is b
17 }
```

GPIO – Debug Signals

As with previous examples, we will configure some of the output ports so that we can generate debug signals:

```
    //  PD5 indicates when the ADC is triggered.
    //
    PD_DDR_DDR5 = 1;
    PD_CR1_C15 = 1;
    PD_CR2_C25 = 1;
    //
    //  PD4 indicated when the ADC has completed.
    //
    PD_DDR_DDR4 = 1;
    PD_CR1_C14 = 1;
    PD_CR2_C24 = 1;
}
```

ADC

The setup method for the ADC is relatively simple as many of the settings we will be using are the defaults after a reset. This method is as follows:

```
void SetupADC()
{
    ADC_CR1_ADON = 1;       //  Turn ADC on, note a second set is required to

#if defined PROTOMODULE
    ADC_CSR_CH = 0x03;      //  Protomodule uses STM8S105 - no AIN4.
#else
    ADC_CSR_CH = 0x04;      //  ADC on AIN4 only.
#endif

    ADC_CR3_DBUF = 0;
    ADC_CR2_ALIGN = 1;      //  Data is right aligned.
    ADC_CSR_EOCIE = 1;      //  Enable the interrupt after conversion complet
}
```

After calling this method, the ADC should be powered on and ready to perform a conversion. Note that the ADC will not perform a conversion until *ADC_CR1_ADON* is set for a second time. This will be performed by the Timer 2 interrupt.

Another point to note is that on the Protomodule board the version of the STM8S does not have the AIN4 channel and so we use AIN3 instead.

The next method we will consider is the ADC ISR. This is where the real work of changing the values for the PWM signal takes place.

```
#pragma vector = ADC1_EOC_vector
__interrupt void ADC1_EOC_IRQHandler()
{
    unsigned char low, high;
    int reading;

    ADC_CR1_ADON = 0;       //  Disable the ADC.
    TIM1_CR1_CEN = 0;       //  Disable Timer 1.
    ADC_CSR_EOC = 0;        //     Indicate that ADC conversion is complete.

    low = ADC_DRL;            //    Extract the ADC reading.
```

```
    low = reading &amp; 0xff;
    high = (reading >> 8) &amp; 0xff;
    TIM1_CCR3H = high;      //  Reset the PWM counters.
    TIM1_CCR3L = low;
    TIM1_CR1_CEN = 1;       //  Restart Timer 1.

    PD_ODR_ODR4 = !PD_ODR_ODR4;     //  Indicate we have processed an ADC int
}
```

Note that we once again use a GPIO port to indicate when the ISR has been called.

Main Program Loop

The main program loop looks pretty much like the programs we have written in previous examples:

```
void main()
{
    //
    //  Initialise the system.
    //
    __disable_interrupt();
    InitialiseSystemClock();
    SetupTimer1();
    SetupTimer2();
    SetupOutputPorts();
    SetupADC();
    __enable_interrupt();
    while (1)
    {
        __wait_for_interrupt();
    }
}
```

Results

If we put all of this together we can do the following:

The LED indicates the duty cycle of the PWM signal. The output on the oscilloscope confirms the changes being made to the PWM signal (the wider the high component of the signal, the brighter the LED should be).

By adjusting the trimmer potentiometer to ground (turning to the right) the LED becomes dimmer as the duty cycle becomes biased towards ground (off more than on). Turning to the left does the reverse, the PWM signal becomes biased to +3.3V (more on than off).

Conclusion

This example may be trivial as we could have easily just connected the LED and the potentiometer together. However, it does show how we can take a reading from an ADC and change the output of the microcontroller based upon the value.

As always, the source code is available for download. This application is compatible with my reference platform, the Variable Labs Protomodule and the STM8S Discovery board.

Source Code Compatibility

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

Tags: Electronics, LED, Software Development, STM8, The Way of the Register

Monday, September 17th, 2012 at 4:15 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

Currently Reading