Engine Control Unit Tuning Module Test System

Author: Efren CABEBE
Supervisor: Dr. Art MACCARLEY
LIST OF FIGURES

6.1 Top-view of analog pre-prototype ................................................................. 8
6.2 Potentiometers and 555 circuit ................................................................. 8
6.3 Pre-prototype module driver circuit .......................................................... 9
6.4 NE555 Astable timer output (800RPM) ..................................................... 9
6.5 Tuning module test tables for pre-prototype ............................................. 10
6.6 Microchip MCP4921 .............................................................................. 11
6.7 Microchip MCP3002 .............................................................................. 11
6.8 AVR ATXmega256A3BU Evaluation Board .................................................. 12
6.9 Flowchart for main() .............................................................................. 12
6.10 spi_set_input_voltage function ............................................................. 13
6.11 spi_set_input_voltage flowchart .............................................................. 13
6.12 spi_read_output_voltage function ........................................................... 14
6.13 spi_read_output_voltage flowchart ........................................................... 14
6.14 RPM to timer count value conversion ....................................................... 15
6.15 RPM to clock period conversion function ................................................ 15
6.16 Timer Counter interrupt callback functions ............................................. 16
6.17 Timer Counter RPM waveform (500RPM) ............................................... 16
6.18 USART functions .................................................................................... 17
6.19 USART flowchart .................................................................................... 17
6.20 eventDispatcher function ....................................................................... 18
6.21 voltage_single_test function .................................................................. 19
6.22 voltage_single_test flowchart .................................................................. 19
6.23 Delicious Tuning desktop application ....................................................... 20
7.1 Test system debug output ......................................................................... 21
10.1 Original development time ....................................................................... 24
10.2 IN1 & IN2 voltage sweep displaying module-programmed voltage offset ......................................................... 38
10.3 RPM sweep displaying module-programmed voltage offset .................. 38
10.4 5V sawtooth input (top) tuning module output (bottom) ......................... 39
10.5 SPI initialization ....................................................................................... 40
10.6 USART initialization ................................................................................ 41
10.7 Timer Counter initialization .................................................................... 42
10.8 main() .................................................................................................... 43
10.9 Test System - Top View ........................................................................... 43

LIST OF TABLES

4.1 Marketing Requirements ........................................................................... 7
5.1 Engineering Specifications .......................................................................... 7
10.1 Bill of Materials ......................................................................................... 23
1 Abstract

A test system has been developed to provide quality assurance and functional testing for soon-to-be mass produced engine tuning modules. The system is comprised of two halves: the front-end graphical user interface and back-end microcontroller platform. The use of the interface is intuitive in starting tests and reporting the status of completed tests. The back-end microcontroller platform performs the physical tests. Two high precision analog-to-digital converters and two digital-to-analog converters provide the inputs to and receive the outputs from the module, with another pulse-width-modulated pin tied to the tachometer input. Alongside the test system infrastructure, a simple suite of tests was developed to provide basic functionality testing of the tuning module.
I would like to thank William Knose of Delicious Tuning for giving me the opportunity to partake in this project. I would like to also thank my senior project advisor, Dr. MacCarley, for the continued guidance and wealth of information learned with this project and in automotive electronics in general. Lastly, I would like to thank all my professors whose instruction has paved the way for me to reach this point.
3 INTRODUCTION

A well respected engine tuning specialist, William Knose Jr. is the President / CEO of the automotive performance start-up Delicious Tuning, as well as Tuning Director of HG Motorsports in San Diego, CA. Mr. Knose has developed a module which modifies input data from a vehicle’s onboard sensors to the engine control unit (ECU). This provides the ability to both improve a vehicle’s engine performance, while still retaining the full functionality of the engine control unit. Mr. Knose is planning on mass producing these modules. This project will have the student develop a test suite system which assesses both the functionality of the components within the module as well as of the device as a whole after each are produced. The task includes: functionality testing of soon-to-be mass produced Delicious Tuning Engine Interceptor Module. The project will involve two sections: the front end GUI and backend test bed system.

3.1 BACKGROUND

Found in modern vehicles, internal combustion engines (ICE) employ sensors which monitor the air intake into the engine. These sensors either measure the mass air flow (MAF) or manifold air pressure (MAP) of the engine’s intake system. Alongside these intake sensors, the speed of the engine, measured in rotations per minute (RPM), are sent into the engine’s engine control unit (ECU), which are used to control various engine parameters. This control system is used to maintain an optimal air-fuel mixture for combustion, or stoichiometry. The tuning module would be inserted in between the engine sensors and engine control unit, and using the outputs of the engine sensors as inputs into the tuning module, intercept and offset the voltages en route to the engine control unit.

![Typical Electronic Engine Control System](image)

Figure 3.1: Typical Electronic Engine Control System[4]
4 REQUIREMENTS

The module was designed to adhere with several marketing requirements. Marketing requirements do not necessarily define implementation, but provide a functional overview of the product.

<table>
<thead>
<tr>
<th>#</th>
<th>Marketing Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Must be able to simulate engine sensor data</td>
</tr>
<tr>
<td>2</td>
<td>UI must display test status in a meaningful and clear fashion</td>
</tr>
<tr>
<td>3</td>
<td>Must be plug-and-play</td>
</tr>
<tr>
<td>4</td>
<td>Time to conduct tests must be relatively low</td>
</tr>
</tbody>
</table>

Table 4.1: Marketing Requirements

5 SPECIFICATIONS

To support the Marketing Requirements from Table 4.1, the following Engineering Specifications were developed.

<table>
<thead>
<tr>
<th>#</th>
<th>Marketing Req. #</th>
<th>Engineering Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Interface high precision Digital-to-Analog (DAC) and Analog-to-Digital (ADC) converters</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Design user interface in Visual Studio with a minimalist interface of a start button, stop button, and text box readout</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Utilizes a FTDI RS232 &quot;USB to Serial&quot; module which is recognized as a Virtual COM Port (VCP)</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Microcontroller platform uses 2MHz clockspeed and 1MHz SPI peripheral clockspeed</td>
</tr>
</tbody>
</table>

Table 5.1: Engineering Specifications
6 Design

6.1 Overview

The proposed test system would include a host module microcontroller (test module) that interfaces with the tuning module through the harness. The test module would hold the microcontroller as well as all the supporting circuitry for power and testing. The test module would be connected to the user’s machine through a USB serial link. This enables communication with the test module. The test module would run test loaded from the user machine. These tests are sent as emulated sensor inputs to the engine module and sent back from the module as outputs back to the test module. The test module would then determine if these outputs are expected. The user machine would have a GUI running, displaying a list of tests that have passed, failed, or have yet to run. The GUI would also control when the test module would start the series of tests.

6.2 Pre-Prototype

Major steps were completed in discovering the behavior of the module. With essentially no knowledge of the behavior of the module when actually offsetting input voltages (which is critical to data acquisition of the module in the functional prototype and final test bed), an obvious first step would be creating a fully analog (no digital waveform generation) test system to ascertain how the system behaves under different input contexts, as well as various module characteristics (Figures 6.1 and 6.2). The circuit schematic can be seen in Figure 6.3.

![Image](image1.png)  ![Image](image2.png)

Figure 6.1: Top-view of analog pre-prototype  Figure 6.2: Potentiometers and 555 circuit
The analog pre-prototype consisted of two 10KΩ potentiometers which provided 0V - 5V to the inputs (IN1/IN2 and IN3) of the tuning module. An active-low astable 555 clock generation circuit drives the RPM pickup (TACH). All of the voltage and clock generation waveforms are controlled by potentiometers R1 & R2 and R3 & R4 respectively.

Figure 6.4: NE555 Astable timer output (800RPM)
6.2.1 Conducted Pre-Prototype Tests

A test was conducted to determine how the tuning module offsets voltages. The test table used is a 12 x 12 with element (3, 5) as +3V offset (Figure 6.5). IN1 & IN2 and TACH were modulated to see the corresponding change of outputs. Sweeping IN1 from 0V to 5V an initial expectation was an offset of voltage at that interpolated voltage division. Instead, the slope of the voltage increases, then decreases to normal after the interpolated voltage division (Figure 8). This characteristic is also displayed using the DockLight "SEND A-D DATA" UART command (Appendix Figure 10.1). The increase in voltage slope seems to encompass the whole interpolated voltage division of the table. The next page displays the DockLight outputs for the "SEND A-D DATA" UART command, displaying the behavior of the module with the IN1 / IN2 & TACH voltage swept. Notice in Appendix Figure 10.1 the start of the increase of OUT1 voltage @ 1.22V, peaking @ 1.61V, and returning to normal (1:1) output @ 2.09V. Voltage sweep was done by hand, so voltage output is a fairly coarse representation. In Appendix Figure 10.2, RPM was swept from 1100RPM to 2000 RPM. At 1100RPM w/ IN1 = 1.65V, the current position within the offset table was within the +3V offset cell. Increase of RPM moves position of the table to the right, which results in a return to normal (1:1) output for OUT1. RPM is modulated through potentiometers R3 & R4 in Figure 6.3. Figure 6.4 displays the waveform for minimum (800RPM) of the astable 555 timing circuit.

![Figure 6.5: Tuning module test tables for pre-prototype](image-url)
6.3 HARDWARE

Critical to the project is the use of Digital-to-Analog and Analog-to-Digital converters. The use of these peripherals enable the test system to replicate engine sensors to be used as inputs to the tuning module, as well as measure output voltages from the tuning module.

6.3.1 DIGITAL-TO-ANALOG CONVERTERS

The digital-to-analog converters used are Microchip's MCP4921 DAC IC. The digital-to-analog converter has a 12-bit resolution and interfaces through a Serial Peripheral Interface (SPI). With a total resolution of 4095 bits, the corresponding voltage-per-bit is on the order of 1 milivolt. Another feature that the MCP4921 employs is double-buffered inputs. There exists a data input and data output register within the component. Once an input is received from the SPI interface, it is placed within the input register. The toggling of the !LDAC pin moves the value within the input register to the output register, where the corresponding voltage is then outputted. With this scheme, the DAC can simultaneously output a voltage while reading SPI input.

6.3.2 ANALOG-TO-DIGITAL CONVERTERS

Likewise, the analog-to-digital converter used are Microchip's MCP3002 ADC IC. This component is a 10-bit device, chosen with the intention of maintaining optimal voltage measurement resolution. Like the MCP4921, this analog-to-digital converter utilizes the SPI interface for communication. The MCP3002 has two channels of voltage measurement input, which can be used for single-ended (reference to GND) or differential (voltage across the first to the second channel) inputs. In this test system, single-ended voltage measurement is used.
6.3.3 \textbf{Microcontroller}

Central to the test system, the microcontroller controls all of the peripherals, runs the test scheme, and communicates with the host machine. The microcontroller used was the AVR ATXMega256A3BU Xplained[3] evaluation board. This board is relatively powerful in functionality, with its rich variety of I/O technologies, large onboard flash memory, and built-in LCD screen.

Features include:

- 256KBytes flash
- 32MHz processor
- Seven 16-bit Timers
- Six USART interfaces
- Two SPI interfaces
- Multi-tier interrupts

6.4 \textbf{Firmware / Software}

Both the firmware written for the microcontroller and software written for the application program determines the functionality of the test bed as a whole. Each can be divided into two sections: front-end and back-end.

6.4.1 \textbf{Microcontroller Back-end}

The microcontroller utilizes a few technologies found on the XMega256A3BU Evaluation board. Namely, the Serial Peripheral Interface, Universal Asynchronous Receiver Transmitter, and Timer Counters are utilized.

The back-end of the microcontroller firmware focuses on the communication to the SPI interface and use of the Timer Counter module. The main program itself only calls initialization functions; the rest of the test system is entirely interrupt driven, as seen in Figure 6.9. The initialization functions can be found in Appendix Figures 10.4, 10.5 and 10.6
Communicating with the analog-to-digital converters requires the use of the Serial Peripheral Interface. A multi-master-multi-slave interface, configurations such as one master addressing multiple slaves is common. This sort of configuration is what is used within the test system. Sending data through the SPI interface is relatively simple: write the value to send in the SPI data buffer, and wait for the transmission to complete. This can be seen in Figure 6.10 below, where after the first byte of data is assigned, the while loop immediately after polls a status bit determining if the transmission has completed. In this case, values corresponding to voltage are sent to the Digital-to-Analog converters.

```c
void spi_set_input_voltage(SPI_t *spi, uint16_t input, uint8_t ss, uint8_t ldac) {
    // bitshift magic (oh, aah)
    input &= 0x0FFF;
    spi_input_buf[0] = 0x30 | input >> 8;
    spi_input_buf[1] = input & 0x0FFF;

    PORTC.OUTCLR = ss; // slave select LOW

    // send first byte of data
    (*spi).DATA = spi_input_buf[0];
    while (!((*spi).STATUS & SPI_IF_bm));

    // send second byte of data
    (*spi).DATA = spi_input_buf[1];
    while (!((*spi).STATUS & SPI_IF_bm));

    PORTC.OUTSET = ss; // slave select HIGH

    // toggle LDAC pin
    PORTB.OUTCLR |= ldac;
    PORTB.OUTSET |= ldac;
}
```

Figure 6.10: spi_set_input_voltage function

![Figure 6.10: spi_set_input_voltage function](image)

Figure 6.11: spi_set_input_voltage flowchart
Similar to the spi_set_input_voltage function, the spi_read_output_voltage function sends configuration data to the analog-to-digital converters in order to receive data values corresponding to voltage. Figures 6.12 and 6.13 outline the function and flowchart of reading voltage measurements from the analog-to-digital converters via SPI.

```c
uint16_t spi_read_output_voltage(SPI_t *spi, uint8_t ss) {
    uint16_t ret;
    char *s = calloc(9, sizeof(char));
    uint8_t data[2] = { 0x00, 0x00 };

    PORTE.OUTCLR = ss; // slave select LOW

    // Send start bit
    (*spi).DATA = 0x01;
    while (!(((*spi).STATUS & SPI_IF_bm)));

    // Send config data
    (*spi).DATA = 0x00;
    while (!(((*spi).STATUS & SPI_IF_bm)));

    // Read first byte from SPI data buffer
    data[0] = (*spi).DATA;

    // Send dummy data to shift in next data
    (*spi).DATA = 0x00;
    while (!(((*spi).STATUS & SPI_IF_bm)));

    // Read last byte from SPI data buffer
    data[1] = (*spi).DATA;

    // slave select HIGH
    PORTE.OUTSET = ss;

    // more bitshift magic
    ret = ((data[0] & 0xff) << 8) | data[1];
    //uart_sendString(USART_FTDI, utoa(ret, s, 10), true);
    // free string
    free(s);

    return ret;
}
```

Figure 6.12: spi_read_output_voltage function

![Flowchart](image-url)
Engine RPM replication comes from using the timer counter module. The waveform desired is a square wave with a fixed high time of one millisecond, and variable period. A characteristic with engine RPM is that the signal that needs to be replicated comes from the engine's coilpack, which high portions of the waveform represent the firing of any given sparkplug. A decreasing period of the coilpack waveform corresponds to an increasing of RPM, and vice versa. This relationship required a conversion model in order to convert from RPM to a corresponding value that the timer module can count to.

![RPM to Timer Count Value Conversion](image)

Figure 6.14: RPM to timer count value conversion

Various data points were found and plotted, and the scatter plot of Figure 6.14 displays the trend. A power trend line was found to be a completely perfect fit, with a \( R^2 \) (correspondence of regression line to data) of 1. Although the trend line has a perfect fit with the data, the coefficient still had to be modified to provide accurate conversions. This function can be seen in Figure 6.15, where RPM_DIV, the coefficient, is 1.547 rather than 2.

```c
uint32_t rpm_to_clkper(uint16_t rpm) {
    return (RPM_DIV * pow(10, 7) / rpm);
}
```

Figure 6.15: RPM to clock period conversion function
This coilpack waveform is generated using timer overflow and compare interrupts. The timer compare interrupt is set to trigger after a fixed length. Once it is called, the timer disables the compare channel and sets the corresponding I/O pin to low. Turning itself off makes sure the compare interrupt will not trigger other than once within the timer’s entire period. Once the timer reaches its period value, it overflows to zero. At this point, the timer enables the compare channel and sets the corresponding I/O pin to high. The compare interrupt triggers after a fixed length, and the cycle starts again. This repeats until power is disconnected.

Figure 6.16: Timer Counter interrupt callback functions

Figure 6.17: Timer Counter RPM waveform (500RPM)
Establishing connection between the microcontroller and host machine is achieved through the use of the USART module and a FTDI USB to Serial adapter. A simple serial interface, USART provides connectivity with relative ease. When the FTDI USB to Serial adapter receives a character from the host machine, the USART_FTDI_RXC_vect interrupt triggers and handles the character accordingly by sending it to the usart_ftdi_recv_handler. The character is either appended to the receive buffer, or once the receive buffer is full or the character is a carriage return, sends the string to the eventDispatcher for processing of the completed command. The usart_sendString function is used to write strings back to the host machine.

```c
void usart_ftdi_recv_handler(uint8_t c) {
    if (c != '\r' && rx_index < USART_BUF_SIZE - 1) {
        // set input into buffer
        rx_buf[rx_index++] = c;
    } else {
        // place carriage return and newline
        usart_putchar(USART_FTDI, '\r');
        usart_putchar(USART_FTDI, '\n');
        // make rx_buf a null-terminated string
        rx_buf[USART_BUF_SIZE - 1] = '\0';
        // send string to event dispatcher
        eventDispatcher((char *)rx_buf);
        // clear rx_buf
        memset(rx_buf, 0, sizeof(rx_buf));
        // reset rx_index counter
        rx_index = 0;
    }
}
```

```c
ISR (USART_FTDI_RXC_vect) {
    // read received byte from USART reg
    ftdi_recv_byte = (USART_FTDI).DATA;
    // echo char back (minicom terminal type)
    // remove this if final hex
    usart_putchar(USART_FTDI, ftdi_recv_byte);
    // send received byte to handler
    usart_ftdi_recv_handler(ftdi_recv_byte);
}
```

```c
void usart_sendString(USART_t *user, char *s, bool crnl) {
    while (*s) {
        usart_putchar(user, *s);
        if (crnl) {
            usart_putchar(user, '\r');
            usart_putchar(user, '\n');
        }
        s++;
    }
}
```

**Figure 6.18: USART functions**

**Figure 6.19: USART flowchart**
6.4.2 MICROCONTROLLER FRONT-END

Any interaction with the user would be with the eventDispatcher function. The eventDispatcher function takes in the string processed from the usart_ftdi_recv_handler and determines the course of action of the program. Other than the utility commands such as being able to set RPM, the inputs to the module and read the outputs from the module, one command is reserved for actually running tests. The command "STS", abbreviation for "Start Test Suite" would run a functionality test. All of these commands can be seen in Figure 6.20 below.

```c
void eventDispatcher(char *s) {
    char cmd[4] = { 0, 0, 0, 0 },
    cmd[4] = { 0, 0, 0, 0, 0 },
    int32_t* voltage;

    // Command processing
    if (strcmp(s, "M"):)
        // Module startup
    else if (strcmp(s, "FRONT-END") == 0)
        // FPGA controller
    else if (strcmp(s, "ABC") == 0)
        // Module stop
    else if (strcmp(s, "DEF") == 0)
        // Module adjust
}
```

The test_suite function runs a certain test depending on the value passed in as the parameter of the command. This function depends on the functionality of the voltage_single_test function to operate. The voltage_single_test definition can be seen as Figure 6.21, and the flowchart in Figure 6.22. The function voltage_single_test takes in five inputs: RPM, both inputs, and both expected outputs. RPM and input voltages are set, and output voltages are read and compared with the expected output voltages. If the output voltages are within a certain percentage (set to 3%), then that test passes.

![Figure 6.20: eventDispatcher function](image-url)
```c
bool voltage_single_test(uint16_t rpm, uint16_t im0, uint16_t iinl, uint16_t out0_exp, uint16_t out1_exp) {
    uint16_t out_voltage0 = 0;
    out_voltage0 = 0;
    char s[5] = \"0, 0, 0, 0, 0\"
    uart_sendString(UART_FTDI, \"NPM\", false);
    uart_sendString(UART_FTDI, utoa(rpm, 10), false);
    char s[5] = \"0, 0, 0, 0, 0\"
    uart_sendString(UART_FTDI, \"in_voltage0\", false);
    uart_sendString(UART_FTDI, utoa(im0, 10), false);
    uart_sendString(UART_FTDI, \"in_voltage1\", false);
    uart_sendString(UART_FTDI, utoa(iinl, 10), false);

    // Set RPM
    tc_write_period(WRITE_TIMER, rpm_to_clkper(rpm));
    // arithmetic delay
    _delay_us(5);
    // Set IM0 input
    spi_set_input_voltage(SPI, im0, IN1_SS_BA, LEADING_BA);
    // Set iinl input
    spi_set_input_voltage(SPI, iinl, IN1_SS_BA, LEADING_BA);
    // propagation delay
    _delay_us(5);
    // Read OUT0
    out_voltage0 = spi_read_output_voltage(SPI, OUT0_SS_BA);
    uart_sendString(UART_FTDI, \"out_voltage0\", false);
    uart_sendString(UART_FTDI, utoa(out_voltage0, 10), false);
    // Read OUT1
    out_voltage1 = spi_read_output_voltage(SPI, OUT1_SS_BA);
    uart_sendString(UART_FTDI, \"out_voltage1\", false);
    uart_sendString(UART_FTDI, utoa(out_voltage1, 10), false);

    // within 3% of expected value
    if (in0 == 0 && in1 == 0)
    {
        if (out0 < (out0_exp + 123)) &&
        (out0 < (out0_exp + 123))
        return (out0 < (out0_exp + 123)) &&
        (out0 < (out0_exp + 123));
    }
    return (out0 < (out0_exp + 123)) &&
    (out0 < (out0_exp + 123)) &&
    (out0 < (out0_exp + 123)) &&
    (out0 < (out0_exp + 123));
}
```

Figure 6.21: voltage_single_test function

![Voltage Single Test Flowchart](image)

- set RPM
- set first voltage input
- set second voltage input
- read first output voltage
- read second output voltage
- if output voltage is within 3% of the expected output voltage
- Yes: return true
- No: return false

Figure 6.22: voltage_single_test flowchart
6.4.3 APPLICATION

A desktop application was designed for ease of use when running the test system. Having a simple user interface, the application displays passed tests, failed tests, and two push buttons for starting / stopping the tests. The desktop application was written in C# within Visual Studio 2013. The FTDI USB to Serial module, when connected to a host PC, shows as a virtual communication port (VCP). Using the System.IO.Ports library, the application can communicate with the virtual communication port to send messages from the application to the microcontroller.

7 TESTING

Testing was done incrementally as the codebase was created. In the end, the display of tests running correctly show that all portions of the codebase is working as expected. Figure 7.1 shows a run of one of the test schemes. The test conducted is called a "blank table test" where input voltages should correspond to outputs; no offsetting of voltages. This is the most fundamental test to determine if inputs to the module and outputs from the module are working as expected. Verification of the test module is achieved by using the DockLight serial monitor (shown below the terminal), which displays the voltages the module itself is measuring in centivolts. The test system compares voltages in terms of values ranging from 0 to 4095. This proved easier to compare rather than having the microcontroller compare floating point numbers. As for testing, this "blank table" test was deemed suitable for testing purposes as of now.

Figure 6.23: Delicious Tuning desktop application
Figure 7.1: Test system debug output
8 Conclusion

A test system was created to provide quality assurance and functional testing for soon-to-be mass produced engine tuning modules. The use of digital-to-analog and analog-to-digital converters along with various microcontroller technologies enabled the test system to come to fruition. The infrastructure was created for tests to be created and added at any time. The comparison percentage is also modifiable. The test system is the start of a very powerful tool that would prove valuable for mass production quality assurance of the Delicious Tuning engine tuning modules.

8.1 Future Work

As the opportunity to work for William Knose of Delicious Tuning for continued development of the test system arose, future work would include creation of more sophisticated test schemes, dedicated board design and further improvements to automate tests.

9 Bibliography

[1] Microchip, "12-bit DAC with SPI Interface" MCP4921, 2004
Summary of Functional Requirements

The overall capabilities of the test system consist of:

- Replicating engine sensor voltage
- Replicating coilpack tachometer waveform
- Measurement of voltages from the tuning module
- Modifiable margin of error
- Ability to append test cases
- Host machine application to start and see the results of tests

Primary Constraints

Overall, the constraints were pretty relaxed due to the freedom of design of the test system as a whole. As a result, a few self-imposed constraints were created:

- Replicating engine sensor voltage
- Replicating coilpack tachometer waveform
- Measurement of voltages from the tuning module
- Modifiable margin of error
- Ability to append test cases
- Host machine application to start and see the results of tests

Economic

<table>
<thead>
<tr>
<th>Part</th>
<th>Quantity</th>
<th>Description</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATXmega256A3BU</td>
<td>1</td>
<td>Host micro-controller</td>
<td>$31.56</td>
<td>$31.56</td>
</tr>
<tr>
<td>Xplained Eval.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCP4921</td>
<td>2</td>
<td>12-bit Digital-to-Analog Converter</td>
<td>$2.36</td>
<td>$4.72</td>
</tr>
<tr>
<td>MCP3002</td>
<td>2</td>
<td>10-bit Analog-to-Digital Converter</td>
<td>$2.15</td>
<td>$4.30</td>
</tr>
<tr>
<td>Docklight Serial Monitor</td>
<td>1</td>
<td>Serial Monitor</td>
<td>$80.49</td>
<td>$80.49</td>
</tr>
<tr>
<td>Roving Networks RN42-XV</td>
<td>1</td>
<td>Bluetooth module</td>
<td>$20.95</td>
<td>$20.95</td>
</tr>
<tr>
<td>FTDI Basic Breakout Board</td>
<td>1</td>
<td>FTDI USB To Serial Adapter</td>
<td>$14.95</td>
<td>$14.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total:</td>
<td>$156.97</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Bill of Materials

Originally, the estimated cost for component parts was $61.53, as I figured I would not need the Docklight serial monitor or the FTDI USB to Serial Adapter. Additional equipment costs of $95.44 incurred due to the components aforementioned.
Environmental impacts would not be directly correlated with the test system, but the use of the tuning module would potentially increase greenhouse gas emissions. This is due to the change of the engine's operation from the tuning map (running richer or leaner).

Manufacturability

As for producing this test system, manufacturability would stem from PCBA manufacturing. For the actual senior project, the whole system was prototyped on a breadboard to determine functionality.

Sustainability

No issues or challenges arose when maintaining the completed system. As far as I know, there are little to no impact the sustainable use of resources, as all of the components used are RoHS compliant. Upgrades for the project would include creating a dedicated PCBA that would mount atop the ATX Mega256A3BU board. Further improvements would be a dedicated PCBA with the microprocessor integrated. Improving the design to achieve the aforementioned upgrades would pose a challenge in design.

Ethical

There are no ethical implications relating to the design, manufacture or use of the test system.
HEALTH AND SAFETY

There are no health and safety implications relating to the design, manufacture or use of the test system.

SOCIAL AND POLITICAL

There are no social or political concerns associated with the design, manufacture or use of the test system.

DEVELOPMENT

During the development of the test system, the use of the DockLight serial monitor, C# for designing the desktop application, engine sensor operation, various automotive electronics concepts and the AVR Xmega architecture were all learned during the course of this project.

10.2 SOURCE CODE

MAIN.H

```c
#ifdef MAIN_H_
define MAIN_H_
#define F_CPU 2000000UL

#include <asf.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include <math.h>
#include "conf_usart.h"
#include "conf_timers.h"
#include "conf_spi_master.h"
#include "test_definitions.h"

#define VOLTAGE_DIV 5
#define RPM_DIV 1.547
#define VOLTAGE_PREC 3

void usart_init(void);
  void usart_ftdi_recv_handler(uint8_t c);
  void usart_sendString(USART_t *usart, char *s, bool crnl);

void tc_init(void);
  void tc_cca_rpm_callback(void);
  void tc_ovf_rpm_callback(void);

void spi_init(void);
  void spi_set_input_voltage(SPI_t *spi, uint16_t input, uint8_t ss, uint8_t ldac);
  uint16_t spi_read_output_voltage(SPI_t *spi, uint8_t ss);

void eventDispatcher(char *s);
  uint32_t rpm_to_clkper(uint16_t rpm);
  uint16_t conv_voltage(float voltage);
```
```c
#include "main.h"

volatile uint8_t ftdi_recv_byte, bt_recv_byte;

uint8_t spi_input_buf[2] = { 0x00, 0x00 };
uint8_t spi_recv_buf[2] = { 0x00, 0x00 };

uint8_t rx_buf[USART_BUF_SIZE],
       tx_buf[USART_BUF_SIZE];

uint8_t rx_index = 0, tx_index = 0;

int main (void)
{
    board_init();
sysclk_init();
spi_init();
uart_init();
tc_init();

    // enable all interrupt levels from interrupt vector table
    irq_initialize_vectors();

    // enable interrupts
    cpu_irq_enable();

    while (1) {

    }

    void eventDispatcher(char *s) {

        char cmd[4] = { 0, 0, 0, 0 },
        val[6] = { 0, 0, 0, 0, 0, 0 };

        char *str = calloc(1, sizeof(char) * 10);

        uint16_t voltage;

        strncpy(cmd, s, 3);
        strncpy(val, s + 4, 5);

        cmd[3] = '\0';
        val[5] = '\0';

        if (strcmp(cmd, "RST") == 0) {
            uart_sendString(USART_FTDI, "Resetting inputs", true);
            spi_set_input_voltage(SPI, 0, IN0_SS_bm, LDACIN0_bm);
            spi_set_input_voltage(SPI, 0, IN1_SS_bm, LDACIN1_bm);
        }

        if (strcmp(cmd, "RPM") == 0) {
            uart_sendString(USART_FTDI, "Setting RPM", true);
            tc_write_period(RPM_TIMER, rpm_to_clkper(atoi(val)));
        }
```

if (strcmp(cmd, "IN0") == 0) {
    voltage = conv_voltage(atof(val));
    usart_sendString(USART_FTDI, " Setting IN0 to: ", false);
    usart_sendString(USART_FTDI, utoa(voltage, str, 10), true);
    spi_set_input_voltage(SPI, voltage, IN0_SS_bm, LDACIN0_bm);
}

if (strcmp(cmd, "IN1") == 0) {
    voltage = conv_voltage(atof(val));
    usart_sendString(USART_FTDI, " Setting IN0 to: ", false);
    usart_sendString(USART_FTDI, utoa(voltage, str, 10), true);
    spi_set_input_voltage(SPI, voltage, IN1_SS_bm, LDACIN1_bm);
}

if (strcmp(cmd, "OT0") == 0) {
    usart_sendString(USART_FTDI, "OUT0 voltage: ", false);
    usart_sendString(USART_FTDI, utoa(spi_read_output_voltage(SPI, OUT0_SS_bm), val, 10), true);
}

if (strcmp(cmd, "OT1") == 0) {
    usart_sendString(USART_FTDI, "OUT1 voltage: ", false);
    usart_sendString(USART_FTDI, utoa(spi_read_output_voltage(SPI, OUT1_SS_bm), val, 10), true);
}

//
if (strcmp(cmd, "BTH") == 0) {
    usart_sendString(USART_FTDI, "Sending command mode string", true);
    usart_sendString(USART_BT, "$$", false);
}

// Start Testing Scheme
if (strcmp(cmd, "STS") == 0) {
    usart_sendString(USART_FTDI, "Running test number: ", false);
    usart_sendString(USART_FTDI, val, true);
    test_suite(atoi(val));
}

uint16_t conv_voltage(float voltage) {
    return (voltage / VOLTAGE_DIV) * 0xFFF;
}

uint32_t rpm_to_clkper(uint16_t rpm) {
    return (RPM_DIV * pow(10, 7) / rpm);
}

void usart_init(void) {

}
/* FTDI USART */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

// set USART settings
static usart_rs232_options_t USART_FTDI_OPTIONS = {
    .baudrate = USART_FTDI_BAUDRATE,
    .charlength = USART_FTDI_CHAR_LENGTH,
    .paritytype = USART_FTDI_PARITY,
    .stopbits = USART_FTDI_STOP_BIT
};

// initialize USART_FTDI in rs232 mode
usart_init_rs232(USART_FTDI, &USART_FTDI_OPTIONS);

// enable USART receive interrupts to LOW
usart_set_rx_interrupt_level(USART_FTDI, USART_INT_LVL_LO);

// disable USART transmit interrupts
usart_set_tx_interrupt_level(USART_FTDI, USART_INT_LVL_OFF);

// disable USART data reg interrupts
usart_set_dre_interrupt_level(USART_FTDI, USART_INT_LVL_OFF);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* BLUETOOTH USART */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

static usart_rs232_options_t USART_BT_OPTIONS = {
    .baudrate = USART_BT_BAUDRATE,
    .charlength = USART_BT_CHAR_LENGTH,
    .paritytype = USART_BT_PARITY,
    .stopbits = USART_BT_STOP_BIT
};

// initialize USART_BT in rs232 mode
usart_init_rs232(USART_BT, &USART_BT_OPTIONS);

// enable USART receive interrupts to LOW
usart_set_rx_interrupt_level(USART_BT, USART_INT_LVL_LO);

// disable USART transmit interrupts
usart_set_tx_interrupt_level(USART_BT, USART_INT_LVL_OFF);

// disable USART data reg interrupts
usart_set_dre_interrupt_level(USART_BT, USART_INT_LVL_OFF);

void tc_init(void) {

    // Enable TCC0 peripheral clock
    sysclk_enable_peripheral_clock(&TCC0);

    // Set RPM_PIN (PC1) to an output
    PORTC.DIRSET = RPM_PIN;

    // Set RPM_PIN initial output to LOW
    PORTC.OUTCLR = RPM_PIN;

    // Enable RPM timer counter (TCC0)
    tc_enable(RPM_TIMER);

    // Set CCA interrupt callback function
    tc_set_cca_interrupt_callback(RPM_TIMER, tc_cca_rpm_callback);
}
// Set OVF interrupt callback function
tc_set_overflow_interrupt_callback (RPM_TIMER, tc_ovf_rpm_callback);

// Set wave generation mode to NORMAL
tc_set_wgm (RPM_TIMER, TC_WG_NORMAL);

// Set channel compare
    /*
    Tweak this value to achieve 1ms wide spike
    */
tc_write_cc (RPM_TIMER, TC_CCA, 500);

// Init RPM to 750 (idle)
tc_write_period (RPM_TIMER, rpm_to_clkper (START_TEST_RPM));

// Enable Compare Channel A on RPM Timer (TCC0)
tc_enable_cc_channels (RPM_TIMER, TC_CCAEN);

// Set interrupt level of Compare Channel A
tc_set_cca_interrupt_level (RPM_TIMER, TC_INT_LVL_MED);

// Set interrupt level of TCC0 overflow
tc_set_overflow_interrupt_level (RPM_TIMER, TC_INT_LVL_MED);

// Sets clock div factor and starts timer
tc_write_clock_source (RPM_TIMER, TC_CLKSEL_DIV4_gc);
}

void spi_init (void) {

    // Enable SPIIC peripheral clock
    sysclk_enable_peripheral_clock (SPI);

    /* Init SS and SPI MOSI, SCK pins as outputs, MISO as input */
    PORTC. DIRSET = IN0_SS_bm | IN1_SS_bm | SPI_DAC_MOSI_bm | SPI_DAC_SCK_bm; // Input 0 DAC
    PORTC. DIRCLR = SPI_ADC_MISO_bm; // Data in (for ADC)

    PORTB. DIRSET = LDACIN0_bm | LDACIN1_bm; // LDAC for Input 0
    PORTB. DIRCLR = OUT0_SS_bm; // Output 0 ADC

    PORTE. DIRSET = OUT1_SS_bm; // Output 1 ADC

    /* Wired and pull up SS outputs */
    PORTC. PIN4CTRL = PORT_OPC_WIREDANDPULL_gc; // IN0
    PORTC. PIN0CTRL = PORT_OPC_WIREDANDPULL_gc; // IN1
    PORTE. PIN0CTRL = PORT_OPC_WIREDANDPULL_gc; // OUT0
    PORTE. PIN1CTRL = PORT_OPC_WIREDANDPULL_gc; // OUT1

    /* Set SS & LDAC outputs to high. (active low operation) */
    PORTC. OUTSET = IN0_SS_bm | IN1_SS_bm;
    PORTB. OUTSET = LDACIN0_bm | LDACIN1_bm;
    PORTE. OUTSET = OUT0_SS_bm | SDAC0_bm;
    PORTE. OUTSET = OUT1_SS_bm | SDAC1_bm;
SPIC::INTCTRL = SPI_INTVL_LO_gc;  // set SPI interrupt level
SPIC::CTRL = SPI_PREScaler_DIV4_gc | // SPI prescaler
   SPI_ENABLE_bm | // SPI enable
   SPI_MASTER_bm | // set as SPI master
   SPI_MODE_0_gc; // set SPI mode
}

void spi_set_input_voltage(SPI_t *spi, uint16_t input, uint8_t ss, uint8_t ldac) {
   // bitshift magic (ohh, aah)
   input &= 0x0FFF;
   spi_input_buf[0] = 0x30 | input >> 8;
   spi_input_buf[1] = input & 0x00FF;

   PORTC.OUTCLR = ss; // slave select LOW
   // send first byte of data
   (*spi).DATA = spi_input_buf[0];
   while (!((*spi).STATUS & SPI_IF_bm));

   // send second byte of data
   (*spi).DATA = spi_input_buf[1];
   while (!((*spi).STATUS & SPI_IF_bm));

   PORTC.OUTSET = ss; // slave select HIGH
   // toggle LDAC pin
   PORTB.OUTCLR |= ldac;
   PORTB.OUTSET |= ldac;
}

uint16_t spi_read_output_voltage(SPI_t *spi, uint8_t ss) {
   uint16_t ret;
   char *s = calloc(9, sizeof(char));
   uint8_t data[2] = { 0x00, 0x00 };

   PORTE.OUTCLR = ss; // slave select LOW
   // Send start bit
   (*spi).DATA = 0x01;
   while (!((*spi).STATUS & SPI_IF_bm));
   // Send config data
   (*spi).DATA = 0xA0;
   while (!((*spi).STATUS & SPI_IF_bm));

   // Read first byte from SPIC data buffer
   data[0] = (*spi).DATA;
   // Send dummy data to shift in next data
   (*spi).DATA = 0x00;
   while (!((*spi).STATUS & SPI_IF_bm));
   // Read last byte from SPIC data buffer
   data[1] = (*spi).DATA;

   // slave select HIGH
   PORTE.OUTSET = ss;
   // more bitshift magic
   ret = ((data[0] & 0x0F) << 8) | data[1];
void tc_cca_rpm_callback(void) {
    /*
     * At this point, the timer has finished outputting
     * the high portion of the waveform.
     */
    tc_disable_cc_channels(&TCC0, TC_CCAEN);
    PORTC.OUTCLR = RPM_PIN;
}

void tc_ovf_rpm_callback(void) {
    /*
     * At this point, the timer has finished outputting
     * the low portion of the waveform.
     */
    tc_enable_cc_channels(&TCC0, TC_CCAEN);
    PORTC.OUTSET = RPM_PIN;
}

void usart_sendString(USART_t * usart, char * s, bool crnl) {
    while (*s) {
        usart_putchar(usart, *s);
        if (crnl) {
            usart_putchar(usart, '');
            usart_putchar(usart, '
');
        }
    }
}

void usart_ftdi_recv_handler(uint8_t c) {
    if (c != '\r' && rx_index < USART_BUF_SIZE - 1) {
        // set input into buffer
        rx_buf[rx_index++] = c;
    } else {
        // place carriage return and newline
        usart_putchar(USART_FTDI, '\r');
        usart_putchar(USART_FTDI, '\n');
        // make rx_buf a null-terminated string
        rx_buf[USART_BUF_SIZE - 1] = '\0';
        // send string to event dispatcher
        eventDispatcher((char*)rx_buf);
        // clear rx_buf
```c
    memset(rx_buf, 0, sizeof(rx_buf));

    // reset rx_index counter
    rx_index = 0;

ISR (USART_FTDI_RXC_vect) {

    // read received byte from USART reg
    ftdi_recv_byte = (*USART_FTDI).DATA;

    // echo char back (mimic terminal type)
    // remove this in final hex
    usart_putchar(USART_FTDI, ftdi_recv_byte);

    // send received byte to handler
    usart_ftdi_recv_handler(ftdi_recv_byte);

ISR (USART_BT_RXC_vect) {

    bt_recv_byte = (*USART_BT).DATA;

    usart_putchar(USART_FTDI, bt_recv_byte);
}
ISR (SPIC_INT_vect) {

}

TEST_DEFINITIONS.H

/*
 * test_definitions.h
 */
*
* Created: 5/30/2014 12:27:07 AM
* Author: Efren
*
#include "main.h"

#define TEST_ARR_SIZE 12
#define TEST_PERCENT_ERR 0.01
#define MAX_MEAS 0xFFF
#define VOLTAGE_DELTA 391
#define RPM_DELTA (550)
#define START_TEST_RPM (500)

uint16_t test_arr[TEST_ARR_SIZE][TEST_ARR_SIZE];

uint16_t out0_voltage, out1_voltage;

bool voltage_single_test(uint16_t rpm, uint16_t in0, uint16_t in1, uint16_t out0_exp, uint16_t out1_exp);

bool test_suite(uint8_t test_num);

TEST_DEFINITIONS.C

bool test_suite(uint8_t test_num) {
```
```c
uint16_t rpm_x;
double vload_y;
char s[5] = { 0, 0, 0, 0, 0 };

// blank test
if (test_num == 1) {
  // reset inputs
  spi_set_input_voltage(SPI, 0, IN0_SS_bm, LDACIN0_bm);
  spi_set_input_voltage(SPI, 0, IN1_SS_bm, LDACIN1_bm);

  for (rpm_x = 0; rpm_x < 15; rpm_x++) {
    for (vload_y = 0; vload_y <= 4.75; vload_y += 0.25) {
      if (voltage_single_test(START_TEST_RPM + (rpm_x * RPM_DELTA), // RPM
                              conv_voltage(vload_y), // IN0
                              conv_voltage(vload_y), // IN1
                              conv_voltage(vload_y), // OUT0 expected
                              conv_voltage(vload_y))) { // OUT1 expected
        usart_sendString(USART_FTDI, " passed", true);
      } else {
        usart_sendString(USART_FTDI, " failed", true);
      }
    }
  }

  spi_set_input_voltage(SPI, 0, IN0_SS_bm, LDACIN0_bm);
  spi_set_input_voltage(SPI, 0, IN1_SS_bm, LDACIN1_bm);

  return true;
}
```

CONF_BOARD.H

```c
/** 
 * \file
 * 
 * \brief XMEGA-A3BU Xplained board configuration template
 * 
 * */
#ifndef CONF_BOARD_H
#define CONF_BOARD_H

// Initialize IO pins for use with USART 0 on port C
#define CONF_BOARD_ENABLE_USARTC0

// Initialize IO pins for use with USART 0 on port D
#define CONF_BOARD_ENABLE_USARTD0

// Initialize IO pins for use with USART 0 on port E
#define CONF_BOARD_ENABLE_USARTE0

// Enable Sensors Xplained board interface
#else define SENSORS_XPLAINED_BOARD
#endif
```
#ifndef CONF_CLOCK_H_INCLUDED
#define CONF_CLOCK_H_INCLUDED

#define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RC2MHZ

#define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RC32MHZ

#define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_XOSC

#define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_PLL

#define CONFIG_SYSCLK_PSADIV SYSCLK_PSADIV_1

#define CONFIG_SYSCLK_PSBCDIV SYSCLK_PSBCDIV_1_1

#define CONFIG_PLL0_SOURCE PLL_SRC_XOSC

#define CONFIG_PLL0_SOURCE PLL_SRC_RC2MHZ

#define CONFIG_PLL0_SOURCE PLL_SRC_RC32MHZ

#define CONFIG_PLL0_MUL (24000000UL / BOARD_XOSC_HZ)

#define CONFIG_PLL0_DIV 1

#define CONFIG_XOSC_RANGE XOSC_RANGE_04TO2

#define CONFIG_XOSC_RANGE XOSC_RANGE_2TO9

#define CONFIG_XOSC_RANGE XOSC_RANGE_9TO12

#define CONFIG_XOSC_RANGE XOSC_RANGE_12TO16

#define CONFIG_OSC_AUTOCAL_RC2MHZ_REF_OSC OSC_ID_RC32KHZ

#define CONFIG_OSC_AUTOCAL_RC32MHZ_REF_OSC OSC_ID_XOSC

#define CONFIG_USBCLK_SOURCE USBCLK_SRC_RCOSC

#define CONFIG_RTC_SOURCE SYSCLK_RTCSRC_ULP

#endif // CONF_CLOCK_H_INCLUDED
#ifndef CONF_SPI_MASTER_H_INCLUDED
#define CONF_SPI_MASTER_H_INCLUDED

#include <spi_master.h>

#define SPI &SPIC
#define SPI_DAC_MOSI_bm PIN5_bm
#define SPI_DAC_SCK_bm PIN7_bm
#define IN0_SS_bm PIN4_bm
#define IN1_SS_bm PIN0_bm
#define LDACIN0_bm PIN0_bm
#define LDACIN1_bm PIN1_bm
#define OUT0_SS_bm PIN0_bm
#define OUT1_SS_bm PIN1_bm
#define SPI_ADC_MISO_bm PIN6_bm

// ! Default Config Spi Master Dummy Field
// #define CONFIG_SPI_MASTER_DUMMY 0xFF
#endif /* CONF_SPI_MASTER_H_INCLUDED */

CONF_TIMERS.H

#ifndef CONF_TIMERS_H_
#define CONF_TIMERS_H_

#include <tc.h>
#define RPM_TIMER &TCC0
#define RPM_OVF_vect TCC0_OVF_vect
#define RPM_PIN PIN1_bm
#endif /* CONF_TIMERS_H_ */

CONF_USART.H

#ifndef CONF_USART_H_
#define CONF_USART_H_

#include "conf_board.h"
#include "sysclk.h"
#define USART_BUF_SIZE 12
#define USART_FTDI &USARTC0
#define USART_FTDI_BAUDRATE 9600
#define USART_FTDI_CHAR_LENGTH USART_CHSIZE_8BIT_gc
#define USART_FTDI_PARITY USART_PMODE_DISABLED_gc
#define USART_FTDI_STOP_BIT false
#define USART_FTDI_RXC_vect USARTC0_RXC_vect
#define USART_BT &USARTE0
#define USART_BT_BAUDRATE 115200
#define USART_BT_CHAR_LENGTH USART_CHSIZE_8BIT_gc
#define USART_BT_PARITY USART_PMODE_DISABLED_gc
#endif /* CONF_USART_H_ */
#define USART_BT_STOP_BIT false
#define USART_BT_RXC_vect USARTD0_RXC_vect

/* CONF_USART_H */

ASF.H

#ifndef ASF_H
#define ASF_H

/*
 * This file includes all API header files for the selected drivers from ASF.
 * Note: There might be duplicate includes required by more than one driver.
 * The file is automatically generated and will be re-written when
 * running the ASF driver selector tool. Any changes will be discarded.
 */

// From module: CPU specific features
#include <ccp.h>
#include <xmega_reset_cause.h>

// From module: GPIO - General purpose Input/Output
#include <gpio.h>

// From module: Generic board support
#include <board.h>

// From module: IOPORT - General purpose I/O service
#include <ioport.h>

// From module: Interrupt management - XMEGA implementation
#include <interrupt.h>

// From module: NVM - Non Volatile Memory
#include <nvm.h>

// From module: PMIC - Programmable Multi-level Interrupt Controller
#include <pmic.h>

// From module: Part identification macros
#include <parts.h>

// From module: SPI - Serial Peripheral Interface
#include <spi.h>

// From module: SPI - XMEGA implementation
#include <spi_master.h>
#include <spi_master.h>

// From module: Sleep Controller driver
#include <sleep.h>

// From module: Sleep manager - XMEGA A/AU/B/D implementation
#include <sleepmgr.h>
#include <xmega/sleepmgr.h>

// From module: System Clock Control - XMEGA AIU/A3U/A3BU/AIU/B/C implementation
#include <sysclk.h>

// From module: TC - Timer Counter
#include <tc.h>
// From module: USART – Universal Synchronous/Asynchronous Receiver/Transmitter
#include <usart.h>

// From module: XMEGA compiler driver
#include <compiler.h>
#include <status_codes.h>

// From module: XMEGA-A3BU Xplained LED support enabled
#include <led.h>

#endif // ASF_H
## 10.3 Figures

<table>
<thead>
<tr>
<th>FIGURES</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 10.2: IN1 &amp; IN2 voltage sweep displaying module-programmed voltage offset</td>
<td></td>
</tr>
<tr>
<td>Figure 10.3: RPM sweep displaying module-programmed voltage offset</td>
<td></td>
</tr>
</tbody>
</table>

### Table: Figure 10.2

<table>
<thead>
<tr>
<th>RPM</th>
<th>OUT2</th>
<th>OUT2.1</th>
<th>OUT2.2</th>
<th>OUT2.3</th>
<th>OUT2.4</th>
<th>OUT3</th>
<th>OUT3.1</th>
<th>OUT3.2</th>
<th>OUT3.3</th>
<th>OUT3.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1147 (RT)</td>
<td>101 (BT)</td>
<td>11 (BT)</td>
<td>101 (BT)</td>
<td>13 (BT)</td>
<td>101 (BT)</td>
<td>193 (RT)</td>
<td>13 (BT)</td>
<td>101 (BT)</td>
<td>193 (RT)</td>
<td>13 (BT)</td>
</tr>
</tbody>
</table>

### Table: Figure 10.3

<table>
<thead>
<tr>
<th>RPM</th>
<th>OUT2</th>
<th>OUT2.1</th>
<th>OUT2.2</th>
<th>OUT2.3</th>
<th>OUT2.4</th>
<th>OUT3</th>
<th>OUT3.1</th>
<th>OUT3.2</th>
<th>OUT3.3</th>
<th>OUT3.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1147 (RT)</td>
<td>216 (BT)</td>
<td>85 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
<td>134 (BT)</td>
</tr>
</tbody>
</table>

---

38
Figure 10.4: 5V sawtooth input (top) tuning module output (bottom)
void spi_init(void) {

    // Enable SPI peripheral clock
    sysclk_enable_peripheral_clock(SPI);

    /* Init SS and SPI MOSI, SCK pins as outputs, MISO as input */
    PORTC.DIRSET = IN0_SS_bm | 
                  IN1_SS_bm | 
                  SPI_DAC_MOSI_bm | // SPI out
                  SPI_DAC_SCK_bm; // SPI clock
    PORTC.DIRCLR = SPI_ADC_MISO_bm; // Data in (for ADC)

    PORTB.DIRSET = LDACIN0_bm | // LDAC for Input 0
                   LDACIN1_bm; // LDAC for Input 1
    PORTE.DIRSET = OUT0_SS_bm | // Output 0 ADC
                   OUT1_SS_bm; // Output 1 ADC

    /* Wired and pull up SS outputs */
    PORTC.PIN4CTRL = PORTC.PIN3CTRL = PORTC_PIN2CTRL = PORTC_PIN1CTRL = PORTC_PIN0CTRL = PORTC_PINCTRL_bm; // IN0
    PORTC.PIN0CTRL = PORTC_PIN1CTRL = PORTC_PIN2CTRL = PORTC_PIN3CTRL = PORTC_PIN4CTRL = PORTC_PINCTRL_bm; // IN1
    PORTE.PIN0CTRL = PORTE_PIN1CTRL = PORTE_PIN2CTRL = PORTE_PIN3CTRL = PORTE_PIN4CTRL = PORTE_PINCTRL_bm; // OUT0
    PORTE.PIN1CTRL = PORTE_PIN2CTRL = PORTE_PIN3CTRL = PORTE_PIN4CTRL = PORTE_PINCTRL_bm; // OUT1

    /* Set SS & LDAC outputs to high. (active low operation) */
    PORTC.OUTSET = IN0_SS_bm |
                   IN1_SS_bm;
    PORTB.OUTSET = LDACIN0_bm |
                   LDACIN1_bm;
    PORTE.OUTSET = OUT0_SS_bm |
                   OUT1_SS_bm;

    SPIC.INTCTRL = SPIC_INT_LVL_LO_gc; // set SPI interrupt level
    SPIC_CTRL = SPI_PRESCALER_DIV4_gc | // SPI prescaler
               SPI_ENABLE_bm | // SPI enable
               SPI_MASTER_bm | // set as SPI master
               SPI_MODE_0_gc; // set SPI mode
}

Figure 10.5: SPI initialization
void usart_init(void) {

    /******************************************************************************
    /* FTDI USART
    /******************************************************************************

    // set USART settings
    static usart_hs232_options_t USART_FTDI_OPTIONS = {
        .baudrate = USART_FTDI_BAUDRATE,
        .charlength = USART_FTDI_CHAR_LENGTH,
        .paritytype = USART_FTDI_PARITY,
        .stopbits = USART_FTDI_STOP_BIT
    };

    // initialize USART in rs232 mode
    usart_init_rs232(USART_FTDI, &USART_FTDI_OPTIONS);

    // enable USART receive interrupts to LOW
    usart_set_rx_interrupt_level(USART_FTDI, USART_INT_LVL_LO);

    // disable USART transmit interrupts
    usart_set_tx_interrupt_level(USART_FTDI, USART_INT_LVL_OFF);

    // disable USART data reg interrupts
    usart_set_dre_interrupt_level(USART_FTDI, USART_INT_LVL_OFF);

    /******************************************************************************
    /* BLUETOOTH USART
    /******************************************************************************

    static usart_hs232_options_t USART_BT_OPTIONS = {
        .baudrate = USART_BT_BAUDRATE,
        .charlength = USART_BT_CHAR_LENGTH,
        .paritytype = USART_BT_PARITY,
        .stopbits = USART_BT_STOP_BIT
    };

    // initialize USART in rs232 mode
    usart_init_rs232(USART_BT, &USART_BT_OPTIONS);

    // enable USART receive interrupts to LOW
    usart_set_rx_interrupt_level(USART_BT, USART_INT_LVL_LO);

    // disable USART transmit interrupts
    usart_set_tx_interrupt_level(USART_BT, USART_INT_LVL_OFF);

    // disable USART data reg interrupts
    usart_set_dre_interrupt_level(USART_BT, USART_INT_LVL_OFF);
}

Figure 10.6: USART initialization
void tc_init(void) {

    // Enable TCC0 peripheral clock
    sysclk_enable_peripheral_clock(&TCC0);

    // Set RPM_PIN (PCI) to an output
    PORTC.DIRSET = RPM_PIN;

    // Set RPM_PIN initial output to LOW
    PORTC.OUTCLR = RPM_PIN;

    // Enable RPM timer counter (TCC0)
    tc_enable(RPM_TIMER);

    // Set CCA interrupt callback function
    tc_set_cca_interrupt_callback(RPM_TIMER, tc_cca_rpm_callback);

    // Set OVF interrupt callback function
    tc_set_overflow_interrupt_callback(RPM_TIMER, tc_ovf_rpm_callback);

    // Set wave generation mode to NORMAL
    tc_set_wgm(RPM_TIMER, TC_WGM_NORMAL);

    // Set channel compare
    /*
     * Tweak this value to achieve 1ms wide spike
     */
    tc_write_cc(RPM_TIMER, TC_CCA, 500);

    // Init RPM to 750 (idle)
    tc_write_period(RPM_TIMER, rpm_to_clkper(START_TEST_RPM));

    // Enable Compare Channel A on RPM Timer (TCC0)
    tc_enable_cc_channels(RPM_TIMER, TC_CCAEN);

    // Set interrupt level of Compare Channel A
    tc_set_cca_interrupt_level(RPM_TIMER, TC_INT_LVL_MED);

    // Set interrupt level of TCC0 overflow
    tc_set_overflow_interrupt_level(RPM_TIMER, TC_INT_LVL_MED);

    // Sets clock div factor and starts timer
    tc_write_clock_source(RPM_TIMER, TC_CLKSEL_DIV4_GC);
}

Figure 10.7: Timer Counter initialization
```c
int main (void)
{
    board_init();
    sysclk_init();
    spi_init();
    uart_init();
    tc_init();

    // enable all interrupt levels from interrupt vector table
    irq_initialize_vectors();

    // enable interrupts
    cpu_irq_enable();

    while (1) {
    }
}
```

Figure 10.8: main()