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.
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 D-Bug12 printf() function sends data to the host PC over SCI0
- 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 µs.
• 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 µs 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 - Three 1's followed by 0's at RT1, 3, 5, 7
(Two of RT3, 5, 7 must be zero -
If not all zero, Noise Flag set)

Data Bit - Check at RT8, 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
  \[ mbox{SCIBaudRate} = \frac{Bus\ Clock}{16 \times SCI1BR[12:0]} \]

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.)
<table>
<thead>
<tr>
<th>SBR12</th>
<th>SBR11</th>
<th>SBR10</th>
<th>SBR9</th>
<th>SBR8</th>
<th>SCI1BDH − 0x00D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBR7</td>
<td>SBR6</td>
<td>SBR5</td>
<td>SBR4</td>
<td>SBR3</td>
<td>SCI1BDL − 0x00D1</td>
</tr>
<tr>
<td>LOOPS</td>
<td>SCI1SR</td>
<td>RSRC</td>
<td>M</td>
<td>WAKE</td>
<td>SCI1CR1 − 0x00D2</td>
</tr>
<tr>
<td></td>
<td>RIEN</td>
<td></td>
<td>ILIE</td>
<td>TE</td>
<td>SCI1CR2 − 0x00D3</td>
</tr>
<tr>
<td></td>
<td>TIE</td>
<td>TCIE</td>
<td>RIE</td>
<td>ILIE</td>
<td>SCI1SR1 − 0x00D4</td>
</tr>
<tr>
<td>TDRE</td>
<td>TC</td>
<td>RDRF</td>
<td>IDLE</td>
<td>OR</td>
<td>SCI1SR2 − 0x00D5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NF</td>
<td>SCI1DRH − 0x00D5</td>
</tr>
<tr>
<td>R8</td>
<td>T8</td>
<td></td>
<td></td>
<td></td>
<td>SCI1DRL − 0x00D7</td>
</tr>
<tr>
<td>R7/T7</td>
<td>R6/T6</td>
<td>R5/T5</td>
<td>R4/T4</td>
<td>R3/T3</td>
<td>R2/T2</td>
</tr>
</tbody>
</table>
Example program using the SCI Transmitter

#include <hcs12.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 
    | | | | | | | | Even Parity 
    | | | | | | | | Parity Disabled 
    | | | | | | | | Short IDLE line mode (not used) 
    | | | | | | | | Wakeup by IDLE line rec (not used) 
    | | | | | | | | 8 data bits 
    | | | | | | | | Not used (loopback disabled) 
    | | | | | | | | SCI1 enabled in wait mode 
    | | | | | | | | Normal (not loopback) mode 
    */

    SCI1CR2 = 0x08; /* 0 0 0 0 1 0 0 0 
    | | | | | | | | No Break 
    | | | | | | | | Not in wakeup mode (always awake) 
    | | | | | | | | Reciever disabled 
    | | | | | | | | Transmitter enabled 
    | | | | | | | | No IDLE Interrupt 
    | | | | | | | | No Reciever Interrupt 
    | | | | | | | | No Transmit Complete Interrupt 
    | | | | | | | | No Transmit Ready Interrupt 
    */
    /****************************************************************
    * 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 "hcs12.h"
#include "vectors12.h"

void INTERRUPT 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
    | | | | | | | |
    | | | | | | | \___ Even Parity
    | | | | | | | Parity Disabled
    | | | | | | \____ Short IDLE line mode (not used)
    | | | | | \______ Wakeup by IDLE line rec (not used)
    | | | | \________ 8 data bits
    | | | \____________ Not used (loopback disabled)
    | | \____________ SCI1 enabled in wait mode
    | \______________ Normal (not loopback) mode
    */

    SCI1CR2 = 0x04; /* 0 0 1 0 0 1 0 0
    | | | | | | | |
    | | | | | | | \___ No Break
    | | | | | | \_____ Not in wakeup mode (always awake)
    | | | | | \_______ Receiver enabled
    | | | | \_________ Transmitter disabled
    | | | \___________ No IDLE Interrupt
    | | \_____________ Receiver Interrupts used
    | \_______________ No Transmit Complete Interrupt
    \_______________ No Transmit Ready Interrupt
    */

    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
     */
}

void INTERRUPT 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;
}