Application Report
SPRA555

Implementation of a Software UART on TMS320C54x Using General-Purpose I/O Pins

Adrienne Prahler Jaffe *C5000 Applications*

Abstract

Asynchronous communication between a digital signal processor (DSP) and another device is a common system requirement. This application report discusses a software implementation of a universal asynchronous receiver and transmitter (UART) to enable asynchronous communication with minimal hardware overhead using the Texas Instruments (TI™) TMS320C54x DSP. Topics covered include:

- Asynchronous communication discussion
- Advantages and disadvantages of a software UART
- Implementation of a software UART on a TMS320C54x
- Performance evaluations of software UART code
- Suggestions for user enhancements
- RS-232 interfacing
- C callable assembly source code

Contents

Figures

Tables

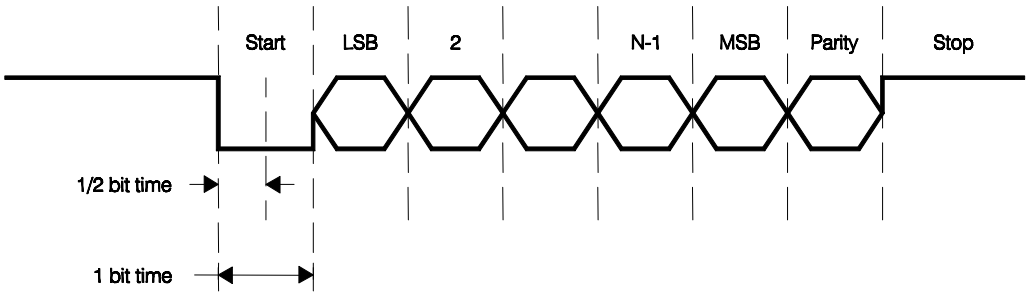
Introduction

Digital signal processors (DSPs) are used in real-time systems that involve the transfer of information among system components. Two types of communication are possible, synchronous, which requires continuous timing references that both devices use, and asynchronous, which does not require continuous timing references. In synchronous communication, the devices use a timing signal that is always present even if there is no data to send. In asynchronous communication, the devices communicate with each other using start and stop bits as timing references. These bits are generated only when there is data present. A software universal asynchronous receiver and transmitter (UART) provides the flexibility of asynchronous serial communication between devices with minimal external hardware. The devices communicating operate under an established protocol that includes the number of data bits, bits per second (bps), and parity. A software implementation of a UART allows increased system functionality without increasing hardware costs.

Data Format

In asynchronous communication, the transmit and receive lines are held high in the idle state when no data is present. The data is sent over a serial line from least significant bit (LSB) to most significant bit (MSB), as shown in Figure 1. The data stream begins with a single start bit, a low, that signals the beginning of the data stream. After the data is sent, the data stream ends with one or two stop bits. Half-duplex operation of the UART is defined as being able to receive and transmit, but only performing one function at a time. Full-duplex operation is achieved when the device can receive and transmit simultaneously. The bit time is the total time the line is held at a value for each data bit as shown in Figure 1. A half bit time is defined as half the time the line is held at a value for each data bit. When operating at 9600 bits per second (bps), the half bit time is 0.05 ms and the 1 bit time is 0.1 ms.

Figure 1. UART Data Stream



Parity Standards

Parity is a simple error-checking routine that can be added by including an additional bit during data transmission. Two types of parity are supported in this implementation, odd and even parity. If odd parity is selected, the parity bit is set to 1 when the number of 1s transmitted (data bits only) is even. If even parity is selected, the parity bit is set to 1 when the number of 1s transmitted (data bits only) is odd. The parity bit in each case is selected such that the total number of 1s transmitted (data bits and parity bits) is odd if odd parity is active and even if even parity is active. In this implementation, odd or even parity or no parity is selectable when the code is assembled and linked.

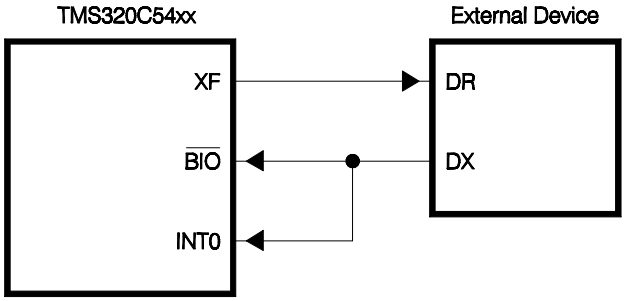
Implementation

This implementation of the UART in software on the ‘C54x allows several options, including:

- Number of data bits(1–16)
- Choice of parity (even or odd) or no parity
- User specified bits per second (BPS)
- Number of stop bits(1–2)

These parameters must be set prior to assembling and linking the code. The code is written in assembly and is not optimized. The assembly functions (setup, transmit, and transmit_delay) are written to be C callable. The method used here allows for the full-duplex operation of the device, while only using two general-purpose I/O pins (BIO\ and XF), an external interrupt (INT0), and a timer. The hardware setup is shown in Figure 2. The XF pin is used to transmit data and the BIO\ and INT0 pins are used to receive data. A UART status register and counter are maintained in memory to control the software and determine what actions should be performed.

Figure 2. Hardware Setup for Software UART



Setup

The software UART has setup code to initialize the C54xx. Once setup, the UART is ready to transmit and receive data. The process flow of the program is shown in Figure 3 through Figure 6. During setup, the memory locations are initialized, interrupts enabled, and the status registers are set.

It is important to note that the UART code is time sensitive. If uninterruptible code is being executed, the latency between when the timer interrupt occurs and when the timer interrupt service routine (ISR) is executed could cause the data to be corrupted. The data bits must be received and transmitted within a specified time period. The appropriate latency to prevent data corruption varies according to the bits per second operation rate of the UART.

The transmit and receive functions require simple tasks for completion; however, ensuring proper performance increases the complexity of the tasks. The two functions will be examined separately and the final goal of achieving full-duplex operation will be discussed.

Receive Function

The receive function recovers and formats the data from the input signal. To receive the data from the input signal correctly, the UART must determine the timing references. The timing references are provided by the start and stop bits that are added to the data stream, so the problem is reduced to detecting the start bit. One method for detection of the start bit is to poll the BIO\ pin at a rate that is greater than the expected bits per second of the data. However, this requires a great deal of CPU overhead and does not provide instantaneous detection of the start bit. Using an external interrupt pin to detect the start bit solves the problem. The low signal is detected immediately with no CPU overhead for polling. By connecting the BIO\ pin and an external interrupt pin together, the external interrupt can detect the start bit and the BIO\ pin can receive the data.

Once the start bit is detected, several things must be done. The interrupt is triggered at the beginning of the start bit, but the best place to test for the value of the data bit is at the half bit time. If the bit was sampled at the beginning or the end of the bit and the timer ISR was not serviced immediately, invalid data could be detected. Sampling at the half bit time also allows some margin if there is latency in the detection of the start bit or in the servicing of future timer ISRs. When the start bit is detected, the external interrupt ISR starts the timer for the half bit time so data samples can be taken at the half bit time.

Proper data reception requires more than just setting up a timer. The software must know that the UART is actively receiving so another section of code (such as a transmit call) will not disrupt the receive function. One way of doing this is to create a quasi UART status register in memory and define a bit, the RCV bit, to determine if the UART is receiving or not. A counter is needed to keep track of the number of data bits received so the program knows when it is finished receiving. The counter for the number of data bits, UART count, then needs to be reset at the start of a new data stream. The received data will also need to be stored, so a memory location is also reset. Once the start bit is detected, the external interrupt must be masked so an interrupt will not occur with the start of every low data bit in the data stream. Once these pieces are in place for the data reception, the code can return to the main program and wait for the timer interrupt to occur.

The UART is setup to receive a data stream and the timer goes off a half bit time after the start bit is detected. The data needs to be sampled and formatted when all the data is recovered. The BIO\ pin is sampled and the data value is stored (in the rcv_data memory location). The UART counter is updated to keep track of the number of data bits received.

There are two iterations of the timer ISR when additional steps must be included. The first is when the counter is cleared to 0 (the first time the timer ISR is triggered for each data stream). When the start bit is detected, the timer is setup and started for only a half bit time for the first iteration of the timer ISR. The next data point needs to be sampled a whole bit time later (sample at ½ bit time of start bit and again at the ½ bit time of data bit 1 so the total time is 1 bit time). If the counter is 0, the timer needs to be started for 1 bit time. As an error checking measure, the BIO\ pin (the receive pin) is sampled at this time (the ½ bit time of the start bit). If the pin is high, the timer is stopped and the program exits immediately because the low detected was a glitch, not a valid start bit.

The second exception to the standard timer ISR is when the final data bit is received. For the data to be useful, it needs to be formatted. To detect when the final bit is received, a comparison of the number of data bits expected and the number received is included in the timer ISR. When the final data bit is received, the received data must be formatted. The start and stop bits must be cleared and if parity is enabled that bit must be cleared after being checked. The exit routine must also reset the RCV bit in the UART status register since data is no longer being received. The external interrupt must be cleared (even though it was masked during data transmission, the interrupt flag was set to be serviced with each logic low) and enabled so the next start bit can be detected.

Transmit

The transmit function formats the data and sends the data out. The data to be communicated must be passed to the transmit function in some way. One method is to create a data memory location, tx_datapass, and prior to a transmit function call, place the data in that location. Once the data is in place and the function is called, the data must be formatted. Another step must be inserted to protect the data if you make back-to-back calls to the transmit function. An additional bit can be used from the UART status register (where a bit for receive has already been defined), the TX bit, to determine if there is data already formatted for transmission. If the TX bit is set, the program can loop until it is reset. Finally, the data formatting can begin. The start, stop and parity bits (if enabled) can be added to the data stream. The data can then be stored in a memory location ready for transmission.

The data is formatted and now needs to be sent out. To know when to send the next data bit the timer needs to be setup. While the receive and transmit functions are operating independently, they share one timer in this implementation. If the UART is in the middle of receiving data, the data would be corrupted if the timer was suddenly used by the transmit function. Prior to starting transmission, the *RCV* bit should be checked to determine if the UART is receiving. Once the UART is not receiving, the data word can be transmitted. For the same reason that the transmit function can not just take over the timer, the receive function must be disabled from capturing the timer while the UART is transmitting. Masking the external interrupt that detects the start bit during transmission can do this. It is important to note that a start bit will not be detected if the UART is busy transmitting. The other device must not begin a transmission if it is already receiving a data stream because the ‘C54x device will not receive the data stream. Additionally, a counter must be set up to keep track of the number of data bits transmitted. When the data is formatted and the timer is started, the program returns and waits for the timer interrupt to occur.

When the timer ISR is executed, the data must be sent serially to the other device. The XF pin is set for the next data bit each time the ISR is iterated, but the timer ISR is a single piece of code. When the ISR is executed, the state of the UART is determined so the proper functions (reception or transmission) are implemented. There is a receive section that was previously described. This is accessed if the RCV bit in the quasi UART status register is set. There is also a transmit section. It makes sense to just use the TX bit that is set when the transmit function call is made to determine if the transmit code needs to be executed. However, when the transmit function call is made, the code loops if the UART is busy receiving to prevent data corruption. At this point though, the TX bit is set and each time the timer ISR is executed, the transmit code would be performed. There is no counter or way of knowing how many bits have been sent or even if the data was actually formatted and ready to be sent (the timer ISR could occur in the middle of the data formatting). To prevent this situation, an additional bit is specified in the quasi UART status register, the UART bit. The UART bit is set prior to starting the timer for the transmit function if the timer is free and the data transmit has commenced. The UART bit can be used to determine what code needs to be executed for the transmit section of code in the timer ISR.

The final steps of the timer ISR for the transmit function are shared with the receive function. The normal exit routine is used after every complete iteration of the timer ISR. The exit routine checks for the final bit count and increments the UART counter. When the final bit is transmitted and received, the exit routine branches to an end routine. During the end routine, the timer is halted, the UART status register is reset from the transmit function call, and the hardware interrupt is enabled after clearing any pending interrupts that occurred during the transmit functions (if a low bit was transmitted, the IFR register flag will be set even while the interrupt is masked).

The timer ISR program flow that includes both the receive code and the transmit code is shown in Figure 3. Notice the two sections of code – one for the receive function and the other for the transmit section. The normal end routine program flow for both the receive function and the transmit function is shown in Figure 4.

Figure 3. Timer ISR for Transmit and Receive Functions

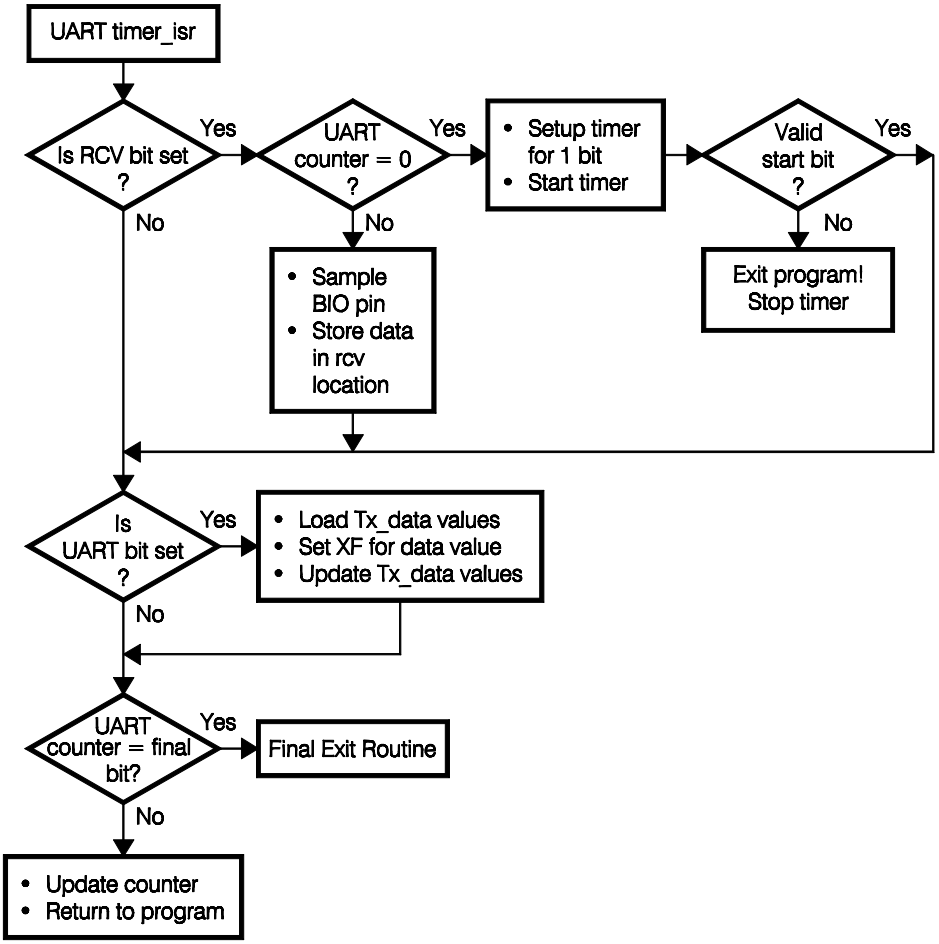
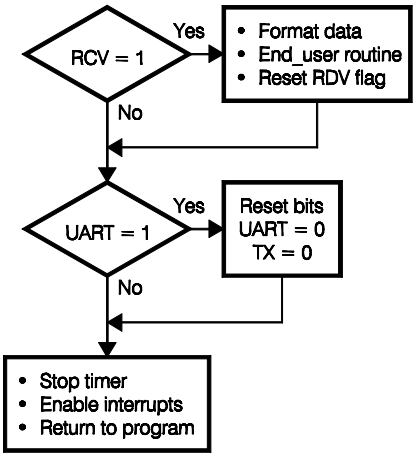


Figure 4. End Routines for Receive and Transmit Functions



Full-Duplex Operation

Receive and transmit have both been examined operating independently. Much of the overhead of the functions like counters and exit routines are the same, so full-duplex operation (both receive and transmit at the same time) is desirable. The limiting factor is the single timer used in the implementation. While independently functioning transmit and receive functions are not possible, dependent full-duplex operation is possible. The receive function is an asynchronous event since the start bit can occur at any time. However, the transmit calls are controlled by the program.

A transmit delay function was implemented in the code to give the flexibility of dependent full-duplex operation. The program flow for the transmit function calls are shown in Figure 5. To support the transmit delay function, another bit in the UART status register, TX-delay, is set when a transmit delay function call is made. The data is formatted and placed in the appropriate memory location, but prior to setting up the timer, the code returns to the main program. The delayed transmit is activated when the next receive function starts. The final program flow for the receive function is shown in Figure 6. When the start bit is detected, the TX-delay bit in the UART status register is checked. If the bit is set, the UART is setup to transmit and receive simultaneously.

Figure 5. Transmit Function

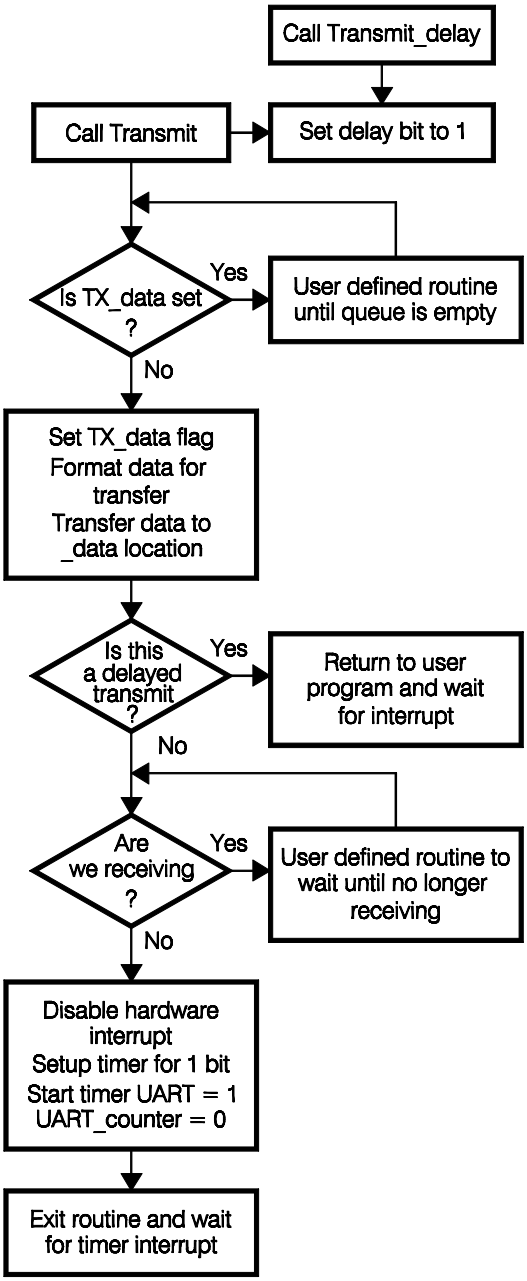
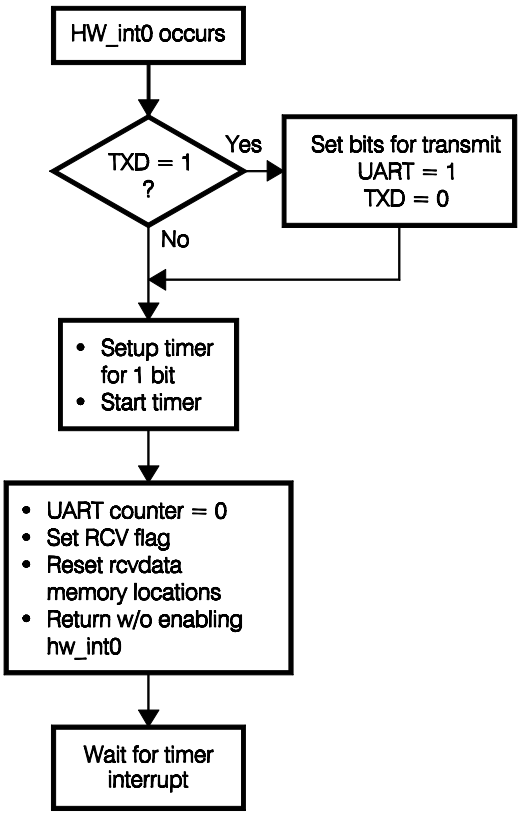
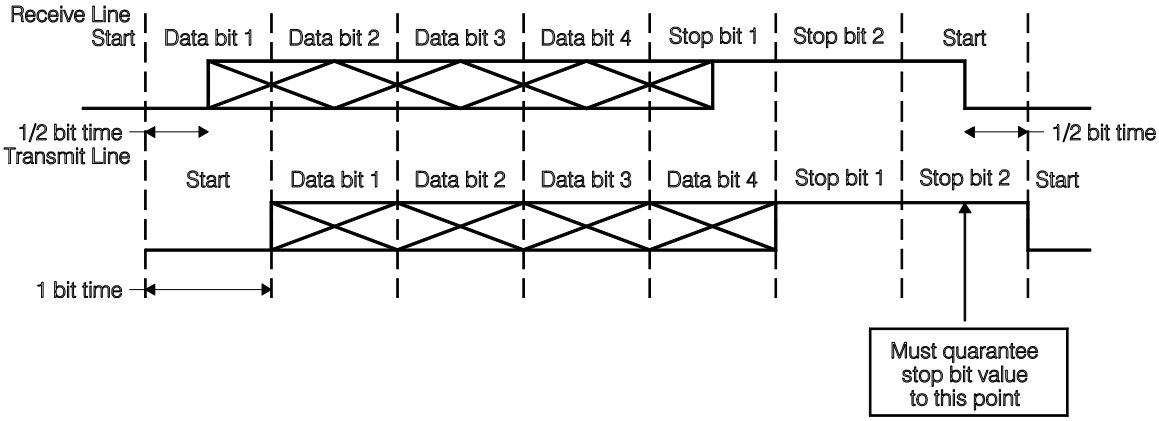


Figure 6. Receive Function



It is possible for full-duplex operation of the UART to be achieved. If the application allows (for example, receive functions occur within a tolerated delay time for the transmit data), you can use transmit delayed function calls. When a start bit is detected, the flag is tested and a full-duplex transmit and receive occurs with a half bit offset of each other as shown in Figure 7. This timing allows for the full-duplex operation of the UART. The offset is created because the first transmit bit is set at the middle of the receive start bit. Continuous communication with a start bit occurring after the final stop bit of the previous data is possible even in full-duplex operation; the earliest a receive start bit could be sent is a ½ bit time after the final stop bit is received. Since the first timer ISR does not occur until a ½ bit into the start bit, the final stop bit for the transmit function will be completed before the next transmit data would be sent. The ½ bit delay allows enough time for the final formatting and the resetting of the system before the next data word is transmitted.

Figure 7. Timing of Receive and Transmit Functions when Operating in Full-Duplex Mode



Parity Algorithm

The parity algorithm used in the software is described in the Texas Instruments Application Brief, *Parity Generation on the TMS320C54x* (literature number SPRA266). The parallel “successive approximations” in software method is used that involves a minimal overhead to generate the parity value. The data is loaded into an accumulator and exclusive ORed with itself shifted by the appropriate bits. Because the values are continually shifted, the parity value is not the least significant bit. In this software, the parity value is stored into a memory location with the appropriate shift so the value is in the least significant bit of the memory location. This allows bit tests to be performed easily to determine what needs to be added for the parity bit or if there is a parity error. Further explanation of the calculation method can be found in the application brief.

Bits Per Second (BPS) Calculation

Prior to assembling and linking the code, the timer registers must be set such that the timer interrupt rate is set for the appropriate bits per second rate. The bits per second rate is determined by the following equation:

$$bps = \frac{1}{clkout * (TDDR + 1) * (PRD + 1)}$$

The clock out value is the effective clockout of the device measured in seconds. Using the clockout value and the desired bits per second rate, the user can determine the appropriate prescalar (TDDR) and period (PRD) prior to compile time.

Settings for the PRD register, assuming a 100 MHz device (10 ns clockout), and a TDDR value of 1, are shown in Table 1. Changes in the clockout value of the device or the value of the TDDR register would alter these values.

Table 1. Values of PRD Register for Different BPS

BPS	Whole bit PRD	Half bit PRD
2400	20832	10416
9600	5207	2603
19200	2603	1301
38400	1301	650
57600	867	433
115200	433	216

Summary of Functions

A summary of the function calls supported in this software UART, the locations of the appropriate data, and the hardware pins used are listed in Table 2. In addition, the timer is used during transmit and receive function calls. Context save and restore is implemented in the code, but status register ST1 is not maintained. During transmit, the XF pin is used and the XF pin state is defined by a bit in the ST1 register. The user should be aware of this and ensure that no other program will use the XF pin during a transmit function call. The data placed in tx_datapass is not preserved.

Table 2. Memory locations of data used in Function Calls

Function calls	Data location	Pins Used
Transmit	tx_datapass	XF
Transmit_delay	tx_datapass	XF
Receive	rcv_datapass	BIO and int0

The number of data bits (1-16), number of stop bits (1 or 2), parity (even, odd, or none), and the bps of the system are specified when the code is assembled and linked. These values could be made variable by specifying the values in a soft register in memory (like the reserved space in the UART status register).

The UART uses several memory locations listed below during its operation. To support the ability of 16-bit data communication (since the ‘C54x is a 16-bit machine) of the UART, two memory locations are used for both receive and transmit data. If the total number of bits to be communicated using the UART including start, stop, parity, and data is less than 16, the second memory location can be eliminated.

- UART_counter
- UART status register, UART_reg
- Tx_datalo
- Tx_datahi
- Rcv_datalo
- Rcv_datahi
- Tx_datapass

- Rcv_datapass
- Parity_value

The UART status register bits are shown in Figure 8. If the bit is 0, that component is not active. The bit descriptions are in Table 3.

Figure 8. UART Status Register Bits

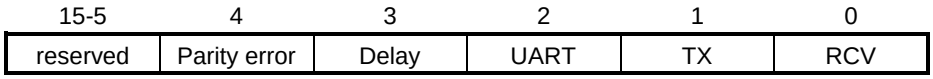


Table 3. Bit Meanings of UART Status Register

Bit	Value	Status
RCV	1	UART receiving data
	0	No data being received
TX	1	Data to transmit
	0	No valid transmit data
UART	1	UART busy transmitting
	0	UART not transmitting
Delay	1	Delayed transmit ready
	0	No delayed transmit
Parity error	1	Parity error occurred in last receive
	0	Parity correct or not used

Processor Configuration

As previously mentioned in this application report, there are several conditions that must be met for this software to operate correctly. The processor must be set with Sign Extension Mode (SXM) suppressed to allow the correct bit manipulations for transmit and receive. During transmit, the status register 1, ST1, can not be context saved and restored since the XF pin is set from that register and is used as the transmit line. You must also ensure interrupt latencies for the timer ISR so data is communicated correctly. When you are incorporating the software UART into a program all of these issues must be addressed. The code listed here assembles error free using the TMS320C54x COFF Assembler Version 1.20. The UART was tested using the C541 EVM that operates at 40 MHz. The UART was connected to a hardware UART on the TMS320F240 to demonstrate valid data transmission and reception.

Performance Evaluations of Code

Without any modifications, the performance of the software UART at 9600 bps is approximately 0.75 MIPS in full-duplex operation. There are several timing references that limit the performance of the software programming. These performance metrics shown in Table 4 are for the software as documented, operating in full-duplex mode with a transmit delay and receive function calls and context save and restore of status registers and accumulators.

Table 4. Performance of Software UART Functions (Number of Clock Cycles)

Function	Transmit, no parity	Transmit, odd parity	Receive, no parity	Receive, odd parity	Full-duplex, no parity	Full-duplex, odd parity
UART_setup	16	16	16	16	16	16
Transmit	71	93				
Tx_delay					45	67
Hw_int0			33	33	33	33
Timer_isr 1st iteration	43	43	59	59	69	69
Timer_isr 2nd - final–1 iterations	43	43	52	52	62	62
Timer_isr final	78	78	102	135	112	145

There are three routines that determine the performance of the software implementation. The timer ISR must be completed in less than one bit time since it must be able to recover each data bit. The setup for the receive function must be completed in less than a half bit time so the timer references are maintained. Additionally, the final timer ISR must be completed in less than a half bit time to allow the UART to detect back to back data streams (final stop bit of one data stream is followed immediately with a start bit, signaling a new data stream). The maximum performance is determined by the total time to complete the number of cycles specified. In this equation, clkfreq is the clock frequency of the device, #cycles is the number of cycles for the function, and bit is the limitation of when the function needs to be complete (1/2 or 1).

$$\max \quad bps = \frac{clkfreq}{\#\,cycles} * bit$$

To put this in perspective, the performance of the unoptimized code for full-duplex operation assuming a 100MHz device would be limited to 400 000 bps. The external ISR for the receive function setup takes approximately 33 cycles, which would limit the performance of the UART to 1 500 000 bps (max bps = 100MHz/(2*33cycles)). The limitation of the timer ISR to one bit time limits the performance to 1 400 000 bps with 69 clock cycles for completion. The longest ISR (the final bit that also includes the end routine call) takes 125 clock cycles to complete, which reduces the maximum performance to 400 000 bps. In this software implementation, the longest timer ISR is the limiting factor. The code used to test this software makes a call to transmit delay when the final stop bit is set. This increases the longest ISR, which decreases the performance of the algorithm. Any modifications of the software will effect the maximum bit per second rates.

The MIPS performance is determined by calculating the average number of cycles for each bit to be transmitted, including overhead cycles. Assuming full-duplex operation with 8 data bits, 2 stop bits, and parity, the average number of cycles for each bit is 78 cycles. Therefore, the total number of MIPS if the UART is in continuous operation is 78 cycles times the desired bps. For the example at 9600 bps, the UART uses approximately 0.75 MIPS. The code size is approximately 2K words. Modifications to the code will alter these performance numbers, but they are provided to give some idea of the overhead required to run this code.

Possible User Alterations

RS-232 Interface

The transmit and receive lines can be used to interface to RS-232 compatible serial ports by hooking up a line driver and receiver to provide the required differential voltages. The only RS-232 signals supported are the transmit (DX) and receive (DR) lines. Other desired signals, such as handshaking could be accommodated by you with additional bit I/O lines as desired. The ‘C5402, ‘C5410, and ‘C5420 have additional general-purpose bit I/O available depending on which peripherals are in use (for example, HPI or McBSP).

Performance Enhancements

Depending on the specific application the software UART is being used for, several adjustments to the code are possible to optimize performance. The code itself can be optimized for speed and/or code size as needed. Reordering some commands to effectively use available delay slots will increase the performance. Some other areas that could be optimized include temporary data storage. If the total number of bits, data, stop, start, and parity is less than or equal to 16, the additional temporary data locations, tx_datahi and rcv_datahi can be discarded. This would free up memory space and reduce the number of instructions in the timer ISR. Additionally, the waiting loops to prevent the user from overwriting data can be modified to return to the main program instead of just looping until the flags are cleared. You can also modify the error checking routines. If parity is enabled, a flag is set in the UART status register if an error occurs. The error routine could also be modified to include requesting the other device to resend data or to sample each data value multiple times to reduce the probability of an error. Modification of the software UART code is possible depending on application requirements.

Conclusions

A software implementation of a UART has been presented for the TMS320C54x DSP. The UART is a universal asynchronous receive and transmit that allows communication between two devices that is not possible if serial port communication is not available. This implementation requires approximately 2K words of memory for code size and uses approximately 0.75 MIPS when operating at 9600bps. The software UART is a valuable tool for connecting two devices with minimal overhead; however, CPU overhead is application dependent.

References

1. Nerge, David.*Parity Generation on the TMS320C54x*, TMS320 DSP Designer's Notebook, Application Brief, SPRA266, February, 1996.
2. Zalac, David. *An Implementation of a Software UART Using the TMS320C25*, Application Report, SPRA077, 1997.
3. Fried, Ted. *Developing a Full-Duplex UART on the TMS320C3x*, TMS320 DSP Designer's Notebook, Application Brief, SPRA254, February, 1995.
4. Ward, Stephen A. and Halstead, Robert H., Jr., Computation Structures. The MIT Press, Cambridge, 1991.
5. *TMS320C54x DSP Reference Set, Volume I: CPU and Peripherals,* SPRU131, 1997.

Appendix A Software UART Code

```
************************************************************
*This program is a s/w UART for the C54x using the BIO and *
*XF, I/O pins, a hardware interrupt, and a timer.          *
*The device operates in full-duplex mode and supports      *
*up to 16 data bits, 1 or 2 stop bits, and parity (even or *
*odd).                                                     *
*                                                          *
*Written by: Adrienne Prahler Jaffe                        *
*Modified: 3/10/99                                         *
*Three function calls are available in this program,       *
*setup, transmit and transmit_delay.                       *
*Functions being made c-callable                           *
************************************************************

       .global _timer_isr, _transmit, _transmit_delay, _setup, _hw_int0,
_tx_datapass
       .global _rcv_datapass, _UART_reg, _parity_value, _one, _UART_count,
_tx_datalo
       .global _rcv_datalo


****************** MAIN SHELL **********************************
       .mmregs               ; allow addressing of memory registers by name


       .bss _UART_reg,1       ; location of status register flags
       .bss _rcv_datalo,1     ; rcv location
       .bss rcv_datahi,1      ; rcv location
       .bss _tx_datalo, 1     ; tx_data location
       .bss tx_datahi, 1      ; tx_data location
       .bss _UART_count,1     ; counter
       .bss _one, 1           ; constant location
       .bss _tx_datapass, 1   ; where transmit data is put for transmission
       .bss _rcv_datapass, 1  ; where receive data is stored
                              ; (striped of stop, start, parity)
       .bss _parity_value,1   ; parity value location
                              ; if total #1's in dataword even then
                              ; lsb = 0, odd lsb = 1
       .bss parity_bit, 1     ; parity_bit value
***************Added for debugging*****************************

       .bss _data_storage, 330, 1 ; where receive data is stored

************************************************************


*********************Timer information**********************
*  timer interrupt rate (baud rate)= 1/(clkout*(TDDR+1)*(PRD+1))*
*  TDDR is timer divide down ratio, 4 bits                    *
*  PRD is the timer period register, 16 bits                  *
*                                                             *
* Knowing the desired baud rate and clkout of device, the values*
* of TDDR and PRD can be determined.                          *
*                                                             *
```

```
* In this example, the TDDR is set to 1, if that is changed, the*
* values of tcr_timer_go needs to be changed so the lower 4 bits*
* are the desired values.                                        *
*                                                               *
* The program uses two timer rates, one for a half bit time and *
* one for a whole bit time.  The value calculated for the PRD   *
* in the above equation is for the whole bit time.              *
*****************************************************************



whole_bit_time   .set 8332d             ; set whole bit time assuming 40MHZ
                                        ; and 2400 baud
half_bit_time    .set 4167d             ; set half bit time assuming 40MHz
                                        ; and 2400 baud
tcr_setup        .set 0000000000010000b ; setup tcr values, stop timer
tcr_timer_go     .set 0000000001100001b ; tcr register values for
                                        ; timer start

*************variables set for operation of UART*****************
* UART_reg:                                                     *
*___15-5_____4_____3_____2_____1_____0____                      *
*|reserved| parity error | delay |  uart  |  tx  |  rcv  |      *
*-------------------------------------------------------        *
*RCV                                                            *
*              0 = not receiving data                           *
*              1 = receiving data                               *
*TX                                                             *
*              0 = no data to transmit                          *
*              1 = data waiting to be transmitted               *
*UART                                                           *
*              0 = not busy transmitting                        *
*              1 = actively transmitting                        *
*Delay                                                          *
*              0 = not delayed                                  *
*              1 = delayed                                      *
*Parity error                                                   *
*              0 = nothing wrong                                *
*              1 = error occurred                               *
*****************************************************************



data_bits        .set 8    ; number of data bits being transferred
stop             .set 2    ; number of stop bits, 1 or 2
start_bits       .set 1    ; number of start bits
parity           .set 0    ; if 0, no parity
                           ; if 1, odd parity
                           ; if 2, even parity


       .if parity != 0     ; if parity set then reserve a bit spot
par_bits         .set 1
parity_or        .set 01b<<data_bits+start_bits
       .else
par_bits         .set 0
       .endif
```

```
************Setting up stopbit formatting values******************
       .if stop=2     ;if 2 stop bits then set up this value for formatting
stop_or       .set 011b<<data_bits+par_bits+start_bits
                     ; setting up for formatting data
       .elseif stop=1 ; if 1 stop bit set up this value for formatting
stop_or       .set 01b<<data_bits+par_bits+start_bits
                     ; setting up for formatting data
       .else
       .emsg "ERROR, must specify the number of stop bits to be 1 or 2"
       .endif

**********Parity Routines**************************************
       .if parity > 2
              .emsg   "ERROR, the parity value must correspond to no
parity, even parity or odd parity!"
       .endif


       .if parity !=0               ; if parity set
              .if data_bits+par_bits+start_bits>=17
N_par  .set 32
              .elseif data_bits+par_bits+start_bits>=9
N_par  .set 16
              .elseif data_bits+par_bits+start_bits>=5
N_par  .set 8
              .elseif data_bits+par_bits+start_bits>=3
N_par  .set 4
              .elseif data_bits+par_bits+start_bits=2
N_par  .set 2
              .elseif data_bits+par_bits+start_bits=1
N_par  .set 1
              .endif
       .endif

******************Main Program**********************

       .text

******dummy main program to call UART setup and a delayed transmit******

start:
        nop
        call _setup
        nop
        nop
        call _transmit_delay
        nop
loop:   nop               ; looping
        nop
        nop
        b  loop
        nop
        nop
```

```
************setup UART*************************
_setup:
            rsbx cpl          ; enable direct addressing mode
            ld #_UART_reg, DP ; setup DP for direct addressing
            RSBX    INTM      ; enable global interrupts
            RSBX    SXM       ; turn off sign extension mode, the UART must
                              ; operate with this setting!
            st #1, @_one      ; create a Smem location with value 1 stored
            st #9h, *(IMR)    ; enable interrupt (timer and hw_int)
            st #0, @_UART_reg ; initialize data locations
            st #0, @_tx_datalo
            st #0, @tx_datahi
**************Added for debugging**************
            stm #100, bk            ; load circular buffer size
            ssbx xf                 ; set xf high
            nop
            nop
            nop
            nop
            stm #_data_storage, ar3 ; setup auxiliary register for
                                    ; receive data
            st #8bh, @_tx_datapass  ; load data into tx_datapass
*****************************************************
            ssbx cpl
            rete                    ; return to main program
************Timer ISR for UART*************************
_timer_isr: pshm st0                ; context save
            pshm AL
            pshm AH
            pshm AG
            rsbx cpl                ; set cpl=0 for assembly functions
            ld #_UART_reg, DP       ; setup DP for direct addressing

rcv:        bitf *(_UART_reg), #1h ; test rcv flag
            nop
            nop
            bc tx, ntc              ; if rcv not set then branch to tx
            cmpm @_UART_count, #0h ; is count = 0?
            nop
            nop
            bc exit_now, nbio, tc  ; if count = 0 and bio =1 then exit now!
                                    ; not a VALID start bit!
            cc timerwhole, bio, tc ; setup timer for whole bit if count=0
                                    ; bio=0
            bc tx, bio, tc          ; after time set up branch to tx routine
            ld @_UART_count, T      ; load counter into T register
                                    ; used for offset since data received
                                    ; lsb first.
            ld @_rcv_datalo, A      ; load rcv_data
            add @rcv_datahi, 16, A
            xc 1, nbio              ; if bio pin high then
            add @_one, TS, A        ; add one shifted by bit position(count)
            stl A, @_rcv_datalo     ; store rcv data in memory location
            sth A, @rcv_datahi
```

```
tx:         bitf @_UART_reg, #4h   ; test UART flag
            nop
            nop
            bc exit_norm, ntc      ; if not transmitting then just exit
            bitf @_tx_datalo, #1h  ; what is lsb?
            ld @_tx_datalo, A      ; load tx_data
            add @tx_datahi, 16, A
            xc 1, tc               ; if lsb=1 set xf=1
            ssbx xf
            xc 1, ntc
            rsbx xf                ; if lsb=0 set xf=0
            stl A, -1, @_tx_datalo ; store low portion of tx out with shift
                                   ; right of one
            sth A, -1, @tx_datahi  ; store hi portion of tx out with shift
                                   ; right of one

******************Exit Routines from timer isr*****************
* Normal exit routine                                         *
* Invalid start bit detected                                  *
* End of rcv data routine                                     *
* End of tx data routine                                      *
**************************************************************
*********Normal Exit routine************************************
exit_norm:  cmpm  @_UART_count, #stop+start_bits+par_bits+data_bits-1
                                  ;test if final bit
            nop
            nop
            cc exit_final, tc       ; if final bit exit
            Addm #1,  @_UART_count  ; update counter
restore:    ssbx cpl                ; reset cpl=1 for c program
            popm AG             ; restore accumulator A contents and status
            popm AH
            popm AL
            popm ST0
timer_end:  rete                    ; return to main program
            nop
            nop
*********Invalid start bit**************************************
exit_now:   bitf @_UART_reg, #4h  ; What is UART bit set to?
            stm #tcr_setup, TCR   ; stop timer by writing a 1 to tss in tcr
            xorm #1h, @_UART_reg  ; reset RCV flag
            xc 2, tc              ; if this was a delayed tx and uart=1
            xorm #0ah, @_UART_reg ; reset txdelay bit, and uart bit!
            st #9h, *(IMR)        ; enable int0 and timer interrupts (timer
                                  ; stopped)
            b restore             ; exit routine
            nop
            nop
************Final Bit exit Routine
exit_final: bitf @_UART_reg, #1h  ; check rcv bit
            ld @_rcv_datalo, A
            add @rcv_datahi, A
            cc end_rcv_routine, tc
            nop
            nop
            bitf @_UART_reg, #4h  ; check uart bit
            stm #tcr_setup, TCR   ; stop timer by writing a 1 to tss in tcr
            nop
```

```
            xc 2, tc
            xorm #6h, @_UART_reg  ; reset TX and UART flags
            ldm  IFR, A           ; load interrupts pending
            nop
            nop
            and #1h, A            ; get rid of int0 pending interrupts
            stlm  A, IFR          ; clear all pending interrupts
            st #9h, *(IMR)  ; enable int0 (timer still active but stopped)
            call _transmit_delay  ; loop back with a transmit delay call
            nop
            nop
            ret                   ; restore saved registers
            nop
            nop

******************end rcv data routine***********************

end_rcv_routine:
formatting  xor #stop_or, A      ; get rid of stop bits

******************Parity Bit Routine*******************
** only compiled if parity set at compile time        **
*****************************************************
            .if parity != 0
                 call parity_calc              ; does this match with the
                                               ; type of parity expected?
                 bitf @_parity_value, #1h      ; check parity value
                 ld @_rcv_datalo, A    ; load rcv_data to edit out parity,
                 add @rcv_datahi, 16, A        ; stop, start
                 stl A, -(start_bits+data_bits), @parity_bit
                                               ; store parity bit
                                               ;to lsb in parity_bit
                 .if parity = 1         ; if odd parity & paritycalc is 0
                     cc parity_error, ntc      ; then call error
                     .elseif parity = 2 ; if even parity & parity bit is 1
                     cc parity_error, tc       ; then call error
                 .endif
                 bitf @parity_bit, #1h         ; check parity bit value
                 xor #stop_or, A               ; get rid of stop bits
                 nop
                 xc 2, tc              ; if parity bit=1 then get rid of it
                 xor #parity_or, A
                 nop
            .endif

*****************************************************

            stl A, -start_bits, @_rcv_datapass ; put unformatted data word
                                               ; in rcv location
```

```
*******************Added for debugging*****************************
            stl A, -start_bits, @_tx_datapass  ; load tx data with just
                                               ; received data
            stl A, -start_bits, *AR3+%         ; error checking routine –
                                               ; putting data into a
                                               ; memory location so it can
                                               ; be checked later

*************************************************************************

            xorm  #1h, @_UART_reg              ; reset RCV flag
            ret
            nop
            nop

*********************** Timer setups**********************

timerhalf:  stm  #tcr_setup, TCR  ; stop timer by writing a 1 to tss in tcr
            stm  #half_bit_time, PRD    ; load timer period
            stm  #tcr_timer_go,TCR      ; start timer
            ret
timerwhole:
            stm  #tcr_setup, TCR  ; stop timer by writing a 1 to tss in tcr
            stm  #whole_bit_time, PRD   ; load timer period
            stm  #tcr_timer_go,TCR      ; start timer
            ret

***************Parity calculation routine*****************
** only compiled if parity set at compile time          **
*******************************************************
parity_calc:  .if parity != 0
                 .if N_par = 32
                       xor A, 16, A
                 .endif
                 .if N_par >= 16
                       xor A, 8, A
                 .endif
                 .if N_par>= 8
                       xor A,4, A
                 .endif
                 .if N_par>=4
                       xor A, 2, A
                 .endif
                 .if N_par>=2
                       xor A, 1,A
                       stl A, -N_par+1, @_parity_value ; load parity status
                                                        ; into lsb
                 .endif
                 ret
                 nop
                 nop
              .endif

*******************Parity error routine************************

parity_error: xorm  #10h, @_UART_reg     ;set parity error flag
              ret
```

```
*****************Transmit Routine*********************************
* Data passed in mem location tx_datapass                         *
* The routine will check if the tx flag is set and wait until     *
* the tx flag is reset so transmit data won't be overwritten      *
* before it is sent.  If it is a delayed transmit the routine     *
* will just format the data and return to the main program until*
* the next receive interrupt occurs.  If it is a transmit the     *
* registers for transmit and the timer are setup and started.     *
*                                                                 *
* If a transmit is started, then the device can not receive.      *
* Full-duplex operation occurs only when a transmit delayed is    *
* called since there is only one timer.                           *
*******************************************************************
_transmit:    pshm  AL          ; context save since Accumulator A used
              pshm  AH
              pshm  AG
              pshm  ST0
              rsbx cpl               ; enable direct addressing mode
              ld #_UART_reg, DP      ; set DP for addressing

busy:         BITF *(_UART_reg), #4h  ; What is UART_flag set to?
              nop
              nop
              BC   busy, tc     ; if UART=1 then can not overwrite data

tx_set        BITf @_UART_reg, #2h   ; What is TX_flag set to?
              nop
              nop
              BC tx_set, tc     ; if tx = 1 then can not overwrite data
              xorm #2h, @_UART_reg   ; if tx = 0 set TX=1

format:
******************Parity formatting of tx data***********
** only compiled if parity set at compile time          **
*********************************************************
              .if parity != 0
                   ld @_tx_datapass, A    ; load temp data for parity check
                   call parity_calc       ; calculate parity_value
              .endif
*********************************************************

              ld @_tx_datapass, start_bits, A  ; load temp data (offset by
                                               ; start_bit)
              nop
              stl A,-1, *AR3+%                 ; added for debugging
              add #stop_or,  A                 ; add stop bits
```

```
**********************Add parity bit**********************
** only compiled if parity set at compile time          **
*********************************************************
              .if parity != 0
                   bitf  @_parity_value, #1h           ; check parity value
                   nop
                   nop
                   .if parity = 1
                       xc 2, ntc
                   add #1h, start_bits+data_bits, A    ; add a 1 to make
                                                       ; odd parity
                                                       ; add #parity_or, A
                                                       ; add a 1 to make
                                                       ; odd parity
                   .elseif parity =2
                       xc 2, tc
                   add #1h, start_bits+data_bits, A    ; add a 1 to make
                                                       ; even parity
                                                       ; add #parity_or, A
                                                       ; add a 1 to make
                                                       ; even parity
                   .endif
              .endif
*********************************************************

                bitf @_UART_reg, #8h   ; is this a delayed transmit?
                sth A,@tx_datahi       ; store out data for transmit
                stl A,@_tx_datalo
                stl A, *AR3+%          ; added for debugging
                                       ; return to user program if delayed
td_end:         bc tx_end, tc          ; transmit - wait for rcv interrupt
                nop
                nop
waiting:        BITf @_UART_reg, #1h   ; what is RCV flag?
                nop
                nop
                BC waiting, tc      ; if recieving have to wait until done
                                    ; can insert user program if desired
                st #8h, *(IMR)         ; disable hw_int0, timer enabled
                call timerwhole        ; setup timer for 1 bit and start
                xorm #4h, @_UART_reg   ; set UART=1, going to transmit
                st #0, @_UART_count    ; initialize UART_count
tx_end:         ssbx cpl               ; reset cpl=1
                popm st0               ; restore contents and status
                popm ag
                popm ah
                popm al
t_end           ret
_transmit_delay: xorm #8h, *(_UART_reg) ; enable delay bit (this is a
                                        ; delayed tx)
                B    _transmit
                nop
                nop
```

```
*********************Receive routine************************************
*This is the hw_int routine that will be executed when a start          *
*bit is detected. The routine checks the status of transmit and starts  *
*the timer for a half bit time so the program can be sure a true start  *
*bit not a glitch was detected.  It also resets the counter, and        *
*receive data locations.                                                *
*                                                                       *
*If the device is finishing a transmit routine, the TX and UART flags   *
*will be reset.  If the transmit_delay flag is set, the routine will    *
*setup for a transmit during the receive.                               *
************************************************************************

_hw_int0:       pshm  st0                ; context save
                rsbx  cpl                ; enable direct addressing mode
                ld    #_UART_reg, DP     ; setup DP for addressing
                bitf  *(_UART_reg), #8h  ; is txd set?
                call  timerhalf        ; setup and start time for half bit
                xc    2, tc              ; if delayed transmit
                xorm  #0ch, @_UART_reg   ; reset TXD bit and set UART bit

end:            ST    #0, @_UART_count   ; initialize UART counter
                st    #8h, *(IMR)        ; disable int0
                st    #0, @_rcv_datalo
                st    #0, @rcv_datahi
                xorm  #1h, @_UART_reg    ; set RCV flag = 1
                                         ; return with timer enabled
                ssbx  cpl
                popm  st0                ; context restore
hw_end:         rete
```

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas
Phone +1(972) 644-5580
Fax +1(972) 480-7800
Email sc-infomaster@ti.com

Europe, Middle East, and Africa
Phone
- Deutsch +49-(0) 8161 80 3311
- English +44-(0) 1604 66 3399
- Español +34-(0) 90 23 54 0 28
- Francais +33-(0) 1-30 70 11 64
- Italiano +33-(0) 1-30 70 11 67

Fax +44-(0) 1604 66 33 34
Email epic@ti.com

Japan
Phone
- International +81-3-3344-5311
- Domestic 0120-81-0026

Fax
- International +81-3-3344-5317
- Domestic 0120-81-0036

Email pic-japan@ti.com

Asia
Phone
- International +886-2-23786800
- Domestic
  - Australia 1-800-881-011
    - TI Number -800-800-1450
  - China 10810
    - TI Number -800-800-1450
  - Hong Kong 800-96-1111
    - TI Number -800-800-1450
  - India 000-117
    - TI Number -800-800-1450
  - Indonesia 001-801-10
    - TI Number -800-800-1450
  - Korea 080-551-2804
  - Malaysia 1-800-800-011
    - TI Number -800-800-1450
  - New Zealand 000-911
    - TI Number -800-800-1450
  - Philippines 105-11
    - TI Number -800-800-1450
  - Singapore 800-0111-111
    - TI Number -800-800-1450
  - Taiwan 080-006800
  - Thailand 0019-991-1111
    - TI Number -800-800-1450
  - Fax 886-2-2378-6808
  - Email tiasia@ti.com

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty, or endorsement thereof.