
Project 6: SPI

The Task

- **Implement various methods to develop the various functionalities offered by the flash memory attached to the SPI header board.**
 - **The timing diagrams provided by the data sheet made it very clear that we had to submit bytes of information in a specific order to make the methods work.**
 - **In every code snippet, random hex codes will be submitted to the SPI board. This is as the timing diagrams specified.**
 - **Almost everything else reads like plain English thanks to the helper methods outlined in the Shortcuts slide.**
-

The Setup

- **Master-Slave:**
 - **Master: MSP430**
 - Executes programmed commands as specified by the methods we write.
 - Transmits requests to the slave through the master-out-slave-in port, otherwise known as MOSI.
 - ALL of the methods outlined are for the master.
 - **Slave: SPI header board**
 - Receives requests and executes functionality, sometimes returning data to the master via the master-in-slave-out port, otherwise known as MISO.
 - The slave's behavior/demands are outlined in the data sheet.

Shortcuts

- **Since most of the methods required lots of enabling and disabling of the memory, as well as the “Write Enable”, some helper methods were written to reduce the amount of boilerplate code present.**

The Code

```
inline void EnableMemory(unsigned char ComponentNumber) {  
    if (ComponentNumber == FLASH_MEMORY_U3) {  
        ENABLE_FLASH_MEMORY_U3;  
    } else {  
        ENABLE_FLASH_MEMORY_U2;  
    }  
}  
  
inline void DisableMemory(unsigned char ComponentNumber) {  
    if (ComponentNumber == FLASH_MEMORY_U3) {  
        DISABLE_FLASH_MEMORY_U3;  
    } else {  
        DISABLE_FLASH_MEMORY_U2;  
    }  
}  
  
inline void WriteEnable(unsigned char ComponentNumber) {  
    EnableMemory(ComponentNumber);  
    SPISendByte(0x06);  
    DisableMemory(ComponentNumber);  
}  
  
inline void WriteDisable(unsigned char ComponentNumber) {  
    EnableMemory(ComponentNumber);  
    SPISendByte(0x04);  
    DisableMemory(ComponentNumber);  
}
```

Flash Memory: ID

- **Returns the identification information of the product.**
- **Not covered in the rubric, so the code will not be presented for this one.**

Status Register: Read

- Reads a register that returns the current status of the active flash memory.
 - Returns a byte representing the properties of the currently active flash memory, such as 'Busy' status and read-only status.
-

The Code

```
unsigned char ReadFlashMemoryStatusRegister(unsigned char ComponentNumber)
{
    EnableMemory(ComponentNumber);

    SPISendByte(0x05);

    unsigned char status = SPIReceiveByte();

    DisableMemory(ComponentNumber);

    return status;
}
```

Status Register: Write

- Tweaks the read-only status of the blocks of memory corresponding to each write protection level, as well as the “mode” of writing (automatic iteration versus writing a single byte to an address).

The Code

```
void WriteFlashMemoryStatusRegister(unsigned char WriteValue,unsigned char ComponentNumber)
{
    EnableMemory(ComponentNumber);

    SPISendByte(0x50);

    DisableMemory(ComponentNumber);

    EnableMemory(ComponentNumber);

    SPISendByte(0x01);

    SPISendByte(WriteValue);

    DisableMemory(ComponentNumber);
}
```

Status Register: Busy

- Returns the busy status of the flash memory.
 - Essentially returns bit 0 of the status register output.
-

The Code

```
unsigned char FlashMemoryBusy(unsigned char ComponentNumber)
{
    return ReadFlashMemoryStatusRegister(ComponentNumber) % 2;
}
```

Flash Memory: Read

- Reads the flash memory at the starting address, and continuously iterates through the addresses.
 - Returns the byte of memory held at each address until disabled.
 - Comes in two flavors: normal, high-speed.
 - High-speed requires an extra “don’t care” byte to be submitted to the slave.
-

The Code

```
void ReadFlashMemory(unsigned Long StartAddress, unsigned char* DataValuesArray,  
    unsigned int NumberOfDataValues, unsigned char ComponentNumber, unsigned char ReadMode)  
{  
    EnableMemory(ComponentNumber);  
  
    SPISendByte(ReadMode);  
  
    SPISendByte((unsigned char) (StartAddress >> 16));  
    SPISendByte((unsigned char) (StartAddress >> 8));  
    SPISendByte((unsigned char) (StartAddress >> 0));  
  
    if (ReadMode == 0x0B) {  
        SPISendByte(0x00);  
    }  
  
    unsigned int i;  
  
    for (i = 0; i < NumberOfDataValues; i++) {  
        DataValuesArray[i] = SPIReceiveByte();  
    }  
  
    DisableMemory(ComponentNumber);  
}
```

Write Protection

- **Smaller scale version of the write method to the status register.**
 - **Adjusts the bits corresponding to write protection levels, enabling read-only status on:**
 - None of the memory
 - Quarter of the memory
 - Half of the memory
 - All of the memory
-

The Code

```
void SetBlockProtection(unsigned char ProtectionLevel, unsigned char ComponentNumber)
{
    ENABLE_WRITE_PROTECT;

    WriteFlashMemoryStatusRegister(ProtectionLevel << 2, ComponentNumber);

    DISABLE_WRITE_PROTECT;
}
```

Flash Memory: Single Byte Write

- **Writes a single byte of memory to the specified address in the active flash memory.**
 - **Does absolutely nothing if read-only status is set for that address.**
 - **Has a wait time - you need to suspend any further actions until then.**
 - **Requires a “Write Enable” instruction.**
-

The Code

```
void ByteProgramFlashMemory(unsigned Long MemoryAddress, unsigned char WriteValue, unsigned char ComponentNumber)
{
    WriteEnable(ComponentNumber);
    EnableMemory(ComponentNumber);

    SPISendByte(0x02);

    SPISendByte((unsigned char) (MemoryAddress >> 16));
    SPISendByte((unsigned char) (MemoryAddress >> 8));
    SPISendByte((unsigned char) (MemoryAddress >> 0));

    SPISendByte(WriteValue);

    while (FlashMemoryBusy(ComponentNumber));

    DisableMemory(ComponentNumber);
}
```

Flash Memory: Data Streams

- **Writes a stream of bytes to the memory starting at the specified start address and beyond.**
 - **Automatically iterates, starting at the specified address, so just keep submitting data until finished.**
 - **An extension of the single byte write functionality.**
 - Has a wait time.
 - Also requires a “Write Enable” instruction.
 - Also does absolutely nothing to the address if read-only status is set there.
-

The Code

```
void AAIProgramFlashMemory(unsigned Long StartAddress, unsigned char* DataValuesArray,
                           unsigned int NumberOfDataValues, unsigned char ComponentNumber)
{
    WriteEnable(ComponentNumber);
    EnableMemory(ComponentNumber);

    unsigned int i;

    for (i = 0; i < NumberOfDataValues; i++) {
        SPISendByte(0xAF);
        if (i == 0) {
            SPISendByte((unsigned char) (StartAddress >> 16));
            SPISendByte((unsigned char) (StartAddress >> 8));
            SPISendByte((unsigned char) (StartAddress >> 0));
        }
        SPISendByte(DataValuesArray[i]);
        DisableMemory(ComponentNumber);
        while (FlashMemoryBusy(ComponentNumber));
        EnableMemory(ComponentNumber);
    }

    WriteDisable(ComponentNumber);

    EnableMemory(ComponentNumber);

    SPISendByte(0x05);

    DisableMemory(ComponentNumber);
}
```

Flash Memory: Wipe

- **Completely wipes all data on the flash memory, setting all bits in the flash memory to the default “erase” value of 1.**
 - **Requires a “Write Enable” instruction.**
 - **Has a wait time.**
 - **Does absolutely nothing if the bits at that address are read-only.**
-

The Code

```
void ChipEraseFlashMemory(unsigned char ComponentNumber)
{
    WriteEnable(ComponentNumber);
    EnableMemory(ComponentNumber);

    SPISendByte(0xC7);
    while (FlashMemoryBusy(ComponentNumber));

    DisableMemory(ComponentNumber);
}
```

Flash Memory: Selective Erase

- **Completely wipes a 32 KB region of memory starting from the specified address.**
 - **Requires a “Write Enable” instruction.**
 - **Has a wait time.**
 - **Does nothing if the bits at the address are read-only.**
-

The Code

```
void SectorBlockEraseFlashMemory(unsigned long StartAddress, unsigned char ComponentNumber, unsigned char EraseMode)
{
    WriteEnable(ComponentNumber);
    EnableMemory(ComponentNumber);












    SPISendByte(EraseMode);

    SPISendByte((unsigned char) (StartAddress >> 16));
    SPISendByte((unsigned char) (StartAddress >> 8));
    SPISendByte((unsigned char) (StartAddress >> 0));
    while (FlashMemoryBusy(ComponentNumber));

    DisableMemory(ComponentNumber);
}
```


Final Verification

- This is just proof that the methods are working.
- Clearly we have passed all 6 tests.

Name	Type	Value	Location
 ID_U2	unsigned int	0xBF48 (Hex)	0x03F8
 ID_U3	unsigned int	0xBF48 (Hex)	0x03F6
 LFSRStateAfterRead_U2	unsigned int	0x0362 (Hex)	0x03F2
 LFSRStateAfterRead_U3	unsigned int	0x0362 (Hex)	0x03EE
 LFSRStateAfterWrite_U2	unsigned int	0x0362 (Hex)	0x03F4
 LFSRStateAfterWrite_U3	unsigned int	0x0362 (Hex)	0x03F0
 NumberOfTestsPassed	unsigned char	0x06 '\x06' (Hex)	Register R13
 StatusRegister_U2	unsigned char	0x0C '\x0c' (Hex)	0x03FB
 StatusRegister_U3	unsigned char	0x0C '\x0c' (Hex)	0x03FA
 TestHasNotFailed	unsigned char[6]	[0x01 '\x01',0x01 '\x01',0x01 '\x01',0x...	0x03E8
 TestNumber	unsigned char	5 '\x05'	Register R10

Fin
