11.1 OVERVIEW

This chapter describes a software implementation of a Universal Asynchronous Receiver/Transmitter (UART). The UART is implemented as a program running on an ADSP-2101, with the Flag In (FI) and Flag Out (FO) signals used as asynchronous receive and transmit lines. The UART software was developed to provide the following features:

- Full-duplex operation—independent receiver and transmitter
- Double-buffered receive and transmit registers, to allow continuous data flow
- Asynchronous operation—no synchronization required between transmitted and received bit streams
- Programmability—to provide a variety of baud rates and flexible data formats (7 or 8 data bits, 1 or 2 stop bits)

11.2 HARDWARE

A general system configuration is shown in Figure 11.1. The ADSP-21xx is interfaced to an RS-232 line driver chip which is in turn connected to any RS-232-compatible device. The RS-232 line driver is needed to convert the 5-volt logic level of the ADSP-21xx to the proper RS-232 line voltages, and vice versa.

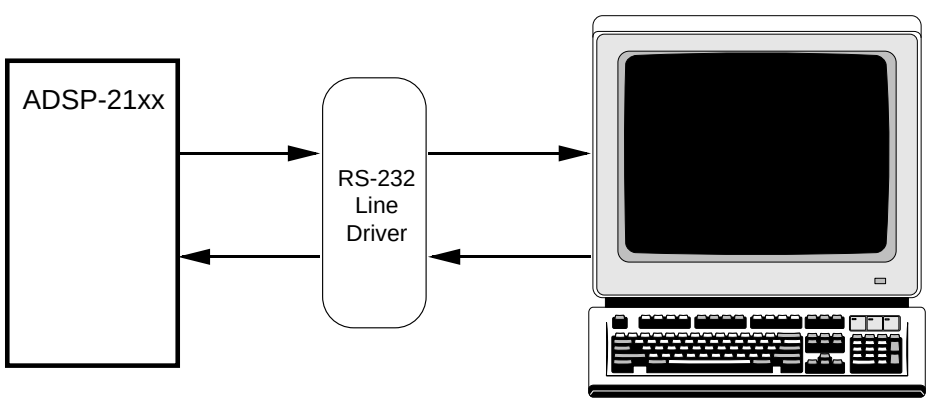
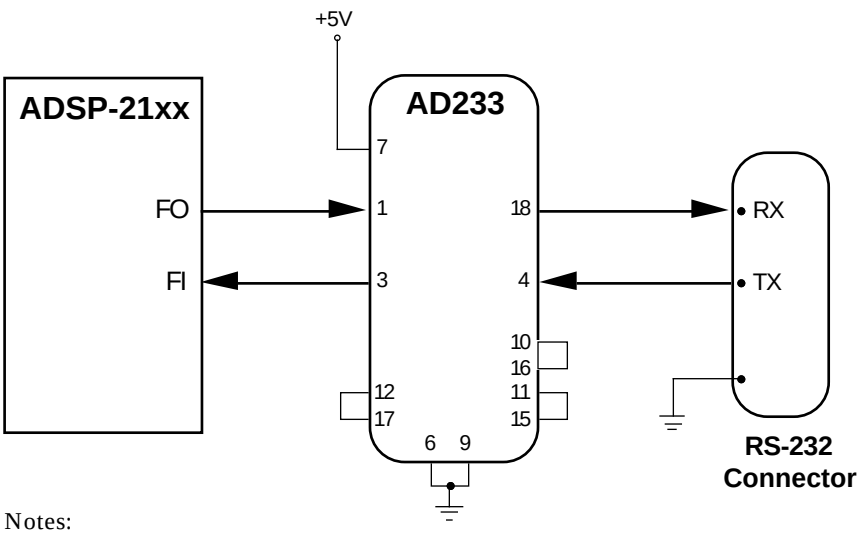


Figure 11.1 General System Configuration

11 Software UART

Figure 11.2 shows a specific example of a hardware implementation for the UART in which an ADSP-2101 processor and AD233 RS-232 Driver / Receiver are used. The Flag In (FI) and Flag Out (FO) pins of the ADSP-21xx are used as independent receive and transmit lines. The AD233 is an ideal choice for the line driver because it requires no external capacitors and is powered by a single 5V supply.



Notes:

1. Pins 2, 5, 8, 13, 14, 19, and 20 on the AD233 have no connection.
2. For autobaud operation, connect the ADSP-21xx IRQ2 pin to the FI pin.

Figure 11.2 Example System Configuration

11.3 SOFTWARE

Two separate sets of receive and transmit registers are used in this UART implementation. One set is used by the UART to to clock data words in and out. The other set is used to read from (and write to) the UART, thus providing access to the UART while it is in operation. This allows a continuous data flow.

The UART program can handle a wide variety of baud rates and data formats by modifying the timer and shifter settings of the ADSP-21xx. Autobaud operation is also possible.

The ADSP-21xx's on-chip timer generates interrupts at three times the baud rate, providing enough clock resolution to handle asynchronously transmitted and received data streams. The timer's interrupt rate may be increased to provide additional clock resolution, if desired—this is described below under "Initialization & Timer Interrupt Routines."

11.3.1 Program Flow

The UART program consists of six subroutines:

- Initialization routine
- Timer interrupt routine
- Transmit character routine
- Receive character routine
- Enable receive routine
- Disable receive routine

The *initialization routine* must be called after a system reset. The *timer interrupt routine* is the heart of the UART program. It transmits and receives bits when necessary. The receive portion of this routine can be disabled by executing the disable receive routine. The timer interrupt routine prepares the UART for use by: 1) setting up the timer to generate its interrupt at the proper rate, 2) configuring SPORT1 (Serial Port 1) as the FI/FO pins (Flag In, Flag Out), 3) setting flags to indicate that the UART is not busy, and 4) clearing any pending interrupts and enabling the timer.

The *transmit character routine* waits for any previously transmitted character to be completely sent, and then sends the next character.

The *receive character routine* waits for a character to be completely received, and then gets the character and returns it to the calling program.

The *enable receive routine* enables the UART receive portion of the timer interrupt routine.

The *disable receive routine* disables the UART receive portion of the timer interrupt routine.

11 Software UART

11.3.2 Initialization & Timer Interrupt Routines

For the following discussion of each subroutine, refer to the code shown in Listing 11.1.

```
{************************************************************
  ADSP-2101 Software UART                                UART.DSP

This program uses FLAG_IN, FLAG_OUT and the TIMER of the ADSP-2101 to
interface to an RS-232 asynchronous serial device such as a VT100 terminal.

for example:

 ADSP-2101 FLAG_OUT ———> AD233 ———> RS232 RX

 ADSP-2101 FLAG_IN  <——— AD233 <——— RS232 TX

                   (TIMER maintains baudrate)

Parameters bits/word, baudrate, stopbits & parity are user-programmable.
An RS-232 line driver chip (such as the AD233) can be used to electrically
interface +5 VDC to the RS-232 line voltage levels.

The operation of the transmitter setup routine is completely independent on
the receiver setup routine operation. Although both tx and rx use the same
timer as a master clock source, the transmitted bits need not be in sync
with the received bits. The default state of the reciever is OFF, so the
“turn_rx_on” subroutine must be used to enable RX.

Calling Argument:
  For autobaud load the baud constant:
    dm(baud_period)=(Proc_frequency/(3*Baudrate))-1

Useful Subroutines:
 init_uart        Must be called after system reset.
 get_char_ax1     Waits for RX input and returns with it in AX1.
 out_char_ax1     Waits for last TX output and transmits data from AX1.
 turn_rx_on       Must be called to enable the receipt of RX data.
 turn_rx_off      Can be used to ignore input RX data.

Useful Flag:
 DM(flag_rx_ready)      If this DM location is all ones it indicates that
                        the UART is ready to receive a new word. If it is
                        zero then data is being received. Can be used for
                        xon/xoff flow control.

************************************************************}
```

Software UART 11

```
.module/boot=1    UART;

{The constants below must be changed to modify the UART parameters}

.const tx_num_of_bits = 10;        {start bits + tx data bits + stop bits}
.const rx_num_of_bits = 8;         {rx data bits (start&stop bits not counted)}
.const RX_BIT_ADD = 0x0100;        {= 1<<rx_num_of_bits }
.const TX_BIT_ADD = 0xfe00;        {= 0xffff<<(tx data bits+1)}

{___These constants can be used if autobaud is not needed___}

.const PERIOD=74;          {13 & 57600}   {PERIOD=(Proc_freq/(3*Baudrate))-1}
{.const PERIOD=112;}       {13 & 38400}   {PERIOD=(Proc_freq/(3*Baudrate))-1}
{.const PERIOD=225;}       {13 & 19200}   {PERIOD=(Proc_freq/(3*Baudrate))-1}
{.const PERIOD=450;}       {13 & 9600}    {PERIOD=(Proc_freq/(3*Baudrate))-1}

{_____Definitions for memory-mapped control registers_____}

.const TSCALE = 0x3ffb;
.const TCOUNT = 0x3ffc;
.const TPERIOD = 0x3ffd;
.const System_Control_Reg = 0x3fff;

{_____________________________________________________________}

.entry init_uart;           {UART initialize baudrate etc.}
.entry out_char_ax1;        {UART output a character}
.entry get_char_ax1;        {UART wait & get input character}
.entry turn_rx_on;          {UART enable the rx section}
.entry turn_rx_off;         {UART disable the rx section}
.entry process_a_bit;       {UART timer interrupt routine for RX and TX}

.global flag_rx_ready;
.global baud_period;

.var flag_tx_ready;            {flag indicating UART is ready for new tx word}
.var flag_rx_ready;            {flag indicating UART is ready to rx new word}
.var flag_rx_stop_yet;         {flag tells that a rx stop bit is not pending}
.var flag_rx_no_word;          {indicates a word is not in the user_rx_buffer}
.var flag_rx_off;              {indicates a that the receiver is turned off}
.var timer_tx_ctr;             {divide by 3 ctr, timer is running @ 3x baudrate}
.var timer_rx_ctr;             {divide by 3 ctr, timer is running @ 3x baudrate}
.var user_tx_buffer;           {UART tx reg loaded by user before UART xmit}
.var user_rx_buffer;           {UART rx reg read by user after word is rcvd}
.var internal_tx_buffer;       {formatted for serial word, adds start&stop bits}
                               {'user_tx_buffer' is copied here before xmission}
.var internal_rx_buffer;
.var bits_left_in_tx;          {number of bits left in tx buffer (not yet clkd out) }
.var bits_left_in_rx;          {number of bits left to be rcvd (not yet clkd in) }
.var baud_period;              {loaded by autobaud routine}
```

(listing continues on next page)

11 Software UART

```
{__________________Initializing subroutine_________________}

init_uart:

  ax0=0;
  dm(TSCALE)=ax0;             {decrement TCOUNT every instruction cycle}

  ax0=dm(baud_period);        {from autobaud or use constant: ax0=PERIOD;}
                              {...and comment in the appropriate constant}
  dm(TCOUNT)=ax0;
  dm(TPERIOD)=ax0;            {interrupts generated at 3x baudrate}
  ax0=0;
  dm(System_Control_Reg)=ax0; {no bmwait,pmwait states, SPORT1=FI/FO}

  ax0=1;
  dm(flag_tx_ready)=ax0;      {set the flags showing that UART is not busy}
  dm(flag_rx_ready)=ax0;
  dm(flag_rx_stop_yet)=ax0;
  dm(flag_rx_no_word)=ax0;
  dm(flag_rx_off)=ax0;        {rx section off}

  set flag_out;               {UART tx output is initialized to high}
  ifc=0x003f;                 {clear all pending interrupts}
  nop;                        {wait for ifc latency }
  imask=b#000001;             {enable TIMER interrupt handling}
  ena timer;                  {start timer now}
  rts;

{_____________________process_a_bit_____________________
                (TIMER interrupt routine)

This routine is the heart of the UART. It is called every timer
interrupt (i.e. 3x baudrate). This routine will xmit one bit at a time
by setting/clearing the FLAG_OUT pin of the ADSP-2101. This routine
will then test if the UART is already receiving. If not it will test
flagin (rx) for a start bit and place the UART in receive mode if true.
If already in receive mode it will shift in one bit at a time by
reading the FLAG_IN pin. Since the internal timer is running at 3x
baudrate, bits need only be transmitted/received once every 3 timer
interrupts.
_____________________________________________________________}

process_a_bit:

  ena sec_reg;                    {Switch to background register set}
  ax0=dm(flag_tx_ready);          {if not in “transmit”, go right to
                                    “receive”}
  ar=pass ax0;
  if ne jump receiver;
```

```
{_________Transmitter Section________}

  ay0=dm(timer_tx_ctr);             {test timer ctr to see if a bit}
  ar=ay0-1;                         {is to be sent this time around}
  dm(timer_tx_ctr)=ar;              {if no bit is to be sent}
  if ne jump receiver;              {then decrement ctr and return}

  sr1=dm(internal_tx_buffer);       {shift out LSB of internal_tx_buffer}
  sr=lshift sr1 by -1 (hi);         {into SR1.  Test the sign of this bit}
  dm(internal_tx_buffer)=sr1;       {set or reset FLAG_OUT accordingly}
  ar=pass sr0;                      {this effectively clocks out the}
  if ge reset flag_out;             {word being xmitted one bit at a time}
  if lt set flag_out;               {LSB out first at FLAG_OUT.}

  ay0=3;                            {reset timer ctr to 3, i.e. next bit}
  dm(timer_tx_ctr)=ay0;             {will be sent after 3 timer interrupts}

  ay0=dm(bits_left_in_tx);          {number of bits left to be xmitted}
  ar=ay0-1;                         {is now decremented by one,}
  dm(bits_left_in_tx)=ar;           {indicating that one is now xmitted}
  if gt jump receiver;              {if no more bits left, then ready}

  ax0=1;                            {flag is set to true indicating}
  dm(flag_tx_ready)=ax0;            {a new word can now be xmitted}

{_________Receiver Section________}

receiver:
  ax0=dm(flag_rx_off);              {Test if receiver is turned on}
  ar=pass ax0;
  if ne rti;

  ax0=dm(flag_rx_stop_yet);         {Test if finished with stop bit of}
  ar=pass ax0;                      {last word or not. if finished then}
  if ne jump rx_test_busy;          {continue with check for receive.}

  ay0=dm(timer_rx_ctr);             {decrement timer ctr and test to see}
  ar=ay0-1;                         {if stop bit period has been reached}
  dm(timer_rx_ctr)=ar;              {if not return and wait}
  if ne rti;

  ax0=1;                            {if stop bit is reached then reset}
  dm(flag_rx_stop_yet)=ax0;         {to wait for next word}
  dm(flag_rx_ready)=ax0;

  ax0=dm(internal_rx_buffer);       {copy internal rx buffer}
  dm(user_rx_buffer)=ax0;           {to the user_rx_buffer}
```

(listing continues on next page)

```
  ax0=0;                          {indicated that a word is ready in}
  dm(flag_rx_no_word)=ax0;        {the user_rx_buffer}
  rti;

rx_test_busy:
  ax0=dm(flag_rx_ready);          {test rx flag, if rcvr is not busy}
  ar=pass ax0;                    {receiving bits then test for start.If it}
  if eq jump rx_busy;             {is busy, then clk in one bit at a time}

  if flag_in jump rx_exit;        {Test for start bit and return if none}

  ax0=0;
  dm(flag_rx_ready)=ax0;          {otherwise, indicate rcvr is now busy}
  dm(internal_rx_buffer)=ax0;     {clear out rcv register}

  ax0=4;                          {Timer runs @ 3x baud rate, so rcvr}
  dm(timer_rx_ctr)=ax0;           {will only rcv on every 3rd interrupt.}
                                  {Initially this ctr is set to 4.}
                                  {This will skip the start bit and will}
  ax0=rx_num_of_bits;             {allow us to check FLAG_IN at the center}
  dm(bits_left_in_rx)=ax0;        {of the received data bit.}

rx_exit:
  rti;

rx_busy:
  ay0=dm(timer_rx_ctr);           {decrement timer ctr and test to see}
  ar=ay0-1;                       {if bit is to be rcvd this time around}
  dm(timer_rx_ctr)=ar;            {if not return, else receive a bit}
  if ne rti;

rcv:                              {Shift in rx bit}
  ax0=3;                          {reset the timer ctr to 3 indicating}
  dm(timer_rx_ctr)=ax0;           {next bit is 3 timer interrupts later}

  ay0=RX_BIT_ADD;
  ar=dm(internal_rx_buffer);
  if not flag_in jump pad_zero;   {Test RX input bit and}
  ar=ar+ay0;                      {add in a 1 if hi}

pad_zero:
  sr=lshift ar by -1 (lo);        {Shift down to ready for next bit}
  dm(internal_rx_buffer)=sr0;

  ay0=dm(bits_left_in_rx);        {if there are more bits left to be rcvd}
  ar=ay0-1;                       {then keep UART in rcv mode}
  dm(bits_left_in_rx)=ar;         {and return}
  if gt rti;                      {if there are no more bits then ...}
                                  {...that was the last bit }
```

```
  ax0=3;                              {set timer to wait for middle of the}
  dm(timer_rx_ctr)=ax0;               {stop bit}
  ax0=0;                              {flag indicated that uart is waiting}
  dm(flag_rx_stop_yet)=ax0;           {for the stop bit to arrive}
  rti;

{_____________invoke_UART_transmit subroutine_____________

This is the first step in the transmit process. The user has now loaded
'user_tx_buffer' with the ascii code and has also invoked this routine.
_____________________________________________________________________}

invoke_UART_transmit:

  ax0=3;                              {initialize the timer decimator ctr}
  dm(timer_tx_ctr)=ax0;               {this divide by three ctr is needed}
                                      {since timer runs @ 3x baud rate}

  ax0=tx_num_of_bits;                 {this constant is defined by the}
  dm(bits_left_in_tx)=ax0;            {user and represents total number of}
                                      {bits including stop and parity}
                                      {ctr is initialized here indicating}
                                      {none of the bits have been xmitted}
  sr1=0;
  sr0=TX_BIT_ADD;                     {upper bits are hi to end txmit with hi}
  ar=dm(user_tx_buffer);              {transmit register is copied into }
  sr=sr or lshift ar by 1 (lo);       {the internal tx reg & left justified}
  dm(internal_tx_buffer)=sr0;         {before it gets xmitted}

  ax0=0;                              {indicate that the UART is busy}
  dm(flag_tx_ready)=ax0;
  rts;

{_________________get an input character____________________

  output:    ax1
  modifies:  ax0
_____________________________________________________________}

get_char_ax1:
  ax0=dm(flag_rx_no_word);
  ar=pass ax0;
  if ne jump get_char_ax1;            {if no rx word input, then wait}

  ax1=dm(user_rx_buffer);             {get received ascii character}
  ax0=1;
  dm(flag_rx_no_word)=ax0;            {word was read}
  rts;
```

(listing continues on next page)

11 Software UART

```
{_________________output a character_____________________

  input:     ax1
  modifies:  ax0, sr1, sr0, ar
_____________________________________________________________}

out_char_ax1:
  ax0=dm(flag_tx_ready);
  ar=pass ax0;
  if eq jump out_char_ax1;          {if tx word out still pending, then wait}
  dm(user_tx_buffer)=ax1;
  call invoke_UART_transmit;        {send it out}
  rts;

{_________________enable the RX section____________________

  modifies:  ax0
_____________________________________________________________}

turn_rx_on:
  ax0=0;
  dm(flag_rx_off)=ax0;
  rts;

{_________________disable the RX section____________________

  modifies:  ax0
_____________________________________________________________}

turn_rx_off:
  ax0=1;
  dm(flag_rx_off)=ax0;
  rts;

.endmod;
```

Listing 11.1 UART.DSP Code

The *initialization routine* (`init_uart`) first sets the timer to generate interrupts at three times the baud rate. (**Note:** autobaud is supported.) This provides sufficient clock resolution to handle the asynchronous data stream in most cases. For applications requiring greater resolution, the interrupt rate may be increased to higher *odd* multiples of the baud rate. The reason only odd multiples of the baud rate are used is to locate the sampling of the bitstream as close to the bit center as possible.

The initialization routine next configures Serial Port 1 as the Flag In and Flag Out pins. It also configures the system for no boot memory or program memory wait states. (The number of wait states may be changed as needed for a particular application.) The routine then sets the following flags: transmit ready (`tx_ready`), receive ready (`rx_ready`), receive stop yet (`rx_stop_yet`), receive no word (`rx_no_word`), and receive section off / disable (`rx_off`). This indicates that the UART is ready to both transmit and receive. However, the UART receive function is initially disabled, and must be enabled by the *enable receive routine* before the UART can receive data. The initialization routine then sets Flag Out (FO) to initialize the UART transmit output high, clears all pending interrupts, and enables the timer interrupt handling. Lastly, the initialization routine enables the timer.

The *timer interrupt routine* (`process_a_bit`) is the central routine of the UART program. It processes the individual bits received and transmitted. The routine is divided into two main sections: a transmitter section and a receive section.

The timer interrupt routine begins by switching to the secondary data register set of the ADSP-21xx. Next, it checks the transmit ready flag to see if the UART is in transmit mode. If transmit mode is active, the transmit section is executed; if not, the routine skips to the receiver section.

The transmitter section decrements the timer transmit counter to determine whether or not a bit must be sent during the current interrupt. (A bit will be sent once every three interrupts in transmit mode, since the timer is running at three times the baud rate.) If no bit is to be sent, the routine jumps to the receiver section. If a bit is to be sent, it is shifted out of the internal transmit buffer, LSB first. This bit will determine whether Flag Out is set high or low. All necessary counters and flags are then updated and the receiver section is executed.

11 Software UART

The receiver section of the timer interrupt routine first checks to see if the receiver is enabled, via the `rx_off` flag. The routine then checks to see if the stop bit of the last word has been received, via the stop bit flag.

If the stop bit has been received, the timer receive counter is decremented and checked to see if the stop bit period has been reached. This must be done because the timer runs at three times the baud rate. If the stop bit of the last word has been received, the routine sets the stop bit flag and the receiver ready flag, copies the internal RX buffer to the user RX buffer, and clears the `rx_no_word` flag to indicate that a word is ready in the user RX buffer.

The routine then returns from the timer interrupt. If the stop bit has not been reached, the routine checks to see if the receiver is either: 1) in the middle of receiving a word, or 2) is waiting for a start bit (via the `rx_ready` flag). If it is waiting for a start bit, the routine checks for one. If one is not found, it exits from the interrupt. If one is found, it sets the appropriate flag to indicate that the receiver is busy, clears out the internal receive register, sets the timer counter to receive on every third interrupt, and sets the "number of bits left to be received" counter.

Initially the timer counter is set to four, not three. This is done to skip the start bit and align the Flag In check at the middle of the received data bit, as shown in Figure 11.3.

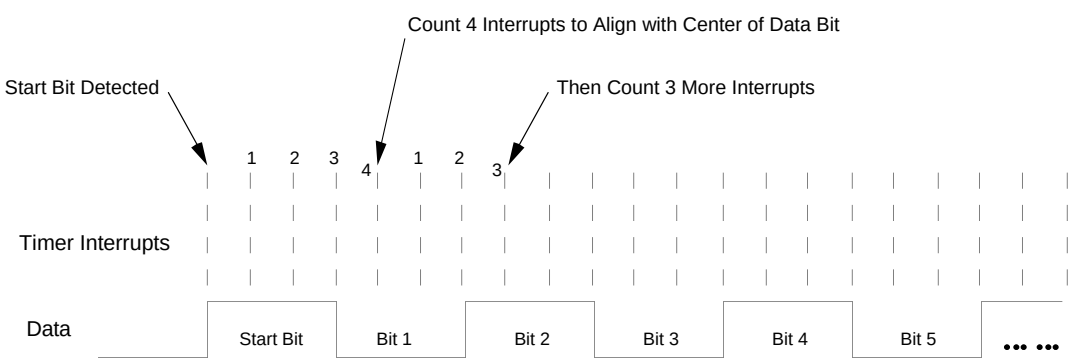


Figure 11.3 Receive Data Timing

The timer routine then exits from the interrupt. If the receiver is in the middle of receiving a word, it decrements and checks the timer RX counter to determine if a bit is to be received during the current interrupt. (A bit will be received every three timer interrupts when receiving, since the timer is running at three times the baud rate.)

If a data bit is not due to be received, it returns from the interrupt; otherwise, it shifts in the bit by checking if Flag In is high or low and shifting in a one or zero to the internal RX buffer. The number of bits left in the RX counter is then decremented, and, if it has expired, a flag is set to indicate that the UART is waiting for a stop bit. The routine then returns from the interrupt.

11.3.3 Transmit & Receive Subroutines

The *transmit character routine* (`out_char_ax1`) first checks the transmit ready flag to determine if a character transmit is still pending. If one is, the routine waits until it is complete. Then the routine copies the character stored in AX1 (by the calling routine) to the user transmit buffer; the invoke_uart_transmit (IUT) subroutine is then invoked.

The IUT subroutine initializes the timer transmit counter to set up operation at the proper baud rate, in this case one-third of the timer interrupt rate since the timer runs at three times the baud rate. The subroutine then initializes the "bits left to be transmitted" counter. Next, the subroutine copies the user transmit register to the internal transmit register and sets a flag indicating that the UART is busy. The IUT subroutine then returns to the transmit character routine, which in turn returns to the calling routine. As described later, the timer interrupt routine handles the actual transmission of this character.

The *receive character routine* (`get_char_ax1`) first checks to see if a character has been received. If not, it waits until one has been received, then returns to the calling routine with the character's ASCII code in AX1.

The *enable receive routine* (`turn_rx_on`) enables the UART to receive by clearing the `rx_off` flag.

(**Note:** The UART program starts with its receive function disabled—it must be activated by calling the enable receive routine in order to receive data. Otherwise, the receive character routine will loop indefinitely, waiting for a character to be received.)

The *disable receive routine* (`turn_rx_off`) disables the UART receive function by setting the `rx_off` flag.

11 Software UART

11.4 BAUD RATES

The worst-case time required for the timer interrupt routine to execute is 55 processor cycles. Since this routine is executed for every timer interrupt, the maximum allowable baud rate can be determined by the formula:

$$\text{Baudrate} = (\text{CLK frequency}) \left(\frac{1}{55 \text{ cycles/interrupt}}\right) \left(\frac{1}{3 \text{ interrupts/baud}}\right)$$

The following table can thus be derived, assuming a timer interrupt rate of three times the baud rate:

Maximum Baud Rate (approx.)	*ADSP-21xx Clock Frequency*
60600	10 MHz
66600	11 MHz
72700	12 MHz
75700	12.5 MHz
78700	13 MHz
101000	16.67 MHz

11.5 AUTOBAUD FEATURE

Listing 11.2 shows `AUTOTEST.DSP`, an example program that automatically selects the proper baud rate for the UART. As noted in the program comments, the interrupt vectors used are for the ADSP-2101, but can easily be modified for other ADSP-21xx processors. Also, the code can be modified for wakeup characters other than the ASCII Bell and for other processor clock frequencies.

It is important to note that in order to use the autobaud feature of the UART, the processor's IRQ2 line, as well as Flag In (FI), must be tied to the serial receive input (RX).

The `AUTOTEST.DSP` program begins by booting page zero and invoking the autobaud program. Autobauding works as follows:

1. The program counts (using the timer) three high baud periods after being awakened.

2. It then compares this amount of time to known amounts of time for different baud rates, and computes the baud rate constant. This constant is left in the AR register and can be used to initialize the UART routine's `baud_period` variable.

The program then boots page one, where the UART is initialized, turned on, and then echoes the characters input.

```
{************************************************************
  ADSP-2101 UART Autobaud Example                AUTOTEST.DSP

This example program waits for a bell character to interrupt IRQ2, then performs an
autobaud on it, loads the autobaud constant, and boots the next page.

************************************************************}

.module/boot=0/abs=0    IO_shell;

{_____________ADSP-2101 Interrupt Vector Table_____________}

chip_reset:    call init_irq; jump main; rti; rti; {Reset Vector}
ext_IRQ2:      jump boot_pg1; rti; rti; rti;       {external IRQ2}
sport0_tx:     rti; rti; rti; rti;                 {sport0 TX}
sport0_rx:     rti; rti; rti; rti;                 {sport1 RX}
s1_tx_IRQ1:    rti; rti; rti; rti;                 {sport1 TX or IRQ1}
s1_rx_IRQ0:    rti; rti; rti; rti;                 {sport1 RX or IRQ0}
timer_done:    rti; rti; rti; rti;                 {Timer not used}

{_____________________main idle loop_____________________}

main:   idle;
        jump main;

{__________initialize interrupt for UART monitor__________}

init_irq:   icntl=0x0007;
            ifc=0x003f;       {make sure that interrupts are clear}
            nop;              {nop is for latency in loading IFC}

            ay0=0x0020;       {enable irq2}
            ar=imask;
            ar=ar or ay0;
            imask=ar;

            mr1=0x0000;       {set flag in and out on sport}
            dm(0x3fff)=mr1;

            set flag_out;     {default state for UART TX}
            rts;
```

(listing continues on next page)

11 Software UART

```
{_______IRQ2 interrupt from PC, Autobaud, Boot page 1________

After the PC monitor sends a Bell character to wakeup the 2101, it waits
for a “1”, counts cycles and waits for a “0”, then selects the
appropriate baud rate. Exit the autoboot with AR = the baud period value
to the uart routine. The maximum baudrate supported is 57600. This
routine can be modified for other wakeup characters and other processor
clock frequencies. It can also be collapsed into one boot page if needed.
The IRQ2 interrupt is used to wakeup the processor; the FLAGIN pin is
used to test the level of rx.

The IRQ2 pin and the FLAG IN pin must be tied to the serial rx input.

Ascii Bell,Start bit,Stop bit = 00000111 0 1

Determine baud constant for the UART = (proc_freq/(3*Baudrate))-1

  9600 = 4062 cycles at 13MHz          if>3046             ar=450
 19200 = 2031 cycles at 13MHz          else if>1523        ar=225
 38400 = 1016 cycles at 13MHz          else if>876         ar=112
 57600 =  676 cycles at 13MHz          else                ar= 74
_____________________________________________________________________}

boot_pg1:   dis timer;            {be sure timer is off}

            ar=0;
            dm(0x3ffb)=ar;        {TSCALE=0, decrement every cycle}
            ar=10000;             {TPERIOD=TCOUNT=10000 to start}
            dm(0x3ffc)=ar;
            dm(0x3ffd)=ar;

wait_1: if not flag_in jump wait_1; {wait for first high bit}
wait_2: if not flag_in jump wait_2; {be sure it’s not a glitch}
wait_3: if not flag_in jump wait_3; {be sure it stays high}

            ena timer;

count:      if flag_in jump count; {determine # cycles for 3 bauds}
            if flag_in jump count; {be sure it’s not a noise spike}

            dis timer;

            ay0=dm(0x3ffc);         {get TCOUNT value}
            ax0=10000;
            ar=ax0-ay0;             {calc elapsed cycles}
            ax0=ar;                 {ax0=3 baud periods}
```

```
            ay0=3046;               {what's the baud rate?}
            ar=ax0-ay0;
            if ge jump b_9600;

            ay0=1523;
            ar=ax0-ay0;
            if ge jump b_19200;

            ay0=876;
            ar=ax0-ay0;
            if ge jump b_38400;

b_57600:    ar= 74; jump baud_done;
b_38400:    ar=112; jump baud_done;
b_19200:    ar=225; jump baud_done;
b_9600:     ar=450;

baud_done: MSTAT=0;                 {clear modes for boot}

            ax0=0x0A58;             {sport 1 in flag/interrupt mode}
            dm(0x3fff)=ax0;         {force a boot to page 1}
.endmod;
```

Listing 11.2 Autobaud Example Program

11.6 CHARACTER ECHO EXAMPLE

Listing 11.3 shows a program called AUTOECHO.DSP that uses the software UART routines to provide a simple example of how to use the UART monitor. The program reads in a character and writes ("echoes") it back out.

The program also uses the autobaud capability of the UART, modified slightly for this example. (The modified autobaud code is included in the AUTOECHO.DSP file, provided on the disk accompanying this handbook.)

The only hardware needed to run this example is an ADSP-2101 EZ-LAB board and an interface board with an RS-232 line driver chip connected to the Flag In (FI) and Flag Out (FO) pins on the EZ-LAB's J2 SPORT Connector. (**Note:** You must supply the input signals to the line driver chip.)

11 Software UART

```
{************************************************************
  ADSP-2101 EZLAB       UART Example            AUTOECHO.DSP

This program uses the software UART routines to provide a simple example of
how to use the UART monitor. The program reads in a character and writes
("echoes") it back out. The program also uses the autobaud capability of
the UART, modified slightly for this example.
************************************************************}

.module/boot=1/abs=0    AUTOEcho;

.external init_uart;          {UART initialize baudrate etc.}
.external turn_rx_on;         {UART enable the rx section of the uart}
.external turn_rx_off;        {UART disable the rx section of the uart}
.external out_char_ax1;       {UART output a character}
.external get_char_ax1;       {UART wait & get input character}
.external process_a_bit;      {UART timer interrupt routine for RX and TX}
.external baud_period;        {UART load with period from autobaud}

{_____________ADSP-2101 Interrupt Vector Table_____________}

jump start; rti; nop; nop;                {Reset Vector}
rti; nop; nop; nop;                       {IRQ2 Interrupt}
rti; nop; nop; nop;                       {SPORT0 Transmit Interrupt}
rti; nop; nop; nop;                       {SPORT0 Receive Interrupt}
rti; nop; nop; nop;                       {SPORT1 Transmit Interrupt}
rti; nop; nop; nop;                       {SPORT1 Receive Interrupt}
jump process_a_bit; rti; nop; nop;        {Timer Interrupt}

{____________________Initialization Routine____________________}

start:   DM(Baud_Period)=AR;           {UART Autobaud}
         CALL Init_UART;               {Initialize UART}
         CNTR=15000;                   {Wait approximately one}
         DO xloop UNTIL CE;            { character to insure last}
xloop:     NOP;                        { one made it through}
         CNTR=15000;
         DO yloop UNTIL CE;
yloop:     NOP;

         CALL Turn_RX_On;              {Enable UART Receive}

{_____________________Main System Loop_____________________}

         DO mloop UNTIL FOREVER;
           CALL Get_Char_AX1;          {Read in character}
           CALL Out_Char_AX1;          {and Echo it back out}
mloop:     NOP;

.endmod;
```

Listing 11.3 Character Echo Program

11.7 PROGRAM FILES

The disk included with this handbook contains the following files for the software UART:

UART.DSP	UART program subroutines
AUTOTEST.DSP	UART example that echoes received characters
AUTOBAUD.DSP	Section of program that automatically selects the correct baud rate
210X.SYS	System Builder source file used to generate the .ACH architecture file
MAKEIT.BAT	Batch file that runs the assembler, linker, and PROM splitter with the files described above

After building the 210x.SYS file, the MAKEIT.BAT file can be used to create the .BNM version of the UART program (called 2101.BNM). The MAKEIT batch program creates this by assembling the UART.DSP, AUTOTEST.DSP, and AUTOECHO.DSP programs.

It then links the assembled versions with the architecture file 210x.ACH to create the executable file 2101.EXE. After this, it runs the PROM splitter on the executable file to generate the file that will be downloaded (2101.BNM).

The UART program and 210X.SYS system file are designed to run on an ADSP-2101 EZ-LAB board connected to an RS-232 line driver chip. The files can be easily modified for other hardware configurations.