Project 5: PWM with ADC Re-STRUCT-ured

Jacob Hollenbeck, Boise State University

04/02/2016

Contents

1 Overview

This project is a code rewrite of Project 3: Pulse Width Modulation with an ADC source. The new code was designed to be more modular and to remove blocking statements. By implementing a second layer of code between the ATmegaADC and the ADC calculations, the ADC can be better controlled to use less power and remove timing errors.

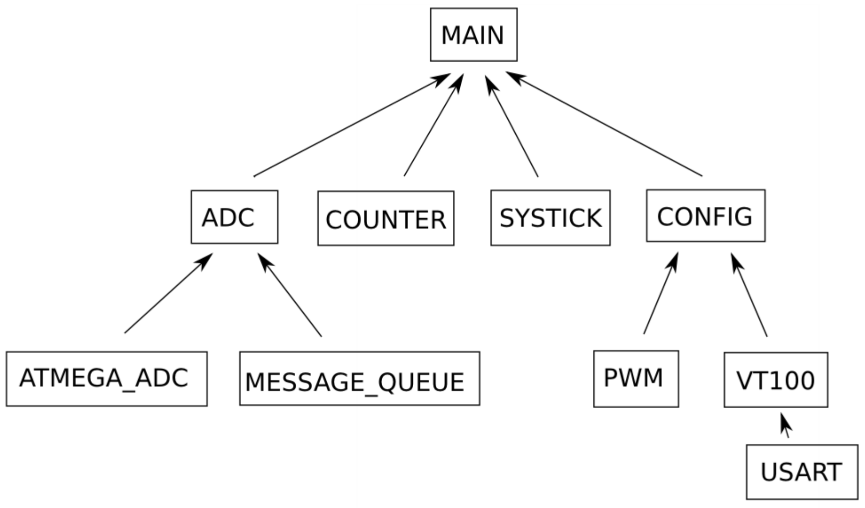

Figure 1: Abstract Overview

Figure 1 displays the ideal layout for the program. Note that there were a few minor tweaks made to the code layout in order to obtain better functionality. This will be covered in later sections.

2 Supplies

- Atmega324p Microprocessor PDIP
- Four-prong button
- 220 ohm Resistor (x4)
- Atmel-ICE programmer with ISP cable
- Breadboard
- FTDI Cable
- JTAG Header
- Tri-color LED
- Potentiometer (x3)

3 ADC

The ADC has been reconfigured into three primary files:

- Atmega_adc.*
- adc.*
- message_queue.*

Atmega_adc contains the adc functionality. This is composed of the functions required to initialize and read the adc. ADC.* contains finite state machine definitions and the state machine processes. Message queue contains functionality for a short queue to be used with the state machine.

3.1 Atmega ADC

Atmega_adc contains the accessor functions for the adc. These functions are necessary to have access to the peripheral. This file is more modularized than the previous one. The adc is initialized in single channel mode with a prescaler of 16 as follows:

```
    ADCSRA |= (1 << ADEN);                              // enable ADC
    ADMUX  |= (0 << REFS1)|(1 << REFS0);                // AVCC = AREF
    ADCSRA |= (0 << ADPS2)|(1 << ADPS1)|(0 << ADPS0);   // set prescaler to 16
```

Instead of reading the ADC using a large function with blocking statements, the function is broken up into four separate pieces. The function adc_atmega_set_channel takes in a channel to be set as an input and sets it as follows:

```
void adc_atmega_set_channel(uint8_t ch)
{
   ADMUX = (ADMUX & 0xF0)|ch;        // Only want MUX bits to be altered
}
```

Note how the ADMUX is & with 0xF0 before | with the input channel. This is done to keep the reference voltage set. The bits used to set the channel are the first 5 bits; however, for the purposes of this program, only the first 3 bits needed to be cleared. By setting it to 0xF0, the first 4 bits are set to 0. The next piece starts the adc conversion. This is done inside of the adc_atmega_start_conversion function:

```
void adc_atmega_start_conversion(void)
{
  ADCSRA |= (1 << ADSC);              // Set start bit
}
```

The start conversion function starts a conversion by setting the ADSC bit high. Once this conversion has been completed, the bit will change to 0. In project 3, this was checked continuously using a while loop as follows:

```
// ...
 while(ADCSRA & (1 << ADIF));        // Wait for flag bit to clear
// ...
```

This creates a blocking statement and will cause timing issues. To prevent this, another function was created called adc_atmeage_converion_complete. This function uses an if loop and returns a 1 if the bit has been changed to 0. This is done as follows:

```
unsigned char adc_atmega_conversion_complete(void)
{
  if(ADCSRA & (1<<ADSC))
     return 0;
  else
```

```c
    return 1;
}
```

Finally, a get data function was implemented. This was implemented as follows:

```c
uint16_t adc_atmega_conversion_data(void)
{
  return ADC;
}
```

This function returns the current ADC value in a 16 bit variable.

3.2 Message Queue

Message queue acts as a buffer for the oncoming channel requests. Each time the ADC is updated, a new channel is queued. The buffer is defined in the header file as 8 bits. The message queue uses two primary methods. The first of which puts the ADC packet onto the back of the queue. This is done as follows:

```c
unsigned char msg_q_put(void * ptr)
{
    // define a new spot for input
  uint8_t new_tail = (tail+1) % Q_SIZE;
  if(new_tail != head) {
    queue[tail] = ptr;  // place input at end of queue
    tail = new_tail;    // Update tail
    return 1;
  } else {
    // If overlap, need a bigger queue
  }
}
```

The second methods takes in a double pointer as input. This pointer is redirected to the newest ADC request. This is done as follows:

```c
unsigned char msg_q_get(void ** ptr)
{
  // Check if queue is empty
  if(msg_q_empty())
    *ptr = NULL;
  else{
    // re-point pointer to head of queue
    *ptr = queue[head];

    // update pointer to new head
    head = (head+1)%Q_SIZE;
  }
  return 1;
}
```

3.3 ADC State Machine

The layer between the ADC functionality and Main is composed of a state machine. The state machine gets the current channel from the queue and checks it against the current process. If it is the same channel then the channel switch is complete and the data can try to be read. If it is a new channel, the current channel is updated and a conversion is started. This process can be seen in Figure 2.

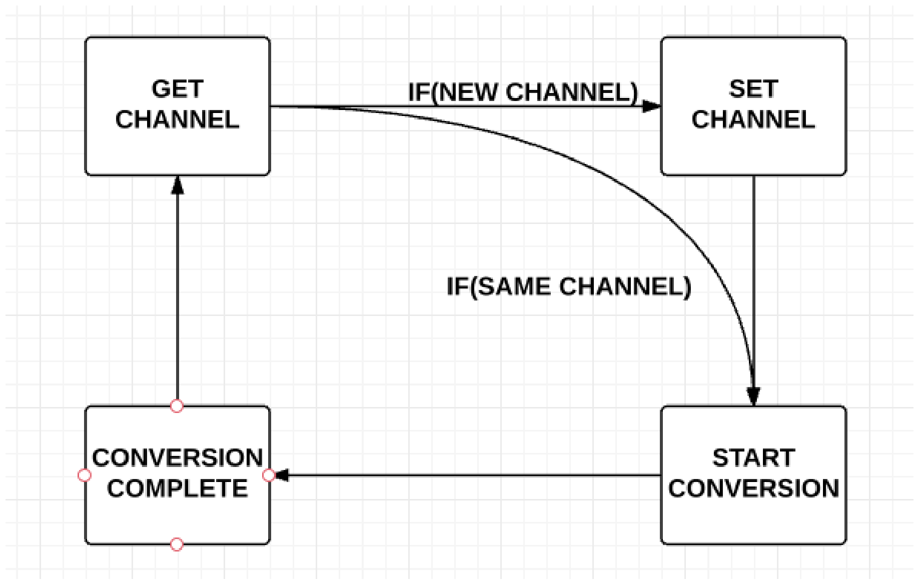

Figure 2: Finite State Machine Overview

The code to do this is a large switch statement. First three states were created by defining a new data type in the header file:

```
typedef enum {
  ADC_STATE_IDLE,
  ADC_STATE_WAIT_CHANNEL,
  ADC_STATE_WAIT_COMPLETE
} adc_state_t;
```

These states were then updated by using the adc_process function. Inside of the adc_process function, three static variable are initialized as follows:

```
  static adc_request_t * current_request   = NULL;
  static adc_state_t current_state     = ADC_STATE_IDLE;
  static uint8_t current_channel       = ~0;
```

By initializing the variables as static, their place on the stack remains constant inside of the method. This will update the defined variables only as requests are gathered. The current state is used as a switch statement to check the current as described above. The full code for the process statement can be seen in the reference section. The remaining functions inside of the adc.c are helper methods. There is an intializer method which intials a data packet for a particular channel, a start method that queues the request, a request complete; which checks if the request has been completed, a data request method which grabs the data, and an id request which grabs the current ID.

4 Main

Finally, Main will be covered. Main pulls all of the functionality together. Main intializes the structures for the three adc requests and three timers. Once a timer has elapsed, a new adc value is requested. If a request has been completed, the new value is printed to the screen. An example of the code is shown below:

```
{
//...

    // Create structs
    adc_request_t ADC0;
  counter_t pc0;

    // Initialize ADC and Systick
  adc_init();
  systick_init();

    // Initialize structs
  adc_request_init(&ADC1, 'A', 0);
  counter_init(&pc0, 0, 50);

    // Sync system clock
    systick_sync();

  while(1){
    // Synchronous edge
    if(systick_edge()){

        // Read in ADC request
      if(counter_elapsed(&pc0))
        adc_request_start(&ADC0);
    } // Asynchronous edge

        // If done reading, do something with data
    if(adc_request_complete(&ADC0))
            adc_consume(&ADC0);

    // State machine
    adc_process();
    }
}
```

Note that the above is only a sample of main. The full code can be seen in the reference section.

4.1 tweaks

The FTDI cable used was damaged resulting in a loss of data if too much was processed at the same time. Because of this, the method ADC consume was not used. Instead, the actual code that would be inside of the ADC consume method was placed inside of the loop as follows:

```
{
  if(adc_request_complete(&ADC0)){
    data0 = adc_request_data(&ADC0);
    set_OCR0A(data0/4);
    OCR0A = read_OCR0A();
    VT100_prints(2,11, "A");
    VT100_printf(2,29, "0x%03X", data0);
    VT100_printf(2,47, "%03d", OCR0A);
    }
}
```

The actual method for adc consume would look like the following:

```
adc_consume(adc_request * p) {
    uint8_t duty;

    // Grab data from pointer
    uint16_t data = adc_request_data(&p);

    switch(p->id){
        case 'A':
            // Set PWM
          set_OCR0A(data/4);
          // Grab duty cycle
          duty = read_OCR0A();

          // Print to VT100
          VT100_printf(2,11, "%c", p->id);
          VT100_printf(2,29, "0x%03X", data);
          VT100_printf(2,47, "%03d", duty);
        break;
        case 'B':
          set_OCR0B(data/4);
          duty = read_OCR0B();

          VT100_printf(4,11, "%c", p->id);
          VT100_printf(4,29, "0x%03X", data);
          VT100_printf(4,47, "%03d", data/4);
        break;
        case 'C'
          set_OCR1A(data/4);
          duty = read_OCR1A();

          VT100_printf(6,11, "%c", p->id);
          VT100_printf(6,29, "0x%03X", data);
          VT100_printf(6,47, "%03d", duty);
        break;
    }

}
```

5 Compilation

Because multiple files are used in this program, a Makefile is used to compile the .elf file. The linker process was covered in the previous report. To use the Makefile included, change to the project directory and use the following commands:

```
$ make
$ make write
```

The make write command will write the compiled .elf file to the MCU.

6 reference

ADC.C

```
void adc_process(void)
{
  // Need to initialize as static. Ensures variables will only
  // be updated inside of this stack. Also makes current_state
  // changeable from ADC_STATE_IDLE.
  static adc_request_t * current_request  = NULL;
  static adc_state_t current_state     = ADC_STATE_IDLE;
  static uint8_t current_channel       = ~0;

  switch(current_state)
  {
    case ADC_STATE_IDLE:
      if(msg_q_get((void **) &current_request)){
        current_request->state = ADC_REQUEST_STATE_ACTIVE;
        if(current_request->channel != current_channel)
        {
          current_channel = current_request->channel;
          adc_atmega_set_channel(current_request->channel);
          current_state = ADC_STATE_WAIT_CHANNEL;
        } else {
          adc_atmega_start_conversion();
        }
      }
      break;

    case ADC_STATE_WAIT_CHANNEL:
      if(adc_atmega_channel_ready(current_request->channel)){
        adc_atmega_start_conversion();
        current_state = ADC_STATE_WAIT_COMPLETE;
      }
      break;

    case ADC_STATE_WAIT_COMPLETE:
      if(adc_atmega_conversion_complete()){
        current_request->data = adc_atmega_conversion_data();
        current_request->state = ADC_REQUEST_STATE_COMPLETE;
```

```c
        current_request = NULL;
        current_state = ADC_STATE_IDLE;
      }
      break;

    default:
      current_request = NULL;
      current_state = ADC_STATE_IDLE;
      break;
  }
}
```

MAIN.C

```c
#include "config.h"
#include "systick.h"
#include "vt100.h"
#include "adc.h"
#include "counter.h"



void main(void)
{
  // Create structs
  adc_request_t ADC1, ADC2, ADC0;
  counter_t pc0, pc1, pc2, dc;

  // Initialize periphials
  adc_init();
  systick_init();
  usart_init();
  init_INT2(0);
  pwm0_init();
  pwm1_init();

    // Initialize structs
  adc_request_init(&ADC1, 'A', 0);
  adc_request_init(&ADC1, 'B', 1);
  adc_request_init(&ADC2, 'C', 2);

  counter_init(&pc0, 0, 50);
  counter_init(&pc1, 0, 50);
  counter_init(&pc2, 0, 50);

    // Sync system clock
    systick_sync();

    // Create headers for VT1000
    headers();

    // Initialize variables
  uint16_t data0, data1, data2, OCR0A, OCR0B, OCR1A;

  data0 = data1 = data2 = 0;
  OCR0A = OCR0B = OCR1A = 0;

  while(1){
    // Synchronous to clock
    if(systick_edge()){
      if(counter_elapsed(&pc0))
      {
        // Request a new thread
        adc_request_start(&ADC0);
      }
      if(counter_elapsed(&pc1))
      {
        adc_request_start(&ADC1);
      }
      if(counter_elapsed(&pc2))
      {
        adc_request_start(&ADC2);
      }
    }
    // Asynchronous
      if(adc_request_complete(&ADC0)){
        // Once a thread has finished, do something with data
        data0 = adc_request_data(&ADC0);
        set_OCR0A(data0/4);
        OCR0A = read_OCR0A();

        VT100_prints(2,11, "A");
        VT100_printf(2,29, "0x%03X", data0);
        VT100_printf(2,47, "%03d", OCR0A);

        //consume_adc_data('A', adc_request_data(&ADC0));
      }

      if(adc_request_complete(&ADC1)){
        data1 = adc_request_data(&ADC1);
        set_OCR0B(data1/4);
        OCR0B = read_OCR0B();

        VT100_prints(4,11, "B");
        VT100_printf(4,29, "0x%03X", data1);
        VT100_printf(4,47, "%03d", data1/4);
      }

      if(adc_request_complete(&ADC2)){
        data2 = adc_request_data(&ADC2);
        set_OCR1A(data2/4);
        OCR1A = read_OCR1A();
        VT100_prints(6,11, "C");
        VT100_printf(6,29, "0x%03X", data2);
        VT100_printf(6,47, "%03d", OCR1A);
      }

    adc_process();

  }
}
```