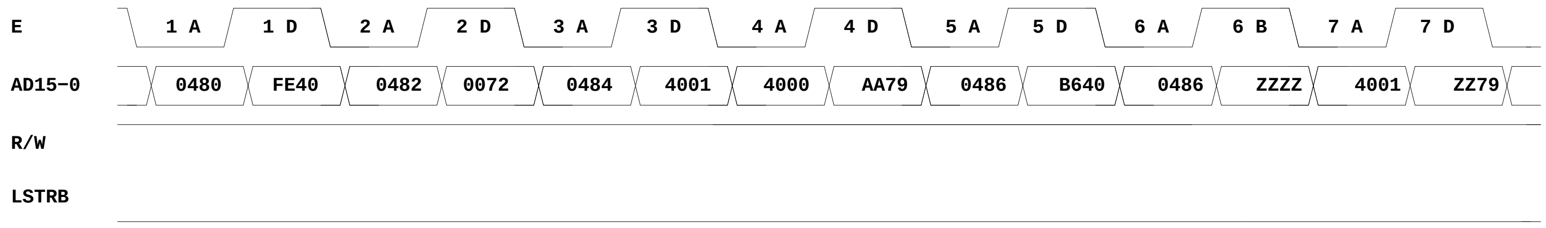
Consider a 9S12 executing the following program loop:

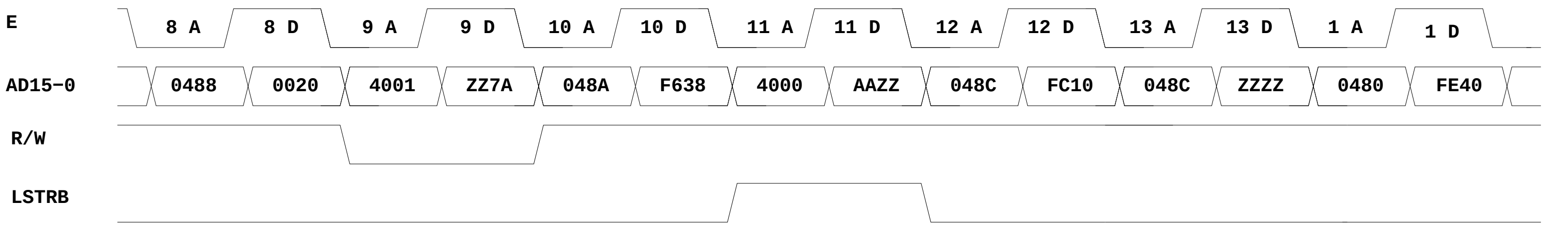
```
                            org    $480
0480  FE4000      l1:       ldx    $4000      % 3 cycles
0483  724001                inc    $4001      % 4 cycles
0486  B64000                ldaa   $4000      % 3 cycles
0489  20F5                  bra    $l1        % 3 cycles
```

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:





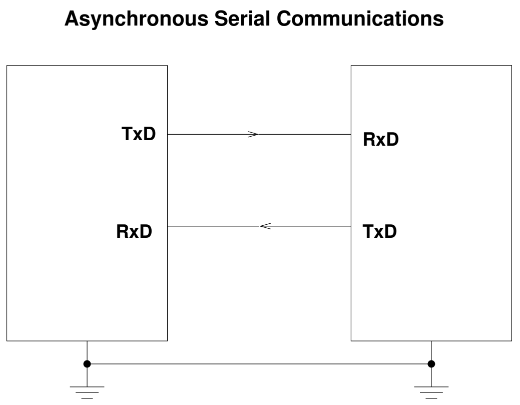
Here is what happens cycle by cycle:

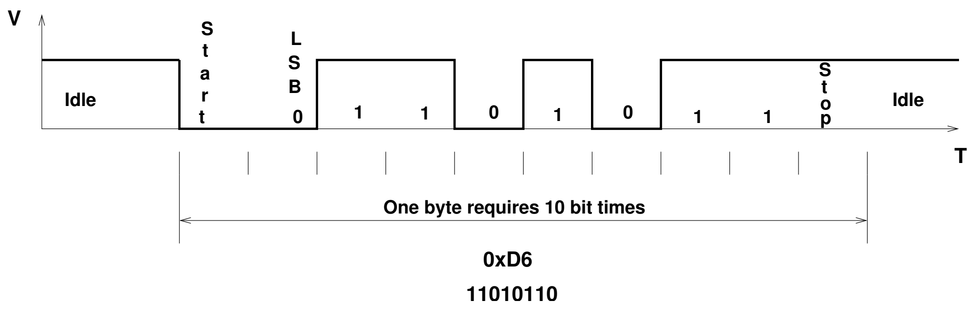
1. 9S12 does a 16 bit read from address \$0480. The memory returns \$FE40, the first two bytes of the `ldx $4000` instruction.
2. 9S12 does a 16-bit read from address \$0482. The memory returns \$0072, the third byte of the the `ldx $4000` instruction and the first byte of the `inc $4001` instruction.
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 specical bit (the stop bit)
- The start bit and the stop bit are used to synchronize the data transfer

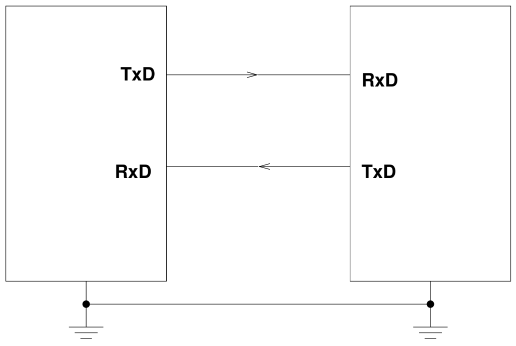


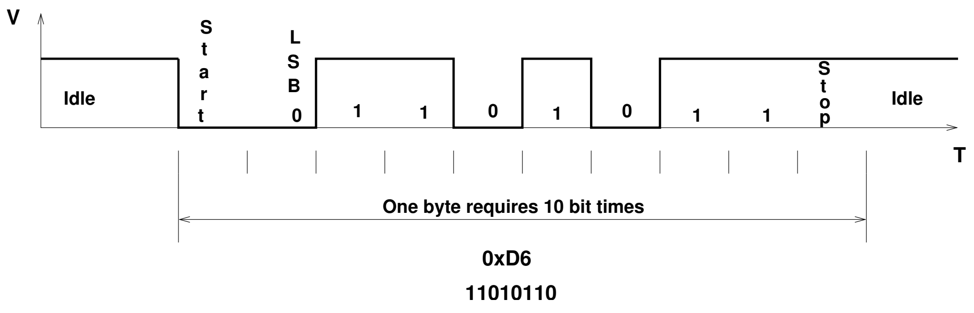


Asynchronous Data Transfer

- The reciever 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 Communications



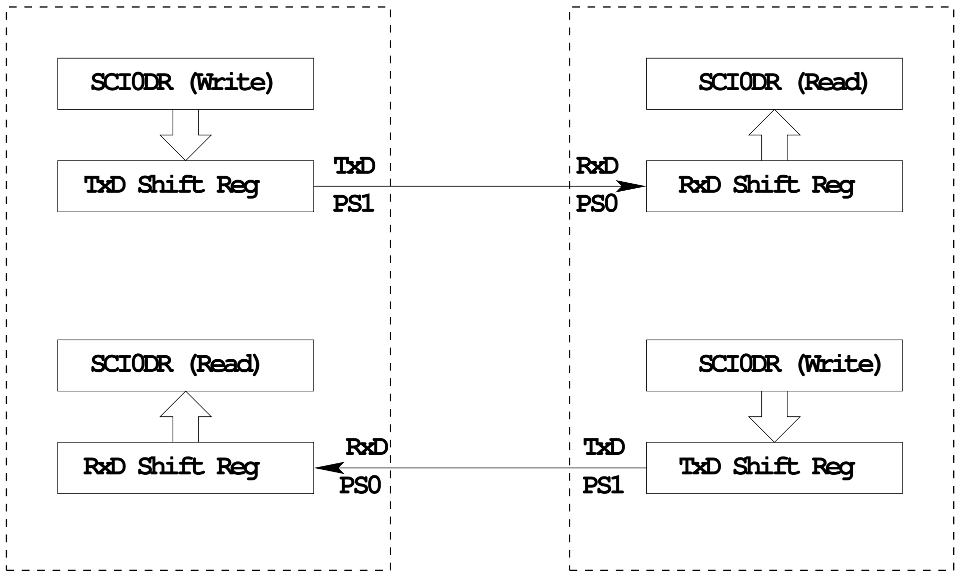


Parity in Ascyncronous Serial Transfers

- 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 tranmit 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

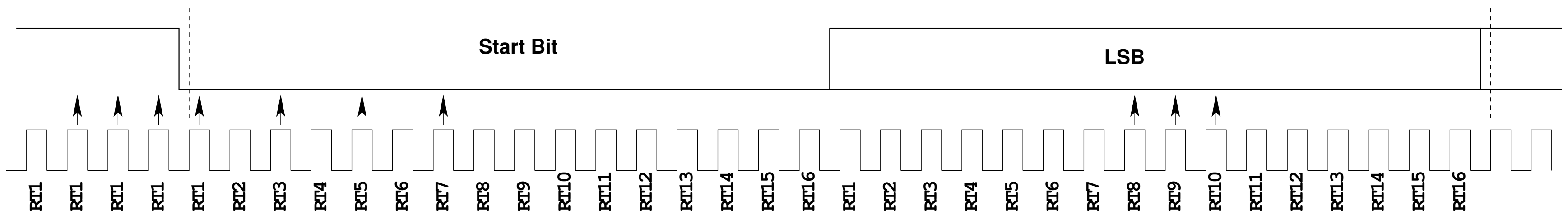
Timing in Asynchronous Data Transfers

- 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 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 reciever 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 COMMUNIATIONS

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

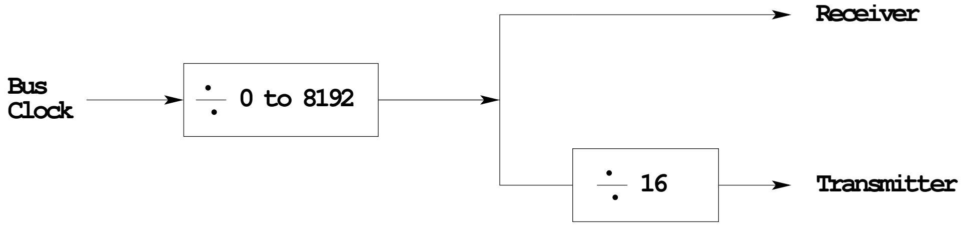
Odd parity -- the number of ones in data word is odd

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

$$mbox\texttt{SCIBaudRate} = \frac{\texttt{Bus Clock}}{16 \times \texttt{SCI1BR[12:0]}}$$



- 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	38,461.5	9,600	0.16
312	76,693.0	38,461.5	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`)
- 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.)

0	0	0	**SBR12**	**SBR11**	**SBR10**	**SBR9**	**SBR8**

SCI1BDH – 0x00D0

SBR7	**SBR6**	**SBR5**	**SBR4**	**SBR3**	**SBR2**	**SBR1**	**SBR0**

SCI1BDL – 0x00D1

LOOPS	SCISWAI	RSRC	**M**	**WAKE**	ILT	**PE**	**PT**

SCI1CR1 – 0x00D2

TIE	TCIE	**RIE**	ILIE	**TE**	**RE**	**RWU**	SBK

SCI1CR2 – 0x00D3

TDRE	TC	**RDRF**	IDLE	**OR**	**NF**	**FE**	**PF**

SCI1SR1 – 0x00D4

0	**0**	**0**	**0**	**0**	**BRK13**	**TXDIR**	RAF

SCI1SR2 – 0x00D5

R8	**T8**	**0**	**0**	**0**	**0**	**0**	**0**

SCI1DRH – 0x00D5

R7/T7	**R6/T6**	**R5/T5**	**R4/T4**	**R3/T3**	**R2/T2**	**R1/T1**	**R0/T0**

SCI1DRL – 0x00D7

```
                Example program using the SCI Transmitter



#include <iodp256.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 Tranmit Complete Interrupt
                        \__________________ No Tranmit 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 "db12.h"
#include <iodp256.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
                        | | | | | | | |
                        | | | | | | | \____ 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)
                        | | | | | \________ Reciever enabled
                        | | | | \__________ Transmitter disabled
                        | | | \____________ No IDLE Interrupt
                        | | \______________ Reciever Interrupts used
                        | \________________ No Tranmit Complete Interrupt
                        \__________________ No Tranmit Ready Interrupt
                     */

    DB12FNP->SetUserVector(UserSCI1,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 interrrupt */
    data[i] = SCI1DRL;
    i = i+1;
    return;
}
```