Harris Semiconductor

APPNOTE

No. 117 March 1989

Harris Digital

IMPLEMENTING A SOFTWARE UART ON THE HARRIS RTX 2000

This application note illustrates how a UART can be implemented on the Harris RTX 2000 in software. Implementing a software UART has many advantages over the typical hardware implementation. The most obvious advantage is the reduction in system chip count. This reduces system cost and power and increases overall system reliability. The other major advantage of implementing a software UART is the added flexibility. With a software UART, a software modification can change signal encoding formats, signal frequency, or error checking techniques. Thus, if a change is made in the communication protocol after the hardware design is complete, the change will result in a software update rather than a hardware redesign. The only disadvantage to implementing a UART in software is the reduction of available processor band width. The software UART in this example will result in less than a 2% reduction in RTX processing band width when continually transmitting and receiving data at 1200 Baud. This small reduction in band width is not significant when executing on the RTX 2000 which runs at > 10 MIPS.

The UART implemented in this example is designed to allow the RTX 2000 to communicate over a serial RS-232 link to a terminal. This example uses 8 data bits, 1 stop and no parity. It would be straight forward to make these options setable via the software. This was not done in this example in order to focus on the most difficult portion of the problem which is the basic sending and receiving of data. The software UART also contains an automatic Baud rate detection routine which detects the Baud rate by timing the bit period of the incoming serial data. The software UART incorporates both a sixteen character receive and transmit buffer to increase efficiency.

Implementing the UART requires a method to output and input serial data. This example uses one of the RTX 2000 external interrupt lines to input data, and uses the RTX 2000 Boot pin to output data. One of the three RTX 2000 on chip timers is used to generate the correct Baud rate. The only external hardware needed to communicate with RS-232 interface is an RS-232 line driver/receiver used to condition the digital data to the levels required by the RS-232 standard. This can be done with a single IC, and is described in section titled "HARDWARE NECESSARY FOR SOFTWARE UART".

Forth Background Information

All the code in this example is written in Forth. This section contains a brief introduction to some of the concepts of the programming language Forth. It is intended to help people who have never used Forth to read through and understand the code in the examples.

More information on Forth can be obtained from one of the books on Forth listed in Appendix A. Forth is a high level language that was originally developed for embedded real time applications. Forth creates fast compact code which makes it well suited for embedded real time applications.

Forth is a stack based language and uses postfix notation. That is, for all operations, the operands come before the operation. For example the following code would add the numbers 5 and 10:

```
5 10 +
```

To execute this code Forth first puts the 5 on the stack, followed by the 10. The + operation adds the top two values on the stack, leaving only the result on the stack. Thus, the above operation would result in a 15 being pushed on to the stack.

In Forth, procedures or subroutines are called words. The : operator denotes the start of a new definition for a word. The ; operator denotes the end of a definition. The following code defines a word called MINUTES-TO-SECONDS which converts a value from minutes to seconds by multiplying the number on the top of the stack by 60; the result is left on the stack.

```
: MINUTES-TO-SECONDS 60 * ;
```

The following code would execute this word converting 3 minutes to 180 seconds.

```
3 MINUTES-TO-SECONDS
```

The above code pushes a 3 on the stack and executes the word MINUTES-TO-SECONDS, which pushes a 60 on the stack and then multiplies the top two items on the stack. The above operation would result in a 180 being pushed on the stack. Forth uses the stack to pass parameters between words.

The operators @ (read fetch) and ! (read store) are used by Forth to access memory. To read data from memory the address is pushed on the stack and then @ is executed. The @ operation reads the data addressed by the value on the top of the stack from memory and pushes it on the stack.

```
100 @       \ Reads memory location 100 and pushes the
            \ data on stack.
COUNT @     \ Assuming COUNT is a variable, The word
            \ COUNT places the address of COUNT on
            \ the stack, and @ reads the data stored in
            \ COUNT from memory and pushes it on the
            \ stack.
```

The \ operator is a comment in Forth. Everything after \ on a line is ignored. Comments can also be enclosed in parentheses; e.g. (comment goes here).

The ! (Store operation) has two operands. The first operand is the data that is to be written to memory, and second operand is the address where the data is to be stored. For example:

```
20 100 !    \ Writes a 20 to memory location 100.
0 COUNT !   \ Writes a zero to memory addressed by
            \ COUNT.
```

Since Forth is stack oriented, numerous operations are provided for manipulating the stack. Following are a few examples of stack manipulation operations.

```
DUP   ( n -- n n )              \ Duplicates the top
                                \ element on the stack
SWAP  ( n1 n2 -- n2 n1 )        \ Swaps top two ele-
                                \ ments on the stack
DROP  ( n -- )                  \ Drops top element
                                \ from the stack
ROT   ( n1 n2 n3 -- n2 n3 n1 )  \ Rotates top 3 items
                                \ on stack
```

The comments in parentheses are typically used to show the effects of an operation on the stack. The data before the -- indicates the stack status before the operation. The data after the -- indicates the stack status after the operation. The top of stack is always right most in the list. For example, before the SWAP operation n2 represents the data on the top of the stack, after the SWAP n1 is on the top of the stack.

Forth also provides an IF statement. Due to the postfix nature of the language, the syntax of the IF THEN ELSE statement is somewhat different than other languages. The statement has the following format:

```
(condition) IF (execute this code if condition is TRUE )
            ELSE (execute this code if condition is FALSE)
            THEN (continue executing here after if )
```

In Forth TRUE is any non-zero value and FALSE is a zero value. The THEN portion of the IF statement is sometimes confusing to people familiar with other programming languages. In Forth THEN represents the end of the IF statement rather than the portion that is done when the IF test is TRUE. There is a mechanism to define a word ENDIF to perform the same function as THEN. To make the software UART code easier to read for people not familar with Forth, assume that ENDIF has been defined to do the same thing as THEN. Following is an example of an IF statement in Forth.

```
HOURS-WORKED @ 40 <   \ Test if hours worked less
                      \ than 40
IF
  NORMAL-WAGES        \ If Hours worked < 40 call
                      \ word NORMAL-WAGES
ELSE
  OVERTIME-WAGES      \ else call word
                      \ OVERTIME-WAGES
ENDIF
  CALCULATE-TAX       \ When done with IF
                      \ CALCULATE-TAX
```

Forth, like other high level languages, provides several structures for looping. An example of one of them is the BEGIN UNTIL loop.

This is similar to the REPEAT UNTIL loop in PASCAL. Following is the format for the BEGIN UNTIL loop.

```
BEGIN
                ( execute this code until condition is true )
( condition )   UNTIL
```

The following loop reads in data (by executing a word READ-DATA not defined here) and then manipulates the data in some manner (by executing the word MANIPULATE-DATA also not defined here .) This loop continues until the word ?END-OF-DATA pushes a TRUE (non-zero) value on the stack. ?END-OF-DATA is a forth word (again not defined here) that checks the data for some ending condition and if found pushes TRUE on the stack, if the end is not found FALSE is pushed on the stack.

```
BEGIN                 \ Beginning of Loop.
  READ-DATA           \ Execute Word that reads in
                      \ data.
  MANIPULATE-DATA     \ Execute Word to Manipulate
                      \ data.
  ?END-OF-DATA        \ Execute Word to detect end
                      \ of data.
UNTIL                 \ Go back to BEGIN until ?END-
                      \ OF-DATA returns TRUE.
```

Hopefully, the above information has given the reader enough information to read through the Forth code for the sample UART. For more information on Forth there are several Forth books on the market. (See Appendix A for list)

UART Implementation Details

Using the Boot Pin to Transmit Data

The boot pin of the RTX 2000 is a software controlled output that can be set or cleared with a bit in the Configuration Register (CR). The boot pin is set to one by the processor after reset.

The boot pin was intended to be used for memory decoding of a PROM containing the initialization program for a system. The initialization program would only be run after a processor

reset. After initialization is complete, the program, could clear the boot pin to zero causing the address decoding to disable the reset PROM and enable a different PROM or RAM for normal system operation.

If this feature is not used, the boot pin can be used as a general purpose software controlled output bit. This example uses the Boot pin as the serial output for the RS-232 communication. If the Boot pin is used for other purposes, it would be quite simple to use an address on the memory bus or ASIC bus for serial data output.

The Boot pin is controlled by setting or clearing bit 3 of the Configuration Register (CR). (See Harris RTX 2000 Programmer's Reference Manual and RTX 2000 data sheet for more information on the Configuration Register.) Following is the Forth code for a word called XMIT1 that sets the Boot pin to one.

```
: XMIT1     \ Word to Transmit a 1 on the Boot Pin.
  CR@       \ CR Fetch, copies Configuration Register to
            \ top of stack.
  08 OR     \ Set bit 3 of CR stored on stack top to 1.
  CR!       \ CR Store, copies stack top to CR setting
            \ boot pin.
;
```

Similarly, the boot pin can be cleared to zero by ANDing the old configuration register with FFF7 instead of ORing it with 08 as in the above example. The Boot pin is used to transmit serial data by setting and clearing bit 3 of the configuration register at the appropriate Baud rate. (The section titled "Using Internal Timer to Generate Baud Rate" describes how the Baud rate is generated.) To transmit data, the Boot pin is connected to the RS-232 line that transmits serial data to the terminal.

Using an External Interrupt to Receive Data

This example uses one of the RTX 2000 external interrupt lines to receive the serial data. The start bit of the RS-232 data will interrupt the processor indicating serial data is starting to be received. The RTX 2000 provides a mechanism for polling the external interrupt lines. This feature is used to read in the serial data at the appropriate Baud rate, after the start bit has been detected via the external interrupt. This example uses external interrupt 3 (EI3) for serial data input.

The following sequence of steps is necessary to poll an external interrupt line on the RTX 2000.

1) Disable interrupts and save current interrupt mask.
2) Mask all interrupts except external line being polled.
3) Read interrupt vector register (IVR) to see if external line being polled has an interrupt pending. If so, then a one was read; if not a zero was read.
4) Restore current interrupt mask and enable interrupts.

Following is the Forth code for the word POLL-EI3 that implements the above sequence for reading external interrupt 3 (EI3):

```
: POLL-EI3          \ Word to read EI3 leaves result on
                    \ top of stack.
  DISABLE-INTERRUPTS
                    \ Word to disable RTX interrupts.
  IMR@              \ IMR fetch, store current interrupt
                    \ mask register on top of stack.
  FBFF IMR!         \ Store FBFF in IMR, thus masking
                    \ all interrupts except EI3.
  IVR@              \ IVR fetch, reads interrupt vector
                    \ register.
  03FF AND          \ Clear upper 5 bits of interrupt vector.
  A0 =              \ Test to see if interrupt pending is EI3.
                    \ This will leave a 1 on stack if EI3 is 1,
                    \ or zero otherwise.
  SWAP IMR!         \ Restore saved IMR .
  ENABLE-INTERRUPTS
                    \ Word to enable RTX interrupts.
;
```

For more information on the Interrupt Mask Register, Interrupt Vector Register, and Configuration Register see Harris RTX 2000 Programmer's Reference Manual and data sheet. Following are the definitions for the words to enable and disable RTX interrupts.

```
: DISABLE-INTERRUPTS
                    \ Word to disable RTX interrupts
  CR@               \ CR fetch, save current Configura-
                    \ tion Register on Stack
  10 OR             \ Set bit 4 of saved configuration
                    \ register on stack
  CR!               \ CR store, Load data from stack into
                    \ Configuration register
;

: ENABLE-INTERRUPTS
                    \ Word to enable RTX interrupts
  CR@               \ CR fetch, save current Configura-
                    \ tion Register on Stack
  CR!               \ CR store, load configuration
                    \ register with value on stack. This
                    \ can be done since CR bit 4 always
                    \ reads as a zero so you can read
                    \ register and write it back to
                    \ enable interrupts.
;
```

To input serial data, the RS-232 line that transmits data from the terminal is connected to external interrupt 3 of the RTX 2000. When a start bit is transmitted, the RTX 2000 will be interrupted. Next external interrupt 3 will be read (using the word POLL-EI3) at the correct Baud rate to input the serial data.

Using Internal Timer to Generate Baud Rate

The RTX 2000 has 3 internal timer/counters. This example uses one of those timers to generate the correct Baud rate to transmit data and one to receive data. Two timers are used to allow data to be asynchronously transmitted and received in full duplex easily. This could be done with a single timer, however, the code would be more complex. Only one timer would be necessary if half duplex transmission were used.

The timers are configured to decrement with the internal time-base (the processor clock speed ICLK) and interrupt the processor when the are decremented to 0 (after each serial data bit period). To do this, a value determined from the Baud rate is loaded into the timer. When the timer decrements to zero the processor is interrupted indicating it is time to read or transmit the next bit. The following formula is used to determine the correct count to load in the timer.

$$\frac{\text{Processor Speed (\#Cycles/sec)}}{\text{Baud Rate (Bits/sec)}} = \text{Timer Count (\#Cycles/Bit)}$$

For example, assume the RTX 2000 is running at 10 MHz (ICLK) and the desired Baud rate is 1200. The value to be loaded into the Timer/Counter preload register would be 10,000,000/1200 = 8,333 = 208DH. The following Forth code would configure timer/counter 0 to interrupt the RTX 2000 after every 208D clock cycles or 1200 times per second. (If the automatic Baud rate detection routine is utilized, the routine will determine the count automatically according to the bit period of the incoming data.)

```
: INIT-TIMER          \ Word to initialize Timer0 for
                      \ 1200 Baud
  IBC@                \ IBC fetch, Pushes Interrupt
                      \ Base/Control reg on stack
  FCFF AND            \ Clear bit 8 and 9 of IBCR to
                      \ configure timers for Internal
                      \ time base.
  IBC!                \ IBC store, Load Interrupt
                      \ Base/Control reg with timers
                      \ configured for internal time
                      \ base.
  208D TC0!           \ Store 208DH into Timer/
                      \ counter 0 pre-load register;
                      \ timer will be loaded on next
                      \ clock.
  UNMASK-TIMER0       \ Word to unmask timer inter-
                      \ rupts.
  ENABLE-INTERRUPTS
                      \ Enable processor interrupts
;
```

Following is the Forth code for the word UNMASK-TIMER0.

```
: UNMASK-TIMER0       \ Word to Unmask timer0
                      \ interrupt
  IMR@                \ IMR fetch, Pushes Interrupt
                      \ mask register on stack top
  FF7F AND            \ Clear Bit 7 of IMR to unmask
                      \ timer/counter 0
  IMR!                \ IMR store, Load Interrupt
                      \ Mask Register from modified
                      \ value on stack.
;
```

When the timer decrements to zero the processor will execute an interrupt handler (provided that the timer is initialized, the timer interrupt is unmasked, and RTX interrupts are enabled.) The programmer must install the desired interrupt handler in the interrupt table. The RTX Forth compiler has a word called !INTERRUPT which performs that function. The !INTERRUPT word expects two values to be on the stack, first the address of the word to be executed when the interrupt occurs, and second the interrupt level. The interrupt level determines which of the 16 RTX interrupts the handler is to be installed for. The interrupt level for timer 0 interrupt is 7. The RTX 2000 interrupt levels are listed in the Harris RTX 2000 data sheet and Programmer's Reference Manual. The following code would cause the word POLL-EI3 to be executed after every timer 0 interrupt; i.e., whenever timer 0 decrements to 0.

```
['] POLL-EI3 7 !INTERRUPT
```

The ['] operator in Forth causes the address of the word following the ['] to be pushed on the stack. The above code pushes the address of the word POLL-EI3 followed by a 7 (for the interrupt level) on the stack and then executes the word !INTERRUPT. The !INTERRUPT word will place a call instruction to the word POLL-EI3 in the interrupt table location corresponding to timer 0 interrupt.

Algorithm for Transmitting Data

As mentioned previously, the boot pin on the RTX 2000 will be used for transmitting data. The UART is setup for sending a start bit followed by eight data bits, no parity, and one stop bit.

The Forth word EMIT is provided to transmit data. EMIT expects a character on the stack, and when executed transmits that character.

In order to decrease the time a program waits for output, the software UART is interrupt driven and has a sixteen word transmit buffer. When the word EMIT is executed the

character is placed in the next available location in the buffer; thus, the program does not need to wait for the character to be transmitted. The program will be interrupted by the timer so the UART can output the next bit of serial data in the buffer. The word WAIT-FOR-EMIT is provided if it is necessary for the program to wait for the character to be transmitted before continuing.

If the transmit portion of the UART is idle when the word EMIT is executed, the character is placed in the buffer, the UART is activated and then control is returned to the calling program. If the UART is busy, but there is space available in the buffer then the character is placed in the buffer and control is returned to the calling program. If the UART is busy and the buffer is full then the word EMIT waits until there is space available in the buffer. When space becomes available the character is placed in the buffer and control is returned to the calling program.

The transmit portion of the software UART is a Finite State Machine (FSM) that exists as the interrupt handler for the timer 0 interrupt. The transmit FSM has the following states IDLE-STATE, START-STATE, DATA-STATE, STOP-STATE, COMPETE-STATE. Following is a brief description of each of the states. The transition between states occurs when a timer interrupt occurs. See Figure 1 for a diagram illustrating the transmit finite state machine. Figure 2 contains a more detailed flow chart for the transmit function. The section titled Forth CODE FOR SOFTWARE UART contains a complete source code listing for the UART.

IDLE-STATE in this state there are no characters waiting to be transmitted. The transmit buffer is empty. In this state the timer 0 interrupt is masked; i.e. the transmit FSM is not running and therefore there is no loss of processor utilization. The UART remains in this state until the word EMIT is executed to transmit a character. When EMIT is executed, the timer 0 interrupt is unmasked and the state changes to the START-STATE thus activating the UART FSM. (If EMIT is executed when the UART is in any other state, then the character is placed in the receive buffer since the UART is not idle.) Immediately upon unmasking the timer interrupt, a timer interrupt will occur since the timers are free running and the transmit algorithm guarantees the timer will have decremented to zero at least once.

START-STATE when the uart is switched to the start state a start bit is transmitted, the baud rate count is stored into the timer, and the state is then changed to the DATA-STATE. Control is then returned to the user. The next timer interrupt will cause the DATA-STATE portion of the FSM to be executed.

DATA-STATE while the FSM is in the data state it is transmitting the eight data bits. Upon entry to the data state a counter is set to one to output the first bit of serial data. The data is transmitted, the counter is incremented, and control is returned to the user. This cycle continues until eight bits are transmitted at which time the state is changed to the STOP-STATE.

STOP-STATE when in this state the UART transmits a stop bit and then the state is changed to the COMPLETE-STATE. The next timer interrupt will cause the processor to switch to the COMPLETE-STATE.

COMPLETE-STATE when in this state the UART is finished transmitting the stop bit. If there are more characters in the transmit buffer then the state is changed to the START-STATE. If the transmit buffer is empty then TIMER 0 interrupt is masked thus changing the state to the IDLE-STATE.

Algorithm for Receiving Data

External Interrupt 3 is used by the software UART to receive data. The UART expects to receive one start bit, eight data bits, no parity, and one stop bit.

The Forth word KEY is provided to receive data. When executed, KEY waits for a character to be received and returns that character on the stack. The receiver portion is also interrupt driven and has a sixteen character input buffer.

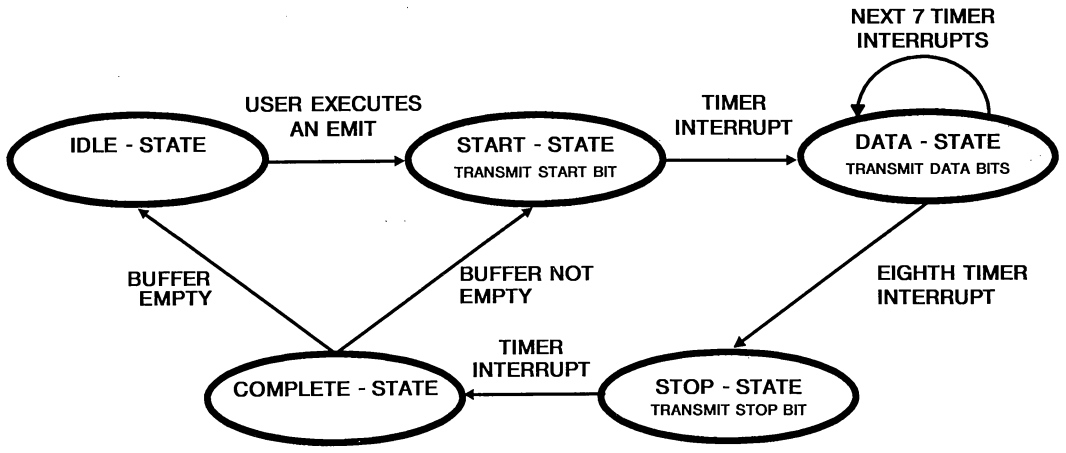

FIGURE 1. TRANSMIT DATA FINITE STATE MACHINE

The buffer allows characters to be read into the buffer while the program is performing other functions. Then, when the program needs input data, it executes the word KEY. If there is already data in the buffer, the program will not have to wait and the data will be returned immediately. If the buffer is empty, KEY will wait until a character is received.

The receive portion of the UART is also a finite state machine. The UART prepares to receive data by unmasking External Interrupt 3 (EI3). When an EI3 interrupt occurs, one of two things might have occurred. It could either indicate that a start bit has been detected or it could indicate a glitch on the receive input. To detect the difference between these cases, a timer is set to expire after 1/2 the bit period. After the timer expires, the external interrupt line is polled to see if the data still represents a start bit. If it is a start bit, then a valid start bit was detected, if not the interrupt was caused by a glitch (See Figure 3).

Once a valid start bit has been confirmed the timer is set up to generate an interrupt after each bit period. Each time the interrupt occurs, the external interrupt pin is polled to read in the next bit of data. As the data is read in it is placed in the RCV-BUFFER.

The Receive portion of the software UART is a Finite State Machine that exists as the interrupt handler for both the timer 1 interrupt and external interrupt 3. The receive FSM has these states IDLE-STATE, INIT-STATE, START-STATE, DATA-STATE. Following is a brief description of each of the states. See Figure 4 for a diagram illustrating the transmit finite state machine. Figure 5 contains a more detailed Flow chart of the receive portion of the UART. The section titled Forth CODE FOR SOFTWARE UART contains a complete source code listing for the UART.

IDLE-STATE in this state there are no characters being transmitted to the UART. The user has 100% of the processor bandwidth. The UART remains in this state until a start bit is received. The start bit will cause external interrupt 3 (EI3). When this occurs, timer 1 will be set to cause an interrupt after 1/2 the bit period. EI3 will be masked so that none of the data bits will cause an interrupt. The timer interrupt will be unmasked so that the UART can interrupt the user program once during each bit period to input the data. The state is then be changed to the INIT-STATE and control will be returned to the user.

INIT-STATE as soon as the timer interrupt is unmasked in the IDLE-STATE, a timer interrupt will occur switching the processor to this state. This immediate interrupt is guaranteed since the RTX timers are free running, and the algorithm for the UART will ensure that the timer will have expired at least once. When this timer interrupt occurs, the UART just changes to the START-STATE and returns control to the user until the timer expires causing another interrupt. This second timer interrupt will be in the center of the start bit.

START-STATE the UART will enter this state after the start bit has been detected and 1/2 of the bit period has expired (The center of teh start bit, see label 2 of Figure 3). The UART will read in the input data and ensure that the data represents a valid start bit. If the data is not a start bit the UART will be switched to the IDLE-STATE, the timer interrupt will be masked and external interrupt 3 will be unmasked in preparation for a new start bit. If the data is a start bit then the timer will be set to cause an interrupt after the next full bit period and the UART will be switched to the DATA-STATE.

DATA-STATE in this state the UART will be reading data from EI3 into the receive buffer. The UART will stay in this state for eight timer interrupts (1 for each bit of data). During each of these interrupts the UART will sample the input data bit store the result in the buffer and return to the user. After receiving the last data bit the UART will update the buffer

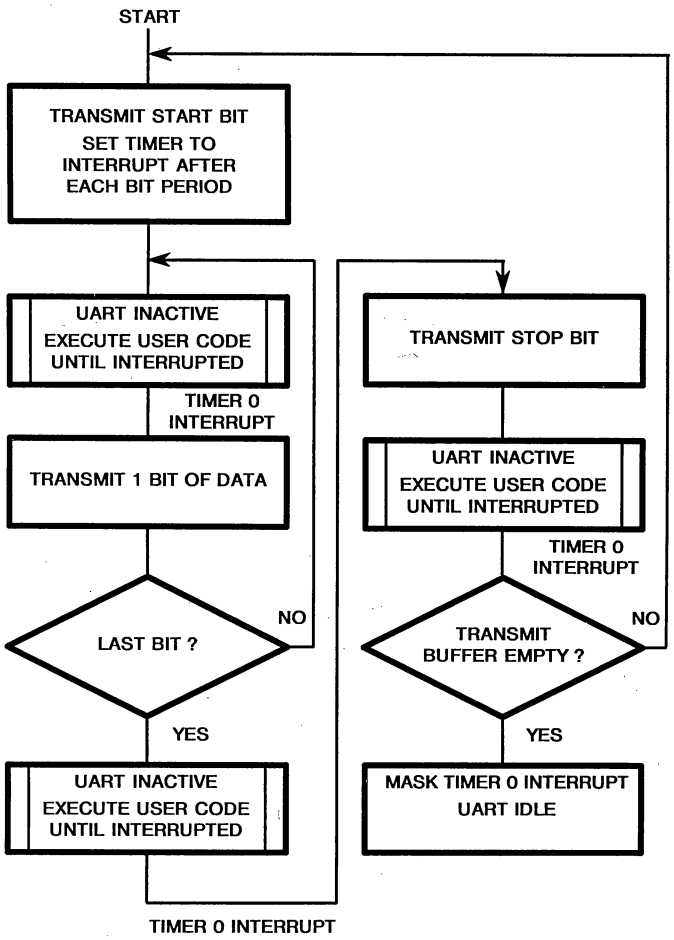

FIGURE 2. FLOW CHART FOR TRANSMIT FUNCTION

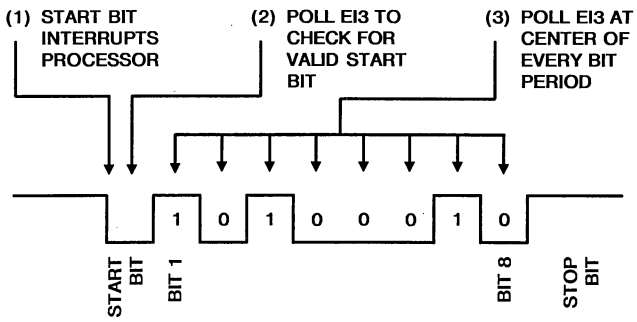

FIGURE 3. POLLING EIE TO RECEIVE SERIAL DATA

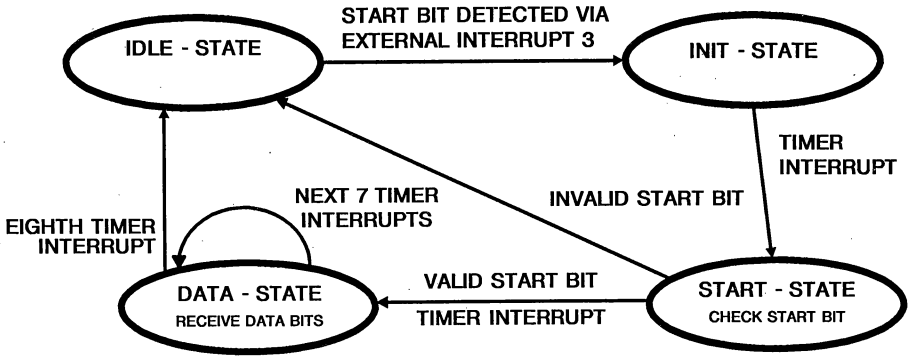

FIGURE 4. RECEIVE DATA FINITE STATE MACHINE

pointer indicating that there is a character available in the buffer. The UART will be switched to the IDLE state, the timer interrupt will be masked, and external interrupt 3 will be unmasked in preparation for a new start bit.

Initializing the UART

A routine called INIT-UART is provided to initialize the software UART. The routine installs the interrupt handlers for timer 0, timer 1, and external interrupt 3. The routine also initializes the receive and transmit buffers and the receive and transmit finite state machine. The final step in the initialization process is detecting the Baud rate. To do this the software UART waits for the user to press the RETURN key. When it detects a start bit the UART times the width of the start bit and uses that value for the bit period. The UART then continues to read in the character to ensure that it is a RETURN character. Once the RETURN character has been read correctly the Baud rate has been detected and the initialization is complete.

Hardware Used for Software UART

The only hardware necessary for the UART is an RS-232 driver/receiver. The boot pin from the RTX 2000 is connected to one of the transmitter input pins on the driver. The output from the driver is connected to the TX pin of the RS-232 connector.

The external interrupt from the RTX 2000 is connected in a similar fashion to the receiver portion of the RS-232 driver receiver, which is in turn connected to the RS-232 RX pin. The only difference is that the received data must be inverted before it is connected to the RTX 2000. The RTX 2000 external interrupts are active-high level sensitive, so it is necessary to invert the serial data to force the start bit to cause an interrupt. Since the RS-232 line driver receiver inverts the data, the received data is routed through the receiver to invert the incoming data. This saves having to add an inverter to the system.

Figure 6 illustrates the schematic for connecting a Maxim MAX235 RS-232 driver/receiver to the RTX 2000. This is the only hardware necessary to implement a software UART that communicates over RS-232 cable to a terminal or other device.

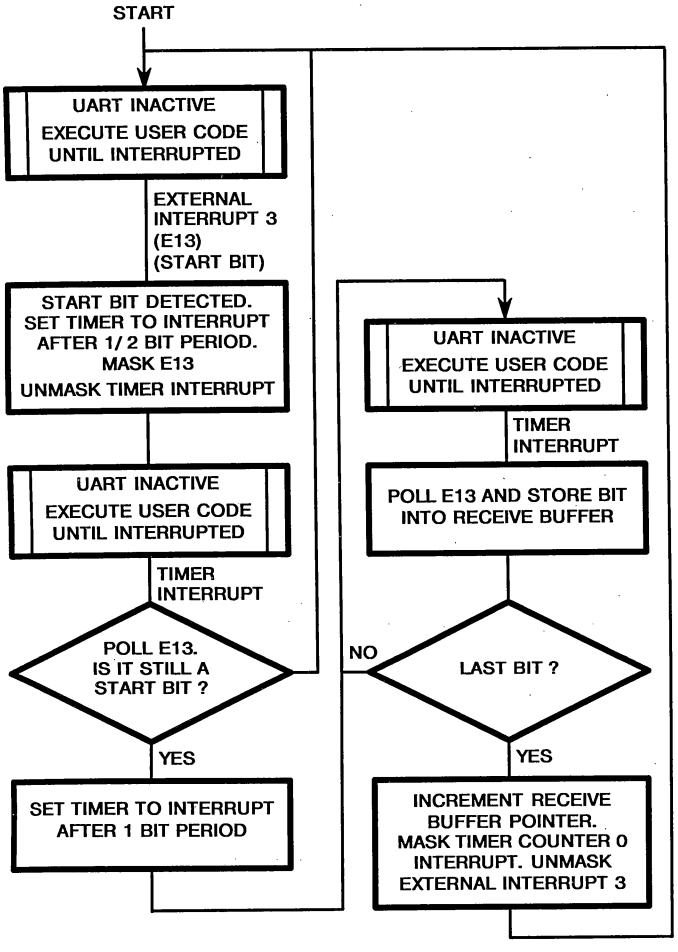

FIGURE 5. TRANSMIT DATA FLOW CHART

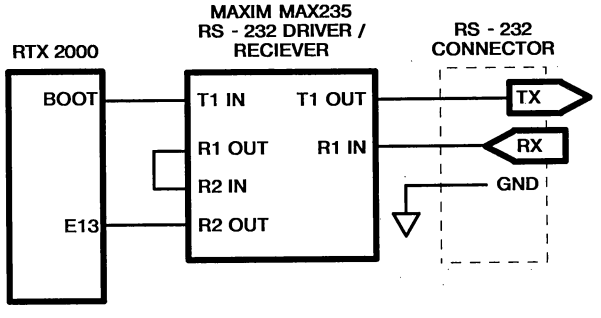

FIGURE 6. HARDWARE NECESSARY FOR SOFTWARE UART

Forth Code for Software UART

This section contains a complete source code listing for the software UART. Forth programs are typically written by developing small words and building more complex words by combining the smaller ones. This results in a bottom up method of coding. The source code listing is presented in that order. Thus the words EMIT is the last word to be defined in the TRANSMIT section and KEY is the last words to be defined in the RECEIVE section. For ease of understanding, it might be useful to begin reading the transmit portion of the UART from the Transmit finite state machine on page 10. The receive portion can be best understood by beginning with receive finite state machine on page 12.

```
\
\                    Software UART for the RTX 2000
\                    By Ted Dimbero Applications Engineer Harris Semiconductor
\
\          To transmit a character place the character on the stack and execute the word EMIT. To
\          receive a character execute the word KEY. This leaves the character received on the stack.
\
```

```
\ Following is the code for the transmit portion of the UART
\
HEX \ Change the default base to hexadecimal for entire program.
10 CONSTANT BUFFER-SIZE                          \ 16 character buffer size.
VARIABLE XMIT-BIT-POSITION                       \ The next bit of the data word to be transmitted
VARIABLE XMIT-STATE                              \ The current state of transmit FSM
VARIABLE XMIT-BUFFER BUFFER-SIZE ALLOT           \ Transmit Buffer
VARIABLE BAUD-RATE                               \ Contains # of cycles per bit period
VARIABLE XMIT-IN                                 \ Position to put character in transmit buffer
VARIABLE XMIT-OUT                                \ Position to take out chars from transmit buffer
VARIABLE ADDR-INIT-UART                          \ Contains address of INIT-UART word

0 CONSTANT XSTART-STATE                          \ states for the transmit FSM
1 CONSTANT XDATA-STATE
2 CONSTANT XSTOP-STATE
3 CONSTANT XCOMPLETE-STATE
0100 CONSTANT LAST-BIT

: ENABLE-INT CR@ CR! ;                           \ Enables all RTX interrupts
: DISABLE-INT CR@ 0010 OR CR! ;                  \ Disables all RTX interrupts

: UNMASK-TIMER0 IMR@ FF7F AND IMR! ;             \ Unmask Timer Counter 0 (TC0) Interrupt
: MASK-TIMER0 IMR@ 0080 OR IMR! ;                \ Mask TC0 interrupt

: UNMASK-EI3 IMR@ FBFF AND IMR! ;                \ Unmask External Interrupt 3 EI3

: MASK-EI3 IMR@ 0400 OR IMR! ;                   \ Mask EI3 interrupt

: INC-PTR ( buffer-ptr -- )                      \ Increments pointer through circular buffer
   DUP @ 1+ DUP BUFFER-SIZE >                    \ If buffer pointer > limit set back to zero
   IF DROP 0 SWAP ! ELSE SWAP ! ENDIF ;          \ else just add one to buffer pointer

: ?TRANSMIT-IDLE                                 \ When timer0 is masked there is no data currently being transmitted.
   IMR@ 0080 AND                                 \ Therefore the Transmit portion of UART is idle
;

: XBUF@                                          \ Reads next character from the transmit buffer
   XMIT-OUT @
   XMIT-BUFFER + C@
;

: XBUF! ( C -- )                                 \ Stores top of stack in next position of the transmit buffer and
   XMIT-IN @                                     \ increments the transmit buffer pointer.
   XMIT-BUFFER + C!
   XMIT-IN INC-PTR
;
```

```
: ?BUFF-FREE                          \ Leaves 0 on stack if buffer is full -1 otherwise
  XMIT-IN @ XMIT-OUT @ <>             \ Buffer is full when in and out index are =
;

: XMIT0 CR@ FFF7 AND CR! ;            \ Writes a zero to the boot pin
: XMIT1 CR@ 0008 OR CR! ;             \ Writes a one to the boot pin

: XMIT-BIT (Bit-position -- )         \ Transmit next bit of data from transmit buffer.
  XBUF@ AND IF                        \ Read data from XMIT buffer AND with current bit position,
    XMIT1                             \ If result is 1 then transmit a 1
  ELSE
    XMIT0                             \ Else if result is 0 then transmit a zero.
  ENDIF
;

\ Transmit next bit of data then update bit position to point
\ to next bit. If this is last bit change state of XMIT FSM
: XMIT-DATA-BIT
  XMIT-BIT-POSITION @ DUP XMIT-BIT    \ Transmit next bit of data.
  2* DUP XMIT-BIT-POSITION !          \ Update the bit position.
  LAST-BIT = IF                       \ If we have transmitted all 8 data bits
    XSTOP-STATE XMIT-STATE !          \ then change to STOP-STATE to transmit stop bit.
  ENDIF
;

\ Set timer and xmit start bit then change XMIT FSM to DATA-STATE
: XMIT-START-BIT ( -- )
  BAUD-RATE @ TC0!                    \ Set timer according to BAUD rate
  XMIT0                               \ Transmit start bit
  1 XMIT-BIT-POSITION !               \ Set bit position to transmit bit 1 of data first
  XDATA-STATE XMIT-STATE !            \ Change to DATA-STATE to begin transmitting data
;

: XMIT-STOP-BIT                       \ Xmit stop bit and change XMIT FSM state
  XMIT1                               \ Transmit Stop bit
  XCOMPLETE-STATE XMIT-STATE !        \ Change to COMPLETE-STATE
;

: XMIT-COMPLETE
  XMIT-OUT INC-PTR                    \ Increment buffer pointer since current character has been transmitted.
  ?BUFF-FREE NOT IF                   \ If Buffer is empty then mask timer 0 indicating UART is idle
    MASK-TIMER0 1 TC0!
  ENDIF
  XSTART-STATE XMIT-STATE !           \ Switch to START-STATE to prepare for next character.
;
```

```
\
\   Transmit Finite State Machine.
\   This routine is the interrupt handler for timer 0.
\

: XMIT
  XMIT-STATE @ DUP                    \ Fetch the state and determine the current state.
  XSTART-STATE = IF
    DROP XMIT-START-BIT               \ Transmit the start bit
  ELSE DUP XDATA-STATE = IF
    DROP XMIT-DATA-BIT                \ Transmit the data bits
  ELSE DUP XSTOP-STATE = IF
    DROP XMIT-STOP-BIT                \ Transmit the stop bits
  ELSE XCOMPLETE-STATE = IF
    XMIT-COMPLETE                     \ Transfer complete update buffer and prepare for next character
  ELSE
    ADDR-INIT-UART @ EXECUTE          \ Not a valid state so initialize the uart.
  ENDIF ENDIF ENDIF ENDIF
;

: EMIT ( C -- )                       \ Transmit the character on the top of the stack
  ?TRANSMIT-IDLE IF                   \ If the UART is idle then
    XBUF!                             \ Store the character in the transmit buffer
    UNMASK-TIMER0                     \ and Unmask timer 0 interrupt to activate the UART.
    ELSE BEGIN                        \ Else if UART is not idle wait until there is space in
      ?BUFF-FREE                      \ the transmit buffer.
    UNTIL
    XBUF!                             \ When space is available store character in the next location.
  ENDIF
;

: WAIT-FOR-EMIT                       \ Waits until all characters in transmit buffer have been
  BEGIN ?TRANSMIT-IDLE UNTIL          \ transmitted.
;
```

\
\
\

CODE FOR THE RECEIVE PORTION OF SOFTWARE UART

```
VARIABLE RCV-BIT-POSITION                \ Position in word of next bit to be received
VARIABLE RCV-STATE                       \ Current state of Receive FSM
VARIABLE RCV-BUFFER BUFFER-SIZE ALLOT    \ 16 character receive buffer
VARIABLE RCV-OUT                         \ Next position to read character from receive buffer
VARIABLE RCV-IN                          \ Next position to write character into buffer

0 CONSTANT DETECT-BAUD                   \ Receive FSM states
1 CONSTANT RIDLE-BAUD
2 CONSTANT RINIT-STATE
3 CONSTANT RSTART-STATE
4 CONSTANT RDATA-STATE

: UNMASK-TIMER1 IMR@ FEFF AND IMR! ;     \ Unmask Timer Counter (TC1) 1 interrupt
: MASK-TIMER1 IMR@ 0100 OR IMR! ;        \ Mask TC1 interrupt

: RBUF@ @ RCV-BUFFER + C@ ;              \ Read a character from RCV buffer
: RBUF! @ RCV-BUFFER + C! ;              \ Write a character to RCV buffer

: POLL-EI3 ( -- BIT)                     \ Reads Status of external interrupt 3 pin
  DISABLE-INT                            \ Disable interrupts
  IMR@                                   \ Save current IMR
  FBFF IMR!                              \ Mask all interrupts except EI3
  IVR@                                   \ Read interrupt vector
  03FF AND 0A0 =                         \ Test if EI3 pending
  SWAP IMR!                              \ Restore IMR value
  ENABLE-INT NOT                         \ Invert the serial data
;

: RCV-BIT ( BIT-POSITION -- BIT-POSITION )   \ Read in 1 bit and store in RCV-BUFFER
  POLL-EI3                               \ Read in the Bit
  IF                                     \ If bit is zero do nothing if bit is one set bit in RCV-BUFFER
    DUP                                  \ Duplicate the bit position
    RCV-IN RBUF@ OR                      \ Read in word from RCV-BUFFER and set the appropriate bit
    RCV-IN RBUF!                         \ Store new value in RCV-BUFFER
  ENDIF
;

: RCV-DATA-BIT                           \ Read in a data bit and update bit position
  RCV-BIT-POSITION @                     \ Read current bit position
  RCV-BIT                                \ Read next data bit into RCV buffer
  2* DUP RCV-BIT-POSITION !              \ Update bit position to point to next bit.
  LAST-BIT = IF                          \ When 8 bits have been read we are done.
    MASK-TIMER1                          \ When done mask timer 1 interrupt,
    RIDLE-STATE RCV-STATE !              \ change UART to idle state,
    RCV-IN INC-PTR                       \ Update RCV buffer pointer, and
    UNMASK-EI3                           \ Unmask EI3 to prepare for new start bit.
  ENDIF
;

\ Receive a bit and see if it is a start bit. If it is Initialize timer and get ready to receive data.
\ If it is not then reset the receive portion of UART.
: RCV-START-BIT
  POLL-EI3 IF                            \ If bit is one then invalid start bit
    MASK-TIMER1                          \ So reset receiver to prepare for new start bit.
    RIDLE-STATE RCV-STATE !
    UNMASK-EI3
  ELSE                                   \ Else if bit is zero then start bit was valid
    BAUD-RATE @ TC1!                     \ Set timer to interrupt after 1 bit period
    RDATA-STATE RCV-STATE !              \ and change to DATA-STATE to prepare to read data
  ENDIF
;
```

```
\
\   RECEIVE Finite State Machine.
\   This routine is the interrupt handler for timer 1.
\

: RECEIVE
  RCV-STATE @                          \ Read current UART state and decide what to do.
  DUP RINIT-STATE = IF                 \ First time out interrupt is meaningless so
    DROP RSTART-STATE RCV-STATE !      \ just change to next state
  ELSE DUP RSTART-STATE = IF
    DROP RCV-START-BIT                 \ Second time out interrupt check for valid start bit
  ELSE RDATA-STATE = IF
    RCV-DATA-BIT                       \ Next 8 time out interrupts read in the serial data
  ELSE
    ADDR-INIT-UART @ EXECUTE           \ Not valid state so re-init uart
  ENDIF ENDIF ENDIF
;

: ?KEY ( -- flag ) \ Returns TRUE if there are any characters in RCV-BUFFER false otherwise.
  RCV-IN @ RCV-OUT @ <>
;

: KEY ( -- C ) \ Receives next character and places it on the stack
  BEGIN ?KEY UNTIL                     \ Wait for a character to be placed in the buffer
  RCV-OUT RBUF@                        \ When available read the character from the buffer
  RCV-OUT INC-PTR                      \ and update the buffer pointer
;

: READ-DATA                            \ Prepare to Read data after start bit has interrupted the processor via EI3
  BAUD-RATE @ 2/ TC1!                  \ Store 1/2 bit period in counter to check for valid start bit in center of bit
  RINIT-STATE RCV-STATE !              \ Change to INIT-STATE
  1 RCV-BIT-POSITION !                 \ Set bit position to indicate receiving first bit
  0 RCV-IN RBUF!                       \ Set data in receive buffer to zero. Only need to change it when 1 is read.
  UNMASK-TIMER1                        \ Unmask timer 1 interrupt when interrupt will check for valid start bit.
;

: TIME-BAUD                            \ Time the length of the start bit to determine bit period
  FFFF TC1!                            \ Set max time in timer and
  BEGIN POLL-EI3 UNTIL                 \ wait until the start bit goes away.
  TC1 @                                \ Save elapsed time
  FFFF SWAP-                           \ Calculate bit period by subtracting value from maximum value
  BAUD-RATE !                          \ Store bit period in BAUD-RATE
;

\ This routine sets the Bit period based on an input character. It assumes the first
\ data bit is <> start bit. It then ensures the UART is in a correct mode to read
\ the rest of the data.
: SET-BAUD
  TIME-BAUD                            \ Determine the width of start bit in clock cycles
  BAUD-RATE @ 2/ TC1!                  \ Set timer to 1/2 bit period and
  BEGIN TC1 @ 10 < UNTIL               \ wait for it to expire.
  1 RCV-BIT-POSITION !                 \ Now we should be in center of first data bit. So we initialize software
  0 RCV-IN RBUF!                       \ UART to the state that will allow it to read in the data word.
  RDATA-STATE RCV-STATE !
  BAUD-RATE @ TC1!
  UNMASK-TIMER1
;
```

```
\
\   This is the EI3 interrupt handler. A serial data start bit will cause
\   this interrupt to be activated.
\
\   If the UART is being initialized it will determine the bit period by
\   calling SET-BAUD.
\
\   For normal UART operation it will prepare the UART to read
\   in the serial data by calling READ-DATA.
\
: START-BIT-DETECT
  MASK-EI3                          \ Mask EI3 so remaining data bits will not cause this interrupt
  RCV-STATE @ DETECT-BAUD = IF      \ If UART is being initialized then
    SET-BAUD                        \ Call SET-BAUD to determine bit period
  ELSE
    READ-DATA                       \ Else prepare UART to read in the data bits
  ENDIF
;

: WAIT-FOR-RETURN                   \ This routine is called during initialization it expects the user to press
  BEGIN                             \ the RETURN key
    DETECT-BAUD RCV-STATE !         \ Indicate that UART is trying to determine bit period
    UNMASK-EI3                      \ Unmask EI3 to wait for a start bit
    KEY 0D =                        \ Loop until we have successfully read in a RETURN key.
  UNTIL
;
\
\ Following word is provided to initialize the UART to a known state after a processor RESET
\
: INIT-UART
  MASK-TIMER0 MASK-TIMER1 MASK-EI3  \ Mask all interrupts associated with UART
  ENABLE-INT                        \ Enable RTX interrupts.
  1 TC0! 1 TC1!                     \ Store small count in timers to ensure an interrupt
                                    \ when they are unmasked
  ['] XMIT 7 !INTERRUPT             \ Store address of transmit timer 0 interrupt handler
  ['] START-BIT-DETECT 0A !INTERRUPT \ Store address of START-BIT-DETECT as EI3 interrupt handler
  ['] RECEIVE 8 !INTERRUPT          \ Store address of RECEIVE as timer 1 interrupt handler
  ['] INIT-UART ADDR-INIT-UART !    \ Store address of this routine in variable ADDR-INIT-UART
  XIDLE-STATE XMIT-STATE !          \ Initialize transmit FSM to idle state
  0 XMIT-IN ! 0 XMIT-OUT !          \ Initialize transmit buffer pointers
  XMIT1                             \ Make sure we are transmitting a stop bit.
  0 RCV-OUT ! 0 RCV-IN !            \ Initialize receive buffer pointers.
  WAIT-FOR-RETURN                   \ Wait for user to enter a return.
;
```

Overhead Associated With Software UART

The software UART in this example takes approximately 45 clock cycles to transmit a bit and 51 clock cycles to receive a bit. Using a Baud rate of 1200 bits per second and an RTX 2000 running at 10 MHz, the number of clock cycles between bits is 8,333 (See section titled "Using Internal Timer to Generate Baud Rate" to calculate this number). Therefore, if data is continually being transmitted and received the software UART will cause a loss of 1.15 % processor bandwidth (1.15 = (45 + 51) / 8,333). The goal of the transmit and receive software is to explain the functionality of the software UART in a straight forward manner, and therefore, no significant effort was made to optimize the code. If this performance penalty is not acceptable then some effort will be necessary to optimize the code to reduce the overhead.

Appendix A

Books Available for Learning Forth

FORTH: A TEXT AND REFERENCE by Mahlon G. Kelly & Nicholas Spies

A textbook approach to Forth with comprehensive references to MMS-FORTH and the 79 and 83 Forth Standards.

FORTH ENCYCLOPEDIA by Mitch Derick & Linda Baker

A detailed look at all fig-Forth Instructions.

MASTERING FORTH by Anita Anderson & Martin Tracy

A step-by-step tutorial including each of the commands of the Forth-83 International Standard; with utilities, extensions and numerous examples.

STARTING FORTH, 2nd Edition by Leo Brodie

The most popular and complete introduction to Forth, examples use the new Forth-83 standard.

THINKING FORTH by Leo Brodie

The sequel to "Starting Forth". An intermediate text on style and form.

All of the Books listed can be purchased in most book stores or through mail order from:

FORTH INTEREST GROUP
P.O.Box 8231
San Jose, CA 95155
(408) 277-0668

Notes

Sales Offices

U.S. HEADQUARTERS
Harris Semiconductor
1301 Woody Burke Road
Melbourne, Florida 32902
TEL: (407) 724-3739

EUROPEAN HEADQUARTERS
Harris Semiconductor
Mercure Centre
Rue de la Fusse 100
Brussels, Belgium 1130
TEL: (32) 246-2201

SOUTH ASIA
Harris Semiconductor H.K. Ltd
13/F Fourseas Building
208-212 Nathan Road
Tsimshatsui, Kowloon
Hong Kong
TEL: (852) 3-723-6339

NORTH ASIA
Harris K.K.
Shinjuku NS Bldg. Box 6153
2-4-1 Nishi-Shinjuku
Shinjuku-Ku, Tokyo 163 Japan
TEL: 81-3-345-8911

DISTRIBUTORS IN U.S.A.
Almac Electronics
Anthem Electronics
Electronics Marketing Corporation
Falcon Electronics
Gerber Electronics
Hall-Mark Electronics
Hamilton/Avnet Corporation
Newark Electronics
Schweber Electronics
Wyle Laboratories

DISTRIBUTORS IN CANADA
Hamilton/Avnet Corporation
ITT Multicomponents

HARRIS