PPS Detection Circuit: An MSP430 Based Hardware Design

Christopher Daniel Taylor

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Recommended Citation
https://louis.uah.edu/honors-capstones/612

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
PPS Detection Circuit: An MSP430 Based Hardware Design

by

Christopher Daniel Taylor

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

November 25, 2019

Honors Capstone Director: Dr. Aleksandar Milenkovic

Professor, Electrical and Computer Engineering Department

Student

Date

11/26/19

Director

Date

11/26/19

Department Chair

Date

11/26/19

Honors College Dean

Date
Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Christopher Daniel Taylor

Student Name (printed)

[Signature]

Student Signature

11/26/19

Date
PPS Detection Circuit: An MSP430 Based Hardware Design

Christopher Taylor, Student

Abstract—Sponsored by ADTRAN. This project served to create a custom circuit board to aide in automated testing of the pulse-per-second functionality of fiber optic products. The final result was a robust and flexible embedded platform with a full suite of timing verification features to meet the needs of test engineers. The design utilizes several functional blocks of the MSP430 microcontroller, external crystal oscillator, custom power supply, RS232 serial communication, USB support, and a variety of feedback and debug related features. The goal of this report is to describe the hardware and software design process and demonstrate the functionality and performance of this custom hardware solution.

I. INTRODUCTION

Upon entering my third co-op term with ADTRAN, I was placed on Hardware Team 4. While this was a departure from my past experiences in software development and testing, it would prove to be a massive learning opportunity. While my team had a variety of tasks I could assist with, I was assigned an independent project that would become my main focus for the term: the PPS (Pulse-per-second) Detection Circuit. The general goal of this project was to create a custom board which can be delivered to ATP (Automated Testing) and integrated into a rack which will test OLTs (Optical Line Terminals) as they are manufactured. In their current test setups, there was no infrastructure the test the dedicated PPS port on OLT and they were instead relying on observing high and low voltages via a multimeter at roughly the right frequency. In this sense, my final product only had to be better than nothing and had a lot of freedom in terms of what metrics I would verify and what features I would include.

This report serves to both present my experience and results from this project as well as give the reader insight into what a modern-day hardware development project might look like. To start, section 2 will discuss the research and planning stages which included meeting with all relevant parties and making top-level design decisions. Section 3 will describe my prototyping phase where hardware and software were experimented on in parallel and functionality was verified. Section 4 explores the hardware development process at the schematic capture level. Section 5 describes development once the hardware is delivered and any final integration steps. Lastly, my results and a brief conclusion can be found in section 7.

II. RESEARCH AND PLANNING

The first step in any hardware project is to identify and understand the problem you are wanting to solve. This stage included researching both the ITU-T G.703 pulse-per-second standard and how the PPS signal is being generated on our products currently. For the former, I referenced both the international standard and implementations on other telecommunication products. While there was variation in different sources for the exact implementation of the PPS signal, I concluded that the following characteristics had to be met:

| Interface is balanced 100Ω differential signal |
| Frequency of 1.0Hz/ Period of 1.0s |
| Period tolerance of ±10ns |
| Positive pulse width ranging from 100ns to 500ms |
| Rise time (10%-90%) of less than 5ns |

Table 1: PPS Signal Characteristics

In addition, my team recommended some simulation effort to both get experience with circuit simulation tools as well as gain a better understanding of the driving circuitry behind the PPS port. To do this, I recreated a section of the OLT schematic which drove the PPS signal in HyperLynx (Figure 1), and simulated both as shown as well as experimenting with fewer buffers and different termination methods. While this did not directly influence the design of my circuit, I believe that being able to simulate and verify signal integrity on a conceptual level increased my understanding of the problem and allowed me to be more effective as a developer. Included are a few plots from this effort showing best and worst case rise times during my simulations.

Figure 1: HyperLynx Driver Circuit
From here, I then met with my customers at ATP to define functionality and general interface requirements for my board. This included a brief presentation about my research and reviewing the characteristics seen in Table I to decide what we want to verify and how that affects the hardware requirements. One main topic of discussion was whether or not high-speed ADC sampling would be required to model rise times and verify the integrity of the signal to a finer level. While this is doable given a high-speed ADC peripheral with internal buffering (of which candidate parts were found), it was omitted since that level of verification is more suited for DVT (design verification testing, done by developers at the end of development) as opposed to ATP where the priority is simply detecting manufacturing defects.

The other main goal of this meeting to best match the interface and features of the board to the rest of the testing infrastructure. Since this was a custom hardware project, we had the luxury to plan exactly what they wanted to connect to and how they would connect to it. Based on their feedback, I planned to include both a serial connection port and drive lines to interface with their relay driver for simple test execution. These lines would allow their relay driver to reset the device, send a start signal, wait until the board asserts the ready signal, and then poll the result for the basic pass/fail case. While more functionality would be defined later, we now had a basis to begin development.

Before development could begin, a target processor had to be selected. At this stage, we were already decided that it would use an MSP430 microcontroller due to a good foundation of existing designs, its low-cost and low-power architecture, and my personal level of experience and confidence from my embedded systems course. The next step was deciding which model of MSP would best suit our needs. The MSP430F5xxx lineup was chosen since they have the fastest supported clocking at 25MHz to get the most accurate timing results. Within this lineup, two specific chips were being used in other designs with both having sufficient in-house inventory. The 5308 was a lower cost model with less flash memory and no USB support whereas the 5528 had much larger flash, more IO pins, and full USB support. Even though I did not need the specific benefits of the 5528, we selected it since cost was not a major concern in such a limited run and it would allow for flexibility if the project were to be continued or expanded at a later date.

To actually begin development, an evaluation board was used (MSP-TS430RGC64USB) which allowed me to socket a MSP430F5528, develop and test code from TI Code Composer, and interface GPIO pins to connect to prototyped hardware. With this setup, my immediate goal was to identify and utilize functional blocks on the MSP430 (seen in figure 3) to verify some amount of functionality discussed in the planning stages. During this time, I implemented code that could measure period and width using timer A, read a signal’s voltage level via ADC channel A0, manage the UCS (Universal Clock System) to instantiate a 4 MHz external crystal and assign it to SMCLK, and interface GPIO pins.
As software was developed for the MSP to demonstrate levels of functionality, I brought in my own breadboard and circuit components to get an idea of what all would interface the microcontroller core. This allowed me to route a signal into the device's GPIO pins to both trigger timer A and get timing data, as well as sample voltage levels through the ADC. In addition, LEDs were added to show outputs visually and a switch was set up to trigger the start line. This meant that I could run tests and verify functionality in the exact way the test engineers hoped to use my final product. This setup proved extremely helpful since I could easily reroute, change pins, or add components as my software matured and I added more features.

As my hardware and software developed, I was able to define what all features would be included in my final design. Specifically, I found out what aspects of the PPS I would go on to verify and the software and hardware additions would be necessary to make that happen. It was also at this point that I refined my acceptance criteria to be the following:

- Provide pass signal on output line upon successful pps signal capture
- Use start/stop and ready lines as requested by ATP
- Allow system to be reset using RST line
- Verify the signal has the following characteristics:
  - Has a period of 1 second
  - Reaches acceptable high and low voltages
  - Has an acceptable pulse width
- Communicate additional info (like fail condition) through serial interface
- Do not exceed size or power requirements

By the end of this stage, I had a working prototype that was demonstrated to my team and ATP to show what levels amounts of functionality I was able to achieve. This was one of the most satisfying parts of the entire project for me since I was able to implement a lot of functionality fairly quickly. This time was also crucial in giving me hope that I simply had to have a hardware design that was compatible with my prototype code and connections instead of creating something new from scratch that may or may not work. During this final demonstration, my PPS input was made using a signal generator so I could modify it and see my setup identify different fail conditions. One limitation of my prototype was the lack of a serial interface. Since this evaluation board did not have a serial connector or driving circuitry, I was getting data from debugging features and printfs to Code Composer. The following figures show both the signal generator setup and my results from my functional demonstration.
IV. HARDWARE DEVELOPMENT

While hardware development was happening in parallel with the prototyping phase, it was mainly after a proof of concept was made that I could create a schematic with confidence that it will meet my functional requirements. The goal of hardware development is to take the functionality and parts used during prototyping and select specific components and connections that will be used in the final product. This also includes making sure that all connections to the MSP430 are within spec and implemented correctly. To start, I used another product's schematic that used the same microcontroller and copied the majority of its power, usb, and other standard connections. For my specific functionality, I organized all the connections and components I would need into a spreadsheet and began to identify usable components from DxDatabook and build my schematic capture. For the remainder of this section, I will examine different features and where they can be seen in the schematic itself.

![Figure 6: Schematic Capture: Power Delivery](image)

In the following figure, you can see the power delivery and reset mechanism for the circuit. As I mentioned, most of this was replicated from other standard designs and cross-referenced with MSP430 literature and the evaluation boards schematic to ensure compatibility. Personal additions made include the drop-in DCO controller in the top-left to act as a 5 to 3.3V power supply. This is rated for up to 300mA which was more than sufficient since the MSP itself consumes on the order of micro-amps and additional peripherals and LEDs would not add much. You can also see the reset circuitry in this section, which allows the MSP430 to be rebooted either by asserting the reset drive line or pressing the reset button. Lastly, a MAX803 microprocessor reset monitor was added to verify stable voltage levels before applying any power the MSP itself.

![Figure 7: Schematic Capture: Drive Lines and LEDs](image)

The next figure shows the first few sets of GPIO pins on the MSP430. The connections seen include a dedicated power LED, signal input for timer capture, drive lines, pass/fail LEDs, and debugging LEDs. Each LED is simply positively driven by the 3.3V supplied when a pin is set to output. Given a 3.3V output, I subtracted the voltage absorbed by the LED and found a good resistor that would result in the current needed for optimal color and brightness. For the signal input, you can see the 50Ω to ground termination which results in the balanced 100Ω differential signal. I also added a zener for protection that can short to ground if the voltage reaches a level that would damage the MSP. Lastly, there are a variety of 0Ω placed as optional filtering points in case any hardware modifications were needed after manufacturing.

![Figure 8: Schematic Capture: External Clock and ADC Channels](image)

Further down the MSP, you will find the 25Mhz external crystal oscillator and ADC channel connections. The crystal used was selected for its cost, availability, and accuracy with the capacitors around it simply defined within its documentation. This was connected to the XT2 pins in the same way that the 4Mhz external crystal was used in the evaluation board during prototyping. For the 12-bit ADC connections, the PPS signal is scaled using voltage division to half its original value (to 1.65V) so that it can be compared to a 2.5V ADC reference without saturating. The equation below demonstrates how I calculated what value to expect in software from the ADC. There are also inputs to monitor voltage levels coming from the power supply even though it did not become necessary to verify in software.

\[
\frac{V_{\text{meas}}}{V_{\text{ref}}} \cdot 2^{12} = \frac{3.3(1/2)}{2.5} \cdot 4096 = 2703
\]
The two sections shown above are different interfaces for the MSP. First, the USB interface and connector is standardized with protection fusing and capacitors borrowed from other designs like the evaluation board. Next, the JTAG connector was directly hooked in to its appropriate pins based on MSP430 documentation and also verified against the evaluation board. Lastly, the serial connection also required a driving circuit to be compatible with the RS232 standard which was sent through a DB9 connector at ATP's request. This was done with a TI MAX3232 with connections copied from other designs that utilized this style of serial interface. Since we are only implementing a simple master/slave UART connection, there was no need to connect control lines like RTS, CTS, DSR, DTR, etc.

Another major part of the hardware design process was getting feedback from other engineers on my team as well as the customer. Two schematic reviews were held in which I walked through my progress and got both technical feedbacks from people on my team and the end users in testing. This was extremely helpful in both keeping everyone on the same page about the status of the project, and ensuring the end result was high quality and met the expectations of the customer. This also served as a chance to add any last second functionality before getting the boards built. One example of this was the later addition of the power LED, which was easy to implement and added some piece of mind for the end user if they needed to troubleshoot. Once the schematic was finalized, it was time to turn it into a physical device.

V. PRODUCT REALIZATION

Now that the schematic capture is completed, it was time to shift focus to a physical design and layout in order to create a board design that can be ordered and built. This meant that I had to rely on engineers in different departments and ensure that their work aligned with the goals of the project. Specifically, I was assigned a mechanical engineer to help create a board outline and physical model. This model and my schematic were then submitted to the layout queue where I was assigned a layout engineer who routed my connections and components onto a pcb. Lastly, my final result post-layout would be reviewed by a variety of departments like safety and compliance before being approved to begin the ordering process. As you might imagine, this process took quite some time since the people I was working with had other priorities and some modifications had to be made as more people had a chance to give feedback about my design.

The first step was to collaborate with a mechanical engineer to get a physical design that layout could use to implement my schematic. From early on, ATP had recommended a few plastic enclosures that could be used for sizing requirements (even though they did not need the final product to be delivered in an enclosure). I simply provided the specifications of the enclosure and the rough location of ports and let layout create a workable model. Since there wasn't very strict form factor requirements, I was very forgiving with the location of components as long as everything was accessible and workable from testing's perspective. Mechanical also had to define and show specific tolerances and clearances in their drawing such that layout would not have any invalid routing or placement conflicts. The following figures show my initial approximation followed by the final mechanical design.

![Initial Physical Design](image)

![Final Physical Design](image)
Once the mechanical drawing was complete, I had to submit a PCB queue request form to get assigned a layout engineer who could map my schematic to the physical model. Thankfully, my layout engineer was also very experienced in constraint managing which I had not yet completed before submitting my form. Constraints exist in both the schematic and layout phase and define the width, clearances, and general characteristics of nets that must be maintained in the final product. Since there were no high-speed or sensitive nets, we just spent an afternoon setting up standard constraints that would keep us out of trouble. Since this was a new product, we also had to decide what PCB stack up would be used. This could be a very large design decision for complex hardware projects but my simple board just used a standard 4-layer stack with a ground plane on layer 3. My layout engineer was also an excellent resource in terms of reviewing and giving feedback on my schematic that resulted in a number of improvements and fixes. These changes included fixing some resistor part choices to be surface mount, verifying part numbers for headers used in the mechanical drawing, and fixing a typo for a specific net. By the end of this process, we had a completed layout that could be sent in for review and eventually ordered.

Figure 14: Layout Design

Once layout was completed, a final review was done before moving forward with ordering. First, the end users at testing were brought in to do a layout review and get feedback on the final design. While they were happy with the look and location of board components, we did decide to add some additional silk-screening text to label headers and also define the debug LED states. Next, the layout and schematic were sent out for a final check with other mechanical, layout, compliance, and safety engineers. This is done so that another set of eyes besides the original designers can check for errors and give their approval before any money is spent ordering and building it. This was all done over email and no one who was included in the review had any new concerns to voice. Once this was finished, the appropriate output files were generated for the design and it was time to order.

The first step was to take the schematic and export its BOM (Bill of Materials) so that a work order could gather the required parts in time for assembly. From here, a work order was submitted to an engineering planner who would run shortage reports for parts used and pull the required kit. With a work order submitted, I was then able to submit a PCB order request which sent an order through an overseas board-house that could manufacture the 4-layer stack up with all required traces and pads. While we were waiting for the boards themselves, a shortage report revealed that two parts used were "status 49," meaning that they are labeled obsolete and not allowed for new designs. One of these was a 50Ω resistor which was easily swapped for a 49.99Ω resistor of a valid status and stock. The other, however, was the DB9 connector which was planned to be used as a serial connector. Since there was not an equivalent part available in the database, we decided to no-load it in the design, source the parts directly from testing, and attach them ourselves manually. From here, it was only a matter of waiting about a week for the PCBs to arrive, then another week for them to get built on an in-house assembly line.

VI. INTEGRATION

At this point, boards have arrived, and it was time to see if anything actually works. The first step was to finish assembly by attaching any parts that were left as no load during assembly. This was done for both the DB9 connector and a few 0Ω resistors that were left for optional modifications. Thankfully, my office was located right next to our lab technician who was very helpful in any modifications I needed to make to my boards. After this, I used a multimeter to check for shorts between signal and power connections. This was just done to verify connections were generally correct before attaching a power supply. Next, the boards were powered up and I attempted to load a basic LED blinking code via the JTAG interface. Thankfully, it flashed the code successfully and I was able to use the MSP to blink LEDs. At this point, I was confident that there were no major problems and I could work to integrate my prototype software into the final product.

Figure 15: Finished Board

Since the GPIO pins were organized ahead of time and used correctly in the schematic, flashing my experimental code produced good results after very little modifications. This meant that I was very quickly able to use my new boards to get the results seen in Table 3 in terms of signal verification. The next step was verifying all additional functionality over the prototype. Specifically, this meant checking if I could get a serial link up and interact with my software via a command line through a serial cable to my pc. This led to some problems since I could not immediately print through the serial interface as I was initially hoping. To help debug this issue, I actually ended up checking my transmit and receive pins with an oscilloscope to check if there were actually outputting what I expected based on the UART standard. I will not get in to too much detail here, but UART is a character oriented serial standard where each message is 10 bits including a start/stop and 8 bits to define the character. I first noticed that I could capture a character being sent, but everywhere I expected to be
a '0' was at 5V and each expected '1' was -5V. This concerned my until I spoke with my mentor who explained the RS-232 standard for transmission actually uses the positive voltage to represent a '0' and negative voltage to represent a '1'.

In reality, my issue was that my schematic attached the transmit pin of my board's MAX3232 integrated driver to the transmit pin of the DB9 connector standard (and receive to receive, naturally). I was under the assumptions that the cable itself would properly connect the two devices so that they would transmit and receive to each other instead of directly mapping each pin to the opposite device (meaning TX to TX and RX to RX). Since this was the case, and my device was considered a slave to the computer, I should have swapped the connections. This was easily verified using a "null modem" cable which does swap TX and RX pins from one end to the other for this exact type of situation. With the connections swapped, I was able to verify my serial communication code and transmit/receive data from a serial connection on the computer. The following figure shows how this pin swap was accomplished using hardware modifications so that users could use any generic serial interface cable.

![DB9 Connector Hardware Modification](image)

The next step for integration was to verify any additional components. Even though we did not have any use for the USB interface, it was included in the schematic for future expandability. While I did not have the time to create prototype code that would transfer data through the USB connector, I did connect a cable and verified that the board could be powered via USB if desired. This also meant that my USB circuit with fusing and protection must be at least somewhat functional. Full testing would include inputting the extremes of allowed voltages for the USB standard and ensuring that the interface does not get damaged. This was also done for the programming interface which acts like a simplified JTAG connector that only allows for flashing code without any debug features. This was done successfully, and I was able to load code using just this connector in case the user did not have a JTAG connector. At the end of this stage, the hardware and software were effectively completed and working. The next steps would be finalizing my software package, creating documentation for users and future developers of the board, and delivering the product to testing.

VII. RESULTS AND CONCLUSION

Thankfully, the end result of this project was a functioning product that met or exceeded initial expectations by ATP and fulfilled all the acceptance criteria I set for myself. To finish development, I packaged my final code and uploaded it to an internal GitHub repository so it could be maintained by future developers. I then defined hardware and software test procedures so any users of the board could verify its functionality. I also created a basic user's guide to deliver to testing since it may get integrated into setups that are used by people who were not present during development and may not be familiar with the operation of my board. This guide is included in the appendix of this report to give a sense of my final results and the operation of the board. Lastly, the boards themselves were all assembled and tested before being handed off to testing so they could begin using them. Near the end of my term, they were actively integrating my board into test racks and modifying their procedures to include it and verify the PPS functionality of future ADTRAN products.

In conclusion, I am overall very satisfied with my final results and thankful for the opportunity to gain experience in hardware development during my co-op experience with ADTRAN. I have also been happy sharing this experience with other ADTRAN co-op students and students at UAH who have expressed interest in hardware development. I hope that this report will give insight to anyone interested in hardware development and what the process might look like. My personal favorite part of this whole process was the embedded software development and interfacing my hardware peripherals using the MSP430 platform. While I still see myself as a software developer, it was truly valuable seeing a hardware project at every level so that I had a complete understanding of my final product.

WORKS CITED

APPENDIX A: USERS GUIDE

I. Physical Overview and Hookup Guide

**DB9 Connector:** Connect with your PC serial port and use putty with the following settings in order to gain access to the serial interface.

**Micro USB:** Not fully supported in this iteration of the design. Can be used as alternate for power source but not recommended.

**Power Leads:** Used for debugging and verifying correct voltages on the board.

**Power Connector:** Use with 5V power supply (P1=5V, P5=GND). Typical power consumption can be expected to range from 20mA to 80mA depending mostly on LED usage.

**Reset Switch:** Press to immediately cut power to the MSP and reboot to the flash.

**COAX Connector:** Connect to signal generator or SDX product PPS port.

**Programming Header:** Alternative for JTAG connector, used to flash code on to the MSP430.

**JTAG Header:** Used to flash code and debug the MSP430.

**Drive Lines:** Connect to relay driver, each pin has the following functionality:

- **P1:** START/STOP, drive high to begin test
- **P2:** RDY, outputs high when test has completed, and output can be read
- **P3:** OUT, output value of test; high=pass and low=fail
P4: RST, drive high to reset the device

Debug LEDs: Three amber LEDs are used to provide additional test information in the form of the following fail codes:

Pass/FAIL LEDs: D6 appears amber on startup, green on test pass, and red on test failure.

II. Programming and Flashing

1. Install TI Code Composer Studio (http://www.ti.com/tool/CCSTUDIO). During development, version 9.0.1 was used.

2. Create/import project. This could be the PPS_Detection_Circuit project or sample code from TI. Verify project settings are configured for the MSP430F5528 as seen below:

3. Modify or add to the code. One example could be turning on the power LED (P1.0)

4. Connect power connections to power supply and either JTAG or Programming header to a TI Flash Emulation Tool (FET). While the board can get power from JTAG/Programming header, there is a chance that CCS will not recognize the device if the power supply is disconnected.

5. Use the following icons in the toolbar to initiation a debug, flash, or build respectively:
III. Running Tests

Note: The following sections “Running Tests” and “Configuring Tests” assume that you have already flashed the PPS_Detection_Circuit code onto the board.

1. Connect power connectors, coax signal, serial connector, and optional drive lines. See section I for details about these connections.

2. Power on the board. The startup state should include the following:
   a. Power LED is on and green
   b. Debug LEDs are all on and amber
   c. Pass/Fail LED is on and amber
   d. The following message was printed through the serial interface:

   ![Startup Message]

3. Entering ‘t’ from this prompt should display the following message if you have a functioning PPS signal attached:

   ![Starting Test Message]

4. If you do not want to always use the serial interface the following procedure will allow the tests to execute using the drive lines:
   a. Drive START/STOP high: The width of this signal is not important; the code is only checking for a rising edge to begin the test.
   b. Wait: During test execution, all LEDs and drive line outputs (except power LED) will be low.
   c. Check READY: Once this line is high then the result has been determined.
   d. Check OUT: After READY is set, this line defines the pass/fail status.
   e. Optional: Debug a failed test using the serial interface or debug LED fail codes.

IV. Configuring Tests

1. From startup or after a test has completed, you can enter ‘c’ on the serial connection in order to access configuration mode:

2. Select from the choices shown in order to exit configuration mode or modify a given test parameter. For this example, let’s increase the test runtime to 10 seconds. Type ‘0’.
3. At each setting, you will be shown the current value and units that it represents. In this case, test duration is stored as the number of pulses it will need to capture to pass and is currently set to 5.

4. Enter a new value and press Enter to submit it. Make sure to verify on the following text that your desired value was set correctly (there could be issues if you type the wrong thing first and attempt to re-enter a value).

5. After setting a value, you will be returned to the screen seen in step 1. You can continue to modify test parameters or exit configuration mode.

6. Verify your changes by running another test. In this case, it now capture 10 samples:

Note: If you wish to change the default values for any of the test parameters. Simply modify their declarations in main where you will also find a brief description:

```c
14/**** TEST PARAMETERS ****/
15// Clock frequency of timers used by msp430, used to define acceptable time values
16const unsigned long int CLK_FREQ = 25000000;
17// Period expected from the signal in milliseconds
18unsigned long int PERIOD = 1000;
19// Level of tolerance from exactly 1 second as a percentage of clock frequency
20unsigned int TIME_TOLERANCE = 1;
21// Min and max pulse width in milliseconds
22unsigned int PWIDTH_MIN = 100;
23unsigned int PWIDTH_MAX = 600;
24// Ideal levels returned by ADC12
25// With 2.5V reference. You get ((3.3/2)/2.5)^2 = 2703
26unsigned int LEVEL_HIGH = 2703;
27unsigned int LEVEL_LOW = 0;
28// Level of tolerance from ideal high and low measured as a percentage of max value
29unsigned int LEVEL_TOLERANCE = 5;
30// Test runtime in seconds (how many samples required to get a passing score)
31unsigned int TEST_DURATION = 5;
```
APPENDIX B: SOURCE CODE

/* main.c
 * Created on: May 17, 2019
 * Author: ctaylor
 */

// ACLK = TBCLK = 1.048Mhz, SMCLK = XT2 sourced from 25MHz crystal

#include <msp430.h>
#include <stdio.h>
#include <string.h>
#include <serial.h>

// TEST PARAMETERS

const unsigned long int CLK_FREQ = 25000000;

unsigned long int PERIOD = 10000;

unsigned int TIME_TOLERANCE = 1;

unsigned int PWIDTH_MIN = 100;
unsigned int PWIDTH_MAX = 600;

unsigned int LEVEL_HIGH = 2703;
unsigned int LEVEL_LOW = 0;

unsigned int LEVEL_TOLERANCE = 1;

unsigned int TEST_DURATION = 5;

// Actual comparison values - These get calculated at the beginning or whenever a test parameter changes
unsigned long int timeout;
unsigned long int second_high;
unsigned long int second_low;
unsigned long int pulse_high;
unsigned long int pulse_low;
unsigned int rise_level_high;
unsigned int rise_level_low;
unsigned int fall_level_high;
unsigned int fall_level_low;

// Counters used to measure timing of rising, falling edge of pulse
unsigned long int count = 0;
unsigned long int rise, fall, offset;

enum status{first_high, first_low, high, low} state;

// Used to identify state of pulse and avoid sampling edge on transition
enum errcode{none, stuck_high, stuck_low, low_freq, high_freq, short_pulse, long_pulse, bad_low_lvl, bad_high_lvl_high, bad_high_lvl_low} e = none;

// Flag to enable/disable test logic if not actively testing
int testing = 0;

// Flag which gets set if user wants to enter configuration mode
unsigned int config = 0;

// runtime test info
unsigned long int rise_times[30];
unsigned long int fall_times[30];
unsigned int high_max = 0;
unsigned int high_min = 4095;
unsigned int low_max = 0;
unsigned int low_min = 4095;

// Fixed char buffer used to store formatted messages before sending to UART transmit buffer
char msg[100];
// Incoming character used to receive commands from UART receive buffer
cchar ch;
// Used to flag if input has been received
unsigned int rx = 0;

// Helper function which calculates comparison values for testing based on test parameters
void assign_constants()
{
    timeout = ((CLK_FREQ/1000)*PERIOD*TEST_DURATION) >> 16;
    second_high = ((CLK_FREQ/1000)*PERIOD + (CLK_FREQ/100)*(TIME_TOLERANCE));
    second_low = ((CLK_FREQ/1000)*PERIOD - (CLK_FREQ/100)*(TIME_TOLERANCE));
    pulse_high = ((CLK_FREQ/1000)*PWIDTH_MAX);
    pulse_low = ((CLK_FREQ/1000)*PWIDTH_MIN);
    rise_level_high = (LEVEL_HIGH + (LEVEL_HIGH/1000)*(LEVEL_TOLERANCE));
    rise_level_low = (LEVEL_LOW - (LEVEL_LOW/1000)*(LEVEL_TOLERANCE));
    fall_level_high = (LEVEL_LOW - (LEVEL_HIGH/1000)*(LEVEL_TOLERANCE));
    fall_level_low = 0;
}

// Interactive menu using UART transmission which allows user to see and modify test parameters
void configure_test()
{
    // 'c' was entered to get here so have to clear rx flag
    rx = 0;
    int choice;
    while(1)
    {
        // Print menu for setting choices
        clear_screen();
        send_msg("*********** Settings ***********\n", 60);
        send_msg("(0) Test Duration\n(1) Signal Period\n(2) Period Tolerance\n(3) Minimum Pulse Width\n(4) Maximum Pulse Width\n(5) Low Voltage Level\n(6) High Voltage Level\n(7) Voltage Level Tolerance\n", 266);
        send_msg("Enter choice ('q' to quit): ", 28);
        while (!rx);
        rx = 0;
        if (ch == 'q')
            break;
        else if ((ch >= '0') && (ch <= '7'))
            choice = ch - '0';
        else
        {
            send_msg("Invalid choice...
", 24);
            continue;
        }
        UCALIE = ~UCRXIE; // Disable USCI_A0 RX interrupt
        // For each menu choice, print the same style of prompt including current value, input to set new value, and confirmation
        switch (choice)
        {
        case 0:
            sprintf(msg, "Current test duration (num pulses): %2u\n", TEST_DURATION);
            send_msg(msg, 40);
            send_msg("Enter new test duration: ", 26);
            TEST_DURATION = read_uint();
            sprintf(msg, "Test duration set: %2u\n", TEST_DURATION);
            send_msg(msg, 4);
            break;
        case 1:
            sprintf(msg, "Current signal period (ms): %4lu\n", PERIOD);
            send_msg(msg, 38);
            send_msg("Enter new signal period: ", 27);
            PERIOD = read_uint();
            sprintf(msg, "Signal period set: %4lu\n", PERIOD);
            send_msg(msg, 3);
            break;
        case 2:
            sprintf(msg, "Current period tolerance (%): %2u\n", TIME_TOLERANCE);
            send_msg(msg, 32);
            send_msg("Enter new period tolerance: ", 26);
            TIME_TOLERANCE = read_uint();
            break;
        }
```c
case 3:
    sprintf(msg, "%\n\rCurrent minimum pulse width (ms): %3u\n\r", PWIDTH_MIN);
    send_msg(msg, 0);
    break;
    sprintf(msg, "Enter new minimum pulse width: ", 3);
    PWIDTH_MIN = read_uint();
    sprintf(msg, "%\n\rMinimum pulse width set: %3u\n\r", PWIDTH_MIN);
    send_msg(msg, 0);
    break;

case 4:
    sprintf(msg, "%\n\rCurrent maximum pulse width (ms): %3u\n\r", PWIDTH_MAX);
    send_msg(msg, 0);
    break;
    sprintf(msg, "Enter new maximum pulse width: ", 3);
    PWIDTH_MAX = read_uint();
    sprintf(msg, "%\n\rMaximum pulse width set: %3u\n\r", PWIDTH_MAX);
    send_msg(msg, 0);
    break;

case 5:
    LEVEL_LOW = LEVEL_LOW;
    send_msg(msg, 15);
    send_msg("Enter new low voltage level: ", 29);
    LEVEL_LOW = read_uint();
    sprintf(msg, "%\n\rLow voltage level set: %4u\n\r", LEVEL_LOW);
    send_msg(msg, 0);
    break;

case 6:
    LEVEL_HIGH = LEVEL_HIGH;
    send_msg(msg, 15);
    send_msg("Enter new high voltage level: ", 29);
    LEVEL_HIGH = read_uint();
    sprintf(msg, "%\n\rHigh voltage level set: %4u\n\r", LEVEL_HIGH);
    send_msg(msg, 0);
    break;

case 7:
    LEVEL_TOLERANCE = LEVEL_TOLERANCE;
    send_msg(msg, 15);
    send_msg("Enter new voltage level tolerance (%): ", 29);
    LEVEL_TOLERANCE = read_uint();
    sprintf(msg, "%\n\rVoltage level tolerance set: %2u\n\r", LEVEL_TOLERANCE);
    send_msg(msg, 0);
    break;

default:
    send_msg("Invalid choice!\n\r", 17);
    break;
}
```

This code snippet appears to be part of a program that sets and reads various system parameters such as pulse width, voltage levels, and tolerance. It includes functions to send messages to the user for input and to display the selected settings. The comments suggest that the program is designed to test various conditions, possibly related to a hardware interface, and provides a simple interface for setting different configurations.
// Take flags and runtime test info to output results. This includes pins/LEDs as well as UART messages.
void output_results()
{
    testing = 0;
P2OUT |= -BIT2;
    char result[30];
    int length;
    switch (e)
    {
        case none:
            strcpy(result, "Test passed!\n\r");
            length = 14;
P2OUT |= BIT2;
P3OUT = 0x01;
            break;
        case stuck_high:
            strcpy(result, "Stuck high!\n\r");
            length = 13;
P3OUT = 0x1E;
            break;
        case stuck_low:
            strcpy(result, "Stuck low!\n\r");
            length = 12;
P3OUT = 0x02;
            break;
        case low_freq:
            strcpy(result, "Frequency too low!\n\r");
            length = 20;
P3OUT = 0x12;
            break;
        case high_freq:
            strcpy(result, "Frequency too high!\n\r");
            length = 21;
P3OUT = 0x0A;
            break;
        case short_pulse:
            strcpy(result, "Pulse too short!\n\r");
            length = 18;
P3OUT = 0x16;
            break;
        case long_pulse:
            strcpy(result, "Pulse too long!\n\r");
            length = 17;
P3OUT = 0x0E;
            break;
        case bad_low_lvl:
            strcpy(result, "Bad low voltage! \n\r");
            length = 18;
P3OUT = 0x1A;
            break;
        case bad_high_lvl_high:
            strcpy(result, "Bad high voltage! (Too high) \n\r");
            length = 30;
P3OUT = 0x0C;
            break;
        case bad_high_lvl_low:
            strcpy(result, "Bad high voltage! (Too low) \n\r");
            length = 29;
P3OUT = 0x06;
            break;
        default:
            break;
    }
    // Output has been decided, set READY bit
    P2OUT |= BIT1;

    // Debug info: period and pulse width
    send_msg("Periods (ms): ", 14);
    int i;
    for (i=0; i<TEST_DURATION; i++)
    {
        if (rise_times[i]==0)
            send_msg("0.000 ", 9);
        else
            sprintf(msg, "%4.3f ", (float)rise_times[i]/((float)CLK_FREQ/1000.0));
            send_msg(msg, 12);
    }
rise_times[i] = 0;
}
}
send_msg("Pulses (ms): ", i);
for (i=1; i<TEST_DURATION; i++)
{
  if (fall_times[i]==1)
    send_msg("0.000 ", i);
  else
    sprintf(msg, ", (float)fall_times[i]/((float)CLK_FREQ/1000.0));
    send_msg(msg, 32);
    fall_times[i] = 0;
}

// Debug info: adc levels (only print when values determined for this)
if (e != stuck_high && e != stuck_low)
{
  sprintf(msg, "Highs: (%04u, %04u) Var: %04u\n", high_min, high_max, high_max-high_min);
  send_msg(msg, 2);  
  high_max = 0;
  high_min = 4095;
  sprintf(msg, "Lows: (%04u, %04u) Var: %04u\n", low_min, low_max, low_max-low_min);
  send_msg(msg, 30);
  low_max = 0;
  low_min = 4095;
}
else
  send_msg("\n", 2);

// Results message
send_msg(result, length);

// Reset error flag for subsequent test runs
e = none;
}

Main function: Performs the following tasks:
- Set comparison values based on test parameters
- Configure all used pins directions, special op, pulldown, etc.
- Configure clock system to DCOLKDIV on ACLK and XT2 (25Mhz) on SMCLK
- Initialize uart serial communication
- Initialize timer b to drive ADC sampling at ~100Hz
- Initialize timer a multiple ccr block for pulse timing measurement
- Initialize ADC to begin sampling
- Sleep/Loop indefinitely while you wait for interrupts or user input

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD; // Stop WDT

  // This function uses global constant values for test parameters and configures specific values
  // used to compare
  assign_constants();

  // Configure pins
  P1DIR |= 0x01; // P1.0 output (power LED)
  P1OUT |= 0x01; // Turn on power LED
  P2DIR |= 0x06; // P2.1, P2.2 output (RDY and OUT signals)
  P2OUT |= 0x01; // Enable interrupts for P2.0, default edge is low-to-high (START/STOP signal)
  P1REN |= 0x02; // P1.1 pulldown enable for signal input (when not present - trends low)
  P1OUT |= 0x02; // Make sure it is in pulldown instead of pullup mode
  P1SEL |= 0x02; // P1.1 SEL used to serve as TAO input
  P5SEL |= 0x0C; // P5.2 and P5.3 SEL used to serve as XT2 input
P6SEL |= 0x01;  // P6.0 ADC option select
P3DIR |= 0x1F;  // P3.0-3.4 used for LED feedback
P3OUT = 0x1F;  // Amber LEDs for startup
P2OUT &= ~BT1;  // Drive lines start low
P2OUT &= ~BT2;

// Configure 13 MHz external clock on eval board
UCSCTL6 &= ~XT20FF;  // Turn on XT2
UCSCTL3 |= SELREF_2;  // PLLref = REF0, avoid XT1OFF flag being set
UCSCTL4 |= SELA_4;  //

do {
    // Wait for XT2 to stablize
    UCSCTL7 &= ~(XT20FFG + XT1LFOFFG + DCOFFG);
    SFRIFG &= ~OFIFG;
}

// Setup UART serial communication
serial_initialize();
serial_initialize();
send_msg("Starting up...
!");

// Use timer_b to drive adc sampling
TBCCCTL0 = CCIE;  // CCRO interrupt enabled
TBCCR0 = 0x85;  // 1048576Hz/1048576 = 100 samples/s
TBCTL = TBSEL_1 + MC_1;  // 16-bit, ACLK, UP mode

// Configure TAO in capture mode, it is bridged with pps signal
// CM3 = rising and falling edge, check in ISR which one is happening
TAOCCCTL0 = CM_3 + CCIS_0 + CAP + CCIE;
TAOCTL = TASSEL_2 + MC_2 + TACLK;  // 16-bit, SMCLK, contmode
// Configure TAI in continuous mode, keep track of running total
TAICCTL = TASSEL_2 + MC_2 + TACLK + TAIE;

// Configure ADC12 to repeatedly sample and verify P6.0 with state
REFCTL0 &= ~REFMSTR;  // Let ADC handle reference voltage generation
ADC12CTL0 = ADC12ON + ADC12SHT0_2 + ADC12REFON + ADC12REF2_5V;  // Turn on ADC12, sampling time of 16 cc, reference of 2.5V

ADC12CTL1 = ADC12SHP + ADC12SSEL_1;  // Use sampling timer, ACLK source
ADC12MCTL0 = ADC12SREF_1;  // Vr+ = VREFP and Vr-=AVSS
ADC12IE |= ADC12IE;  // Enable ADC12IFG.0
ADC12CTL0 |= ADC12ENC;  // Enable conversions
ADC12CTL0 |= ADC12SC;  // Start conversion

//configure_test();
// Enable interrupts
__bic_SR_register(GIE);  // Enter LPW0_bits, enable interrupts
send_msg("******** System Ready! ********
!
-Enter 't' to begin a test
-Enter 'c' to change test settings
-Interface using drive lines
!");

// Main loop, wake up periodically to check if user wants to configure
while(1) {
    if (config)
        configure_test();

    //__bic_SR_register(LPM0_bits + GIE);  // Enter LPM0_bits, enable interrupts
}

// Timer B0 interrupt service routine - drives sampling period for ADC
#pragma vector=TIMERB0_VECTOR
__interrupt void TIMERB0_ISR (void)
{
    // Clear IFG
    TBCCCTL0 &= ~CCIFG;
    // Start sampling/conversion
ADC12CTL0 |= ADC12SC;

// Timer A0 ISR, triggers on edges of incoming signal and captures timer values
#pragma vector TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR (void) {

  unsigned long int val;
  // Clear IFG
  TAOCCTL0 &= ~CCIFG;
  // Stop timer to fetch value, then restart
  TA0CTL &= ~MC_3;
  TAICTL &= ~MC_3;
  val = TAOCCR0;
  TA0CTL |= MC_2 + TACLR;
  TAICTL |= MC_2 + TACLR;
  // If rising edge...
  if (TA0CCINL & CCI)
  {
    rise = (0xFFF * count) + val + offset;
    count = 0;
    state = first_high;
    if (testing)
    {
      // Verify frequency
      if (runtime > 0)
      {
        rise_times[runtime-1] = rise;
        if (rise > second_high)
          e = low_freq;
        else if (rise < second_low)
          e = high_freq;
      }
      if (runtime == TEST_DURATION)
        output_results();
      runtime++;
    } // If falling edge...
    else
    {
      offset = val;
      fall = (0xFFF * count) + val;
      state = first_low;
      // Verify pulse width
      if (testing && runtime > 0)
      {
        fall_times[runtime-1] = fall;
        if (fall > pulse_high)
          e = long_pulse;
        else if (fall < pulse_low)
          e = short_pulse;
      }
    }
  }
}

// Timer A1 ISR, keeps running total for each timeout between pulses
#pragma vector TIMER1_A1_VECTOR
__interrupt void TIMER1_A1_ISR (void) {

  // Clear IFG
  TAICCTL0 &= ~CCIFG;
  // TA1IV 14th entry = overflow
  if (TA1IV == 14)
    count++;
}
// If we reach timeout then output appropriate stack signal
if (testing && count == timeout)
{
    send_msg("Timed out...
\r", 14);
    if (PIN & BIT1)
        e = stuck_high;
    else
        e = stuck_low;
    output_results();
}

// ADC12 ISR, periodically sample signal and verify it matches with current state
#pragma vector = ADC12_VECTOR
interrupt void ADC12_ISR(void)
{
    unsigned int adc_val;
    // ADC12IV 6th entry = ADC12IFG0
    if (ADC12IV == 6)
    {
        // Clear IFG
        ADC12IFG &= ~ADC12IFG0;
        adc_val = ADC12MEMO;
        switch (state)
        {
            case first_high: state = high; break;
            case first_low: state = low; break;
            case high: if (testing && runtime > 0) // Verify high voltage falls within range
            {
                // If sample exceeds current max/min then store it
                if (adc_val > high_max) high_max = adc_val;
                if (adc_val < high_min) high_min = adc_val;

                // Verify high voltage level
                if (adc_val > rise_level_high)
                    e = bad_high_lvl_high;
                else if (adc_val < rise_level_low)
                    e = bad_high_lvl_low;
            }
            break;
            case low: if (testing && runtime > 0) // Verify low voltage falls within range
            {
                // If sample exceeds current max/min then store it
                if (adc_val > low_max) low_max = adc_val;
                if (adc_val < low_min) low_min = adc_val;

                // Verify low voltage level
                if (adc_val > fall_level_high)
                    e = bad_low_lvl;
            }
            break;
            default: break;
        }

        // ADC12IV 2nd entry = overflow
        else if (ADC12IV == 2)
        {
            if (testing)
                send_msg("adc overflow\n\r", 14);
        }
    }

    // Port 2 interrupt service routine for START/STOP signal on P2.0
    #pragma vector = PORT2_VECTOR
    #pragma vector = PORT2_VECTOR
    interrupt void Port2_ISR(void)
    {
        // Debouncing delay
__delay_cycles(10000);

P2IFG &= BIT0; // Clear IFG
if (config)
{
    send_msg("\n\rTest aborted, exit configuration mode.\n\r", 4); 
    return;
}
send_msg("\nStarting test...\n\r", 10);

// Reset clocks
count = 0;
// Enable test logic
testing = 1;
// Make sure runtime starts at 0
runtime = 0;
// Turn off drive lines/LEDs
P2OUT &= -BIT1;
P2OUT &= -BIT2;
P3OUT = 0x00;
}

// USClAl interrupt service routine to receive incoming characters
#pragma vector = USCl_A1 VECTOR
__interrupt void USCl_A1 ISR(void)
{
    switch(__even_in_range(UCAlIV,4))
    {
    case 0: break; // Vector 0 - no interrupt
    case 2: ch = UCAlRXBUF;
            rx = 1;
            if (ch == 'c' && !config)
                config = 1;
            else if (ch == 't' && !config)
                P2IFG |= BIT0; //UCAlTXBUF = UCAlRXBUF;
                break;
    case 4: break; // Vector 4 - TXIFG
    default: break;
    }
}