Review for Exam 3

A/D Converter

• Power-up A/D converter (ATD0CTL2)

• Write 0x05 to ATD0CTL4 to set at fastest conversion speed and 10-bit conversions

• Write 0x85 to ATD0CTL4 to set at fastest conversion speed and 8-bit conversions

• Select number of conversions in a sequence (ATD0CTL3)

• Select type of conversion sequence and the analog channels sampled (ATD0CTL5)
– Right/left justified
– signed/unsigned
– Continuous Scan vs. Single Scan
– Multichannel vs. Single Channel conversions

• How to tell when conversion is complete – ATD0STAT0 register

• How to read results of A/D conversions – ATD0DR[7 − 0]H (8-bit leftjustified conversions)

• How to read results of A/D conversions – ATD0DR[7 − 0]L (8-bit rightjustified conversions)

• How to read results of A/D conversions – ATD0DR[15 − 6] (10-bit leftjustified conversions)

• How to read results of A/D conversions – ATD0DR[9 − 0] (10-bit rightjustified conversions)
– Be able to convert from digital number to voltage, and from voltage to digital number (need to know VRH and VRL).

• How long does it take to make a conversion?

SPI

• Pins used – SCLK, MOSI, MISO, SS

• Difference of use in Master and Slave mode

• SPI0CR1 Register
– Enable SPI
– Master or Slave
– Enable interrupts
– Clock polarity
– Clock phase
– Automatically operate SS for single-byte transfers
– LSB or MSB first

• SPI0CR2 Register
– Get into bidirectional mode
– Enable or disable the output buffer on the data pin in bidirectional mode

• SPI0BR Register — Set speed (master only)

• SPI0SR Register — SPIF and SPTEF flag - clear SPIF flag by reading
SPI0SR, the read SPI0DR; clear the SPTEF flag by reading SPI0SR, then write the SPI0DR.

• SPI0DR Register — shift register – master starts transfer by writing data to SPI0DR

Interfacing

• Getting into expanded mode — MODA, MODB, MDOC pins or MODE Registers

• PEAR Register — enable ECLK, LSTRB, R/W on external pins

• Ports A and B in expanded mode
– Port A – A/D 15-8 (Port A is for data for high byte, even addresses)
– Port B – A/D 7-0 (Port B is for data for low byte, odd addresses)

• E clock
– Address on A/D 15-0 when E low, Data on A/D 15-0 when E high
– Need to latch address on rising edge of E clock
– On write (output), external device latches data on signal initiated by falling edge of E
– On read (input), HCS12 latches data on falling edge of E
– E-clock stretch - MISC register

• R/W Line

• LSTRB line

• Single-byte and two-byte accesses
– 16-bit access of even address – A0 low, LSTRB low – accesses even and odd bytes

– 8-bit access of even address – A0 low, LSTRB high – accesses even byte only
– 8-bit access of odd address – A0 high, LSTRB low – accesses odd byte only
– A0 high and LSTRB high never occurs on external bus.

• Address Decoding – interfacing using MSI chips

- **What's on the 9S12 bus as it executes a program**
- **The 9S12 Serial Communications Interface**
- 9S12 Serial Communications Interface (SCI) Block Guide V02.05
- Huang, Sections 9.2-9.6

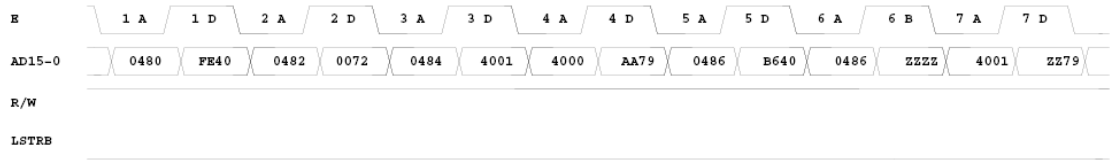
Consider a 9S12 executing the following program loop:

```
                    org $480
0480 FE4000 l1:     ldx $4000    % 3 cycles
0483 724001         inc $4001    % 4 cycles
0486 B64000         ldaa $4000   % 3 cycles
0489 20F5           bra $l1      % 3 cycles
```

If you assemble this program, you get the following:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0480	**FE**	**40**	**00**	**72**	**40**	**01**	**B6**	**40**	**00**	**20**	**F5**	**3B**	**FC**	**10**	**18**	**F3**

Here is what is on the bus during these 13 cycles:



Here is what happens cycle by cycle:

1. 9S12 does a 16 bit read from address $0480. The memory returns $FE40, the first two bytes of the ldx $4000 instruction.

2. 9S12 does a 16-bit read from address $0482. The memory returns $0072, the third byte of the the ldx $4000 instruction and the first byte of the inc $4001 instruction.

3. 9S12 does a 16-bit read from address $0484. The memory returns $4001, the second and third byte of the inc $4001 instruction.

4. 9S12 does a 16 bit read from address $4000 (it is executing the ldx $4000 instruction); the memory returns $AA79.

5. 9S12 does a 16-bit read from address $0486. The memory returns $B640, the first two bytes of the ldaa $4000 instruction.

6. 9S12 does nothing on bus (it puts the last address it used on the bus, during the address cycle, and nothing on the bus during the data cycle). It is completing the ldx $4000 instruction.

7. 9S12 does an 8 bit read from address $4001 (it is executing the inc $4001 instruction, and has to read the byte at address $4001); the memory returns $AA79.

8. 9S12 does a 16-bit read from address $0488. The memory returns $0020, the third byte of the the ldaa $4000 instruction and the first byte of the bra l1 instruction.

9. 9S12 does an 8 bit write to address $4001 (it is executing the inc $4001 instruction, and has to write the incremented byte to address $4001); it puts a $7A on the low byte and nothing of the high byte.

10. 9S12 does a 16-bit read from address $048A. The memory returns $F53B, the second byte of the the bra l1 instruction and the next byte in memory.

11. 9S12 does an 8 bit read from address $4000 (it is executing the ldaa $4000 instruction); the external device put a $AA on the high byte and nothing of the low byte.

12. 9S12 does a 16-bit read from address $048C. The memory returns $FC10, next two bytes in memory. The 9S12 has not yet figured out that it has to branch, so it is reading the next to bytes to fill its instruction pipeline.

13. 9S12 does nothing on bus (it puts the last address it used on the bus, during the address cycle, and nothing on the bus during the data cycle). It is figuring out where it needs to branch to.

1. The loop executes again.

Asynchronous Data Transfer

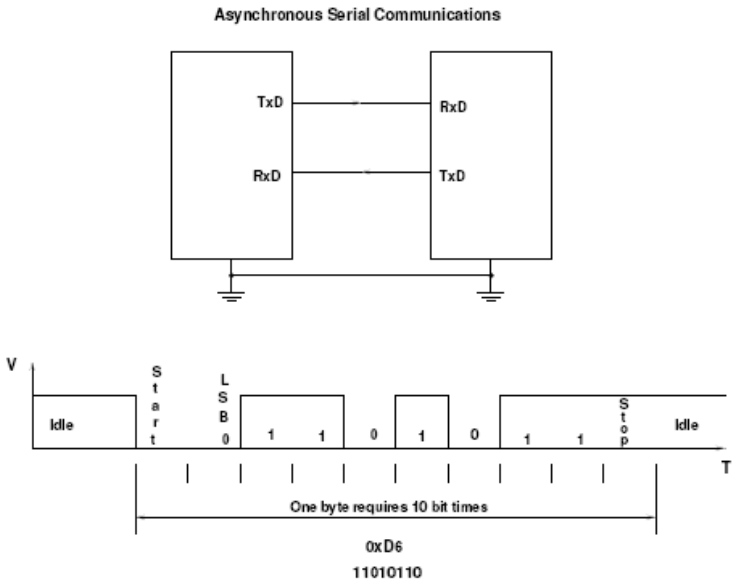
• In asynchronous data transfer, there is no clock line between the two devices

• 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 MSB is replaced with a parity bit.

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

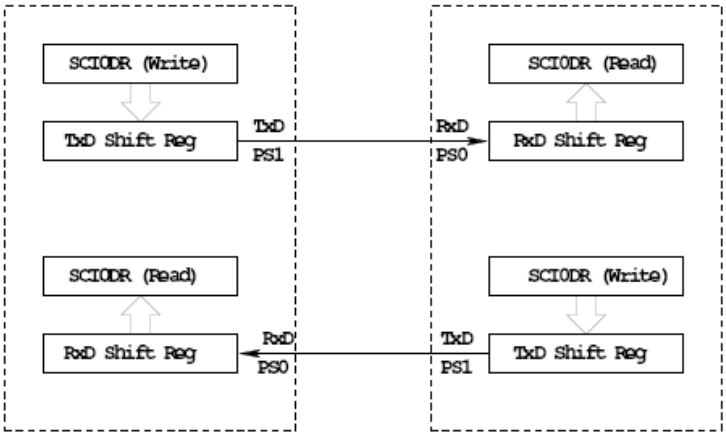
• When using D-Bug12 you normally cannot independently operate SCI0 (or you will lose your communications link with the host PC)

• 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 trasmitting 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.

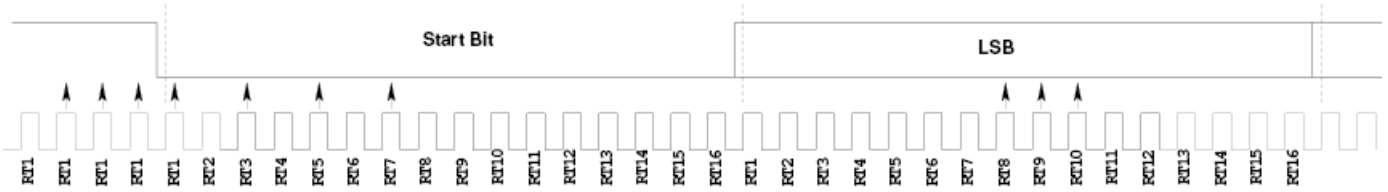
• If the receiver sees a start bit, but fails to see a stop bit, there is an error. Most likely the two clocks are running at different frequencies (generally because they are using different baud rates). This is called a framing error.

• 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

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

Stop Bit – Check at RT8, 9, 10
(Majority decides value)
(If not all same, noise flag set)

Baud Rate Generation

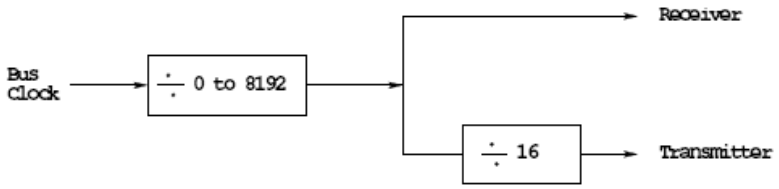
• The SCI transmitter and receiver operate independently, although they use the same baud rate generator.

• 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

SCIBaudRate =(Bus Clock)/(16 × SCI1BR[12:0])



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

Bits SPR[12:0]	Receiver Clock (Hz)	Transmitter Clock (Hz)	Target Baud Rate	Error (%)
39	615,384.6	38,461.5	38,400	0.16
78	307,692.3	19,230.7	19,200	0.16
156	153,846.1	9,615.3	9,600	0.16
312	76,693.0	4,793.3	4,800	0.16

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)

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

• Control register SCI1CR2 is used for normal SCI operation.

• 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.

• 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.)

SCI Baud Rate Generation

Module Base + 0x_0000

	7	6	5	4	3	2	1	0
R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
W								
Reset	0	0	0	0	0	0	0	0

Module Base + 0x_0001

	7	6	5	4	3	2	1	0
R	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
W								
Reset	0	0	0	0	0	1	0	0

= Unimplemented or Reserved

Figure 1-3. SCI Baud Rate Registers (SCIBDH and SCIBDL)

The baud rate for the SCI is determined by these 13 bits
SBR12-SBR8: SCI baud rate control register high
SBR7 - SBR0: SCI baud rate control register low

SCI Baud Rate = SCI module clock / (16 ×BR)

SCI Operation

Module Base + 0x_0002

	7	6	5	4	3	2	1	0
R	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT
W								
Reset	0	0	0	0	0	0	0	0

Figure 1-4. SCI Control Register 1 (SCICR1)

Read: Anytime

Write: Anytime

LOOPS: loop select bit. In loop operation, the RxD pin is disconnected from the SCI and the transmitter output is internally connected to the receiver input
0 = normal operation enabled
1 = loop operation enabled

SCISWAI: Enalbes/disables the wait mode
0 = SCI enabled in wait mode
1 = SCI disabled in wait mode

RSRC: Receiver source bit. When LOOPS = 1 RSRC determines the source of the receiver shift register
0 = Receiver input connected internally to transmitter output
1 = Receiver input connected externally to the transmitter

M: Data format mode bit. Determines whether data characters are 8 or 9 bits.
0 = 1 start bit, eight data bits, one stop bit
1 = 1 start bit, nine data bits, one stop bit

PE: Parity enable bit. Enables the parity function. When enabled, the parity function inserts a parity bit in the MSB
0 = Parity function disabled
1 = Parity function enabled

PT: Parity type bit. Determines whether the SCI generated and checks for even parity or odd parity. With even parity, an even number of 1s clears the parity bit and an odd number of 1s sets the parity bit. With odd parity, an odd number of 1s clears the parity bit and an even number of 1s sets the parity bit
0 = Even parity
1 = Odd parity

Module Base + 0x_0003

	7	6	5	4	3	2	1	0
R W	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
Reset	0	0	0	0	0	0	0	0

Figure 1-5. SCI Control Register 2 (SCICR2)

Read: Anytime

Write: Anytime

TIE: Transmitter interrupt enable bit. Enables the transmit data register empty flag, TDRE, to generate interrupt requests
0 = TDRE interrupt requests disabled
1 = TDRE interrupt requests enabled

TCIE: Transmission complete interrupt enable bit. TCIE enables the transmission complete flag, TC, to generate interrupt requests
0 = TC interrupt requests disabled
1 = TC interrupt requests enabled

RIE: Receiver full interrupt enable bit. Enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupts requests
0 = RDRF and OR interrupt requests disabled
1 = RDRF and OR interrupt requests enabled

ILIE: Idle line interrupt enable bit. Enables the idle line flag, IDLE, to generate interrupt requests
0 = IDLE interrupt requests disabled
1 = IDLE interrupt requests enabled

TE: Transmitter enable bit. Enables the SCI transmitter and configures the TxD pin as being controlled by the SCI
0 = Transmitter disabled
1 = Transmitter enabled

RE: Receiver enable bit. RE enables the SCI receiver
0 = Receiver disabled
1 = Receiver enabled

Module Base + 0x_0004

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
Reset	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 1-6. SCI Status Register 1 (SCISR1)

Read: Anytime

Write: Has no meaning or effect

TDRE: Transmit data register empty flag.
0 = No byte was transferred to transmit shift register
1 = Byte transferred to transmit shift register; transmit data register empty

TC: Transmit complete flag.
0 = Transmission in progress
1 = No transmission in progress

RDRF: Receive data register full flag.
0 = Data not available in SCI data register
1 = Received data available in SCI data register

IDLE: Idle line flag. Is set when 10 consecutive logic 1s (if M=0) or 11 consecutive logic 1s (if M=1) appear on the receiver input.
0 = Receiver input is either active now
1 = Receiver input has become idle

OR: Overrun flag. Is set when the software fails to read the SCI data register before the receive shift register receives the next frame.
0 = No overrun
1 = Overrun

NF: Noise flag. Is set when the SCI detects noise on the receiver input. NF bit is set during the same cycle as the RDRF flag but does not get set in the case of an overrun
0 = No noise
1 = Noise

FE: Framing error flag.
0 = No framing error
1 = Framing error

PF: Parity error flag. Is set when the parity enable bit is set and the parity of the received data does not match the parity type bit
0 = No parity error
1 = Parity error

Module Base + 0x_0006

	7	6	5	4	3	2	1	0
R	R8	T8	0	0	0	0	0	0
W								
Reset	0	0	0	0	0	0	0	0

Module Base + 0x_0007

	7	6	5	4	3	2	1	0
R	R7	R6	R5	R4	R3	R2	R1	R0
W	T7	T6	T5	T4	T3	T2	T1	T0
Reset	0	0	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 1-8. SCI Data Registers (SCIDRH and SCIDRL)

Read: Anytime; reading accesses SCI receive data register

Write: Anytime; writing accesses SCI transmit data register; writing to R8 has no effect

R8: Received Bit 8. R8 is the ninth data bit received when the SCI is configured for 9-bit data format (M=1)
T8: Transmit Bit 8. T8 is the ninth data when M=1

R[7:0],T[7:0]: Received Bits/Transmit Bits. Received and transmit Bits 7-0 for 9-bit or 8-bit formats.

Data Format

The SCI uses the standard NRZ mark/space data format shown below:

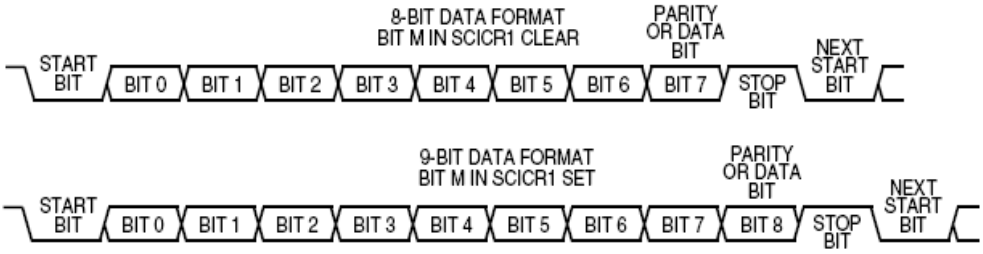


Figure 1-10. SCI Data Formats

Each data character is contained in a frame that includes a start bit, eight or nine data bits, and a stop bit. Clearing the M bit in SCI control register 1 configures the SCI for 8-bit data characters. A frame with eight data bits has a total of 10 bits, and a frame with nine data bits has a total of 11 bits.

Examples of 8-data bit formats:

Start Bit	Data Bits	Parity Bit	Stop Bit
1	8	0	1
1	7	1	1

Examples of 9-data bit formats:

Start Bit	Data Bits	Parity Bit	Stop Bit
1	9	0	1
1	8	1	1

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)
                       | | | | | \______ Receiver disabled
                       | | | | \________ Transmitter enabled
                       | | | \__________ No IDLE Interrupt
                       | | \____________ No Receiver 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"
#include "DBug12.h"
#define enable() asm(" cli")
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 = 0x24; /* 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  */
    while ((SCI1SR1 & 0x20) == 0); /* Wait for data available */
    data[i] = SCI1DRL;
    i = i+1;
    return;

}
```