Setting up MSP430G2553 for SPI Interfacing

Student Name: Kyle Grager
Student PID: A36628433
**Introduction**

The Controller Area Network (CAN) was developed by Bosch in 1985. With the increasing amount of electronic devices is vehicles, heavy and expensive wiring harnesses were required to have full functionality. The CAN protocol gives vehicles a light-weight solution to this problem. Instead of using dedicated wiring for all of the electronic devices, one is able to connect each device in a network and reduce wiring\(^1\).

The Michigan State University Solar Car Team was having the issue of bulky and heavy wiring for their brake-lighting system. Currently, the team does have a CAN controller for the braking system, but all of the lights are connected to a single node. This requires a lot of wire. As a solution, Ian Grosh, Head of Electrical Engineering for the Solar Car Team, decided to put a node at each blinker, brake light, and head light.

All of the nodes will be connected to a CAN bus. The CAN bus sends an asynchronous CAN data frame. The CAN transceiver collects the data frame from the bus and sends it to the CAN controller at the voltages that the CAN controller can detect. The CAN controller decides whether or not to continue with the processing by analyzing a field of the data frame called an arbitration ID. See Figure 1 for an example of a CAN data frame. If the arbitration ID matches that of the braking system, the CAN controller will then send the data field to the microcontroller.

![Figure 1: Microchip MCP2515 CAN data frame](image-url)
**Objective**

One of the steps to implementing the CAN protocol is to code the CAN controller and microcontroller. This code dictates what the CAN controller and the microcontroller does as a result of receiving the CAN input. The idea is for the CAN controller to recognize that the arbitration ID matches that of the blinker system. Should it not match, no further action is required. If the arbitration ID does match, it sends the data frame to the microcontroller. **The CAN controller and the microcontroller communicate over 4-wire SPI Serial Peripheral Interface (SPI) interface.** The microcontroller then decides what to output based on the data frame. In our case, the CAN nodes do not have to communicate with one another, so this is not a factor. There is no need to send any type of data, although it is possible. In summation, simply accept a data frame and output on/off signals to the blinkers/brake light/head lamp. Table 1 shows the nodes and the desired output of the results.

In the case of this application note, the microcontroller will send information to the CAN controller via the SPI interface upon receiving power. This is almost the same idea as uploading firmware to the CAN controller. Once the CAN controller loses power, the code must be uploaded to the CAN controller again.

**SPI Background**

SPI devices communicate over 4 wires. Each wire sends a logic signal. The logic signals are: Serial Clock (SCLK), Master Output Slave Input (MOSI), Master Input Slave Output (MISO), and Slave Select (SS). From the logic signals MOSI and MISO, one can note that the master can communicate with the slave and vice versa. In other words, they can read/write with one another. SCLK is the main clock synchronizing both devices. SS selects the current active slave. Figure 2 shows the basic setup of the connection of SPI interface.

![Figure 2: Logic signals of the SPI interface.](image)

For the sake of our application, the CAN controller is considered the slave and the microcontroller is considered the master.
MSP430 with SPI Background

The MSP430G2553 has two SPI interfaces. Figure 3 shows the pinout from the data sheet.

![Pinout of the MSP430G2553](image1)

Above, the pins that can be used for SPI interface are: P1.1 - UCA0SOMI, P1.2 - UCA0SIMO, P1.4 - UCA0CLK. These pins can be configured, which will be shown below.

Another consideration one must make is setting up the clock cycles. One can setup the clock to be active high or active low. They can also setup whether the clock transition is on the rising or falling edge. Figure 4 shows the clock cycles.

![Clock Cycles of the Universal Serial Communication port on the MSP430](image2)
Setting up SPI on the MSP430G2553 Microcontroller

Now that one knows the considerations of which they must be aware, they can set up the MSP430 using Code Composer Studio. This is a software found on TI’s website. Using this software, they code in C and upload the code to the microcontroller. According to page 17 of the user manual, the MSP430G2553 has a Universal Serial Communications Interface (USCI) that has the ability to output via SPI, I2C UART, enhanced UART, or IrDA. The USCI is supported on many pins in the microcontroller. First, one must set up the pins as inputs/outputs. In this case, the code will look as it appears in Figure 5.

```
P1OUT |= BIT5;
P1DIR |= BIT5;
P1SEL = BIT1 | BIT2 | BIT4;
P1SEL2 = BIT1 | BIT2 | BIT4;
```

Figure 5: Code to set up pin directions and communication.

Above, pin P1.5 is set up to be the SS. To work with Universal Serial Communications Interface, both P1SEL and P1SEL2 must be set to 1. P1.1, P1.2, and P1.4 are all set.

Then, one must set up the UCACNTL registers. This portion of the code will just tell the microcontroller to use SPI interface. Figure 7 is an example of the code for setting up a 3-pin, 8-bit SPI master.

```
UCA0CTL1 = UCSWRST;
UCA0CTL0 |= UCRRF + UMCEB + UMST + UCSYNC; // 3-pin, 8-bit SPI master
UCA0CTL1 |= UCSSSEL_2; // SMCLK
```

Figure 6: Setting up the UCAXCTL Register.

Above, the microcontroller is set to master by setting the UCMST bit. It is then set to send the most significant bit first by setting the UCMSB. UCSYNC is the bit that sets the Universal Communication Interface to SPI mode.

One must also set up the a source clock. To do so, they simply set the UCA0BR0 to the desired bit. For the sake of simplicity, we will set the UCSSEL_2 bit in the UCACNTL1 register. This sets the source clock to the SMCLK. Also, it is good to note here that the USCI module must have the input clock divided by at least 2.

From Figure 6, one can see that the UCSWRST bit of UCA0CTL1 is set. This puts the module is reset mode so that the operations to set up the microcontroller can run. This must
be taken off at the end of the setup. Here, one should set the UCA0CTL1 register to “not”UCSRST, or \( \sim \)UCSRST.

The microcontroller now has all of the proper registers set to communicate via the SPI interface. The next step is to actually send information.

**Communicating via SPI Interface**

In this section, as an example, a byte will be sent. Figure 7 has some example code.

```c
while (!(IFG2 & UCA0TXIFG)); // USCI_A0 TX buffer ready?
UCA0TXBUF = 0xAA; // Send 0xAA over SPI to Slave
while (!(IFG2 & UCA0RXIFG)); // USCI_A0 RX Received?
received_ch = UCA0RXEUF; // Store received data
P1OUT |= (BIT5); // Unselect Device
```

Figure 7: Example code of transmitting and receiving a byte via the SPI interface. [4]

Above, a the TX buffer and RX buffer are not yet filled. In both cases, the UCA0TXIFG and the UCA0RXIFG are flags for the transmission and reception of a signal, respectively. The while loop for each flag just waits for a transition from 0 to 1. If a flag is set, the buffer then fills with the TXBUF from above (in this case 0xAA). After the buffer fills, the SPI will generate SCLK and transfer the bits. It is possible to confirm that the slave has received all of the data by creating another flag having the slave return a signal, confirming reception.

**Conclusion**

The SPI interface what the CAN Controller uses. Since the MSP430 is capable of many uses of the Universal Serial Interface, it is an ideal candidate for this task.
References


