Matt Zimmerer
Daniel Jennings
Lab1
RTOS Spring 2013

```
/***********************************************************
 * Filename: main.c
 * Blinks LED0 and LED2 if SW0 is pressed
 * Blinks LED4 and LED6 if SW0 is not pressed
 * Authors: Daniel Jennings, Matt Zimmerer
 * Revision log:    created 4/2/13
 **********************************************************/

#define __DELAY_BACKWARD_COMPATIBLE__
#define F_CPU 8000000UL

// Calculating delay count (HARDWARE SPECIFIC):
#define DESIRED_FREQ_HZ 2
//    Given a desired frequency, a naive count can be taken as follows:
#define NAIVE_COUNT F_CPU / DESIRED_FREQ_HZ
//    For our application, we want to double the frequency of our dumb delay.
//    This can be understood by observing that the delay is used to toggle an
//    LED, and two toggles will complete on full period.
#define NAIVE_HALF_COUNT NAIVE_COUNT / 2
//    A for loop involving one 32bit value, requires 23 atomic operations
//    per loop (observed from '-O3' optimised assembly). Thus, an acurate count
//    can be calculated as follows:
#define ACCU_COUNT NAIVE_HALF_COUNT / 23

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include "init.h"

// A state variable that reflects the state of button0
static volatile uint8_t state = 0;

int main(void)
{
   // A count variable. It must be declared volatile so that it survives any
   // optimisation used.
   static volatile uint32_t i;

   DDRA = 0x00; //Data direction register port A set to input
   DDRB = 0xFF; //Data direction register port B set to output

   initISR(); // Initialize registers for both timer interrupts
   PORTB = 0xFF; // Turn off all LEDs (active low)
```

```
  while(1){
    if(PINA & (1 << PA0))
    {
      // If PORTA pin 0 is pressed
      state = 1; // Notify ISR handlers that button is pressed
      for (i = 0; i < ACCU_COUNT/4; i++); // Delay of 500ms
      PORTB ^= 1<<PB4; // Toggle LED4
      PORTB |=  1<<PB0; // Turn off LED0
    }
    else
    {
      // If PORTA pin 0 is not pressed
      state = 0; // Notify the ISR handlers that button is not pressed
      for (i = 0; i < ACCU_COUNT/2; i++); // Delay of 500ms
      PORTB ^= 1<<PB0; // Toggle LED0
      PORTB |= 1<<PB4; // Turn off LED4
    }
  }

  return 0;
}

ISR(TIMER0_COMPA_vect)
{
  if (state == 0){ // If button is not pressed
    PORTB ^= 1<<PB2; // Toggle LED2
    PORTB |= 1<<PB6; // Turn of LED6
  }
}

ISR(TIMER1_COMPA_vect)
{
  if (state == 1){ // If button is pressed
    PORTB ^= 1<<PB6; // Toggle LED6
    PORTB |= 1<<PB2; // Turn off LED2
  }
}
```

```
/***********************************************************
 * Filename: init.h
 * Initializes the ISR for timers used in blinky LEDs
 * Authors: Daniel Jennings, Matt Zimmerer
 * Revision log:    created 4/2/13
 **********************************************************/

#ifndef _INIT_H
#define _INIT_H

// Initialize interrupts for ISR(TIMER0_COMPA_vect) and ISR(TIMER1_COMPA_vect)
// Both timers use a prescaler of 256, and a count of 2. The final frequency
// is then 15625Hz.
void initISR();

#endif
```

```c
/*************************************************************
 * Filename: init.c
 * Initializes the ISR for timers used in blinky LEDs
 * Authors: Daniel Jennings, Matt Zimmerer
 * Revision log:    created 4/2/13
 *************************************************************/

#include<avr/io.h>
#include "init.h"
#include <avr/interrupt.h>

void initISR()
{
   // Timer 0
   TCCR0A = 0xC2; //set timer0 to CTC mode, sets OC0A on compare match
   TCCR0B = 0x04; //set prescaler to divide by 256 (frequency at 31250Hz)
   OCR0A  = 0x01; //output compare match register set frequency to 15625Hz
   TIMSK0 = 0x02; //enables interrupt on compare match A

   // Timer 1
   TCCR1A = 0xC0; //set timer0 to CTC mode, sets OC1A on compare match
   TCCR1B = 0x0C; //set prescaler to divide by 256 (frequency at 31250Hz)
   OCR1AH = 0x00; //set the top 8 bits of output compare reg to 0
   OCR1AL = 0x01; //set the bottom 8 bits of output compare reg to 4
   TIMSK1 = 0x02; //enables interrupt on compare match A

   sei(); //enable interrupts
}
```

```
##################################################################################
# Filename: Makefile
# Authors: Daniel Jennings, Matt Zimmerer
# Revision log:    created 4/2/13
##################################################################################
all:
	avr-gcc -mmcu=atmega2560 main.c -c -o main.o -O3
	avr-gcc -mmcu=atmega2560 init.c -c -o init.o -O3
	avr-gcc -mmcu=atmega2560 main.o init.o -o main.elf -O3
	avr-objcopy -j .text -j .data -O ihex main.elf main.hex
	sudo avrdude -c stk600 -p atmega2560 -P usb -v -v -U flash:w:main.hex

clean:
	rm -f *.o *~ main.elf main.hex
```

Questions (note to professor: we messed up and did this independently {not the code}, just this time!)

1. Unless you used an interrupt for SW0, there are some “issues” regarding when exactly your program starts following the desired procedure. In the context of your program, describe these issues.

Since the state of SW0 is checked inline with the specified 'dumb delay loops', any delays that are being executed while a button is pressed much first finish (and a few support instructions), before the state of SW0 is actually read in.

2. What was the “Atmel AVR design methodology” spoken of in this experiment’s objectives? Briefly but completely describe.

1. Write code for the desired target.
2. Compile the code using the appropriate tool-chain.
   a) Compile each object for the correct processor.
   b) Link the object files into an ELF format
   c) Translate the ELF file into a HEX file suitable for flashing
3. Flash the HEX file to target using appropriate programmer (stk600), part (atmega2560), and port (USB), using a tool such as avrdude.

3. What are the primary differences between the dumb delay loops and the avr-libc delay functions? Is one better than the other?

The avr-libc delay function takes a double to a number of ms to delay, uses F_CPU to calculate the appropriate number of ticks, and executes these ticks using macros to select snippets of code for certain flags. A dumb delay simply increments a variable a certain amount that is determined either by practice, or intelligent calculation. The avr-libc version is better in my opinion, because it is well tested. However, more precise timing may be achievable in practice by using experimentally derived dumb delays.

Conclusion (Matt Zimmerer)

In the context of this course, this project probably serves to warm us up to programming this target, and to act baseline of comparison against real time operating systems. For instance, the use of dumb delays in this project is terribly wasteful. In an RTOS, these delays wont just increment a variable, but instead allow the scheduler to choose other tasks while waiting. But as seen in this project, the delays completely block the execution of time critical events (the pressing of a button).