Project 2: Synchronous Clock Signals

Jacob Hollenbeck, Boise State University 02/27/2016

Contents

1 Overview

This document discusses the implementation of synchronous clock signals generated by an MCU and displayed via UART. The signals are displayed as four different clock signals. The signals increment at intervals of 100, 500, 1000, and 5000 milliseconds. These signals are displayed on a terminal emulator, which reads UART data from the com port of the user CPU. The following image displays the graphical overview:

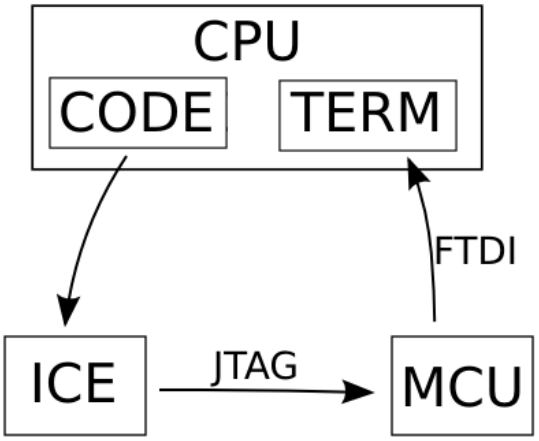

Figure 1: Code overview

The code is compiled on the CPU and loaded onto the MCU using the atmel-ICE programmer. The timers are then displayed on the terminal emulater using UART protocol over an FTDI cable.

2 Supplies

- Atmega324p Microprocessor PDIP
- Four-prong button
- 220 ohm Resistor
- Atmel-ICE programmer with ISP cable
- Breadboard
- FTDI Cable
- JTAG Header

3 Timer

The synchronous signals are generated by using the timer feature of the MCU. This can be configured in several different modes. For this specific project, the timer will be set to normal mode. In normal mode, the timer counts to a given value and then does something at that value. The timer is used to set off an interrupt when the the counter is equal to the compare value. Inside of this interrupt, a global variable called _systick is incremented. This signal is used to generate counters synchronous to it. The following code demonstrates how the timer was initialized:

```
1    TCCR1B |= (1 << CS10);              // Initialize timer with no prescale
2    TIMSK1 |= (1 << OCIE1A);            // Enable compare interrupts
```

```
    TCNT1 = 0;                          // Initialize counter register
    OCR1A = 7999;                       // Set compare value
    _systick = 0;                       // Begin count variable
```

The compare value, OCR1A is determined by using the following formula:

$$NumberOfTicks = \frac{Delay}{ClockPeriod} - 1 \tag{1}$$

whereas the delay is the required time interval and the clock period is the inverse of the period.

3.1 Synchronous Signals

Now that the hardware timer is working, a software timer is necessary. This software timer is synced with the hardware timer with the following code:

```
    _mytick = _systick;
    while(1){
        if(_mytick != _systick){
            _mytick++;                  // _mytick loop
            // insert synchronous code here
        }
        // insert asynchronous code here
    }
```

There will be some minuscule delay introduced to the code when this is run. That delay is accounted for in the _mytick loop. Any code placed inside of the _mytick loop will be synchronous to the _systick counter, which counts at the designated frequency of 1 millisecond. The counters are introduced in the synchronous part of the loop.

Because _systick is an 8-bit variable, it will only count up to 255. The required counts are 100, 500, 1000, and 5000. To get around this problem, only one timer is incremented inside of the _mytick loop. The other timers are incremented inside of new timer's loop.

```
        if(_mytick != _systick){
            _mytick++;
            timer1++;

            // timer 1 loop
            if(timer1 >= 100){      // count to 100 milliseconds
                timer2++;           // start timer 2
                timer3++;           // start timer 3
                timer4++;           // start timer 4

                counter1++;         // count 1 millisecond
                timer1 = 0;         // reset 1 millisecond timer
            }

            // timer 2 loop
            if(timer2 >= 5){        // count to 500 milliseconds
                counter2++;         // increment counter
                timer2 = 0;         // reset 5 millisecond timer
            }
```

```
20
21            // timer 3 loop
22            if(timer3 >= 10){        // count to 1 second
23                counter3++;          // increment 1 second counter
24                timer3 = 0;          // reset 1 second timer
25            }
26
27            // timer 4 loop
28            if(timer4 >= 50){        // count to 5 seconds
29                counter4++;          // increment 5 second counter
30                timer4 = 0;          // reset 5 second timer
31            }
32        }
```

Now that the signals are being generated, it is time to see them on the terminal.

4 UART

In order to send data from the MCU to the computer, a method of communication is needed. The atmega324p features a communication protocol known as UART or Universal Asynchronous Receiver Transmitter. To transmit data, the UART loads data into a register and shifts that data out through the designated pin. To receive data, the UART checks for a particular flag to be raised. If that flag has been raised, the UART begins reading data from the data buffer that had been sent to the data buffer. Because the UART for this project is only required to send data, the Rx function will not be covered.

4.1 Initialization

Because the UART operates asynchronously to a clock signal, the two devices communicating must agree on a particular set of parameters. This set of parameters is known as the frame format. The frame format consists of an agreed upon speed (baud rate), the number of data bits, the parity of the data, and the number of stop bits. This parameters must be the same for the two devices communicating. The following code demonstrates the intialization of the UART:

```
1    baud = 0x33;                          // set baud to 9600
2    UBRR0H = (unsigned char)(baud>>8);    // set baud rate high bytes
3    UBRR0L = (unsigned char)baud;         // set baud rate low bytes
4
5    UCSR0B = (1<<RXEN0)|(1<<TXEN0);       // enable receiver and transmitter
6    UCSR0C = 0x6;                         // set frame format: 8data, 1stop
```

The baudrate code of 0x33 is found in a table inside of the atmega324p datasheet. This also from the datasheet [1].

4.2 Transmission

To send data, the function uart0_tx is used. The data register is checked. If it is empty and the tx functionality is enabled, a byte of data is output.

```c
void uart0_tx(unsigned char data)
{
    //wait for empty transmit buffer
    while(!(UCSR0A & (1<<UDRE0))){;}
    //Put data into buffer and send data
    UDR0 = data;
}
```

This code is available from the atmega datasheet [1].

4.3 Printf

The UART reads and transmits bytes as hex and ASCII. The use of the C function "printf()" allows for the output of dynamic variables. To use this with the UART, several steps are necessary. First a FILE pointer must be created. The FILE pointer tells the compiler that data will be read from or written to a file. Next, the standard output must be changed. The standard output points to the command line by default. This needs to be changed to point to the address of the file pointer that was created. The next step is updating the transmit method. The transmit method needs the FILE pointer appended as a parameter. Finally, the stdio.h header is necessary. This is demonstrated in the following code:

```c
#include <stdio.h>

void uart0_tx(unsigned char data, FILE *stream)
{
    //wait for empty transmit buffer
    while(!(UCSR0A & (1<<UDRE0))){;}
    //Put data into buffer and send data
    UDR0 = data;
}

// create FILE stream. Note that the parameters of FDEV_SETUP_STREAM use uart0_tx
static FILE uart_out = FDEV_SETUP_STREAM(uart0_tx, NULL, _FDEV_SETUP_WRITE);

void main(void)
{
    stdout=&uart_out;       // set stdout to hyper terminal
    printf("test");         // test the printf
}
```

This is a summary of the code from the avr-libc [3]. The timer values can now be printed to the terminal.

4.4 Formatting

The terminal emulator selected for this project must run a version of VT100. This includes VT102. The VT100 terminal allows for output to the terminal to be customized in many ways by sending a configuration code to it. These codes are input as bytes in this document. To display

the four timers in four seperate quadrants, the cursor position is adjusted by use of these codes. It is done as follows:

```
    printf(no_cursor);       // removes blinking cursor
                             // no_cursor = 0x1b, 0x5b, 0x3f, 0x32, 0x35, 0x6c, 0x00
    printf(set_home_0_0);    // set position to 0,0: set_home_0_0 =
                             // 0x1b, 0x5b, 0x30, 0x3b, 0x30, 0x48, 0x00
    printf("counter 1: %04d",counter1); // display 1 millisecond
```

Each of the listed bytes are ASCII. A full list of terminal commands can be found at the following reference [5]. This is done for every counter and placed inside of its respective loop.

```
    // timer 2 loop
    if(timer2 >= 5){         // count to 500 milliseconds
        counter2++;          // increment counter
        printTimer2();       // packaged method for printing timer value
        timer2 = 0;          // reset 5 millisecond timer
    }
```

5 Reset

A reset button is required for this project. This reset button clears the four timers back to 0. This is done by setting an interrupt every time an input signal on PB3 is applied.

```
   EICRA   = (1 << ISC21);    // Fall edge of INT2 generates int
  EIMSK    = (1 << INT2);     // Enable ints for INT2
  EIFR   = (0 << INTF0);    // Clear int buffer
```

To enable this functionality, the DDRB at pin 3 is set to 0 (by default). This enables an input signal on PB3. PB3 is a special pin used for the interrupt vectors known as INT2. When the button is pushed, the counters and timers are cleared to 0.

5.1 Hardware

The circuit for this button is shown below.

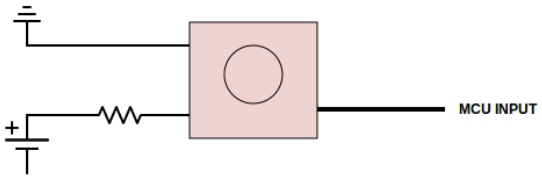

Figure 2: Button Circuit

A pull up resistor is required to keep the button high until it is pressed. The signal is applied from the MCU at pin 3. The button also needs to be grounded.

6 FTDI

To physically view the incoming data, a terminal emulator and an FTDI cable are necessary. The FTDI cable requires the installation of the FTDI drivers, which can be found at the FTDI website [2]. Any terminal emulator may be used so long as the terminal parameters are functionable with the UART parameters. After the drivers have been installed and the terminal emulater setup, the physical interfacing takes place. The TTL end uses six signals. They are as pictured below:

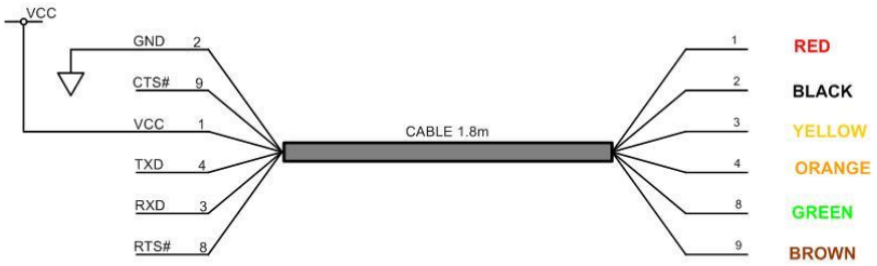

Figure 3: FTDI Layout [2]

For this project, only three signals are needed. The V_{cc}, GND, and RXD. This is because this cable is receiving the transmission. Connect the three wires to their respective partners. The RXD wire should be connected to the TX0 pin (assuming the UART0 is enabled) of the MCU. This is pin 15 of the atmega324p.

7 Configuration

This project requires the MCU frequency to be set to 8 MHz. To do this, the user must change the fuse bits of the MCU. The fuse bits of the MCU are the parameters used to control the MCU settings. To program these fuse bits, JTAG mode is necessary.

7.1 JTAG

JTAG programming is protocol used for communication between the atmel-ICE programmer and the MCU. The atmel-ICE includes a JTAG to JTAG cable. The MCU also features JTAG pins. The JTAG protocol uses 10 wires.

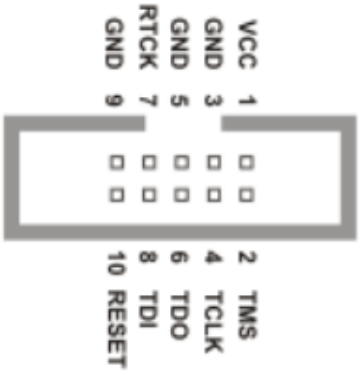

Figure 4: JTAG Header [4]

To use this, simply connect the female header of the cable to the respective partner pins of the MCU. Due to the size of the pin-holes used on the JTAG cable, it is recommended that a JTAG breakout board be used. This allows for easy configuration between the cable and the header.

7.2 Fuses

The previous report demonstrated how to communicate with the MCU using command line. To set the clock frequency, avrdude in terminal mode will be used. To enter terminal mode, use the following commamd:

```
$ avrdude -c atmelice -p m324p -u -t
```

The -u parameter disables safemode and allows for the fuse bits to be changed and the -t parameter allows the program to enter terminal mode. To set the clock frequency to 8 MHz, the clock divider bits needs to be disabled. The default fuse register is set equal to 0x62. The clock divider bit is bit 7. So the fuse should now be 0xe2. To program this, use the following command:

```
$ w lfuse 0 0xe2
$ q
```

7.3 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.

References

[1] *8-bit Atmel Microcontroller with 16K/32K/64K Bytes In-System Programmable Flash*. URL: http://www.atmel.com/images/atmel-8011-8-bit-avr-microcontroller-atmega164p-324p-644p_datasheet.pdf.

[2] *FTDI TTL Serial*. URL: http://www.ftdichip.com/Products/Cables/USBTTLSerial.htm.

[3] *Standard IO*. URL: http://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html.

[4] *Target Connectors*. URL: http://www.keil.com/support/man/docs/ulink2/ulink2_hw_connectors.htm.

[5] *VT100 Escape codes*. URL: http://www.ccs.neu.edu/research/gpc/MSim/vona/terminal/VT100_Escape_Codes.html.