Lecture 15: Serial I/O Devices
Quick overview of serial communication in general
SCI — Serial Communication Interface
SCI on the HCS12
Serial communication transmits one bit of information at a time.
One bit is sent, a time delay occurs, the next bit is sent.
Used to interface to printers, keyboards, scanners, etc.
Universal asynchronous receiver/transmitter (UART) is the interface chip that implements the transmission.
A serial channel is a collection of signals (or wires) that implement the communication.
Data terminal equipment (DTE) is the computer.
Data communication equipment (DCE) is the modem.
**A Serial Channel**

<table>
<thead>
<tr>
<th></th>
<th>CMOS Level</th>
<th>RS232 Level</th>
<th>RS422 Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>True/Mark</td>
<td>+5 V</td>
<td>TxD = -12 V</td>
<td>(TxD⁺ - TxD⁻) = -3 V</td>
</tr>
<tr>
<td>False/Space</td>
<td>+0.1 V</td>
<td>TxD = +12 V</td>
<td>(TxD⁺ - TxD⁻) = +3 V</td>
</tr>
</tbody>
</table>
A **frame** is a complete and nondivisble packet of bits. Includes both information (e.g. data, characters) and overhead (start bit, error checking, and stop bits).

*Parity* is generated at the transmitter and checked at the receiver to help detect errors in transmission.

*Even parity* makes number of 1s even (data+parity).

*Odd parity* makes number of 1s odd (data+parity).

*Bit time* is the time between each bit.
**Bandwidth**

*Baud rate* is total number bits transmitted per time.

*Information* is data user wishes to transmit:

Characters to be printed.

*Overhead* is bits added to achieve transmission:

Start bit(s), stop bit(s), parity, etc.

\[
\text{Bandwidth} = \frac{\text{information bits/frame}}{\text{total bits/frame}} \times \text{baud rate}
\]
SCI versus SPI
Various Serial Channels

Simplex

Full duplex

Half duplex

Open collector
RS232 Cables
## RS232 DB9 Pin Assignments

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Description</th>
<th>True</th>
<th>DTE</th>
<th>DCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DCD</td>
<td>Data Carrier Detect</td>
<td>+12</td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>2</td>
<td>RxD</td>
<td>Receive Data</td>
<td>−12</td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>3</td>
<td>TxD</td>
<td>Transmit Data</td>
<td>−12</td>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>4</td>
<td>DTR</td>
<td>Data Terminal Rdy</td>
<td>+12</td>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>5</td>
<td>SG</td>
<td>Signal Ground</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
<td>Data Set Ready</td>
<td>+12</td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>7</td>
<td>RTS</td>
<td>Request to Send</td>
<td>+12</td>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>8</td>
<td>CTS</td>
<td>Clear to Send</td>
<td>+12</td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>9</td>
<td>RI</td>
<td>Ring Indicator</td>
<td>+12</td>
<td>In</td>
<td>Out</td>
</tr>
</tbody>
</table>
RS232 Interface
More RS232 Interface Chips
RS232 Specifications

Must withstand
  Short to ground
  Short to any other wire
Operating range
  True \(-15 \leq V_{out} \leq -5\) V
  False \(+5 \leq V_{out} \leq +15\) V
Maximum output voltage
  \(-25 \leq V_{out} \leq +25\) V
Short circuit current
  \(I_{out} \leq 0.5\) A
Transition \((-3\) to \(3\)) range time \(\leq 4\%\)

Maximum Slew Rate
  \(dV_{in}/dt \leq 30\) V/\(\mu\)s

Operating Range
  True \(-15 \leq V_{in} \leq -3\) V
  Translation \(-3 < V_{in} < +3\) V
  False \(+3 \leq V_{in} \leq +15\) V

Input Resistance
  \(3000\Omega \leq R_{in} \leq 7000\Omega\)

Input Capacitance including cable
  \(C_{in} \leq 2500\) pF

Input open circuit voltage
  \(E_{in} \leq 2\) V
## RS422/RS423/RS485 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>RS232D</th>
<th>RS423A</th>
<th>RS422</th>
<th>RS485</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drivers on one line</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Receivers on one line</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>Max distance (ft)</td>
<td>50</td>
<td>4,000</td>
<td>4,000</td>
<td>4,000</td>
</tr>
<tr>
<td>Max data rate</td>
<td>20kb/s</td>
<td>100kb/s</td>
<td>10Mb/s</td>
<td>10Mb/s</td>
</tr>
<tr>
<td>Max driver output</td>
<td>±25V</td>
<td>±6V</td>
<td>-0.25/+6V</td>
<td>-7/+12V</td>
</tr>
<tr>
<td>Receiver input</td>
<td>±15V</td>
<td>±12V</td>
<td>±7V</td>
<td>-7/+12V</td>
</tr>
</tbody>
</table>

![RS422/RS423/RS485 Circuit Diagram](image-url)
Single host computer controls the USB. Host controls all transactions using a token-based protocol. Uses a tiered star topology with host at the center. Up to 127 devices can be connected to one USB bus. Plug’n’play implemented with dynamically loadable drivers. Host detects new devices and loads appropriate driver. Can operate at high (480Mb/s), full (12Mb/s), or low (1.5Mb/s) speeds.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Color</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>VBUS (5V)</td>
</tr>
<tr>
<td>2</td>
<td>White</td>
<td>D-</td>
</tr>
<tr>
<td>3</td>
<td>Green</td>
<td>D+</td>
</tr>
<tr>
<td>4</td>
<td>Black</td>
<td>Ground</td>
</tr>
</tbody>
</table>
Current Loop Serial Channel
### Modem Serial Interface

<table>
<thead>
<tr>
<th>Logic</th>
<th>Originate</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>1270 Hz</td>
<td>2225 Hz</td>
</tr>
<tr>
<td>False</td>
<td>1070 Hz</td>
<td>2025 Hz</td>
</tr>
</tbody>
</table>
Optical Channel

Diagram:

- SCI
- TxData
- RxData
- Ground
- Fiber optics
- Light
- Sensor
- Two optic cables
- Not connected
- SCI
- RxData
- SCI
- RxData
- Ground
- TxData
- Ground
Most embedded microcomputers support SCI.
Common features include:

- A baud rate control register used to select transmission rate.
- A mode bit M used to select 8-bit (M=0) or 9-bit (M=1) data frames.

Each device can create its own serial port clock with period that is integer multiple of the E clock period.
Transmitting in Asynchronous Mode

Common features in the transmitter:
- TxD data output pin, with TTL voltage levels.
- 10- or 11-bit shift register, not directly accessible.
- Serial communications data register (SCDR), write only, separate from receive reg. though same address.
- T8 data bit for 9-bit data mode.
Control Bits for the Transmitter

Transmit Enable (TE), set to 1 to enable transmitter.
Send Break (SBK), set to 1 to send blks of 10 or 11 0s.
Transmit Interrupt Enable (TIE), set to arm TDRE flag.
Transmit Complete Enable (TCIE), set to arm TC flag.
Transmit Data Register Empty flag (TDRE), set when SCDR empty, clear by reading TDRE and writing SCDR.
Transmit Complete flag (TC), set when transmit shift register done shifting, cleared by reading TC flag then writing SCDR.
Pseudo Code for Transmission Process

**TRANSMIT**
- Set TxD=0
- Wait 16 clock times
- Set n=0
- Output start bit

**TLOOP**
- Set TxD=bn
- Wait 16 clock times
- Set n=n+1
- Output data bit
- Goto TLOOP if n<=7
- Set TxD=T8
- Wait 16 clock times
- Output T8 bit
- Set TxD=1
- Wait 16 clock times
- Output a stop bit
Receiving in Asynchronous Mode

Common features in the receiver:

- RxD data input pin, with TTL voltage levels.
- 10- or 11-bit shift register, not directly accessible.
- Serial communications data register (SCDR), read only, separate from transmit reg. though same address.
- R8 data bit for 9-bit data mode.
Receiver Enable (RE), set to 1 to enable receiver.
Receiver Wakeup (RWU), set to 1 to allow a receiver input to wake up the computer.
Receiver Interrupt Enable (RIE), set to arm RDRF flag.
Idle Line Interrupt Enable (ILIE), set to arm IDLE flag.
Status Bits Generated by the Receiver

Receive Data Register Full flag (RDRF), set when new data available, clear by reading RDRF and SCDR.

Receiver Idle flag (IDLE), set when receiver line is idle, clear by reading IDLE, then reading SCDR.

Overrun flag (OR), set when input data lost because previous frame not read, clear by reading OR and SCDR.

Noise flag (NF), set when input is noisy, clear by reading NF flag, then reading SCDR.

Framing error (FE), set when stop bit is incorrect, clear by reading FE, then reading SCDR.
Figures for Receiving
Pseudo Code for Receive Process

RECEIVE
  Goto RECEIVE if RxD=1
  Wait 8 clock times
  Goto RECEIVE if RxD=1
  Set n=0

RLOOP
  Wait 16 clock times
  Set bn=RxD
  Set n=n+1
  Goto RLOOP if n<=7
  Wait 16 clock times
  Set R8=RxD
  Wait 16 clock times
  Set FE=1 if RxD=0
  Wait for start bit
  Wait half a bit time
  False start?
  Wait 1 bit time
  Input data bit
  Wait 1 bit time
  Read R8 bit
  Wait 1 bit time
  Framing error if no stop bit
One SCI port using Port S bits 1 and 0. Least significant 13 bits of SCIBD register determine baud rate. **SCICR2** register contains control bits for SCI (see Table 7.11). **SCICR1** register contains other miscellaneous SCI control bits.

**LOOPS**: disconnects receiver from RxD pin.

**RSRC**: when **LOOPS** = 1, **RSRC** = 0 connects receiver to transmitter internally while **RSRC** = 1 connects receiver to TxD.

**WAKE**: 0 wakeup on IDLE line, 1 wakeup on address mark.

**ILT**: determines if idle line count starts from start or stop bit.

**SWAI**: setting this bit causes SCI to shutdown and pause any communication.

**PE**: setting this bit enables parity checking.

**PT**: 0 is even parity while 1 is odd parity.
Flags in **SCISR1** register can be read but not modified by software.

The error conditions are also reported in the **SCISR1** register including parity flag, **PF**, set on parity errors.

The **SCISR2** register contains two mode control and one status bit.

- **BRK13**: break character is 13 or 14 bits when 1 and 10 or 11 bits when 0.
- **TXDIR**: specifies direction of the TxD pin in single-wire mode.
- **RAF**: 1 when frame is being received.

The **SCIDRL** register contains data transmitted and received.

The **SCIDRH** register contains the 9th data bits.
Simultaneous input and output requires two FIFOs. RxFifo passes data from InSCI handler to main thread. TxFifo passes data from main thread to OutSCI handler. Since TxFifo initially empty, transmit interrupts initially disarmed. When main thread calls OutChar, transmit interrupts armed. Interrupt handler disarms transmit interrupts when TxFifo becomes empty.
void SCI_Init(void)
{
    asm sei
    RxFifo_Init(); // empty FIFOs
    TxFifo_Init();
    SCIBD = 26;    // 9600 bits/sec
    SCICR1 = 0;   // M=0, no parity
    SCICR2 = 0x2C; // enable, arm RDRF
    asm cli       // enable interrupts
}
SCI Interface ISR

// RDRF set on new receive data
// TDRE set on empty transmit register
interrupt 20 void SciHandler(void){
char data;
    if(SCISR1 & RDRF){
        RxFifo_Put(SCIDRL); // clears RDRF
    }
    if((SCICR2&TIE)&&(SCISR1&TDRE)){
        if(TxFifo_Get(&data)){
            SCIDRL = data; // clears TDRE
        }
        else{
            SCICR2 = 0x2c; // disarm TDRE
        }
    }
}
SCI In/Out Character

// Input ASCII character from SCI
// spin if RxFifo is empty
char SCI_InChar(void){ char letter;
    while (RxFifo_Get(&letter) == 0){};
    return(letter);
}

// Output ASCII character to SCI
// spin if TxFifo is full
void SCI_OutChar(char data){
    while (TxFifo_Put(data) == 0){};
    SCICR2 = 0xAC; // arm TDRE
Printer bandwidth may be less than the maximum bandwidth supported by the serial channel.

Special characters may require more time to print.
Most printers have internal FIFOs that could get full.
The printer may be disconnected.
The printer may be deselected.
The printer power may be off.

*Flow control* is needed to synchronize computer with variable rate output device

DTR: data terminal ready
XON/XOFF
SCI Simplex Interface with DTR Handshaking
void Printer_Init(void){
    asm sei
    TxFifo_Init(); // empty FIFOs
    SCIBD = 52;     // 9600 bits/sec
    SCICR1 = 0;     // M=0, no parity
    SCICR2 = 0x0C;  // enable disarm TDRE
    TIOS &=~0x08;   // PT3 input capture
    DDRT &=~0x08;   // PT3 is input
    TSCR1 = 0x80;   // enable TCNT
    TCTL4 |= 0xC0;  // both rise and fall
    TIE |= 0x08;    // Arm IC3
    TFLG1 = 0x08;   // initially clear
    asm cli        // enable interrupts
}
Serial Output Using DTR Synchronization ISRs

// TDRE set on empty transmit register
interrupt 20 void SciHandler(void){
  char data;
  if((SCICR2&TIE)&&(SCISR1&TDRE)){
    if(TxFifo_Get(&data)){
      SCIDRL = data; // clears TDRE
    }
  } else{
    SCICR2 = 0x2c; // disarm TDRE
  }
}

// IC3 interrupt on any change of DTR
void interrupt 11 IC3Han(void) {
  TFLG1 = 0x08; // Ack, clear C3F
  checkIC3(); // Arm SCI if DTR=+12
}
Serial Output Using DTR Synchronization

```c
void checkIC3(void){
    if(PTT&0x08) // PT3=1 if DTR=-12
        SCICR2 = 0x0C; // busy, so disarm
    else
        SCICR2 = 0x8C; // not busy, so arm
}
// Output ASCII character to Printer
// spin if TxFifo is full
void Printer_OutChar(char data){
    while (TxFifo_Put(data) == 0){};
    checkIC3();
}
```
Use of XON/XOFF to Interface a Printer
Synchronous Communication Using the SPI

Two devices communicating with SCI operate at same frequency but have 2 separate (not synchronized) clocks.

Two devices communicating with SPI operate using the same (synchronized) clock.

Master device creates the clock while slave device(s) use the clock to latch data in or out.
A Synchronous Serial Interface
Motorola SPI includes four I/O lines:

- **SS** - slave select, used by master to indicate the channel is active.
- **SCK** - 50% duty cycle clock generated by the master.
- **MOSI** (master-out slave-in) - data line driven by master.
- **MISO** (master-in slave-out) - data line driven by slave.

Transmitting device uses one edge of clock to change data, and receiving device uses other edge to accept data.

When data transfer occurs combined 16-bit register is serially shifted eight bit positions (data exchanged).
Common control features of the SPI module include:

- A baud rate control register
- A mode bit in the control register to select master versus slave, clock polarity, clock phase.
- Interrupt arm bit
- Ability to make outputs open-drain.

Common status bits for the SPI module include:

- SPIF, transmission complete
- WCOL, write collision
- MODF, mode fault
Pseudo Code for SPI Communication

TRANSMIT
Set n=7

TLOOP
On fall of Sck, set Data=bn
Set n=n-1
Goto TLOOP if n>=0
Set Data=1

RECEIVE
Set n=7

RLOOP
On rise of Sck, read data
Set bn=Data
Set n=n-1
Goto RLOOP if n>=0
Synchronous Serial Modes
MC9S12C32 SPI Details

Uses four pins, \( PM3 = \overline{SS} \), \( PM5 = SCLK \), \( PM4 = MOSI \), and \( PM2 = MISO \).

If 6812 is master, set DDRM to make PM5, PM4, and PM3 outputs.

Can be in run, wait, or stop mode.

<table>
<thead>
<tr>
<th>SPR2</th>
<th>SPR1</th>
<th>SPR0</th>
<th>Div</th>
<th>Freq</th>
<th>Bit Time</th>
<th>Freq</th>
<th>Bit Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2 MHz</td>
<td>500 ns</td>
<td>12 MHz</td>
<td>83.3 ns</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1 MHz</td>
<td>1 ( \mu s )</td>
<td>6 MHz</td>
<td>166.7 ns</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>500 kHz</td>
<td>2 ( \mu s )</td>
<td>3 MHz</td>
<td>333.3 ns</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>250 kHz</td>
<td>4 ( \mu s )</td>
<td>1.5 MHz</td>
<td>666.7 ns</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>125 MHz</td>
<td>8 ( \mu s )</td>
<td>750 kHz</td>
<td>1.33 ( \mu s )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>64</td>
<td>62.5 kHz</td>
<td>16 ( \mu s )</td>
<td>375 kHz</td>
<td>2.67 ( \mu s )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>128</td>
<td>31.25 kHz</td>
<td>32 ( \mu s )</td>
<td>187.5 kHz</td>
<td>5.33 ( \mu s )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>15.625 kHz</td>
<td>64 ( \mu s )</td>
<td>93.75 kHz</td>
<td>10.67 ( \mu s )</td>
</tr>
</tbody>
</table>
**SPIDR** is 8-bit register used for both input and output.

**SPICR1** register species SPI mode of operation.

- **SPE**: enables the SPI system.
- **SPIE**: arms interrupts on the **SPIF** flag.
- **SPTIE**: arms interrupts on the **SPTEF** flag.
- **LSBF**: if 1 transmits least significant bit first.

**SPISR** register contains flags for the SPI system.

- **SPIF**: indicates that new data is available to be read.
- **SPTEF**: indicates that SPI data register can accept new data.
- **MODF**: mode error interrupt status flag.
# Mode Selections

<table>
<thead>
<tr>
<th>MODFEN</th>
<th>SSOE</th>
<th>Master Mode (MSTR=1)</th>
<th>Slave Mode (MSTR=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>PM3 not used with SPI</td>
<td>PM3 is SS input</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>PM3 not used with SPI</td>
<td>PM3 is SS input</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>PM3 is SS input w/MODF</td>
<td>PM3 is SS input</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>PM3 is SS output</td>
<td>PM3 is SS input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin Mode</th>
<th>MSTR</th>
<th>SPC0</th>
<th>BIDIROE</th>
<th>MISO</th>
<th>MOSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Master In</td>
<td>Master Out</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MISO not used</td>
<td>Master In</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>Slave Out</td>
<td>Slave In</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Slave In</td>
<td>MOSI not used</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Slave In</td>
<td>Slave I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>Slave I/O</td>
<td>MOSI not used</td>
</tr>
</tbody>
</table>
SPI Modes in the 6812

<table>
<thead>
<tr>
<th>Normal mode SPC0=0</th>
<th>Master mode (MSTR=1)</th>
<th>Slave mode (MSTR=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial out</td>
<td>SI</td>
</tr>
<tr>
<td>SPI</td>
<td>MO</td>
<td>PM4</td>
</tr>
<tr>
<td></td>
<td>DDRM4 MI</td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td>PM2</td>
<td>PM2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bidirectional mode SPC0=1</th>
<th>Master mode (MSTR=1)</th>
<th>Slave mode (MSTR=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial out</td>
<td>SI</td>
</tr>
<tr>
<td>SPI</td>
<td>MOMI PM4</td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td>PM2</td>
<td>PM2</td>
</tr>
<tr>
<td></td>
<td>BIDIROE</td>
<td>BIDIROE</td>
</tr>
</tbody>
</table>
Digital-to-Analog Converter

[Diagram of Digital-to-Analog Converter]
SPI Interfacing Issues

Word size - need to transmit 12 bits to DAC, send 16-bits, first 4 bits will be ignored.

Bit order - DAC8043 requires most significant bit first.

Clock phase/polarity - DAC8043 samples on rising edge, so SPI must change data on falling edge.

Bandwidth - DAC8043 has minimum clock low width of 120ns, so shortest SPI period is 250ns.
void DAC_Init(void){ // PM3=LD=1
    DDRM |= 0x38; // PM5=CLK=SPI clock out
    PTM |= 0x08; // PM4=SRI=SPI master out
    /* bit SPICR1
    7 SPIE = 0 no interrupts
    6 SPE  = 1 enable SPI
    5 SPTIE= 0 no interrupts
    4 MSTR = 1 master
    3 CPOL = 0 output changes on fall,
    2 CPHA = 0 clock normally low
    1 SSOE = 0 PM3 regular output, LD
    0 LSBF = 0 most sign bit first */
    SPICR1 = 0x50;
    SPICR2 = 0x00; // normal mode
    SPIBR = 0x00;} // 2MHz
D/A Interface Using the SPI

#define SPIF 0x80
void DAC_out(unsigned short code){
   // better implementation using SPTEF
   unsigned char dummy;
   while((SPISR&SPTEF)==0); // wait for transmit empty
   SPIDR = (code>>8); // msbyte
   dummy = SPIDR; // clear SPIF
   while((SPISR&SPTEF)==0); // wait for transmit empty
   SPIDR = code; // lsbyte
   dummy = SPIDR; // clear SPIF
   Timer_Wait(10); // wait for SPI output completion
   PTM &= ~0x08; // PM3=LD=0
   PTM |= 0x08; // PM3=LD=1
}
Analog-to-Digital Converter
SPI Interfacing Issues

- **Word size** - need to transmit 8 bits to ADC, then receive 12 bits back. MAX1247 sends it as 2 8-bit transmissions.

- **Bit order** - MAX1247 requires most significant bit first.

- **Clock phase/polarity** - MAX1247 samples on rising edge, so SPI must change data on falling edge.

- **Bandwidth** - MAX1247 has maximum SCLK frequency of 2 MHz, so shortest SPI period is 500ns.
void ADC_Init(void){  // PM3=CS=1
    DDRM |=0x38;  // PM5=SCLK=SPI clock out
    DDRM &=~0x04; // PM2=DOUT=SPI master in
    PTM |= 0x08; // PM4=DIN=SPI master out
    /* bit SPICR1
    7 SPIE = 0    no interrupts
    6 SPE = 1     enable SPI
    5 SPTIE= 0    no interrupts
    4 MSTR = 1    master
    3 CPOL = 0    output changes on fall,
    2 CPHA = 0    clock normally low
    1 SSOE = 0    PM3 regular output, LD
    0 LSBF = 0    most sign bit first */
    SPICR1 = 0x50;
    SPICR2 = 0x00;  // normal mode
    SPIBR = 0x00;}  // 2MHz
unsigned short ADC_in(unsigned char code){
    unsigned short data; unsigned char dummy;
    PTM &= ~0x08; // PM3=CS=0
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = code; // set channel,mode
    while((SPISR&SPIF)==0); // gadfly wait
    dummy = SPIDR; // clear SPIF
    SPIDR = 0; // start SPI
    while((SPISR&SPIF)==0); // gadfly wait
    data = SPIDR<<8; // msbyte of ADC
    SPIDR = 0; // start SPI
    while((SPISR&SPIF)==0); // gadfly wait
    data += SPIDR; // lsbyte of ADC
    PTM |= 0x08; // PM3=CS=1
    return data>>3; // right justify
}
Temperature Sensor/Controller
// Interface between MC9S12C32 and DS1620 using the SPI
// bit status Configuration/Status Register meaning
// 7 DONE 1=Conversion done, 0=conversion in progress
// 6 THF 1=temperature above TH, 0=temperature below TH
// 5 TLF 1=temperature below TL, 0=temperature above TL
// 1 CPU 1=CPU control, 0=stand alone operation
// 0 1SHOT 1=one conversion and stop, 0=continuous conversions
// temperature  digital value (binary) digital value (hex)
// +125.0 C  011111010 $0FA
// + 64.0 C  010000000 $080
// +  1.0 C  000000010 $002
// +  0.5 C  000000001 $001
//   0 C    000000000 $000
// - 0.5 C  111111111 $1FF
// -16.0 C  111100000 $1E0
// -55.0 C  110010010 $192
void DS1620_Init(void){ // PM3=RST=0
    DDRM |= 0x38; // PM5=CLK=SPI clock out
    PTM &=~0x08; // PM4=DQ=SPI bidirectional data
/* bit SPICR1
  7 SPIE = 0    no interrupts
  6 SPE  = 1    enable SPI
  5 SPTIE= 0    no interrupts
  4 MSTR = 1    master
  3 CPOL = 1    output changes on fall
  2 CPHA = 1    and input clocked in on rise
  1 SSOE = 0    PM3 regular output DS1620 RST
  0 LSBF = 1    least significant bit first */
    SPICR1 = 0x5D;
    SPICR2 = 0x01; // bidirectional mode
    SPIBR = 0x01;} // 1MHz could be 2MHz
#define SPIF 0x80

void out8(char code){
    unsigned char dummy;
    // assumes DDRM bit 4 is 1, output
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = code;
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    dummy = SPIDR;       // clear SPIF
}

void out9(short code){
    unsigned char dummy;
    // assumes DDRM bit 4 is 1, output
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = code;          // lsbyte
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    dummy = SPIDR;         // clear SPIF
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = (code>>8);     // msbyte
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    dummy = SPIDR;         // clear SPIF
}
unsigned char in8(void){
    short n; unsigned char result;
    DDRM &=~0x10; // PM4=DQ input
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = 0; // start shift register
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    result = SPIDR; // get data, clear SPIF
    DDRM |= 0x10; // PM4=DQ output
    return result;}

short in9(void) {
    short result;
    DDRM &=~0x10; // PM4=DQ input
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = 0; // start shift register
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    result = SPIDR; // get LS data, clear SPIF
    while((SPISR&SPTEF)==0); // wait for transmit empty
    SPIDR = 0; // start shift register
    while((SPISR&SPIF)==0); // gadfly wait for SPIF
    if(SPIDR&0x01) // get MS data, clear SPIF
        result |= 0xFF00; // negative
    else
        result &~=~0xFF00; // positive
    DDRM | = 0x10; // PM4=DQ output
    return result;
}
void DS1620_Start(void){
    PTM |= 0x08;  // RST=1
    out8(0xEE);
    PTM &=~0x08;}  // RST=0

unsigned char DS1620_ReadConfig(void){
    unsigned char value;
    PTM |= 0x08;  // RST=1
    out8(0xAC);
    value = in8();
    PTM &=~0x08;  // RST=0
    return value;}

unsigned short DS1620_ReadT(void){
    unsigned short value;
    PTM |= 0x08;  // RST=1
    out8(0xAA);
    value = in9();
    PTM &=~0x08;  // RST=0
    return value;}

```c
void DS1620_WriteConfig(char data){
    PTM |= 0x08;  // RST=1
    out8(0x0C);
    out8(data);
    PTM &=~0x08;} // RST=0

void DS1620_WriteTH(short data){
    PTM |= 0x08;  // RST=1
    out8(0x01);
    out9(data);
    PTM &=~0x08;} // RST=0

void DS1620_WriteTL(short data){
    PTM |= 0x08;  // RST=1
    out8(0x02);
    out9(data);
    PTM &=~0x08;} // RST=0
```