MILESTONE 2: BASIC C & MCU OUTPUT

LAB OBJECTIVES

1. Gain a familiarity with the basic functions of the onboard MCU system timers.
2. Gain a familiarity with basic output.
3. Create physical interconnects between the MCU and the world.
4. Apply fundamentals of C programming for MCU systems.

DELIVERABLES

1. **Knight Rider Display**
   a. Set up and implement a **MCU timer** to create a LED display for a 'Knight Rider' (https://www.youtube.com/watch?v=Q2yh1j1VQzQ). https://www.youtube.com/watch?v=ydUCLzoNgFU
   b. Provide a demonstration of your 'Knight Rider display' with at **LEAST TWO** LEDs continuously appearing to **shift** from bit 0, bit 1, ..., to bit 7, and then from bit 7 to bit 0, bit 1, bit2.... The time between steps is no less than **200 ms**. A 'shift' could be considered as an LED light appearing to **shift** position from one bit to the next. You **MUST** use bit shifting to achieve this effect. The demo will occur at the beginning of the following lab.
   c. Provide your nicely formatted code, with your student numbers and names to the TAs by the beginning of the **FOLLOWING** lab.
   d. Submit the correct, and completed Milestone 2 assignment to your TAs, during your first lab session prior to hooking up your MCU.

INSTRUCTIONS

1. PREPARATIONS

In this lab, you will complete two exercises. Both of which will assist you in the steps toward completion of your final project (the inspection system). The first exercise is to initiate one of the onboard system timers. These timers will provide all the timing functionality and be critical in implementing many MCU systems. **Ensure that you understand how these timers work for both practical reasons**. It is not imperative that you memorize the details, but **ensure that you understand how to configure the system timers** within the

requirements of these labs. The second exercise is to provide an interface between the MCU and the real world and then to light up some LEDs in a 'Knight Rider' like fashion.

For the purposes of this lab it may be worthwhile to use the divide-and-conquer approach. **IMPORTANT**: Firstly, you should meticulously hook up your board (for prototyping purposes) in a way that minimizes the chances of accidently creating a discontinuity between the MCU and the outside world. This includes NOT using electrical tape to couple wires together or leaving the wires exposed such that they are easily shorted with another connector. Do the job right the first time; otherwise you'll be guaranteed to be troubleshooting later. If the connection is not correct, when problems occur, you will have to check the hardware and software at the same time.

A wire connection example will be provided in class and/or as an image on the website.

This software for this exercise and many other portions of this course can be developed completely in the AVR Studio 4. This includes writing and debugging. For many applications, you can even see the output on 'virtual' ports while using the debugging option. Unfortunately, we cannot show you all the functionality of this tool so you will be required to figure much of it out on your own. Therefore, it is very simple to use the divide-and-conquer approach for the software and the hardware in this course; however, ensure that both members in your group understand both elements of the course (i.e. both hardware and software).

2. DIGITAL OUTPUT

The ports on the ATMEL AT90USB1287 are bidirectional, meaning they can serve as both input and output. You must set the pins to input or output by setting the bits in the data direction register (DDRX) as you did in the first milestone. The 'X' in DDRX represents the PORT ID (e.g. A, B, C, D, E, or F). To set the port pin to input, assign the bit to 0. To set the pin to output, assign the pin to 1. Ensure that you don't overwrite previously assigned bits in a previous part of your code. You can do this by using the OR or AND functions in C to set the bits or clear them respectively. For example,

```
/* Set the 4 MSB to output, the 4LSB to input */
DDRD = 0b11110000;      /* 0xf0 would be fine too. */

/* Set the 4 MSB to input, the 4LSB to output */
DDRD = 0b00001111;      /* 0x0f would be fine too. */

/* Note, this is just an example of how to see the
direction of the I/O bits.*/

/* You should AVOID changing the ports from input to
output, given that they likely are either input or output
ONLY in your application. */
```

Ensure that you understand the distribution of the port pins from the socket (little black 10 pin socket) on your boards (hint: look at page AT90USBKey Hardware User Guide Pg. 6-19 provided online). This schematic will come in handy, so ensure you have it nearby. Notice that there is small white lettering near the port sockets to help you determine where pin 1 is on the port.

3.THE SYSTEM TIMER

MCUs use system timers for many applications, which do exactly as their name implies: provide timing provisions. Having a means of turning on an alarm after a period of time, activating a warning if a response is not received from hardware or logging the elapsed time for data entries are all examples of why system timers are required on MCUs. System timers can work on the same clock frequency, f_c, that is obtained by a crystal oscillator. Timers can also be set to use a scaled portion of f_c. These settings are easily set by changing values in registers prior to using a system timer, and can be changed throughout the application if necessary with some restrictions outlined in the documentation.

The AT90USB1287 MCU uses an 8 MHz crystal oscillator, which can be scaled down by 8, 64, 256 and 1024 of f_c. This MCU provides several timers all with different capabilities. Some of the timers are 8 bit timers and some of the timers are 16 bit timers. An 8-bit timer means that the timer can only count from 0x00 (BOTTOM) to 0xFF (MAX) or a portion thereof. A 16-bit timer can count from 0x0000 (BOTTOM) to 0xFFFF (MAX).

Since the timers on any MCU are limited by a specified word length, meaning we can only count up to a certain number, we need to know either (i) when we have exceeded the maximum count (in the case of a 16-bit timer = 0xffff) or (ii) when we have exceeded a preset value that we would like the counter to count up or down to (i.e. count up to 0x03E8 in the code below). In this lab, you will learn how to set the value of an **Output Compare Register** (OCR1A) so that the timer will reset when it equals the value (0x03E8) and set a flag indicating it reached this TOP value. This method will also be very useful when we need to perform pulse width modulation (PWM). You will learn more about this method in class, but the method is quite straightforward if you set all the registers properly (must read-the-fine-manual).

To perform the function of counting from BOTTOM to TOP (not MAX) we first load the **Output Compare Register** (OCR1A) with the 16-bit value that we want the timer to reset. In the example below that value is (0x03E8). Therefore, we want the timer to count from 0x0000 to 0x03E8 and reset to 0x0000 and begin the count again. However, we need to know when the timer has a reset so we can use a multiple of the time it takes to count for this loop. When the timer reaches the TOP value it sets a FLAG in the Timer Interrupt Flag Register 1 (TIFR1) to high. It indicates that the timer has reached the preset value. Depending on how the registers are set, it may be the programmer's responsibility to set that value back to zero.

In this lab exercise you will learn how to implement a 16 bit 'millisecond' time in the code below, and then use it in the 'Knight Rider' display application. Develop your code with the intent of 'code reuse'. That means you could develop a library of functions so-to-speak with the intent of future use in your major project. The following is an example of the use of the Timer1 on the MCU, but will NOT work in the necessary method required for PWM used later on. **REMEMBER THIS BEFORE CODING!!!**

A program example for using the system Timer1 to light up the LEDs is displayed below. Read those codes and try to understand them. Those codes could be used to as a part of your experiment.

```
#include<avr/io.h>       /* Need for performing I/O */
#include<avr/interrupt.h>   /* Needed for interrupt functionality*/


void mTimer(int count);


int main(int argc, char *argv[]){

/* Timer instructions*/
/* Sets timer 1 to run at CPU clock, disable all function and use as pure timer */

TCCR1B |=_BV(CS10);                  /* _BV sets the bit to logic 1    */
                                     /* Note the register is TCCR1B0 */
                                     /* TCCR1 is the Timer/counter control register 1 */
                                     /* B is the 'B' register and 0is the 0th bit         */
                                     /* CS means clock select, is the pre-scaler */


/*Set all pins on PORTD to output */
DDRD = 0xff;

while(1){

        PORTD = 0b01000000;          /* D5 GREEN*/
        mTimer(1000);

        PORTD = 0b00010000;          /* D2 RED*/
        mTimer(1000);

   } /* while*/
}

/* This is the driver for the timer */
void mTimer(int count){
   /* The system clock is 8 MHz. You can actually see the crystal oscillator
   which is a silver looking cap on the board. The non-prescaled timer actually will work at
   1 MHz and you can pre-scaler that value by 8, 64, 256 ect.
   This means the period = 1 us. See manual 7593-AVR-11/08(i.e. full manual)
   and look up "16-bit Timer/Counter (Timer/Counter1 and Timer/Counter3)" */

   /*Variable declarations*/

   int i;            /* used to keep track of the loop # */
```

```
/*Initialize the loop counter*/

i = 0;

/* Set the Waveform Generation mode bit description to Clear Timer
on Compare Math mode(CTC) only*/

TCCR1B |=_BV(WGM12);          /* This will set the WGM bits to 0100, see page 142 */
                              /* Note WGM is spread over two register.            */



/* Set Output Compare Register for 1000 cycles = 1ms */
OCR1A = 0x03e8;

/* Set the initial value of the Timer Counter to 0x0000*/
TCNT1 = 0x0000;

/* Enable the output compare interrupt enable */
TIMSK1 = TIMSK1 |0b00000010;

/* Clear the timer interrupt flag and begin timer */
/* If the following statement is confusing please ask for clarification!*/
TIFR1 |= _BV(OCF1A);

/* Poll the timer to determine when the timer has reached 0x03e8 */
while (i<count){
        if ((TIFR1 & 0x02) == 0x02){
           /*clear the interrupt flag by WRITING a ONE to the bit*/

            TIFR1 |=_BV(OCF1A);

            i++;     /* increment the loop counter*/

            /* NOTE: The Timer resets on its own from our WGM setting above*/
           }/* end if*/
    }   /* while end*/

  return;
} /*mTimer*/
```

Verify the timer, by using a stopwatch, and set the timer to 1000ms. Did you figure out how much time does it take to count for 0x03E8 from 0x0000?

Once you have confirmed the operation of the timer, build your 8 LED bank and wire one of the PORTS to the bank. See the Milestone2 Assignment to assist you in determining the wiring and a suggested component. After wiring a port to the LED bank, test the LEDs prior to coding the 'Knight Rider' display. Code the 'Knight Rider' display, and use the code above to provide timing services for your applications.

MILESTONE 2: ASSIGNMENT

1. INSTRUCTIONS

Complete the following questions before the beginning of the lab session and show the values to your TAs. You might find a useful link to the online manual "AVR 8-bit Microcontroller with 64/128K Bytes of ISP Flash and USB Controller" to find values of the maximum current output for the ports (pg.398).

2. QUESTIONS

Study the simple circuit diagrams for each diode required and provide the missing information. Note that the output voltage for this microcontroller is the same as VCC (the supply voltage) and is 3.3 V (a low voltage device). This means that the output voltage for the I/O pin is also 3.3 V (not 5 V).

Look at the product specifications of the LEDs provided at the end of this write up to confirm what the voltage ($V_{f(ON)}$) is required for the LEDs in your kit and the maximum current rating ($I_{f(MAX)}$) for the LED. Using Ohms law to determine the ideal resistor that could be used to satisfy the recommended current through the diode, and then find a practical value for this resistor (R_{actual} = a typical value that you could purchase). You also need to evaluate the **minimum** resistance of the resistors ($R_{\min}$), which means that all the devices in the circuit will not be damaged, i.e., the current will not exceed its limit in all devices when using the resistor.

NOTE: You have been provided 100 Ω resistors, which may not be the ideal solution. A document providing typical resistor values is located at (http://www.analog.com/library/analogDialogue/archives/31-1/Ask_Engineer.html).

MCU

V_{OUT} = 3.3V

$I_{OUT(MAX)}$ = _____

R_{min} = _____

Resistors

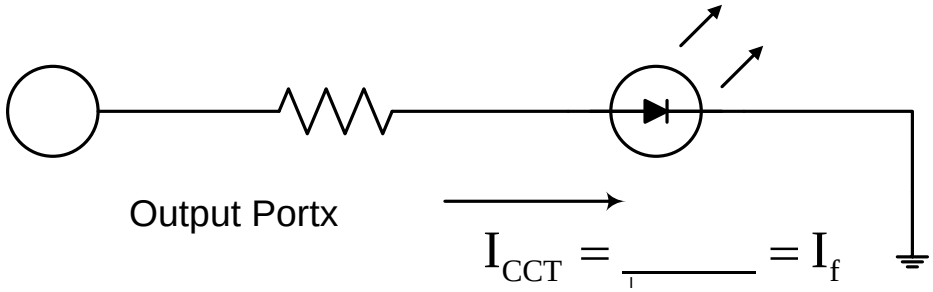
R_{ideal} = _____

R_{actual} = _____

LED

$I_{f(MAX)}$ = _____

$V_{f(ON)}$ = _____



PRODUCT IDS (CONFIRM WITH YOUR TA)

LED	Product ID	I_f (Recommended)	$I_{f(MAX)}$ (Max current before damage)	V_f (required to turn the LED on)
Red and Green	55-552-0	20 mA	$I_{f(MAX)}$ = 100 mA	V_f = 2.0 V (typical)

Lab Session:

Name 1:_____ Student ID: _____

Name 2:_____ Student ID: _____