Serial RAM
Adding Memory to Microcontrollers

• A parallel access memory module would require too many pins
  – You would need ~16 pins for the address, 8 pins for the data, plus more for control signals
  – Microcontrollers are usually pin-limited

• Instead, you can use a serial memory
  – Address and data are sent and received in a bit-serial fashion
  – The SPI (serial peripheral interface) is fast and requires only 4 pins
One type of memory - EEPROM

- **EEPROM**
  - Electrically Erasable Programmable Read-Only Memory
  - Non-volatile
  - Realized as arrays of floating gate transistors

- **Architecture**
  - No charge on floating gate => transistor operates normally
  - Charge on the floating gate => transistor is always off

\[ V_e = V_{DD} \text{ so that transistor conducts} \]

To program, raise \( V_e \) to high voltage (~12V); this causes some electrons in the channel to “tunnel” through the insulator to the floating gate.
EEPROM

- Programming an EEPROM (i.e., storing data) is relatively slow (~milliseconds)
  - Reading is fast, though
  - You can program multiple bytes in a single operation (this is called a page write)

- The number of times an EEPROM can be reprogrammed is limited
  - About one million write operations
  - So it shouldn’t be used like a normal random access memory
  - It is typically used to store configuration information
  - Or it could be used for a (slow) data logging application in an embedded system
Another type of memory: SRAM

- SRAM: “Static Random Access Memory”
- Retains data as long as power is supplied
- No limit on the number of writes
- Unlike dynamic RAM (DRAM) it doesn’t have to refreshed; easier to interface
- SRAM is faster, more expensive and less dense than DRAM
- Primary uses
  - Storage in embedded systems (appliances, LCD displays, cameras)
  - Caches in CPUs
  - Buffers in routers
SRAM Architecture

- SRAM cell is made up of 2 cross-connected inverters to form a latch
- The latch is opened or closed under control of the “word line”
- Data is read or written using the “bit” line (bit is the complement of bit)
- When word line is low, transistors T1 and T2 are off and the latch retains its state

SPI – a “synchronous” serial interface

• “Synchronous” means there is a clock signal in addition to the data signal
• One device is the “master” (typically the MCU), and generates the clock signal; other devices are “slaves” (typically peripheral chips)
• Data is shifted serially from a shift register in the master to a shift register in the slave

- The shift registers are connected in a “ring” configuration
  - When the master shifts its data to the slave, it automatically gets back the data that was in the slave’s shift register

Figure 10.8 Master/slave transfer block diagram
An HCS12 can have multiple SPI channels called SPI0, SPI1, SPI2

Our chip (the C version) has only one SPI channel

It uses Port M pins (PM2:PM5)
Serial SRAM

- A 512 Kbit serial SRAM
  - Organized into 64K 8-bit bytes
  - 32 byte pages

- 20 MHz max clock frequency (at Vcc = 5V)

- You can read & write as fast as you can send & receive data

- We will use the PDIP (8-pin) package

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Vcc Range</th>
<th>Temp. Ranges</th>
<th>Dual I/O (SDI)</th>
<th>Quad I/O (SDI)</th>
<th>Max. Clock Frequency</th>
<th>Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>23A512</td>
<td>1.7-2.2V</td>
<td>I, E</td>
<td>Yes</td>
<td>Yes</td>
<td>20 MHz (1)</td>
<td>SN, ST, P</td>
</tr>
<tr>
<td>23LC512</td>
<td>2.5-5.5V</td>
<td>I, E</td>
<td>Yes</td>
<td>Yes</td>
<td>20 MHz (1)</td>
<td>SN, ST, P</td>
</tr>
</tbody>
</table>

Note 1: 16 MHz for E-temp.

Features:
- SPI-Compatible Bus Interface:
  - 20 MHz Clock rate
- SPI/SDI/SDI mode
- Low-Power CMOS Technology:
  - Read Current: 3 mA at 5.5V, 20 MHz
  - Standby Current: 4 µA at +85°C
- Unlimited Read and Write Cycles
- Zero Write Time
- 64K x 8-bit Organization:
  - 32-byte page
- Byte, Page and Sequential mode for Reads and Writes
- High Reliability
- Temperature Ranges Supported:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C
- RoHS Compliant
- 8-Lead SOIC, TSSOP and PDIP Packages

Description:
The Microchip Technology Inc. 23A512/23LC512 are 512Kbit Serial SRAM devices. The memory is accessed via a simple Serial Peripheral Interface (SPI) compatible serial bus. The bus signals required are a clock input (SCK) plus separate data in (SDI) and data out (SO) lines. Access to the device is controlled through a Chip Select (CS) input. Additionally, SDI (Serial Dual Interface) and SDQ (Serial Quad Interface) is supported if your application needs faster data rates. This device also supports unlimited reads and writes to the memory array.

The 23A512/23LC512 is available in standard packages including 8-lead SOIC, PDIP and advanced 8-lead TSSOP.

Pin Function Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Chip Select Input</td>
</tr>
<tr>
<td>SO/SIO1</td>
<td>Serial Output/SDI/SDI Pin</td>
</tr>
<tr>
<td>STS1</td>
<td>SPI Pin</td>
</tr>
<tr>
<td>STS0</td>
<td>Ground</td>
</tr>
<tr>
<td>SI/SIO0</td>
<td>Serial Input/SDI/SDI Pin</td>
</tr>
<tr>
<td>SCK</td>
<td>Serial Clock</td>
</tr>
<tr>
<td>R/WSI/SI3</td>
<td>Hold/SQI Pin</td>
</tr>
<tr>
<td>VCC</td>
<td>Power Supply</td>
</tr>
</tbody>
</table>
Pins

- Key signals:
  - /CS
  - SO
  - SI
  - SCK

- We won’t use pin 3 (SIO2)
- Tie pin 7 (/HOLD) to +5V
Sending a byte to SRAM

- To send a byte to the SRAM
  - Assert /CS (i.e., bring it low)
  - Pulse the clock (polarity = “high”) 8 times
  - Serial data (SI) is clocked in on first (rising) edges of SCK

Data is sent most significant bit (MSB) first

Note: need a minimum of 25 ns before next chip select (this is less than one clock cycle, with a 24 MHz clock)
Receiving a byte from SRAM

- To receive a byte from the SRAM
  - Assert /CS (i.e., bring it low)
  - Pulse the clock 8 times … serial data is clocked out on SO

From Microchip 23LC512 datasheet
Protocol to write a byte to an address

- Select chip (i.e., pull signal /CS low)
- Send 4 bytes on signal SI:
  - the “write” instruction (0x02)
  - the 16 bit address
  - the data byte to be written
- Then deselect chip (pull /CS high)

**FIGURE 2-2: BYTE WRITE SEQUENCE (SPI MODE)**

From Microchip 23LC512 datasheet
Protocol to read a byte from an address

- Select chip (i.e., pull /CS low)
- Send 3 bytes on signal SI:
  - the “read” instruction (0x03)
  - the 16 bit address
- Then receive the byte on signal SO

- Then deselect chip (i.e., pull /CS high)

From Microchip 23LC512 datasheet
Hardware interface

- We will use the SPI (serial peripheral interface) system on the MCU to interface to the SRAM

- Note on chip select
  - The /SS (slave select) signal on the MCU’s SPI system (pin PM3) could theoretically be used to drive the /CS input on the SRAM
  - /SS is automatically is asserted (goes low) when transmission starts and then is de-asserted (goes high) when transmission finishes
  - However, we need more control over it (ie, should stay low for 4 byte sequence)
  - So just use a digital output pin (such as PT0) to drive /CS (with some software)
SPI System Registers

• You write to (or read from) the 8-bit data register SPIDR
  – A write to this register allows the byte to be queued and transmitted
  – If your system is the master, the queued byte is transmitted immediately after the previous transmission is complete

• Flags (such as the transmission complete flag) are in the status register SPISR

• You set the baud rate using the SPIBR register

• Control register SPICR1
  – There is another control register, SPICR2, but we won’t use it
SPI Control Register

- **SPICR1**
  - **SPE** – SPI system enable
  - **SPIE** – SPI interrupt enable
  - **MSTR** – set to 1 for master mode
  - **CPOL, CPHA** – clock format
  - **SSOE** – enable SS output for master
  - **LSBF** – set to 1 to send LSB first

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SPIE</td>
<td>0 = SPI disabled. 1 = SPI enabled.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SPE</td>
<td>0 = SPI disabled. 1 = SPI enabled and pins PS4-PS7 are dedicated to SPI function.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SPTIE</td>
<td>0 = SPTEF interrupt disabled. 1 = SPTEF interrupt enabled.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MSTR</td>
<td>0 = slave mode. 1 = master mode.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CPOL</td>
<td>0 = active high clocks selected; SCK idle low. 1 = active low clocks selected, SCK idle high.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CPHA</td>
<td>0 = The first SCK edge is issued one-half cycle into the 8-cycle transfer operation. 1 = The SCK edge is issued at the beginning of the 8-cycle transfer operation.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SSOE</td>
<td>Slave select output enable bit. The SS output feature is enabled only in master mode by asserting the SSOE bit and the MODFEN bit of the SPIxCR2 register.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LSBF</td>
<td>0 = data is transferred most significant bit first. 1 = data is transferred least significant bit first.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.1 SPI control register 1 (SPIxCR1, x = 0, 1, or 2)
Clock Signal

- There are four possible combinations of clock polarity and phase

  • **CPOL**
    - Clock polarity
    - 0: clock pulses are high
    - 1: clock pulses are low

  • **CPHA**
    - Clock phase
    - 0: data is valid on 1\(^{\text{st}}\) edge
    - 1: data is valid on 2\(^{\text{nd}}\) edge
SPI Baud Rate Register

- **SPIBR**

  - **Example**
    - Set baud rate to 2 MHz (assuming a 24 MHz E clock)
    - We need a divisor of 12 (because 24 MHz/12 = 2 MHz)
    - This can be done using: $12 = (2+1) \times 2^{(1+1)} = 3 \times 2^2 = 12$
    - So SPPR2:SPPR0 = 010, and SPR2:SPR0 = 001

![Figure 10.3 SPI baud rate register (SPIxBR, x = 0, 1, or 2)](image)
SPI Status Register

- **SPISR**
  - **SPIF** – flag set when receiver register is full
  - **SPTEF** – flag set when transmitter register is empty

- **Clearing flags**
  - **SPIF**: clear by reading SPISR, then reading from SPIDR
  - **SPTEF**: clear by reading SPISR, then writing to SPIDR

---

**Figure 10.4 SPI status register (SPIxSR)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SPIF: SPI interrupt request bit</td>
</tr>
<tr>
<td>6</td>
<td>SPTEF: SPI data register empty interrupt flag</td>
</tr>
<tr>
<td>5</td>
<td>MODF: mode error interrupt status flag</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset value = 0x20
## Summary of SPI Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPICR1</td>
<td>SPIE</td>
<td>SPE</td>
<td>SPTIE</td>
<td>MSTR</td>
<td>CPOL</td>
<td>CPHA</td>
<td>SSOE</td>
<td>LSBFE</td>
</tr>
<tr>
<td>SPICB</td>
<td>0</td>
<td>SPPR2</td>
<td>SPPR1</td>
<td>SPPR0</td>
<td>0</td>
<td>SPR2</td>
<td>SPR1</td>
<td>SPR0</td>
</tr>
<tr>
<td>SPISR</td>
<td>SPIF</td>
<td>0</td>
<td>SPTEF</td>
<td>MODF</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Master writes to SPIDR to start transmission
- SPIF – flag set when receiver register is full
- SPTEF – flag set when transmitter register is empty
- Baud rate divisor = (SPPR+1) * 2^(SPR+1)

- SPE – SPI system enable
- SPIE – SPI interrupt enable
- MSTR – set to 1 for master mode
- CPOL, CPHA – clock format
- SSOE – enable SS output for master
- LSBF – set to 1 to send LSB first
SPI initialization

- Turn on SPI system
- Set baud rate to 1 MHz
- Set as master
- No interrupts
- Clock is pulsed high
- Data is clocked on first clock edge
- Transfer most significant bit first

```c
// Initialize SPI system
void initializeSPI(void) {
    // Set baud rate.  Baud rate divisor = (SPPR+1) * 2^(SPR+1)
    // Bits: 0 SPPR2 SPPR1 SPPR0 0 SPR2 SPR1 SPR0
    SPIBR = ??

    // SPICR1 bits: SPIE SPE SPTIE MSTR CPOL CPHA SSOE LSBFE
    SPICR1 = ??

    // Use defaults for SPICR2
    SPICR2 = 0x00;
}
```
SPI initialization

// Initialize SPI system
void initializeSPI(void) {
    // Set baud rate. Baud rate divisor = (SPPR+1) * 2^(SPR+1).
    // for baud = 1 MHz divisor = 24  SPPR=5, SPR=1 -> 0x51
    SPIBR = 0x51;  // Bits: 0 SPPR2 SPPR1 SPPR0 0 SPR2 SPR1 SPR0

    // SPICR1 bits: SPIE SPTIE SPE MSTR CPOL CPHA SSOE LSBFE
    // SPIE = 0  no interrupts
    // SPE = 1   enable SPI system
    // SPTIE = 0 no interrupts
    // MSTR = 1  we are master
    // CPOL = 0  clock is pulsed high
    // CPHA = 0  data is clocked on first clock edge
    // SSOE = 0  set to 1 to enable SS output
    // LSBF = 0  transfer most signif bit first
    SPICR1 = 0x50;

    // Use defaults for SPICR2
    SPICR2 = 0x00;
}
Send and receive functions

// Send a character out through SPI
void putcharSPI(char cx)
{
    char temp;
    while(!(SPISR & 0x20));  // wait until SPTEF=1 (transmit reg empty)
    SPIDR = cx;              // output the byte to the SPI
    while(!(SPISR & 0x80));  // wait until SPIF=1 (receive reg full)
    temp = SPIDR; // clear the SPIF flag
}

// Get a character from SPI
char getcharSPI(void)
{
    while(!(SPISR & 0x20)); // wait until SPTEF=1 (transmit reg empty)
    SPIDR = 0x00; // trigger 8 SCK pulses to shift in data
    while(!(SPISR & 0x80));  // wait until SPIF=1 (receive reg full)
    return SPIDR; // return the character
}
Flowchart to write a data byte to SRAM at a certain address

Start
select slave
send “write” instruction
send MSB of address
send LSB of address
send data byte
deselect slave
Stop

Select slave by pulling PT0 low
Deselect slave by pulling PT0 high

Flowchart to read a data byte from SRAM from a certain address

Start
select slave
send “read” instruction
send MSB of address
send LSB of address
get data byte
deselect slave
Stop
Note on SSMI board

- On the SSMI board, pin PM4 (the MOSI signal) is hardwired to the speaker.
- You can’t use the speaker if you are using the SPI system.
- The speaker can be disconnected by removing jumper W2 (but don’t do that for this lab).
Bit banging

• If your MCU doesn’t have a dedicated SPI interface, you can always implement the SPI interface using any digital I/O pins and software
  – Software would have to implement the serial clock, signal timing, synchronization, etc.

• This is called “bit banging” – using software instead of dedicated hardware to directly set and sample signals

• Actually this method is common for embedded systems

• Example – this application note describes how to interface a serial EEPROM using bit banging and the 8051 MCU:
Summary

• A serial RAM is a memory that you can read and write to via a serial interface. It is a good way to expand the memory of an embedded system.
  – Using the SPI serial interface, this takes only _???_ pins.
  – A “static” RAM retains its data as long as power is applied.

• Other types of memory that are commonly used:
  – EEPROM – non-volatile, is typically used for storing small amounts of data (like calibration tables or device configuration)
  – Flash memory – is faster to write than EEPROM, but you have to write a whole page at once.