Using the MC9S12 IIC Bus with DS 1307 Real Time Clock

- DS1307 Data Sheet
- The MC9S12 Serial Communication Interface (SCI)
- MC9S12 SCI Block Guide V02.05
- Huang, Sections 9.2-9.6

Dallas Semiconductor DS1307 Real Time Clock

- The DS 1307 is a real-time clock with 56 bytes of NV (non-volatile) RAM
- It uses the IIC bus, with address 1101000₂
- It stores date and time
  - Data are stored in BCD format
- It uses a 32.768 kHz crystal to keep time
- It can generate a square wave output
  - Frequency of square wave can be 1 Hz, 4.096 kHz, 8.192 kHz or 32.768 kHz
- It uses a battery to hold the date and time when your board is not powered
Using the Dallas Semiconductor DS1307 Real Time Clock

• Set up the IIC bus
  – Find the SCL frequency, SDA hold time, Start and Stop hold times
  – Determine the value to write to IBFD to meet those times

• To set the time,
  – Send the Start condition
  – Write address of clock (with R/W low)
  – Write a 0 (to select seconds register),
  – Write second, minute, hour, day of week, day of month, month, year, control
  •  Control determines whether or not to enable square wave, and selects frequency
  – Send the Stop condition

• To read the clock,
  – Send the Start condition
  – Write the address of the clock (with R/W low), then write a 0 (to select seconds register).
  – Send the Stop condition
  – Send the Start condition
  – Write the address of the clock (with R/W high)
  – Read the time registers.
  – Send the Stop condition
  • If you want to store some data which will remain between power cycles, you can write it to the 56 bytes of NV RAM
Asynchronous Data Transfer

• In asynchronous data transfer, there is no clock line between the two devices

• Both devices use internal clocks with the same frequency

• Both devices agree on how many data bits are in one data transfer (usually 8, sometimes 9)

• A device sends data over an TxD line, and receives data over an RxD line
  – The transmitting device transmits a special bit (the start bit) to indicate the start of a transfer
  – The transmitting device sends the requisite number of data bits
  – The transmitting device ends the data transfer with a special bit (the stop bit)

• The start bit and the stop bit are used to synchronize the data transfer
Asynchronous Data Transfer

• The receiver knows when new data is coming by looking for the start bit (digital 0 on the RxD line).

• After receiving the start bit, the receiver looks for 8 data bits, followed by a stop bit (digital high on the RxD line).

• If the receiver does not see a stop bit at the correct time, it sets the Framing Error bit in the status register.

• Transmitter and receiver use the same internal clock rate, called the Baud Rate.

• At 9600 baud (the speed used by D-Bug12), it takes 1/9600 second for one bit, 10/9600 second, or 1.04 ms, for one byte.

Asynchronous Serial Protocols
• The SCI interface on the MC9S12 uses voltage levels of 0 V and +5 V. The RS-232 standard uses voltage levels of +12 V and -12 V.
  – The Dragon12-Plus board uses a Maxim MAX232A chip to shift the TTL levels from the MC9S12 to the RS-232 levels necessary for connecting to a standard serial port. 0 V from the SCI is converted to +12 V on the DB-9 connector and +5 V from the SCI is converted to -12 V on the DB-9 connector.
  – The RS-232 standard can work on cables up to a length of 50 feet.

• Another asynchronous standard is RS-485. Dragon12-Plus board can use SCI1 in RS-485 mode
  – RS-485 is a two-wire differential asynchronous protocol
  – Multiple devices can connect to the same two wires
  – Only one device on the RS-485 bus can transmit; all the other devices are in receive mode
  – The Dragon12-Plus DS75176 differential-to-single ended converter to convert the single-ended SCI1 data to differential RS-485 data
  – Bit 0 of Port J determines if the RS-485 should be in receive mode or transmit mode
  – RS-485 can work with cables up to a length of 1,000 feet.
Parity in Asynchronous Serial Transfers

• The HCS12 can use a parity bit for error detection.
  – When enabled in SCI0CR1, the parity function uses the most significant bit for parity.

  – There are two types of parity – even parity and odd parity
    * With even parity, and even number of ones in the data clears the parity bit; an odd number of ones sets the parity bit. The data transmitted will always have an even number of ones.
    * With odd parity, and odd number of ones in the data clears the parity bit; an even number of ones sets the parity bit. The data transmitted will always have an odd number of ones.

  – The HCS12 can transmit either 8 bits or 9 bits on a single transfer, depending on the state of M bit of SCI0CR1.

  – With 8 data bits and parity disabled, all eight bits of the byte will be sent.

  – With 8 data bits and parity enabled, the seven least significant bits of the byte are sent; the MSB is replaced with a parity bit.

  – With 9 data bits and parity disabled, all eight bits of the byte will be sent, and an additional bit can be sent in the sixth bit of SCI0DRH.
    * It usually does not make sense to use 9 bit mode without parity.

  – With 9 data bits and parity enabled, all eight bits of the byte are sent; the ninth bit is the parity bit, which is put into the MSB of SCI0DRH in the receiver.
Asynchronous Data Transfer

- The HCS12 has two asynchronous serial interfaces, called the SCI0 and SCI1 (SCI stands for Serial Communications Interface)

- SCI0 is used by D-Bug12 to communicate with the host PC

- When using D-Bug12 you normally cannot independently operate SCI0 (or you will lose your communications link with the host PC)

- The SCI0 TxD pin is bit 1 of Port S; the SCI1 TxD pin is bit 3 of Port S.

- The SCI0 RxD pin is bit 0 of Port S; the SCI1 RxD pin is bit 2 of Port S.

- In asynchronous data transfer, serial data is transmitted by shifting out of a transmit shift register into a receive shift register.

SCI0DR receive and transmit registers are separate registers, distributed into two 8-bit registers, SCI0DRH and SCI0DRL.

An overrun error is generated if RxD shift register filled before SCI0DR read.
Timing in Asynchronous Data Transfers

- The BAUD rate is the number of bits per second.

- Typical baud rates are 1200, 2400, 4800, 9600, 19,200, and 115,000

- At 9600 baud the transfer rate is 9600 bits per second, or one bit in 104 $\mu$s.

- When not transmitting the TxD line is held high.

- When starting a transfer the transmitting device sends a start bit by bringing TxD low for one bit period (104 $\mu$s at 9600 baud).

- The receiver knows the transmission is starting when it sees RxD go low.

- After the start bit, the transmitter sends the requisite number of data bits.

- The receiver checks the data three times for each bit. If the data within a bit is different, there is an error. This is called a noise error.

- The transmitter ends the transmission with a stop bit, which is a high level on TxD for one bit period.

- The receiver checks to make sure that a stop bit is received at the proper time.

- If the receiver sees a start bit, but fails to see a stop bit, there is an error. Most likely the two clocks are running at different frequencies (generally because they are using different baud rates). This is called a framing error.

- The transmitter clock and receiver clock will not have exactly the same frequency.

- The transmission will work as long as the frequencies differ by less 4.5% (4% for 9-bit data).
Timing in Asynchronous Data Transfers

ASYNCHRONOUS SERIAL COMMUNICATIONS

Baud Clock = 16 x Baud Rate

Start Bit

Data Bit

Start Bit — Three 1’s followed by 0’s at RTI, 3, 5, 7
(Two of RTI, 5, 7 must be zero —
If not all zero, Noise Flag set)

Data Bit — Check at RTB, 9, 10
(Majority decides value)
(If not all same, noise flag set)

If no stop bit detected, Framing Error Flag set

Baud clocks can differ by 4.5% (4% for 9 data bits)
with no errors.

Even parity — the number of ones in data word is even
Odd parity — the number of ones in data word is odd

When using parity, transmit 7 data + 1 parity, or 8 data + 1 parity
Baud Rate Generation

- The SCI transmitter and receiver operate independently, although they use the same baud rate generator.

- A 13-bit modulus counter generates the baud rate for both the receiver and the transmitter.

- The baud rate clock is divided by 16 for use by the transmitter.

- The baud rate is

  \[ \text{mboxSCIBaudRate} = \frac{\text{Bus Clock}}{16 \times \text{SCI1BR}[12:0]} \]

- With a 24 MHz bus clock, the following values give typically used baud rates.

<table>
<thead>
<tr>
<th>Bits SPR[12:0]</th>
<th>Receiver Clock (Hz)</th>
<th>Transmitter Clock (Hz)</th>
<th>Target Baud Rate</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>615,384.6</td>
<td>38,461.5</td>
<td>38,400</td>
<td>0.16</td>
</tr>
<tr>
<td>78</td>
<td>307,692.3</td>
<td>19,230.7</td>
<td>19,200</td>
<td>0.16</td>
</tr>
<tr>
<td>156</td>
<td>153,846.1</td>
<td>38,461.5</td>
<td>9,600</td>
<td>0.16</td>
</tr>
<tr>
<td>312</td>
<td>76,693.0</td>
<td>38,461.5</td>
<td>4,800</td>
<td>0.16</td>
</tr>
</tbody>
</table>
SCI Registers

• Each SCI uses 8 registers of the HCS12. In the following we will refer to SCI1.

• Two registers are used to set the baud rate (SCI1BDH and SCI1BDL)

• Control register SCI1CR2 is used for normal SCI operation.

• SCI1CR1 is used for special functions, such as setting the number of data bits to 9.

• Status register SCI1SR1 is used for normal operation.

• SCI1SR2 is used for special functions, such as single-wire mode.

• The transmitter and receiver can be separately enabled in SCI1CR2.

• Transmitter and receiver interrupts can be separately enabled in SCI1CR2.

• SCI1SR1 is used to tell when a transmission is complete, and if any error was generated.

• Data to be transmitted is sent to SCI1DRL.

• After data is received it can be read in SCI1DRL. (If using 9-bit data mode, the ninth bit is the MSB of SCI0DRH.)
Example program using the SCI Transmitter

#include "derivative.h"

/* Program to transmit data over SCI port */

main()
{
    /****************************************************************
     * SCI Setup
     ****************************************************************/

    SCI1BDL = 156; /* Set BAUD rate to 9,600 */
    SCI1BDH = 0;
    SCI1CR1 = 0x00; /* 0 0 0 0 0 0 0 0 |  |  |  |  |  |  |  | |  |  ... Reciever Interrupt |  
__ No Tranmit Complete Interrupt 
__ No Tranmit Ready Interrupt */

    SCI1CR2 = 0x08; /* 0 0 0 0 1 0 0 0 */

    /****************************************************************
     * End of SCI Setup
     ****************************************************************/

}
SCI1DRL = 'h'; /* Send first byte */

while ((SCI1SR1 & 0x80) == 0) ; /* Wait for TDRE flag */
SCI1DRL = 'e'; /* Send next byte */

while ((SCI1SR1 & 0x80) == 0) ; /* Wait for TDRE flag */
SCI1DRL = 'l'; /* Send next byte */

while ((SCI1SR1 & 0x80) == 0) ; /* Wait for TDRE flag */
SCI1DRL = 'l'; /* Send next byte */

while ((SCI1SR1 & 0x80) == 0) ; /* Wait for TDRE flag */
SCI1DRL = 'o'; /* Send next byte */

while ((SCI1SR1 & 0x80) == 0) ; /* Wait for TDRE flag */
Example program using the SCI Receiver

/* Program to receive data over SCI1 port */

#include "derivative.h"
#include "vectors12.h"

interrupt void sci1_isr(void);
volatile unsigned char data[80];
volatile int i;

main()
{

 /******************************************************************************
 * SCI Setup
 ******************************************************************************/
 SCI1BDL = 156; /* Set BAUD rate to 9,600 */
 SCI1BDH = 0;
 SCI1CR1 = 0x00; /* 0 0 0 0 0 0 0 0 |  |  |  |  |  |  |  | |  |  ... Interrupts used |  \
________________ No Tranmit Complete Interrupt \
__________________ No Tranmit Ready Interrupt */

 SCI1CR2 = 0x04; /* 0 0 1 0 0 1 0 0 */

}
UserSCI1 = (unsigned short) &sci1_isr;
i = 0;
enable();

/***************************************************************************/
* End of SCI Setup
/***************************************************************************/
while (1) {
    /* Wait for data to be received in ISR, then */
    /* do something with it */
}

interrupt void sci1_isr(void) {
    char tmp;
    /* Note: To clear receiver interrupt, need to read */
    /* SCI1SR1, then read SCI1DRL. */
    /* The following code does that */
    
    if ((SCI1SR1 & 0x20) == 0) return; /* Not receiver interrupt */
data[i] = SCI1DRL;
i = i+1;
return;