Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Simulating an RS-232 UART using the Synchronous Serial Ports on the ADSP-21xx Family DSPs

Last Modified : 4/21/1999
Submitted by : Dan L & Greg G.

Introduction

It is possible to use the synchronous serial ports on the ADSP-21xx digital signal processors to perform bi-directional RS-232 communications.

An RS-232 port has two data signals (data transmit and data receive), and a host of hand-shaking signals (DTR,DSR,CTS,RTS). There is, however, no clock signal and no framing signal which makes it difficult to connect it to ports which require a clock and frame sync. This application note will present a method which allows the synchronous serial ports to communicate in such a manner. To fully understand the content of this document, it is recommended that the reader review Chapter 5, Serial Ports of the ADSP-21xx Family User's Manual.

An Overview of RS-232 Communications

In a typical RS-232 interface, data is transmitted at a predetermined bit rate – this is commonly referred to as the BAUD rate (bits per second). Since there is no clock, both devices need to know this information before data can be effectively communicated. There are also framing and parity bits included in each data word. These bits are commonly referred to as start-bits, stop-bits and parity-bits. RS-232 data can be between 5 and 8-bits and is always sent LSB first.

The start and stop bits occur at the beginning and end of the data transmission respectively.

A parity bit can be included in the transmission and there are 5 parity options in the RS-232 standard : even, odd, mark, space or none. If even parity is used, then the last data bit transmitted will be a logical1 if the data transmitted had an odd amount of '1' bits. If odd parity is used, then the last data bit transmitted will be a logical 1 if the data transmitted had an even amount of '1' bits. If mark parity is used, then the last transmitted data bit will always be a logical 1. If space parity is used, then the last transmitted data bit will always be a logical 0. If no parity is used, then there is no parity bit transmitted.

There are 3 options for the stop-bit. These are 1, 1½ and 2.

So, for the sake of example, let's examine what the transmission of the bit pattern '01010101' over the RS-232 port with a typical RS-232 set-up with 8 data bits, no parity, and one stop-bit. Here, we would see

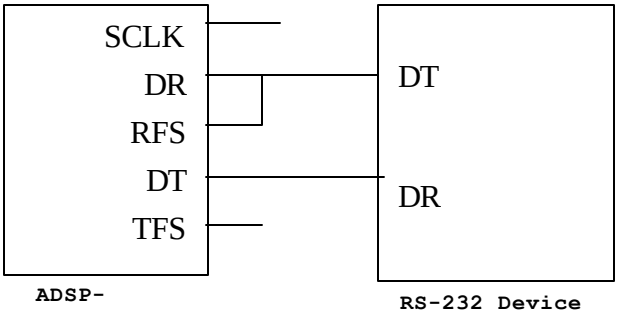
LSB(1 0 1 0 1 0 1 0 1 0) MSB

Or

(start-bit)(8 data bits)(stop bit)

Overview of DSP Interface

The diagram below shows the basic interface between the DSP and the RS-232 port. This diagram does not include level-shifting devices (RS-232 uses 9 Volt signals) nor does it make use of any handshaking signals (DSR, DTR, RTS, CTS).



The serial ports require a slightly different configuration for transmitting and receiving data, therefore, it will be impossible to transmit and receive data at the same time which is common amongst full-duplex serial port communications systems. This problem can be overcome by using a combination of the handshaking signals to ensure that data is not being sent to the DSP while it transmits. For example, we could de-assert the DSR (data send ready) signal on the RS-232 port thereby instructing the device not to send and data while we are transmitting data.

Data Reception

For data reception, we will setup the serial port of the DSP to generate an internal clock, require external frame syncs.

As shown in the diagram above, the clock signal does not connect to anything. We are using the clock internally so that we can sample the incoming data on the RS-232 DR line at a multiple of the bit rate it is being sent at. We are sampling each bit more than once to ensure correct data. Remember, we only know the rate that the data is coming in but have no information regarding the phase relation of the data to our clock. If we over-sample the data with the serial clock, we'll always be sure that at least 2 of the 3 bits we sample for each sent bit is accurate. So, if the RS-232 device we are communicating with is transmitting data at 9600 BAUD (9600 bits/second), we want to sample the data at 3x9600 bits/second, or 28,000 bits/second. The diagram on the next page helps to illustrate this concept.

As shown in the diagram above, the DT signal from the RS-232 port connects to both the DR and the RFS signals on the DSP. This is because we will be using the start-bit of the RS-232 transmission as the RFS signal.

Remember, the DSP only pays attention to the RFS signal while its waiting for a new word *and* when operating in framed mode. While the word is being received, the RFS signal is not polled.

We end up sampling each bit 3 times, plus 2 samples for the start-bit (the 1st bit of the start bit is interpreted as the RFS, not data) plus another 3 samples for the stop bit. So, we need to capture 29 bits of data. The serial ports on the ADSP-21xx family can only read in 16-bit words through the serial ports so we are going to break it up over two words. To achieve this, we will switch the serial port into unframed mode after the first word (word 0) is received and then set it back to framed mode after the second word (word 1) has been received. This, in effect, will allow us to receive one contiguous block of bits from the serial port.

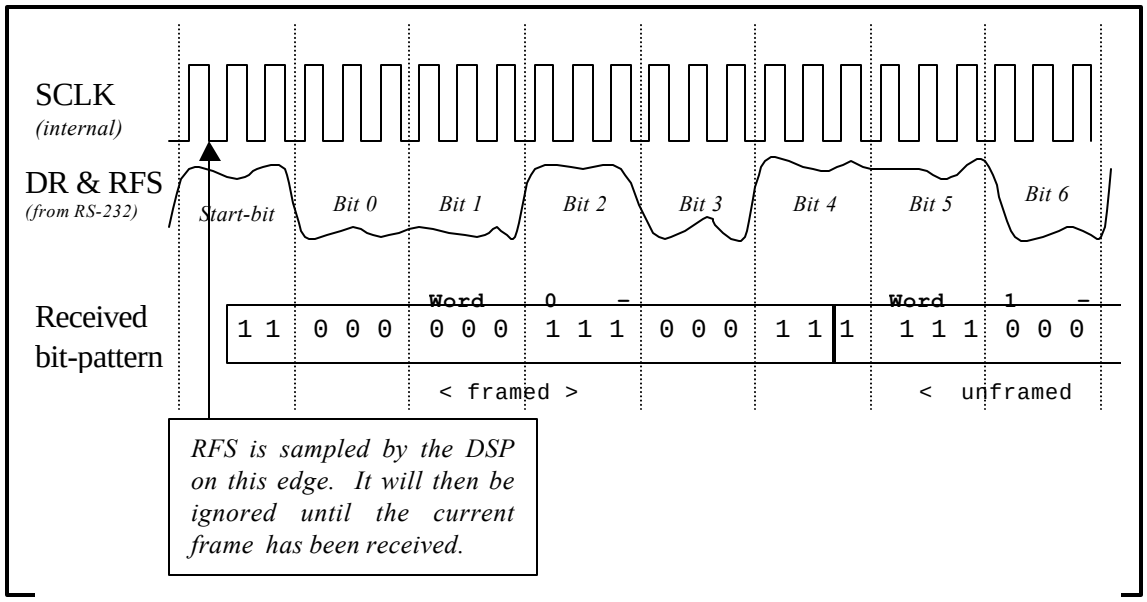


Figure 2 : Example RS-232 word reception using the ADSP-21xx serial ports.

Below is a simple state-diagram to help illustrate this process…

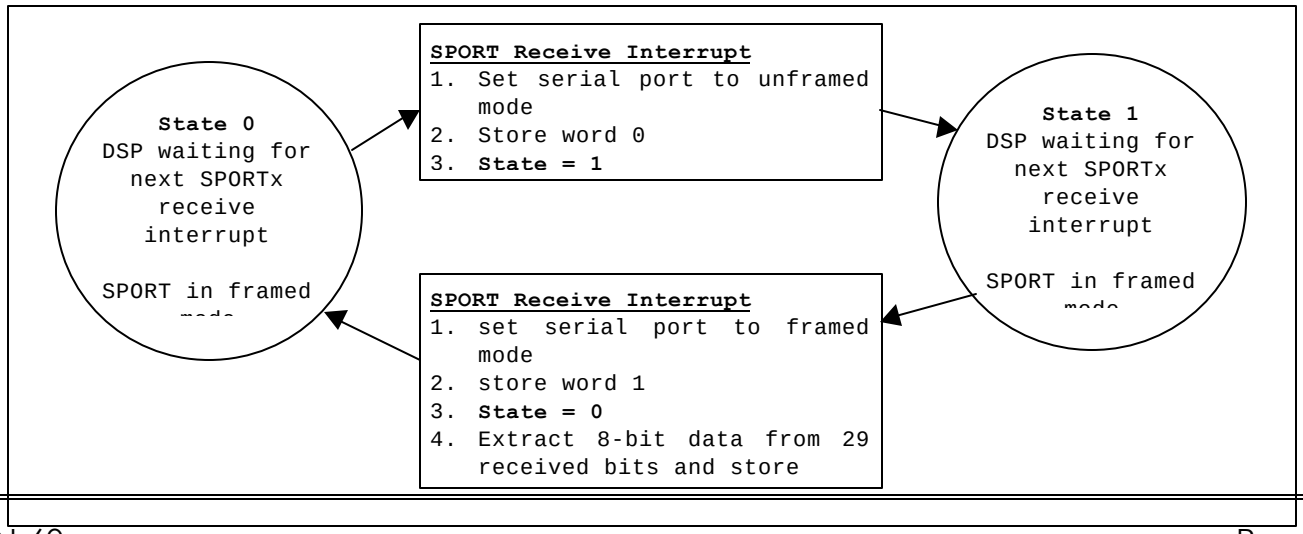


Figure 3 : Receive State Diagram

Data Transmission

The data transmit side of this interface is far simpler than the receive side. Again, we are using the same hardware configuration but on transmit side, we have DT from the DSP connected to DR of the RS-232 device.

We will set up the serial port for internal clock, but also internal frame sync. When transmitting, the clock will be set at the actual BAUD rate instead of a higher multiple. We will also need to manually insert start, stop and parity bits in as well. So, if we want to transmit the following bit pattern:

b#01010101

We would need to use the shifter to add the start and stop bits. What we end up with is:

B#1010101010

The start and stop bits here are underlined and we are using no parity.

Code Examples

The following code example was written for an ADSP-218x part but can be easily modified to support non-8x parts like the ADSP-2101. This code performs RS-232 reads and writes and also calculates and generates parity information.

A soft version of this code can be found on our ftp site at:

ftp.analog.com/pub/dsp/app_note/UART

```
.MODULE/RAM/ABS=0x00 UART_21xx;

{ software uart state machine definitions }
#define UartTxStateInactive       0
#define UartTxStateWord1                1
#define UartTxStateWord2                2
#define UartTxStateWord3                3
#define UartTxStateError                4

#define UartRxStateWord1                0
#define UartRxStateWord2                1
#define UartRxStateError                2

#define TxLowerLimit           0x30
#define TxUpperLimit           0x7f

{ serial port control register stuff }
#define InvertReceiveFrameSync           6
#define InternalTransmitFrameSync       9
#define TransmitFrameAlternate           10
```

```
#define TransmitFrameRequired            11
#define ReceiveFrameAlternate            12
#define ReceiveFrameRequired             13
#define InternalSerialClock       14
#define SerialWordLength_16bits          15

#define ReceiveAutoBufferEnable          0
#define TransmitAutoBufferEnable         1

{ processor specific stuff  }
{ memory mapped core registers of the ADSP2181 }

.CONST  SYSCNTL                          =   0x3fff;
.CONST  MEMWAIT                          =   0x3ffe;
.CONST  TPERIOD                          =   0x3ffd;
.CONST  TCOUNT                     =   0x3ffc;
.CONST  TSCALE                     =   0x3ffb;
.CONST  Sport0_Rx_Words1                 =   0x3ffa;
.CONST  Sport0_Rx_Words0                 =   0x3ff9;
.CONST  Sport0_Tx_Words1                 =   0x3ff8;
.CONST  Sport0_Tx_Words0                 =   0x3ff7;
.CONST  Sport0_Ctrl_Reg                  =   0x3ff6;
.CONST  Sport0_Sclkdiv                   =   0x3ff5;
.CONST  Sport0_Rfsdiv                    =   0x3ff4;
.CONST  Sport0_Autobuf_Ctrl       =   0x3ff3;
.CONST  Sport1_Ctrl_Reg                  =   0x3ff2;
.CONST  Sport1_Sclkdiv                   =   0x3ff1;
.CONST  Sport1_Rfsdiv                    =   0x3ff0;
.CONST  Sport1_Autobuf_Ctrl       =   0x3fef;

{ miscellaneous definitions }
#define ZERO      m0       { m0=0 always }
#define ONE       m1       { m1=1 always }

#define TRUE      ONE
#define FALSE     ZERO


{************************************************* }
{                                                   }
{       Variable definitions                        }
{                                                   }
{************************************************* }

.var/dm/ram TxState; { uart 0 tx state variable }
.var/dm/ram TxData;  { data byte to tx     }
.var/dm/ram RxState; { uart 0 rx state variable }
.var/dm/ram RxData;  { data byte to rx     }

.var/ram/dm/circ    TestBuffer[32];  { test capture buffer }
.VAR/RAM/DM RxUpperWord;
.VAR/RAM/DM RxLowerWord;
.VAR/RAM/DM TxUpperrWord;
.VAR/RAM/DM TxLowerWord;

{ To select which serial port to use, either define SPORT0_UART or SPORT1_UART }
```

```
#define SPORT1_UART   { use sport1 as the UART }


{ Here is where we set the BAUD rate for our system }
#define BAUD 2400

{ Set the system clock frequency here - 33.3Mhz for 2181}
.CONST   CLKFREQ    = 33333333;
.CONST   UARTCLK=(CLKFREQ/((3*BAUD)*2)) - 1;

{------------------------------------------------------------------------}
{ CODE STARTS HERE }


jump STARTUP;        rti; rti; rti;     {00: reset }
rti; rti; rti; rti;     {04: IRQ2 }
rti; rti; rti; rti;     {08: IRQL1 }
rti; rti; rti; rti;     {0c: IRQL0 }
jump UartTx;         rti; rti; rti;                     {10: SPORT0 tx }
jump UartRx;         rti; rti; rti;     {14: SPORT1 rx }
rti; rti; rti; rti;     {18: IRQE }
rti; rti; rti; rti;     {1c: BDMA }
jump UartTx;         rti; rti; rti;     {20: SPORT1 tx or IRQ1 }
jump UartRx;         rti; rti; rti;     {24: SPORT1 rx or IRQ0 }
rti; rti; rti; rti;     {28: timer }
rti; rti; rti; rti;     {2c: power down }


STARTUP:

   call InitStuff;   { setup various registers }
   call InitUart;

   IFC = 0x00;      {Clear any pending interrupts  }

#ifdef SPORT0_UART
   ay0 = 0x060;
#else
   ay0 = 0x006;
#endif

   ar = IMASK;
   ar = ar OR ay0;  { enable sport tx and receive ints }
   IMASK = ar;

   ar=0xffff;     { start serial transmit - all 1's }

#ifdef SPORT0_UART
   tx0=ar;
#else
   tx1=ar;
#endif

MainLoop:               { dummy loop, interrupts handle everything }
   nop;
```

```
   jump MainLoop;

{-------------------------------------------------------------------------}
InitUart:

#ifdef SPORT0_UART
  ar=0;
  ar=SETBIT InternalSerialClock of ar;
  ar=SETBIT ReceiveFrameAlternate of ar;
  ar=CLRBIT ReceiveFrameRequired of ar;

  ar=SETBIT InternalTransmitFrameSync of ar;
  ar=SETBIT TransmitFrameAlternate of ar;
  ar=SETBIT TransmitFrameRequired of ar;

  ar=SETBIT InvertReceiveFrameSync of ar;

  ay0=SerialWordLength_16bits;
  ar=ar or ay0;

  dm(Sport0_Ctrl_Reg)=ar;

  ar=UARTCLK;

  dm(Sport0_Sclkdiv)=ar;

  ar=dm(Sport0_Autobuf_Ctrl);
  ar=CLRBIT TransmitAutoBufferEnable of ar;
  ar=CLRBIT ReceiveAutoBufferEnable of ar;
  dm(Sport0_Autobuf_Ctrl)=ar;

  dm(TxState)=ZERO;       { uart state machine counter }
  dm(RxState)=ZERO;

  ar = TxLowerLimit;
  dm(TxData)=ar;

  ar=dm(SYSCNTL);     { enable serial port 0 }
  ar=setbit 12 of ar;
  dm(SYSCNTL)=ar;

#else    { sport 1 uart setup }
  ar=0;
  ar=SETBIT InternalSerialClock of ar;
  ar=SETBIT ReceiveFrameAlternate of ar;
  ar=CLRBIT ReceiveFrameRequired of ar;

  ar=SETBIT InternalTransmitFrameSync of ar;
  ar=SETBIT TransmitFrameAlternate of ar;
  ar=SETBIT TransmitFrameRequired of ar;

  ar=SETBIT InvertReceiveFrameSync of ar;

  ay0=SerialWordLength_16bits;
  ar=ar or ay0;
```

```
  dm(Sport1_Ctrl_Reg)=ar;

  ar=UARTCLK;

  dm(Sport1_Sclkdiv)=ar;

  ar=dm(Sport1_Autobuf_Ctrl);
  ar=CLRBIT TransmitAutoBufferEnable of ar;
  ar=CLRBIT ReceiveAutoBufferEnable of ar;
  dm(Sport1_Autobuf_Ctrl)=ar;

  dm(TxState)=ZERO;       { uart state machine counter }
  dm(RxState)=ZERO;

  ar = TxLowerLimit;
  dm(TxData)=ar;

  ar=dm(SYSCNTL);     { enable serial port 0 }
  ar=setbit 11 of ar;
  ar=setbit 10 of ar;
  dm(SYSCNTL)=ar;
#endif
  rts;

{-------------------------------------------------------------------------}
UartTx:
   { use this as the entry point for the tx isr }
   ena sec_reg;

   m7=dm(TxState);  { remember that m7, i7 are used }
   i7=^UartTxCommands;
   modify(i7,m7);
   call (i7);
   rti;

UartTxCommands:
   jump UartInactive;
   jump UartWord1;
   jump UartWord2;
   jump UartWord3;
   jump UartError;

UartInactive:
   ar=0xffff;  { send out all stop bits...  }
   #ifdef SPORT0_UART
      tx0=ar;     { stay in state 0 - inactive }
   #else
      tx1=ar;
   #endif
   rts;

UartWord1:
   { call xmit;  here is where the other stuff gets called }
   call PrepareTxWords;
   ar = UartTxStateWord2;
   dm(TxState)=ar;   { move on to tx next word }
```

```
   #ifdef SPORT0_UART
      tx0=dm(TxUpperrWord);     { write the word out... }
   #else
      tx1=dm(TxUpperrWord);
   #endif
   rts;

UartWord2:
   ar = UartTxStateWord3;
   dm(TxState)=ar;   { move on to tx next word }
   #ifdef SPORT0_UART
      tx0=dm(TxLowerWord);    { write the word out... }
   #else
      tx1=dm(TxLowerWord);
   #endif
   rts;

UartWord3:
   ar = UartTxStateInactive;
   dm(TxState)=ar;   { move on to inactive tx state when done }
   ar = 0xffff;         { send out dummy stop bits }

   #ifdef SPORT0_UART
      tx0=ar;
   #else
      tx1=ar;
   #endif
   rts;

UartError:
   dm(TxState)=ZERO;      { just clear the state machine... }
   ar=0xffff;
   #ifdef SPORT0_UART
      tx0=ar;
   #else
      tx1=ar;
   #endif

   rts;

PrepareTxWords:
   ar=dm(TxData);
   sr=lshift ar by -8 (hi);   { sr=0000xx00 data byte to xmit }
   mr0=sr0;             { mr0 contains byte to shift out }
   sr=lshift ar by 32 (lo);   { clear sr }
   ar=8;                    { process 8 bits }
   af=pass ar;              { bit counter in af }
   mr1=0;                { parity counter }

txMirrorImage:
   ay0=mr0;             { get the byte... }
   ar=mr0+ay0;              { mr0=mr0<<1      }
   mr0=ar;
   if ac jump txItsAOne;
      ar=sr1;
      jump txGwon;
```

```
txItsAOne:
   ar=mr1+1;
   mr1=ar;    { increment parity count if it's a 'one' }
   ay0=0xe000;
   ar=sr1 or ay0; { ar=111 in msb's this is a 'one' }

txGwon:
   sr=lshift sr0 by -3 (lo);     { 32 bit shift right 3 bits }
   sr=sr or lshift ar by -3 (hi);
   af=af-1;                { decrement bit counter }
   if ne jump txMirrorImage;

   dm(TxUpperrWord)=sr1;  { upper word to transfer }
   ar=sr0;    { get lower word }
   ar=setbit 0 of ar;
   ar=setbit 1 of ar;   { set the stop bits }
txParity:
   af=tstbit 0 of mr1;  { test the parity even/odd }
   if eq jump prepareDone;
      ay0=0x001c;      { xxxx xxxx xxxP PP11 }
      ar=ar or ay0;

prepareDone:
   dm(TxLowerWord)=ar;
   rts;
{---------------------------------------------------------------------------}

{ generate some test data - increment byte to send out }

TxTestData:
   ar=dm(TxData);
   ar=ar+1;
   ay0=TxUpperLimit;
   af=ar-ay0;
   if le jump reload;
   ar=TxLowerLimit;
reload:
   dm(TxData)=ar;
   dm(TxState)=ONE;
   rts;
{---------------------------------------------------------------------------}

UartRx:         { isr for sport 0 rx interrupts }
   ena sec_reg;

   ar =dm(RxState);      { get the present state }
   af =pass ar;
   if ne jump rxWord2;

      #ifdef SPORT0_UART
        dm(RxUpperWord)=rx0;         { store upper word - 1st word received }
      #else
        dm(RxUpperWord)=rx1;         { store upper word - 1st word received }
      #endif
```

```
      dm(RxState)=ONE;     { advance rx state machine - word 2 }

    ar=dm(RxUpperWord);
    dm(i2,m1)=ar;

   #ifdef SPORT0_UART
          ar=dm(Sport0_Ctrl_Reg);    { set up frame sync to re-sync next byte }
          ar=SETBIT ReceiveFrameRequired of ar;
          dm(Sport0_Ctrl_Reg)=ar;
       #else
       ar=dm(Sport1_Ctrl_Reg);    { set up frame sync to re-sync next byte }
          ar=SETBIT ReceiveFrameRequired of ar;
      dm(Sport1_Ctrl_Reg)=ar;
   #endif

       rti;

rxWord2:

   #ifdef SPORT0_UART
          DM(RxLowerWord)=rx0;
   #else
          DM(RxLowerWord)=rx1;
   #endif

    ar=dm(RxLowerWord);
    dm(i2,m1)=ar;


   call unpackRxWords;
!   call Receive;    { main recieve processor... }

   dm(RxState)=ZERO;     { have unpacked rx byte, get ready for next one }

   #ifdef SPORT0_UART
          ar=dm(Sport0_Ctrl_Reg);  { frame sync not required for 2nd word }
          ar=CLRBIT ReceiveFrameRequired of ar;
          dm(Sport0_Ctrl_Reg)=ar;
       #else
          ar=dm(Sport1_Ctrl_Reg);  { frame sync not required for 2nd word }
          ar=CLRBIT ReceiveFrameRequired of ar;
          dm(Sport1_Ctrl_Reg)=ar;
   #endif

       rti;

rxInactive:
   ar=rx0;    { flush rx buffer }
   rti;

unpackRxWords:
   si=dm(RxLowerWord);
   mr0=dm(RxUpperWord);    { get the rx words }
   ar=tstbit 14 of mr0; { check start bit... }
   if ne jump startBitError;
```

```
   ar=8;
   ay0=0;
   af=pass ar, ar=ay0;        { bit counter      }
   mr2=0;
   mr1=0;            { clear parity counter, etc. }
   se = -3;
   sr=lshift si by -3 (lo);    { move center parity bit to lsb }
   ar=tstbit 0 of sr0;
   if eq jump rxParityZero;
      mr2=ONE;            { parity bit is set, else mr2=0 }

rxParityZero:
   sr=sr or lshift mr0 (hi);
   si=sr0;                { entire 32 bits shifted }
   mr0=sr1;

   ar=0;
rxMirror:
   sr=lshift si by -3 (lo);    { move next bit to lsb of sr    }
   sr=sr or lshift mr0 (hi);
   si=sr0;
   mr0=sr1;
   ar=ar+ay0;          { ar=ar<<1              }
   ay0=ar;
   ar=tstbit 0 of sr0;       { bit to shift in to form byte  }
   if eq jump rxItsAZero;

   ar=mr1+1;         { increment parity counter }
   ar=pass 1 ; mr1=ar;

rxItsAZero:
    ar=ar or ay0;
    ay0=ar;               { new bit is installed  }
    af=af-1;             { check bit counter         }
    if gt jump rxMirror;    { do all 8 bits }

    dm(RxData)=ar;
    ar=tglbit 5 of ar;
    dm(TxData)=ar;
    dm(TxState)=ONE;
    dm(i2,m1)=ar;
   rts;

startBitError:
   dm(RxState)=ZERO;
   dm(RxData)=ZERO;    { null for start bit error... }
   rts;
{-------------------------------------------------------------------------}
{ InitStuff - setup various processor registers and initial conditions... }

InitStuff:
   m0=0;
   m1=1;
   l7=0;

   ICNTL = 0x000;    {Configure interrupt format       }
```

```
   i2 = ^TestBuffer;
   l2 = %TestBuffer;

rts;
{-------------------------------------------------------------------------}
.ENDMOD;
```