# M95P32 SPI Flash Chip on PSoC 5

Ian Lacy

July 2024

## 1 Overview

This tutorial shows how to configure the Cypress PSoC 5 to write to and read from a M95P32 flash memory chip using the SPI (Serial Peripheral Interface) protocol.

These sorts of memory modules are very handy if you need to store data for your PSoC to access, such as sound clips or sprites, without taking up space on the on-board memory, and without having to re-write the chip between power cycles. The M95P32 chip is a 32 Mbit flash, and runs at 3.3 V. It comes in an SMD package, so needs to be mounted on a surfboard to be used on a breadboard. Of the three flash memories we have guides for, this one is "The Big One"

## 2 Hardware

### 2.1 PSoC 5LP "Big Board"

In Figure 1 below, pins boxed in RED are used for this project. Table 1 lists the pin connections.

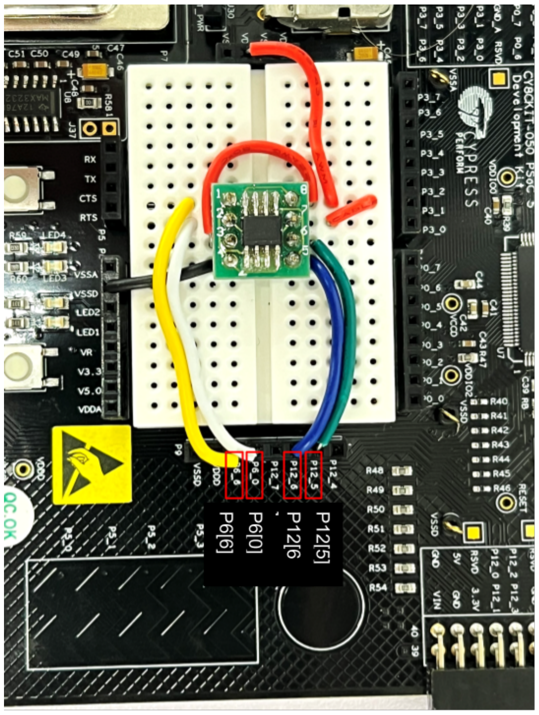


Figure 1: The Pins used in this tutorial.

| Pin | Connection |
|---|---|
| 1 | P6_6 |
| 2 | P6_0 |
| 3 | +3.3V |
| 4 | GND |
| 5 | P12_6 |
| 6 | P12_5 |
| 7 | +3.3V |
| 8 | +3.3V |

Table 1: Connections between the M95P32 and the PSoC

## 2.2 M95P32 Flash Chip

The M95P32 is a simple 32 Mbit flash chip, storing 4 megabytes of data, arranged into 512-byte pages and 4-kilobyte erasable sectors. Despite how the specifications make it sound, it can be read and written at the individual byte level. Unlike some flash chips, the M95P32 will allow you to overwrite a byte, an operation in which it will implicitly erase and program that byte.

Figure 3 below shows the pinout of the M95P32 Chip. Table 2 gives a description of the pins and their use.

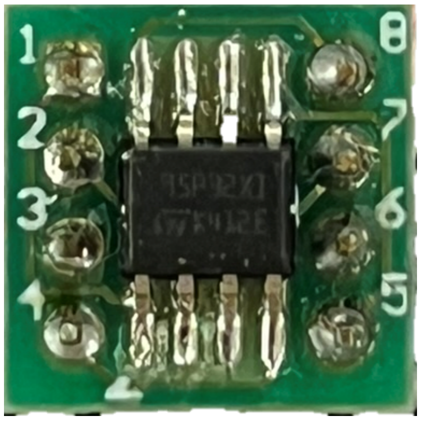


Figure 2: The M95P32 Flash Memory Used in This Example

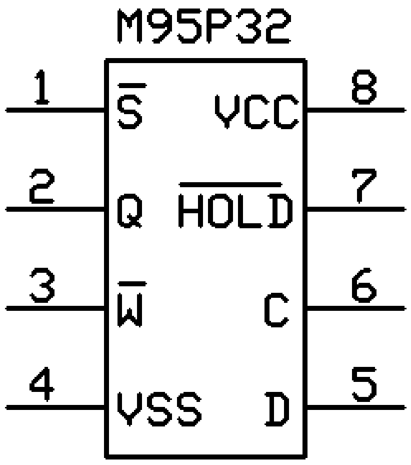


Figure 3: The Pinout/Wiring Diagram for the M95P32 Chip.

| Pin | Description |
|---|---|
| !S | Select (Active low) |
| Q | Serial Data Out |
| !W | Write Protect (Active low) |
| Vss | Ground |
| D | Serial Data In |
| C | Serial Clock |
| !HOLD | Serial hold (Active Low) |
| VCC | +3.3V |

Table 2: Pin Descriptions of the M95P32 Chip

Since this flash chip uses an SPI interface, let's first go over the SPI protocol, then we'll discuss SPI implementation on PSoC5, and finally go over the use of the the M95P32.

## 3 The SPI Protocol

The Serial Peripheral Interface (SPI) protocol, sometimes called "4-wire serial," was introduced by Motorolla in the mid-80's. It is intended as a shared-bus protocol for serial communications. It connects one master device (also called a controller) to one or more slave devices (also called peripherals.) They are connected by 4 or more wires- 3 bus wires that are shared by all devices, and an individual select line for each peripheral device. These wires are:

- SCLK- SPI Clock. This line controls the timing of the bus.
- MOSI- Master Out Slave In. This line carries data from the master to the peripheral.
- MISO- Master In Slave Out. This line carries data from the peripheral to the master.
- SS- Slave select. Active low. These lines controls which peripheral on the bus is activated.

Typically, these lines are driven in a 3-state strong-drive configuration. Shoot-through is protected against by each peripheral being in high-Z mode while not selected, and only one device being selected to drive the line at once. In an inactive state, the data lines are undefined, though they are often pulled low via resistor. The SCLK line is held high or low depending on the "Clock Polarity," a component of the SPI modes, which are detailed below.

The SCLK, MOSI, and MISO lines are shared by all devices on the bus, but each peripheral has its own SS line. In contrast to other serial protocols, such as $I^2C$, SPI pin usage increases with the number of devices on the bus. This drawback can be mitigated rather easily, however, with supporting hardware such as port extenders or demultiplexers.

At its most basic, SPI is implemented as an input and output shift register in each device. As the clock ticks, data is shifted out of one register and into the other. This happens in both directions at the same time, meaning each device is both transmitting and receiving at once. [1] This data flow has some implications when it comes to using SPI, which are covered later.

### 3.1 SPI Modes

SPI uses shift registers, which are simple enough, but does lead to one complication: the source shift register's output must be stable before the destination shift register's input is sampled, or the data will be compromised. The solution isn't too complicated: have the source register shift out on one direction of clock transmission, and the destination register shift in ('sample' the data) on the opposite. There's two ways to do this, which we call "Clock PHAse," or CPHA for short. It has two states:

- CPHA = 0: Data is sampled on rising edge, and shifted out on falling edge
- CPHA = 1: Data is sampled on falling edge, and shifted out on the rising edge.

[1] This is what's known as "full duplex" communication. This is opposed to "half duplex" communication, where data can go both directions, but not at the same time, or "simplex," where data can only flow one way.

As mentioned before, the resting state of SCLK is also something we have control over, which we call the "Clock POLarity", or CPOL. It has two states as well:

- CPOL = 0: Noninverted clock polarity. The clock is low in the idle state.
- CPOL = 1: Inverted clock polarity. The clock is high in the idle state.

Together, this gives us a total of four SPI modes. We typically represent this as the numbers 0-3, or the binary equivalent, with CPOL being the more significant bit. For example, mode 2 (or 1,0), is inverted clock polarity with data sampled on the rising edge. The differences in SPI mode mean that all devices on the bus must match. In most implementations, SPI master devices can change their mode as needed, but peripherals are usually locked to one mode, or, less often, can switch between two modes. Mode 0 is the most common mode.

## 3.2 The Consequences of Full-duplex Communication

The full-duplex nature of SPI gives us a couple of things to reckon with. Since data is shifted in and out at the same time, there is no difference between a read and a write– both necessarily happen simultaneously. The difference between an SPI read and an SPI write is abstract, having more to do with whether we consider the data placed on the MOSI line to be meaningful (a 'write'), or the data sampled from the MISO line (a 'read'), or both lines at once ('transceiving').

On a hardware level, SPI only has one function: transceive data. However, **some** microcontroller programming environments will abstract this detail away. The 'write' function provided will write data to the bus, and throw away the received data, while the 'read' function will send dummy data (often 0x00 or 0xFF) automatically, and return the received data from that write. **This is not the case on the PSoC 5.** This will be discussed in more detail below.

## 3.3 Signaling

Figure 4 shows an example of the timing of SPI signals.

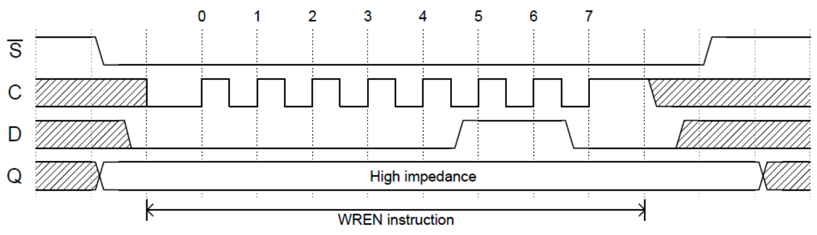


Figure 4: An Example of SPI Timing. $\overline{S}$ is SS, C is SCLK, D is MOSI, Q is MISO. See the M95P32 datasheet for more details.

In contrast to other protocols such as I$^2$C or USB, SPI doesn't particulary have 'signals' per se. Rather, there is a "Start" condition, and a "Stop" condition, and data is shifted both directions for every clock tick in between them.

Each transmission takes the following form:

- A 'Start' condition is generated by driving the select line low while the clock is in its idle state.
- Then, for each clock tick, a bit is shifted into and out of both the master and the peripheral.
- After all data is exchanged, a 'Stop' condition is generated by driving the select line back to high, which tells the peripheral that the transmission is finished.

It is extremely important to remember that the 'Stop' condition is what tells the peripheral the transmission is finished, so raising the select line essentially 'resets' the transmission. In other words, **the select line must be low for the entirety of a transmission, and must be raised at the end of a transmission.**

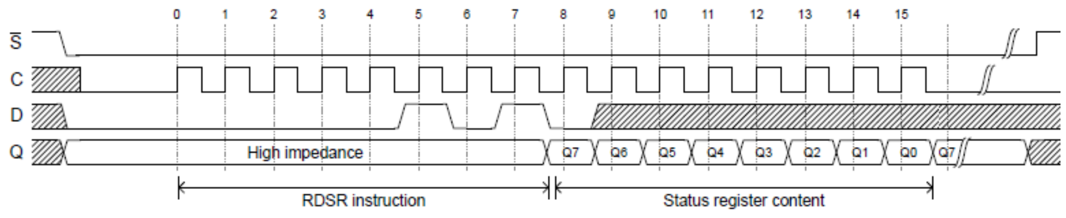


Figure 5: The Timing Diagram for the RDSR Command. $\overline{S}$ is SS, C is SCLK, D is MOSI, Q is MISO. See the M95P32 datasheet for more details.

### 3.4 Example Transmission

As an example, Figure 5 depicts the "RDSR" or "Read Status Register." The sequence is as follows:

1. The transmission begins by by lowering the select line.
2. The RDSR command byte (0x05) is shifted from the transmit buffer of the master to the receive buffer of the peripheral, on the MOSI line. This lasts 8 clock ticks.
3. Notice that dummy data, 0xFF is simultaneously shifted from the transmit buffer of the peripheral to the receive buffer of the master.
4. With the command byte received, the peripheral will now shift its status register over the MISO line. This lasts 8 clock ticks.
5. As with the command byte, notice that the master is shifting dummy data (0xAA) into the peripheral over the MOSI line. It doesn't actually matter what the data is, but it is necessarily shifted out/in, even though we are "reading" from the peripheral.

There are, of course, other forms for transmissions, and they are almost always chip-specific. For the M95P32, for example, the 'read' command consists of the command byte, a 3-byte start address, and then 1 to 4 million bytes of dummy data, during which the chip will sequentially read its memory.

## 4 SPI on PSoC

PSoC Creator 3.3 includes an SPI Master schematic macro, found in the component catalog under **Cypress > Communications > SPI**.

The **SPI Master** component, shown in Figure 6 is used for the example project. It has four terminals by default: **MOSI**, **MISO**, **SCLK**, and **SS**. For this demo, MOSI should be routed to 12[6], MISO should be routed to 6[0], and SCLK should be routed to 12[5]. SS on the SPI master should be left disconnected, with an additional pin that we'll manually control routed to 6[6].

MOSI, SCLK, and SS should be strong drive, and MISO should be high impedance digital.

The settings for the SPI Master component have options for data rate, clock polarity, and shift direction. The 'Advanced' tab has options to change the clock source and buffer size, as well as the ability to enable or disable several types of interrupts. We will be using the default buffer size of four, and no interrupts. Our data rate will be set to 100 kbps, with 8 data bits and a shift direction of MSB first.

**Be sure to enable the SPI master with the SPIM_1_Start() function before attempting to use it.**

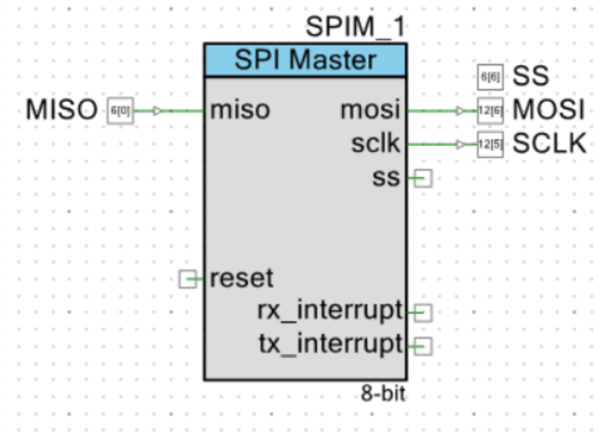
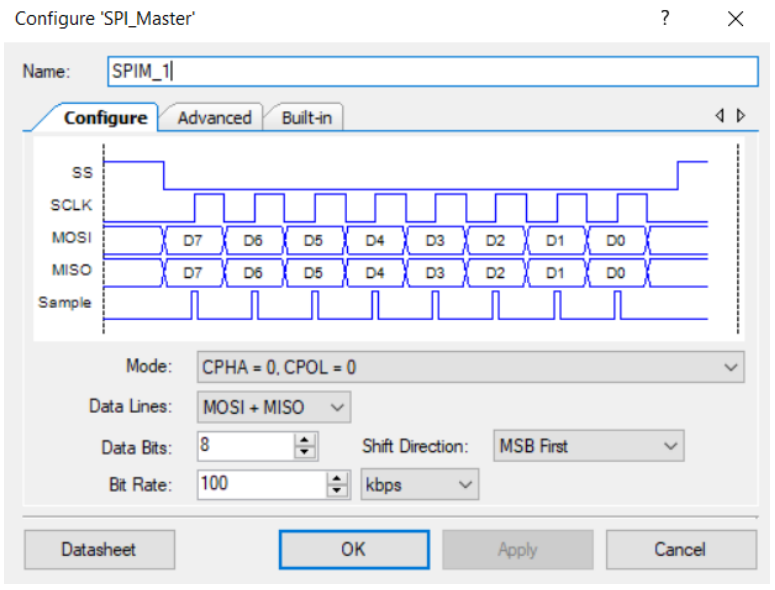


Figure 6: The SPI Master Schematic Macro Connected to three Pins Named 'MOSI', 'MISO', and 'SDA.' Notice that the SS pin is not connected to the component.



Name: SPIM_1

Mode: CPHA = 0, CPOL = 0

Data Lines: MOSI + MISO

Data Bits: 8

Shift Direction: MSB First

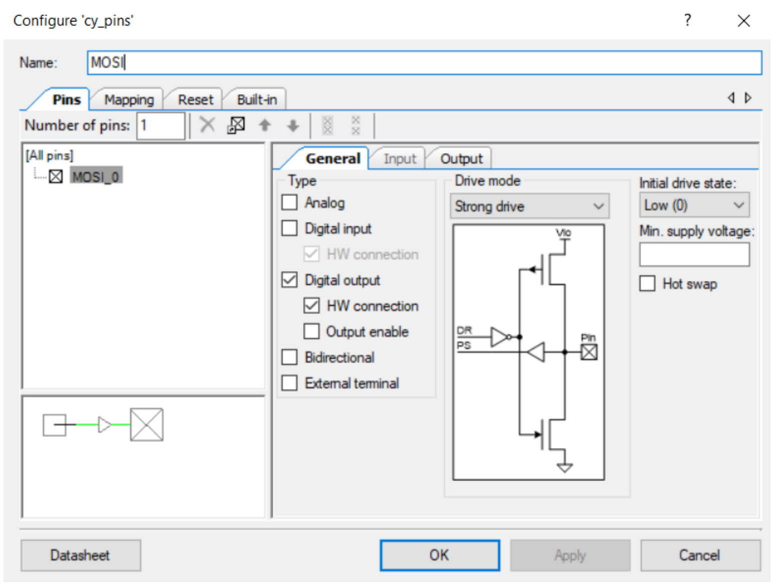
Bit Rate: 100 kbps

Figure 7: The Settings of the SPI Master Schematic Component

## 4.1 SPI Master API Functions

We will use five API functions for this project. Some of these may not seem needed at first, but their necessity will be explained. These functions are:

- SPIM_1_WriteTxData(uin8_t byte) - This function places a single byte into the transmit buffer. It blocks if there is not room in the buffer. It returns once the byte has been placed in the buffer, ***EVEN IF THE DATA HAS NOT BEEN TRANSMITTED.***
- SPIM_1_PutArray(uint8_t * array) - This function places an entire array into the transmit buffer. It blocks while the buffer is full, until the entire array has been placed in the buffer. It returns once the last byte of the array has been placed in the buffer, ***EVEN IF THE DATA HAS NOT BEEN TRANSMITTED.***
- SPIM_1_ReadTxStatus(void) - This function returns the status of the the SPI master's transmit buffer, including whether there is room in the buffer, whether the buffer is full, and whether the SPI Master is done with all transmissions.



Name: MOSI

Number of pins: 1

[All pins]
MOSI_0

Type
- [ ] Analog
- [ ] Digital input
  - [✓] HW connection
- [✓] Digital output
  - [✓] HW connection
  - [ ] Output enable
- [ ] Bidirectional
- [ ] External terminal

Drive mode: Strong drive

Initial drive state: Low (0)

Min. supply voltage:

- [ ] Hot swap

Figure 8: The Settings of the SS, MOSI, and SCLK Pins

- SPIM_1_ReadRxData(void) - This functions pulls the next byte from the FIFO receive buffer. If the buffer is empty, invalid data is returned. ***This function DOES NOT run the SPI clock. ONLY transmitted data runs the SPI clock. This is elaborated on below.***
- SPIM_1_ClearRxBuffer(void) - This function clears out the receive buffer, allowing it to read in more data.

## 4.2 Intricacies of SPI on PSoC

As alluded to, the way SPI is implemented on PSoC can be both powerful and tedious. Of primary issue is the SS control and the way the buffers work. Let's start with the hardware implementation of the SS pin, and then we'll cover the way the buffers and API interact with the serial clock and SPI Master as a whole.

### 4.2.1 SS Pin

The SPI Master on PSoC implements an SS pin in hardware, which sounds convenient (and it certainly can be!!), but it is actually a hindrance for simpler uses of the SPI Master.

Essentially, the SS line on the Master is automatically driven low when the SPI Master is transmitting. As soon as the SPI Master stops transmitting, the SS line is returned to high, marking the transmission as finished. Unfortunately for us, the SPI Master only transmits while there is data in the transmit buffer, which means if it ever ends up empty, the SS line will go high, ***Whether we're done with the transmission or not.***

With interrupts and Direct Memory Access (DMA), this can be avoided- we can simply make sure to top up the transmit buffer before it's emptied out.

Since we're not using DMA or interrupts, though, it poses a big issue, as the peripheral will think we're done with our transmission before we actually are.

We avoid this issue by manually controlling the SS pin in our firmware. That way, we can make sure all of our data has made it to the peripheral before deselecting it.

Our problems don't end here, though

### 4.2.2 Buffers and SCLK

The PSoC SPI Master has two buffers: the transmit buffer and the receive buffer. These buffers are how we interface with the SPI Master hardware, but **they are not the SPI Master hardware.**

Rather, the buffers are a mediating layer. When we place data into the transmit buffer with API functions such as “SPIM_1_WriteTxData(uint8_t byte)”, we are moving data to the intermediary buffer, and this data is then sent by the SPI Master hardware, according to the component datasheet, “at the next bus time.”

In other words, when we write to the Tx buffer, we're not sending the data, *we're queuing it to be sent when the hardware is ready.* This operation is not synchronized to the CPU, meaning we can get into a situation where we put data in the transmit buffer, and then move on to executing other code well before the data is actually sent.

This isn't necessarily a problem if we're interfacing with the SPI Master through Direct Memory Access channels or hardware interrupts, but it does complicate things when we want to use the hardware in a more rudimentary fashion. Since we're manually controlling the SS pin, if our code moves on before all of our data is sent/received, we can end up driving SS back high before the data has actually been sent. This will tell the peripheral we're done before we actually are, and corrupt the transmission.

We avoid this with that third API function- `SPIM_1_ReadTxStatus()`. If we bitwise and, `&`, the result with the built-in macro value `SPIM_1_STS_SPI_DONE`, we have a value that only evaluates as true when everything in the transmit buffer has been sent. We can combine this with a while loop to get `while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));`, a single line that will pause code execution until the entire transmit buffer has been sent through the SPI Master.

That gives us one last issue to deal with; the SPI clock.

#### 4.2.3 SCLK, the SPI Clock

As mentioned previously, SPI is a full-duplex protocol. This is especially important on PSoC. ***Every write is a read, and every read is a write. Both happen at once.***

Unlike some other microcontroller environments, such as the Atmel SAM L10 family, *PSoC does not implicitly run the clock when using the read function provided by the API.* The **ONLY** way to run the clock, and therefore to shift data in from the peripheral, is to write dummy data into the transmit buffer to be sent over SPI. This also means that when you write data over SPI, ***the receive buffer is often filled with dummy data***, in much the same way that we must send dummy data to the peripheral.

To deal with these last two problems, we write dummy data for every byte we want to receive, and also clear the receive buffer before writing our dummy data. This allows us to only read the data we care about, at whatever speed we wish.

## 5 Interfacing with the M95P32

Now that we understand the SPI protocol and how to effectively use it on the PSoC, we can go over our communications with the M95P32.

### 5.1 Commands

We'll be using 4 commands with the M95P32. Each is sent by transmitting the command byte, then (optionally) 3 address bytes, then (optionally) 1 or more data bytes, and finally, optionally, the flash responds with 1 or more response bytes.

The commands we will use are:

- WREN (0x06)- The Write Enable command. This must be sent individually each time we want to write to the chip, whether it's the memory or the write protect register. No address bytes, no data bytes, no response bytes.

- WRSR (0x01) - The Write Status Register command. This sets the write protect values in the status register. Writing them all to 0 allows writing to all parts of the memory. This command does not work if the write protect pin is pulled low. No address bytes, one data byte, no response bytes.
- WRITE (0x02) - The Write command. This allows us to write to memory. We can write sequentially, with the address automatically incrementing as we do so, but be aware that *this aliases over at multiples of 512 bytes, regardless of the starting address.* In other words, if we start writing at address 510 (in decimal), and write three bytes, we'll write at address 510, then 511, then 0, and then address 1. We will NOT write at addresses 512 or 513. Unlike some flash chips, we don't need to manually erase the entire page before writing- this chip will implicitly erase and reprogram single bytes with the write command. Three address bytes, 1 or more data bytes, no response bytes.
- READ(0x03) - The Read command. This reads sequentially from memory, starting at the address. This **DOES** increment across 512-byte boundaries, unlike the write command. Three address bytes, no data bytes, 1 or more response bytes.

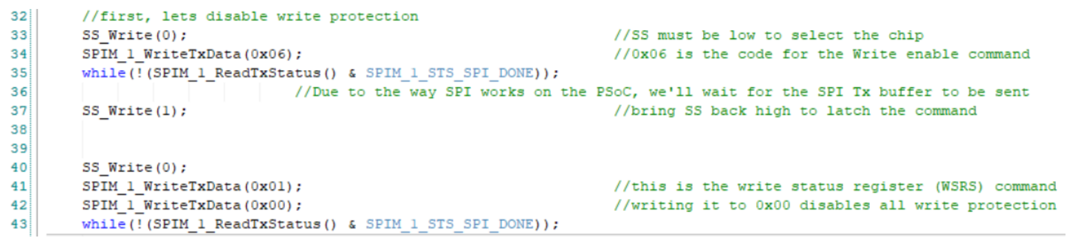
Be sure to lower SS before each command, and raise it after each command. Since WREN is a different command than WRSR or WRITE, *be sure to raise it and lower it again between the commands.*

Refer to the M95P32 datasheet for timing diagrams of each command.

## 6 The Demo

Finally, we're ready to talk about the demo code.

### 6.1 Setup



```
    //first, lets disable write protection
    SS_Write(0);                                            //SS must be low to select the chip
    SPIM_1_WriteTxData(0x06);                               //0x06 is the code for the Write enable command
    while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));
                        //Due to the way SPI works on the PSoC, we'll wait for the SPI Tx buffer to be sent
    SS_Write(1);                                            //bring SS back high to latch the command


    SS_Write(0);
    SPIM_1_WriteTxData(0x01);                               //this is the write status register (WSRS) command
    SPIM_1_WriteTxData(0x00);                               //writing it to 0x00 disables all write protection
    while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));
```

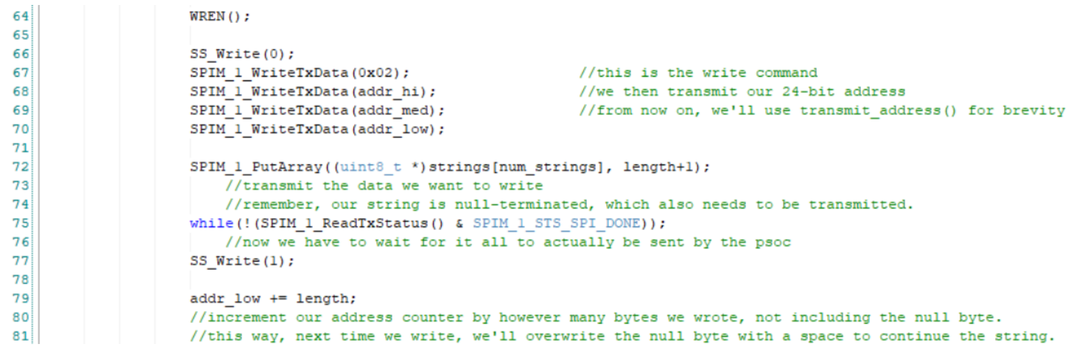
Figure 9: The Demo Code to Set Up the Chip

The first thing we want to do is disable the write protect on the chip, which is on by default after power-cycling.

We do this by sending the WREN command, which we wrap into a helper function after the first time, and then sending the WRSR command with a data byte of 0x00, turning off all write protection.

### 6.2 Writing

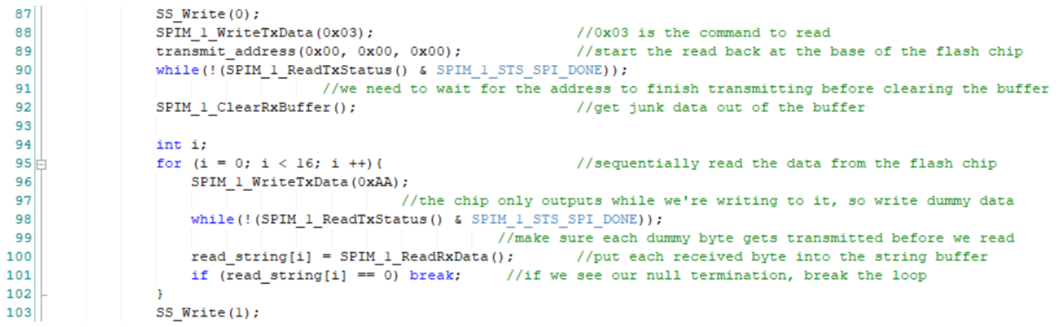
To write to the chip, we first send the WREN command, and then a WRITE command. After the command byte, we transmit the start address, and then the data of our string. Since the string automatically includes a null terminator byte, we don't need to manually send one. After writing to the chip, we increase our local address variable, to hold our place for the next segment. Since the next string segments start with space characters, we make sure to increment by the string length *not including* the null terminator, so that the terminator gets overwritten.



```
WREN();

SS_Write(0);
SPIM_1_WriteTxData(0x02);                     //this is the write command
SPIM_1_WriteTxData(addr_hi);                  //we then transmit our 24-bit address
SPIM_1_WriteTxData(addr_med);                 //from now on, we'll use transmit_address() for brevity
SPIM_1_WriteTxData(addr_low);

SPIM_1_PutArray((uint8_t *)strings[num_strings], length+1);
    //transmit the data we want to write
    //remember, our string is null-terminated, which also needs to be transmitted.
while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));
    //now we have to wait for it all to actually be sent by the psoc
SS_Write(1);

addr_low += length;
//increment our address counter by however many bytes we wrote, not including the null byte.
//this way, next time we write, we'll overwrite the null byte with a space to continue the string.
```

Figure 10: The Demo Code to Perform a WRITE

## 6.3 Reading



```
SS_Write(0);
SPIM_1_WriteTxData(0x03);                       //0x03 is the command to read
transmit_address(0x00, 0x00, 0x00);             //start the read back at the base of the flash chip
while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));
                    //we need to wait for the address to finish transmitting before clearing the buffer
SPIM_1_ClearRxBuffer();                         //get junk data out of the buffer

int i;
for (i = 0; i < 16; i ++){                      //sequentially read the data from the flash chip
    SPIM_1_WriteTxData(0xAA);
                         //the chip only outputs while we're writing to it, so write dummy data
    while(!(SPIM_1_ReadTxStatus() & SPIM_1_STS_SPI_DONE));
                                    //make sure each dummy byte gets transmitted before we read
    read_string[i] = SPIM_1_ReadRxData();       //put each received byte into the string buffer
    if (read_string[i] == 0) break;     //if we see our null termination, break the loop
}
SS_Write(1);
```

Figure 11: The Demo Code to Perform a READ

To read to the chip, we send the READ command, and then the address of the start of our string. Because of the full-duplex communication, we need to wait until all of our meaningful data has been transferred, and then we can clear our Rx buffer. After that, we transmit dummy bytes, reading the response bytes from the Rx buffer and storing them in an array until we see the null terminator. We can then print the read string to our LCD panel.

## 6.4 Running the Demo

This demo is pretty simple. There's a string, "6.115 is cool!!", split into three parts, delineated by the space characters. Each time you press SW2, another part of the string will be written to the flash memory. It starts over when you reset the demo, but critically *it does not erase the flash chip.* When you press SW3, it will read from the flash memory and print to the LCD. This means that you can write some fraction of the string, read it out, turn the board off and back on, and without writing anything more, read back what was written before you turned off the board. This is what makes flash handy- It's nonvolatile!

Have fun with the demo! Play around with changing the string fragments. Notice that if you write the original string to the flash, then edit it in PSoC Creator and reprogram the PSoC, you'll still read the original string from the flash until you hit SW2 to write the new string over it!

## 7 Conclusion

All in all, flash memories and SPI on the PSoC are incredibly powerful tools, despite some of the quirks of its implementation. With this chip, we can store segments of music, sound effects, sprites... just about anything you can come up with. More broadly, SPI allows you to greatly expand the functionality of your PSoC, whether you're adding a bit of flash storage space, communicating with more DACs or ADCs, or interfacing with a 2TB SD card. Flash memories and SPI communication are a wonderful tool to have on your belt.

## 8 References/Resources

1. M95P32 Chip Datasheet (Also mirrored on class website)- `https://www.st.com/resource/en/datasheet/m95p32-i.pdf`