The NS16550A: UART Design and Application Considerations

National Semiconductor
Application Note 491
Martin S. Michael
Daniel G. Durich
July 1987

BACKGROUND

UARTs like other system components have evolved for many years to become faster, more integrated and less expensive. The rise in popularity of the personal computer with its focus and competition primarily centered on an architecture introduced by IBM®, has driven both UART performance and software compatibility issues. As transmission rates have increased, the amount of time the CPU has for other tasks while handling an active serial channel has been sharply reduced. One byte of data received at 1200 baud (8.3 ms) is received in $\frac{1}{16}$th the time at 19.2 kbaud (520 μs). Software compatibility among the PC-based UARTs is critical due to the thousands of existing programs which use the serial channel and the new programs continually being offered.

Higher baud rates and compatibility requirements influence new UART designs. These two constraints result in UARTs that are capable of higher data rates, increasingly independent of CPU intervention and providing more autonomous features, while maintaining software compatibility. These development paths have been brought together in a new UART from National Semiconductor designated the NS16550A.

The NS16550A has all of the registers of its two predecessor parts (INS8250 and NS16450), so it can run all existing IBM PC, XT, AT, RT and compatible serial port software. In addition, it has a programmable mode which incorporates new high-performance features. Of course, all of these advanced features are useful in any asynchronous serial communications application regardless of the host architecture.

The reader is assumed to be familiar with the standard features of the NS16450, so this paper will concentrate mainly on the new features of the NS16550A. If the reader is unfamiliar with these UARTs it is advisable to start by reading their data sheets.

The first section reviews some of the design considerations and the operation of the NS16550A advanced features. The second section shows an NS16550A initialization routine written in 80286 assembly code with an explanation of the routine. The third section gives a detailed example of communications drivers written to interface two NS16550As on individual boards. These drivers are written for use with National Semiconductor's DB32032 evaluation boards, but can be ported to any NS32032-based system containing an NS32202 (ICU).

1.0 Design Considerations and Operation of the New UART Features

In order to optimize CPU/UART data transactions, the UART design takes into consideration the following constraints:

1. The CPU is usually much faster than the UART at transferring data. A high speed CPU could transfer a byte of data to/from the UART in a minimum of 280 ns. The UART would take over 1800 times longer to transmit/receive this data serially if it were operating at 19.2 kbaud.
2. There is a finite amount of wasted CPU time due to software overhead when stopping its current task to service the UART (context switching overhead).
3. The CPU may be required to complete a certain portion of its current task in a multitasking system before servicing the UART. This delay is the CPU latency time associated with servicing the interrupt. The amount of time that the receiver can accept continuous data after it requests service from the CPU constrains CPU latency time.

The design constraints listed above are met by adding two FIFOs and specialized transmitter/receiver support circuitry to the existing NS16450 design. The FIFOs are 16 bytes deep—one holds data for the transmitter, the other for the receiver (see *Figure 1*). Similarity between the FIFOs stops with their size, as each has been customized for special

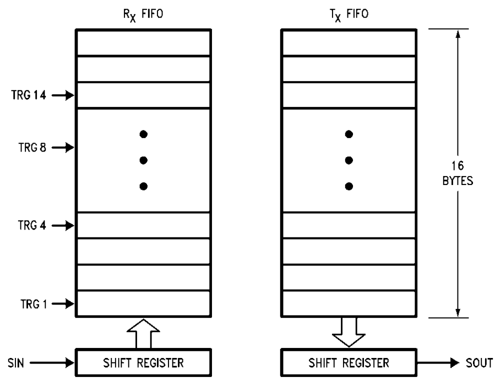


FIGURE 1. Rx and Tx FIFOs

transmitter or receiver functions. Each has support circuitry to minimize software overhead when handling interrupts. The NS16550A **receiver** optimizes the CPU/UART data transaction via the following features:

1. The depth of the Receiver (Rx) FIFO ensures that as many as 16 characters will be ready to transfer when the CPU services the Rx interrupt. Therefore, the CPU transfer rate is effectively buffered from the serial data rate.
2. The program can select the number of bytes required in the Rx FIFO (1, 4, 8 or 14) before the UART issues an interrupt. This allows the software to modify the interrupt trigger levels depending on its current task or loading. It also ensures that the CPU doesn't continually waste time switching context for only a few characters.

3. The Rx FIFO will hold 16 bytes regardless of which trigger level the CPU selects. This makes allowances for a variety of CPU latency times, as the FIFO continues to fill after the interrupt is issued.

The NS16550A **transmitter** optimizes the CPU/UART data transaction via the following features:

1. The depth of the Transmitter (Tx) FIFO ensures that as many as 16 characters can be transferred when the CPU services the Tx interrupt. Once again, this effectively buffers the CPU transfer rate from the serial data rate.
2. The Transmitter (Tx) FIFO is similar in structure to FIFOs the user may have previously set up in RAM. The Tx depth allows the CPU to load 16 characters each time it switches context to the service routine. This reduces the impact of the CPU time lost in context switching.
3. Since a time lag in servicing an asynchronous transmitter usually has no penalty, CPU latency time is of no concern to transmitter operation.

TX AND RX FIFO OPERATION

The Tx portion of the UART transmits data through SOUT as soon as the CPU loads a byte into the Tx FIFO. The UART will prevent loads to the Tx FIFO if it **currently** holds 16 characters. Loading to the Tx FIFO will again be enabled as soon as the next character is transferred to the Tx shift register. These capabilities account for the largely autonomous operation of the Tx.

The UART starts the above operations typically with a Tx interrupt. The NS16550A issues a Tx interrupt whenever the Tx FIFO is empty and the Tx interrupt is enabled, except in the following instance. Assume that the Tx FIFO is empty and the CPU starts to load it. When the first byte enters the FIFO, the Tx FIFO empty interrupt will transition from active to inactive. Depending on the execution speed of the service routine software, the UART may be able to transfer this byte from the FIFO to the shift register before the CPU loads another byte. If this happens, the Tx FIFO will be empty again and typically the UART's interrupt line would transition to the active state. This could cause a system with an interrupt control unit to record a Tx FIFO empty condition, even though the CPU is currently servicing that interrupt. Therefore, after the first byte has been loaded into the FIFO the UART will wait one serial character transmission time before issuing a new Tx FIFO empty interrupt.

This one character Tx interrupt delay will remain active until at least two bytes have been loaded into the FIFO, concurrently. When the Tx FIFO empties after this condition, the Tx interrupt will be activated without a one character delay.

Rx support functions and operation are quite different from those described for the transmitter. The Rx FIFO receives data until the number of bytes in the FIFO equals the selected interrupt trigger level. At that time if Rx interrupts are enabled, the UART will issue an interrupt to the CPU. The Rx FIFO will continue to store bytes until it holds 16 of them. It will not accept any more data when it is full. Any more data entering the Rx shift register will set the Overrun Error flag. Normally, the FIFO depth and the programmable trigger levels will give the CPU ample time to empty the Rx FIFO before an overrun occurs.

One side-effect of having a Rx FIFO is that the selected interrupt trigger level may be above the data level in the FIFO. This could occur when data at the end of the block contains fewer bytes than the trigger level. No interrupt would be issued to the CPU and the data would remain in the UART. To prevent the software from having to check for this situation the NS16550A incorporates a timeout interrupt.

The timeout interrupt is activated when there is at least one byte in the Rx FIFO, and neither the CPU nor the Rx shift register has accessed the Rx FIFO within 4 character times of the last byte. The timeout interrupt is cleared or reset when the CPU reads the Rx FIFO or another character enters it.

These FIFO related features allow optimization of CPU/UART transactions and are especially useful given the higher baud rate capability (256 kbaud). However, in order to eliminate most CPU interactions, the UART provides DMA request signals. Two DMA modes are supported: single-transfer and multi-transfer. These modes allow the UART to interface to higher performance DMA units, which can interleave their transfers between CPU cycles or execute multiple byte transfers.

In single-transfer mode the receiver DMA request signal (Rx RDY) goes active whenever there is at least one character in the Rx FIFO. It goes inactive when the Rx FIFO is empty. The transmitter DMA request signal (Tx RDY) goes active when there are no characters in the Tx FIFO. It goes inactive when there is at least one character in the Tx FIFO. Therefore, in single-transfer mode active and inactive DMA signals are issued on a one byte basis.

In multi-transfer mode Rx RDY goes active whenever the trigger level or the timeout has been reached. It goes inactive when the Rx FIFO is empty. Tx RDY goes active when there is at least one unfilled position in the Tx FIFO. It goes inactive when the Tx FIFO is completely full. Therefore in multi-transfer mode active and inactive DMA signals are issued as the FIFO fills and empties. With 2 DMA channels (one for each Rx and Tx) assigned to it, the NS16550A could run somewhat independently of the CPU when the DMA unit transfers data composed of blocks with checksums.

SYSTEM OPERATION: THE NS16550A VS THE NS16450

Consider the typical system interface block diagram in *Figure 2*. This is a simple diagram, but it includes all of the components that typically interact with a UART. The advantages of the NS16550A over the NS16450 can be illustrated by comparing some of the system constraints when each UART is substituted into this basic system.

Both RS-232C and RS-422A interfaces can be used with either UART, however, the NS16550A can drive these interfaces up to 256 kbaud. Regarding the RS-422A specifica-

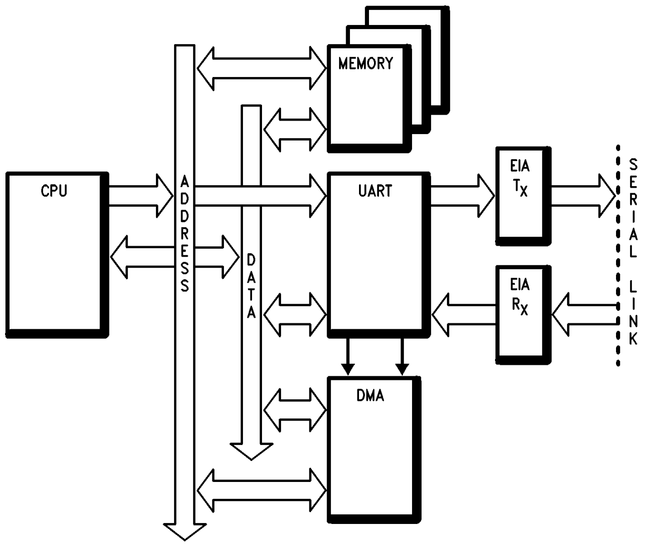


FIGURE 2. Typical System Interface

tion (max. 10 Mbaud) this is significantly faster than the NS16450 (max. 56 kbaud).

The NS16450 has no DMA request signals, so the DMA unit would not interact with the NS16450. The NS16550A, however, has DMA request signals and two modes of data transfer, as previously described, to interface with a variety of DMA units.

The greatest advantages of the NS16550A over the NS16450 are seen when considering the CPU/UART interface. Some characteristics of the transactions occurring between the CPU and the UART were previously cited. However, optimizing these transactions involves two issues:

1. Decreasing the amount of time the CPU interacts with the UART.
2. Increasing the amount of data transferred between the CPU and UART during their interaction time.

These optimization criteria are directly opposed to each other, but various features on the NS16550A have improved both.

One of the more obvious ways to decrease the CPU/UART interaction time is to decrease the time it takes for the transaction to occur. The NS16550A has an access cycle time that is almost 25% shorter than the NS16450. In addition, other timing parameters were made faster to simplify high speed CPU interactions.

The actual software required to transfer the data between the CPU and the UART is a small percentage of that required to support this transfer. However, each time a transfer occurs in the NS16450, this support software (overhead) must also be executed. With the NS16550A each time the UART needs service the CPU can theoretically transfer 16 bytes while only running through its overhead once. Tests have shown that this will increase the performance by a factor of 5 at the system level over the NS16450.

Another time savings for the CPU is a new feature of the UART interrupt structure. Unlike most other UARTs with Rx FIFOs, the NS16550A will issue an interrupt when there are characters below the interrupt trigger level after a preset time delay. This saves the extra time spent by the CPU to check for bytes that are at the end of a block, but won't reach the interrupt level.

Since the NS16550A register set is identical to the NS16450 on power-up, all existing NS16450 software will run on it. The FIFOs are only enabled under program control.

All of this added performance is not without some tradeoffs. Two of the NS16450 pins, no connect (NC) and chip select out (CSOUT) have been replaced by the RxRDY and TxRDY pins. Most serial cards that currently use the NS16450 don't use these pins, so in those situations the NS16550A could be used as a plug-in upgrade. The software drivers for the NS16550A operating in FIFO mode need to be a little more sophisticated than for the NS16450. This will not cause a great penalty in CPU operating time as there is only one additional UART register to program and one to check during the initialization. One additional service routine is required to handle Rx timeout interrupts. This routine does not execute, except during intermittent transmissions or as described above.

All of these speed improvements and allowances for software constraints will make the NS16550A an optimal UART for both multi-tasking systems and multiport systems. Multitasking systems benefit from the increased time and flexibility offered to the CPU during context switching. Multiport systems, such as terminal concentrators, benefit from the on-board FIFOs and relatively autonomous functions of the UART.

SYSTEM INTERRUPT GENERATION

As a prelude to the topic of the next section (80286™-based system initialization) a review of a typical PC hardware interrupt path is given. This concerns only the interrupt path between the UART and the CPU (see *Figure 3*).

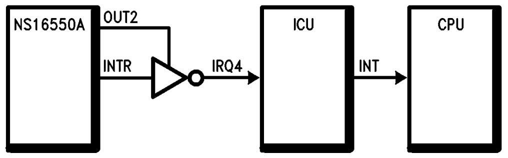


FIGURE 3. Typical PC Interrupt System Hardware

In order to enable interrupts from the UART to the CPU each hardware device must be correctly initialized. While initializing the hardware path, CPU interrupts are turned off to avoid false interrupts from this path. This initialization should be as short as possible to avoid other devices "stacking up" interrupts during this time.

After the NS16550A is initialized the bits 0–3 in the Interrupt Enable Register (IER) are set enabling all UART interrupts. Also, bit 3 in the Modem Control Register (MCR) is set to enable the buffer between the UART and the ICU.

The ICU has bit 4 of its Interrupt Mask Register (IMR) cleared, allowing interrupts occuring on IRQ4 to be transferred to the CPU via the group interrupt (INT). Finally, CPU interrupts are enabled again via the STI instruction.

The programmer should be aware that the ICU will be initialized for edge-triggered interrupts and that the UART always produces level active interrupts. This allows the system to get into a situation where the UART has multiple interrupts pending (signaled via a constantly high INTR), but the ICU fails to respond because it expects an edge for each pending interrupt. To avoid this situation, the programmer should disable all UART interrupts via the IER when entering each UART interrupt service routine and then reenable all UART interrupts that are to be used just before exiting each interrupt service routine.

SUMMARY

Up to this point the features of the NS16550A have been described, some of the design goals that resulted in these features have been reviewed, and a comparison has been given between it and the NS16450. Increases in bus speed and specialized functions make this part both faster from the hardware point of view and more efficient from the software point of view.

2.0 NS16550A Initialization

This initialization can be used on any 80286-based system; it enables both FIFOs and all interrupts on the UART. Additional procedures would have to be written to actually transfer data and service interrupts. These procedures would be similar in form to the 32000-based example in the next section, but the code would be different. The general flow of the initialization is shown in *Figure 4* and described below.

DETAILED SOFTWARE DESCRIPTION

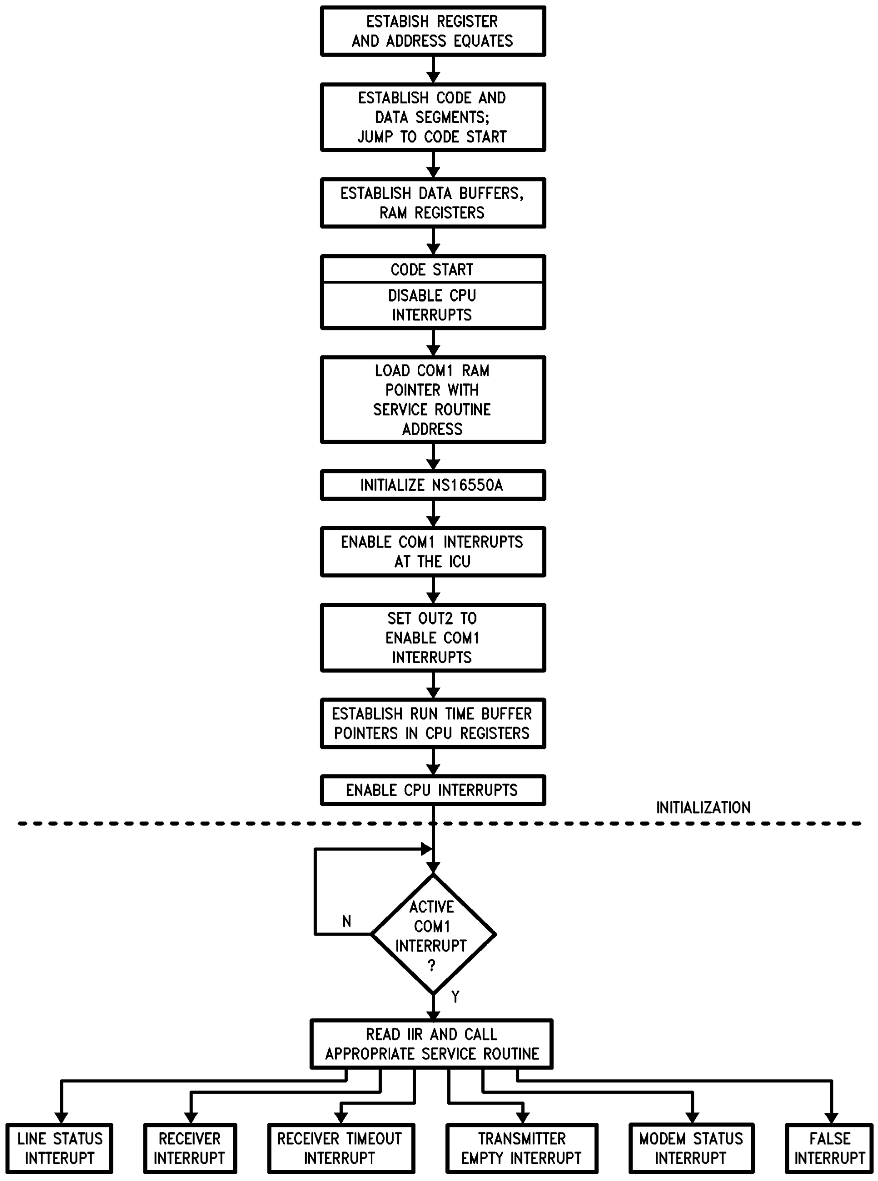
The first block in the initialization establishes abbreviations for the NS16550A registers and assigns addresses to them. The next three blocks establish code and data segments for the 80286. After jumping to the code start, the program disables CPU interrupts (CLI) until it has finished the initialization routine. Other interrupts may be active while CPU interrupts are masked, so the section of code following CLI should be as short as possible. The next block replaces the existing COM1 interrupt vector with the address of NS16550A interrupt handler (INTH in this case).

Initialization of the NS16550A is similar to the NS16450, except that there is one additional register to program which controls the FIFOs (Refer to the datasheet for a complete description). The sequence shown here sets bit 7 (DLAB) of the line control register (LCR), which enables access to the baud rate generator divisor. The divisor programmed is 0006 (19.2 kbaud) in this example. Programming the LCR again resets bit 7 (allowing access to the operational registers) and programs each frame for 7 data bits, one stop bit and even parity. The additional register that needs to be programmed in the NS16550A is the FIFO control register (FCR). The FCR data is 1100 0001. Bits 6 and 7 set the Rx FIFO interrupt trigger level at 14 characters. Bits 5 and 4 are reserved. Bit 3 keeps the DMA signal lines in mode 0. Setting bits 2 and 1 clear the Tx and Rx FIFOs, but this is done automatically when the FIFOs are first enabled by setting bit 0. Bit 0 of the FCR should ALWAYS BE SET whenever changes are to be made to the other bits of the FCR and the UART is to remain in FIFO Mode. When the FIFOs on the NS16550A are enabled bits 6 and 7 in the Interrupt Identification Register are set. Thus the program can distinguish between an NS16450 and an NS16550A, taking advantage of the FIFOs.

Sending a 0F to the Interrupt Enable Register enables all UART interrupts. The next two register accesses, reading the Line Status Register and the Modem Status Register, are optional. They are conservatively included in this initialization in order to defeat false interrupt indications in these registers caused by noise on the external lines.

The next block of code enables the interrupt signal to go beyond the UART through the system hardware. In many popular 80286-based personal computers, an interrupt control unit (ICU) has its mask register at I/O address 21H. To enable interrupts through this ICU for COM1 without disturbing other interrupts, the Interrupt Mask Register (IMR) is read. This data is combined with 1110 1111 via an AND instruction to unmask the COM1 interrupt and then loaded it back to the IMR. On these personal computers there is also a buffer on the interrupt line between the UART and ICU. This buffer is enabled by setting the OUT2 bit of the MODEM Control Register in the UART.

Before enabling CPU interrupts (STI) pointer registers to the data buffers of each service routine are loaded. After enabling CPU interrupts this program jumps to a holding loop to wait for an interrupt, whereas most programs would continue initializing other devices or jump to the system loop.


ESTABISH REGISTER AND ADDRESS EQUATES
ESTABLISH CODE AND DATA SEGMENTS; JUMP TO CODE START
ESTABLISH DATA BUFFERS, RAM REGISTERS
CODE START
DISABLE CPU INTERRUPTS
LOAD COM1 RAM POINTER WITH SERVICE ROUTINE ADDRESS
INITIALIZE NS16550A
ENABLE COM1 INTERRUPTS AT THE ICU
SET OUT2 TO ENABLE COM1 INTERRUPTS
ESTABLISH RUN TIME BUFFER POINTERS IN CPU REGISTERS
ENABLE CPU INTERRUPTS
INITIALIZATION
ACTIVE COM1 INTERRUPT ?
N
Y
READ IIR AND CALL APPROPRIATE SERVICE ROUTINE
LINE STATUS INTTERUPT
RECEIVER INTERRUPT
RECEIVER TIMEOUT INTERRUPT
TRANSMITTER EMPTY INTERRUPT
MODEM STATUS INTERRUPT
FALSE INTERRUPT

TL/C/9313–4

FIGURE 4. NS16550A Initialization and Driver Flowchart

```
        TITLE   550APP.ASM - NS16550A INITIALIZATION
;
;ESTABLISH NS16550A REGISTER ADDRESS/DATA EQUATES
;
;************ UART REGISTERS ************************
;
rxd     EQU 3F8H         ;RECEIVE DATA REG
txd     EQU 3F8H         ;TRANSMITT DATA REG
ier     EQU 3F9H         ;INTERRUPT ENABLE REG
dll     EQU 3F8H         ;DIVISOR LATCH LOW
dlh     EQU 3F9H         ;DIVISOR LATCH HIGH
iir     EQU 3FAH         ;INTERRUPT IDENTIFICATION REG
fcr     EQU 3FAH         ;FIFO CONTROL REG
lcr     EQU 3FBH         ;LINE CONTROL REG
mcr     EQU 3FCH         ;MODEM CONTROL REG
lsr     EQU 3FDH         ;LINE STATUS REG
msr     EQU 3FEH         ;MODEM STATUS REG
scr     EQU 3FFH         ;SCRATCH PAD REG
;
;**************** DATA EQUATES *****************
;
bufsize EQU 7CFH         ;TX AND RX BUFFER SIZE
dosrout EQU 25H          ;DOS ROUTINE SPECIFICATION
intnum  EQU 0CH          ;INTERRUPT NUMBER (0CH = COM1)
icumask EQU 0EFH         ;ICU INTERRUPT ENABLE MASK
divacc  EQU 80H          ;DIVISOR LATCH ACCESS CODE
lowdiv  EQU 06H          ;LOWER DIVISOR
uppdiv  EQU 00H          ;UPPER DIVISOR
dataspc EQU 1AH          ;DLAB = 0, 7 BITS, 1 STOP, EVEN
fifospc EQU 0C1H         ;FIFOS ENABLED, TRIG = 14, DMA MODE = 0
setout2 EQU 08H          ;SETTING OUT2 ENABLES INTRs TO THE ICU
intmask EQU 0FH          ;UART INTERRUPT ENABLE MASK
;
;*********** ESTABLISH CODE AND DATA SEGMENTS ******************
;
cseg    SEGMENT PARA PUBLIC "code"
        ORG     100H
        ASSUME  CS:cseg,DS:cseg
INIT:
        PUSH    CS
        POP     DS
        JMP     START
;
;********* ESTABLISH DATA BUFFERS AND RAM REGISTERS ********
;
msflag  DB      0
txflag  DB      0
sbuf    DB      bufsize DUP ("S")       ; STRING BUFFER
rbuf    DB      bufsize DUP ("R")       ; RECEIVE BUFFER
sbufe   EQU     sbuf + bufsize          ; END OF STRING BUFFER
rbufe   EQU     rbuf + bufsize          ; END OF RECEIVE BUFFER
;
START:
        CLI                     ;>>> DISABLE CPU INTERRUPTS <<<
```

```
;
;****** LOAD NEW INTERRUPT SERVICE ROUTINE POINTER FOR COM1 ***
;
       PUSH    DS               ;SAVE EXISTING DATA SEG
       MOV     AH,dosrout       ;DESIGNATE FUNCTION NUMBER
       MOV     AL,intnum        ;DESIGNATE INTERRUPT
       PUSH    CS               ;ALIGN CODE SEG
       POP     DS               ;WITH DATA SEG
       MOV     DX,OFFSET INTH   ;SPECIFY SERVICE ROUTINE OFFSET
       INT     21H              ;REPLACE EXISTING INTR VECTOR
       POP     DS               ;RESTORE CURRENT DATA SEG
;
;****************  INITIALIZE NS16550A  **********************
;
;This enables both FIFOs for data transfers at 19.2 kbaud using
;7 bit data, 1 stop bit and even parity. The Rx FIFO interrupt
;trigger level is set at 14 bytes.
       MOV     AL,divacc        ;SET-UP ACCESS TO DIVISOR LATCH
       MOV     DX,lcr
       OUT     DX,AL
       MOV     AL,lowdiv        ;LOWER DIVISOR LATCH, 19.2 kbaud
       MOV     DX,dll
       OUT     DX,AL
       MOV     AL,uppdiv        ;UPPER DIVISOR LATCH
       MOV     DX,dlh
       OUT     DX,AL
       MOV     AL,dataspc       ;DLAB = 0, 7 BITS, 1 STOP, EVEN
       MOV     DX,lcr
       OUT     DX,AL
       MOV     AL,fifospc       ;FIFOS ENABLED, TRIGGER = 14,
       MOV     DX,fcr           ;DMA MODE = 0
       OUT     DX,AL
       MOV     AL,intmask       ;ENABLE ALL UART INTERRUPTS
       MOV     DX,ier
       OUT     DX,AL
       MOV     DX,lsr           ;READ THE LSR TO CLEAR ANY FALSE
       IN      AL,DX            ;STATUS INTERRUPTS
       MOV     DX,msr           ;READ THE MSR TO CLEAR ANY FALSE
       IN      AL,DX            ;MODEM INTERRUPTS
;
;*************** ENABLE COM1 INTERRUPTS *********************
;
       IN      AL,21H           ;CHECK IMR
       AND     AL,icumask       ;ENABLE ALL EXISTING AND COM1
       OUT     21H,AL
       MOV     AL,setout2       ;SET OUT2 TO ENABLE INTR
       MOV     DX,mcr
       OUT     DX,AL
;
;*********  ESTABLISH RUN TIME BUFFER POINTERS IN REGISTERS ***
;
       MOV     SI,OFFSET sbuf
       MOV     DI,OFFSET rbuf
       MOV     BX,OFFSET sbuf
       MOV     BP,OFFSET rbuf
       STI      ;>>> ENABLE CPU INTERRUPTS <<<
```

3.0 Board to Board Communications with the NS16550A

The following section describes the hardware and software for a fully asynchronous two board application. The two boards communicate simultaneously with each other via the NS16550As. Predetermined data is exchanged between the NS16550As and checked by the software for accuracy. Any data mismatches are flagged and stop the programs. Any data errors (i.e. overrun, parity, framing or break) will also stop the program. The NS16550A interface schematic, software flow chart and software are provided.

HARDWARE REQUIREMENTS

Running this application requires two NS32032-based boards. Each board must have one CPU, one ICU (NS32202), 256k of RAM (000000–03FFFF), the capability of running a monitor program (MON 32) and the capability of interfacing with a terminal. If MON 32 is not available, the display monitor service calls (SVC) must be altered to interface properly to the available terminal driver routines. In addition to these requirements, the NS16550A is enabled starting at address 0d00000.

The system described above was implemented on two DB32032 boards and used as an alpha site to test the NS16550A during its development. An NS16550A and appropriate decode logic were wirewrapped to each board (see *Figure 5*). As shown, an 8 MHz crystal is used to drive the baud rate generator, but for baud rates at or below 56 kbaud a 1.8432 MHz crystal can be substituted with changes to the divisor. Once this hardware is on both boards 5 connections between the NS16550As must be made—SIN to SOUT, SOUT to SIN, $\overline{\text{CTS}}$ to $\overline{\text{RTS}}$, $\overline{\text{RTS}}$ to $\overline{\text{CTS}}$, and GND to GND. Each DB32032 board has a port for attaching a terminal and a port available for downloading code. The applications software for these boards is downloaded from a VAX™ running the GNX™ debugger (V1.02). Once the downloads are complete to both boards the program D1APPS.EXE is started, then D2APPS.EXE is started.

If a VAX or the GNX debugger is not available the code can be loaded into PROMs and run directly.

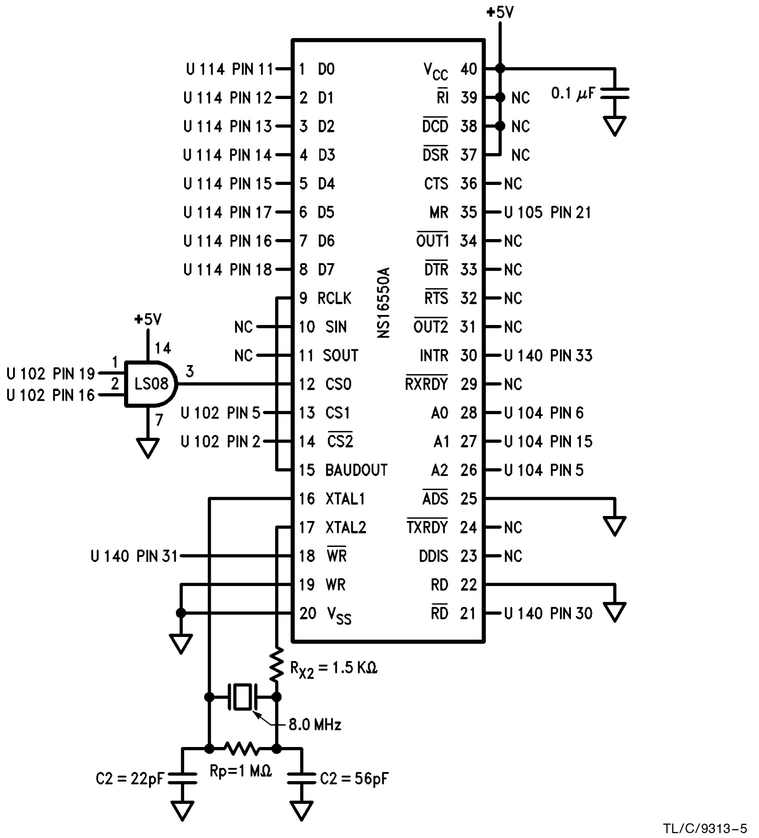


FIGURE 5. NS16550A and DB32032 Board Interconnections

SOFTWARE OVERVIEW

The programs shown at the end of this application note are the assembly listings for D1APPS.ASM and D2APPS.ASM. These can be assembled, linked and loaded to form the executable (.EXE) files. The flowchart shown before them illustrates both programs.

Both programs are interrupt driven. D1APPS.EXE has its transmitter empty interrupt disabled until it receives its first 16 bytes from D2APPS.EXE. This allows the two programs to be started at different times. Data flow is controlled between the programs via $\overline{\text{RTS}}$ and $\overline{\text{CTS}}$ handshakes. D1APPS.EXE is started first and it loops until the first data from D2APPS.EXE arrives. As D1APPS.EXE exits its receiver interrupt routine, it enables its transmitter interrupt and begins to send bytes to D2APPS.EXE.

Transmission of a block of 16 bytes occurs when the Tx FIFO of the NS16550A is empty, the Tx interrupt is enabled and the receiver activates its clear to send ($\overline{\text{CTS}}$) signal. Each transmitter sends the next sequential block of data from a 256 byte buffer. When the bottom of the buffer is reached, the transmitter starts at the top of the buffer, again. The data transmitted from D1APPS.EXE to D2APPS.EXE is 00 to FF and from D2APPS.EXE to D1APPS.EXE is FF to 00. Since these are bench test programs for the NS16550A, the receiver subroutines compare the data they receive with the data they expect. This is done on a block-by-block basis and any mismatches result in both a message sent to the terminal and the program stopping.

DETAILED SOFTWARE DESCRIPTION

Initialization begins by equating NS16550A and ICU (NS32202) registers to the addresses in memory. The equates finish with a list of offsets associated with the static base register. These offsets give the starting locations for the RAM areas assigned to be data buffers. These include the UART interrupt entry offset (irl__mod); the string (sbuf), receive (rbuf), compare (cbuf) buffers and the interrupt table offset (intable).

At the code start (START::) the processor is put in the supervisor mode so that the interrupt dispatch table can be transferred from ROM to RAM. This transfer is essential in order to change the starting address of the UART interrupt service routine. To do this the interrupt service routine offset from the code start is calculated (isr-start). Combining this with the module table address (set-up by the linker, i.e., 9020) results in the interrupt table descriptor entry for UART interrupt service routine (isrent).

The next two sections of code load the data to be transmitted and compared into the RAM buffers sbuf and cbuf, respectively. The two programs differ at this point—D1APPS.EXE transmits 00 to FF and compares FF to 00 sequentially. D2APPS.EXE transmits FF to 00 and compares 00 to FF sequentially.

The NS16550A initialization starts with setting the divisor latch access bit, so the divisor can be loaded. It then determines the serial data format and disables all UART interrupts. The NS16550A initialization finishes by enabling and resetting the FIFOs and programming the receiver interrupt level for 14 bytes.

Next the ICU interrupt registers are set-up and interrupts are enabled. In program D1APPS.ASM the initialization finishes by enabling the receive data and line status interrupts. Since the transmitter FIFO empty interrupt is disabled D1APPS.EXE will stay in its hold loop until it receives data from D2APPS.EXE. D2APPS.EXE has its transmitter FIFO empty interrupt enabled at the end of its initialization, so it will send one block of 16 characters to D1APPS.EXE immediately.

When there are no interrupts pending and no service routines being executed, the programs run in a holding loop until the next interrupt.

Whenever the CPU enters the service routine (isr:) it checks the interrupts identification register (IIR) for the type of interrupt pending and branches to the appropriate subroutine. If the IIR value doesn't match a known interrupt condition, an invalid interrupt message is sent to the terminal and the program stops. Out of the five possible interrupts, two (line status and receiver timeout) have simple routines that only send a message to the terminal and then branch to the receiver data available routine. Modem status interrupts send a message to the CRT and then stop the program. Two robust interrupt service routines exist—one for the receiver and one for the transmitter.

The receiver interrupt service routine (rdai:) does the following:

1. Disables the $\overline{\text{RTS}}$ signal which stops the transmitter on the other board from sending more data.
2. Transfers all data from the UART Rx FIFO to the RAM receiver buffer (rbuf).
3. Branches to the compare subroutine when all data is transferred from the Rx FIFO.
4. Enables Tx interrupts in D1APPS.EXE.
5. Enables the $\overline{\text{RTS}}$ signal which allows the transmitter on the other board to send another block of data.

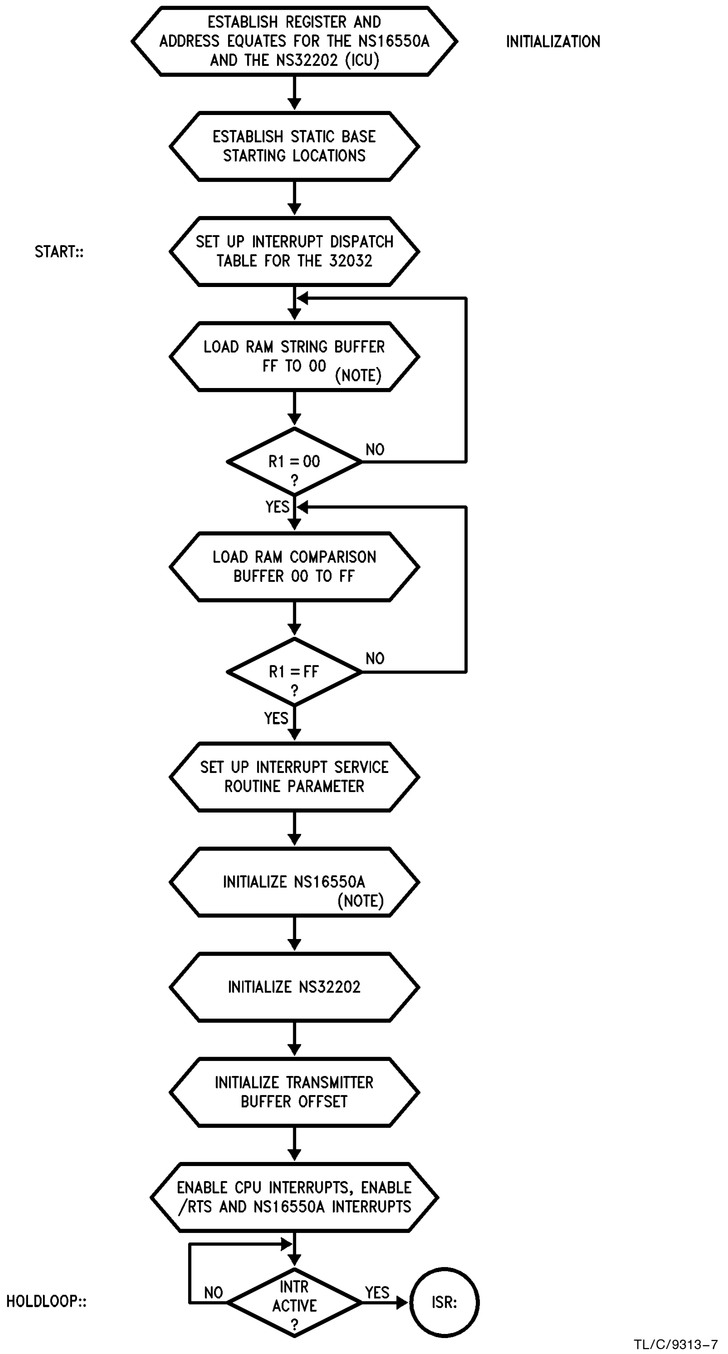
The compare interrupt service routine (compare:) does the following:

1. Aligns the receive buffer pointer to the last character taken in to the receive buffer (rbuf).
2. Compares each new byte in rbuf with the expected value (data stored in cbuf).
3. Sends a data mismatch message to the terminal and stops the program if the rbuf data fails to match the cbuf data.
4. Returns to rdai: when all of the new data in rbuf has compared successfully.

The transmitter interrupt service routine (threi:) does the following:

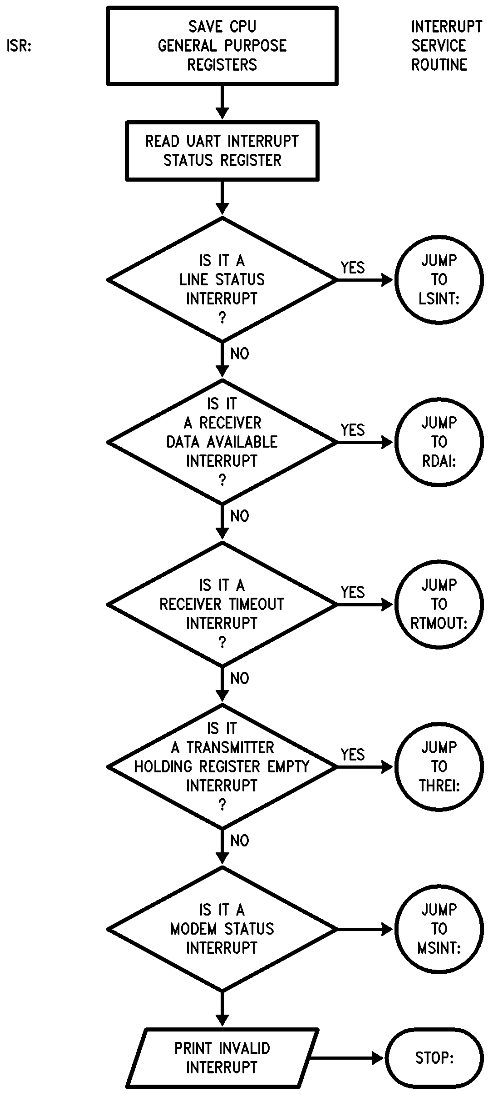
1. Decides whether to send 16 or 15 bytes in a block of data. **Note:** This decision is for testing purposes.
2. Sends one byte of data.
3. Checks for an active $\overline{\text{CTS}}$ condition. If it is active then it sends another byte of data. It continues to check and send a byte of data until all 15 or 16 bytes are sent.

DIAPPS.ASM Flow Chart

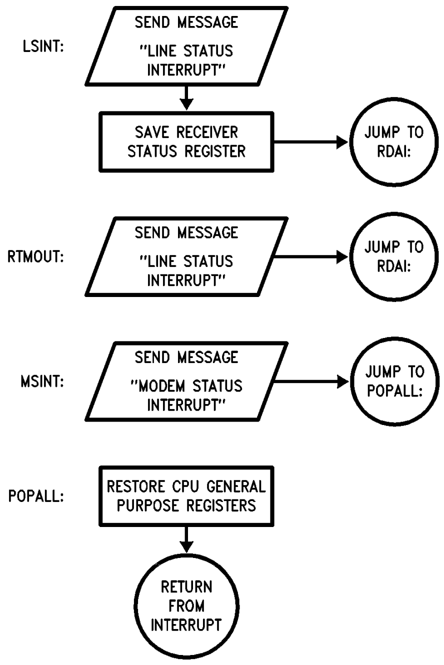

ESTABLISH REGISTER AND ADDRESS EQUATES FOR THE NS16550A AND THE NS32202 (ICU)
INITIALIZATION
ESTABLISH STATIC BASE STARTING LOCATIONS
START::
SET UP INTERRUPT DISPATCH TABLE FOR THE 32032
LOAD RAM STRING BUFFER FF TO 00 (NOTE)
R1 = 00 ?
NO
YES
LOAD RAM COMPARISON BUFFER 00 TO FF
R1 = FF ?
NO
YES
SET UP INTERRUPT SERVICE ROUTINE PARAMETER
INITIALIZE NS16550A (NOTE)
INITIALIZE NS32202
INITIALIZE TRANSMITTER BUFFER OFFSET
ENABLE CPU INTERRUPTS, ENABLE /RTS AND NS16550A INTERRUPTS
HOLDLOOP::
NO
INTR ACTIVE ?
YES
ISR:

TL/C/9313–7

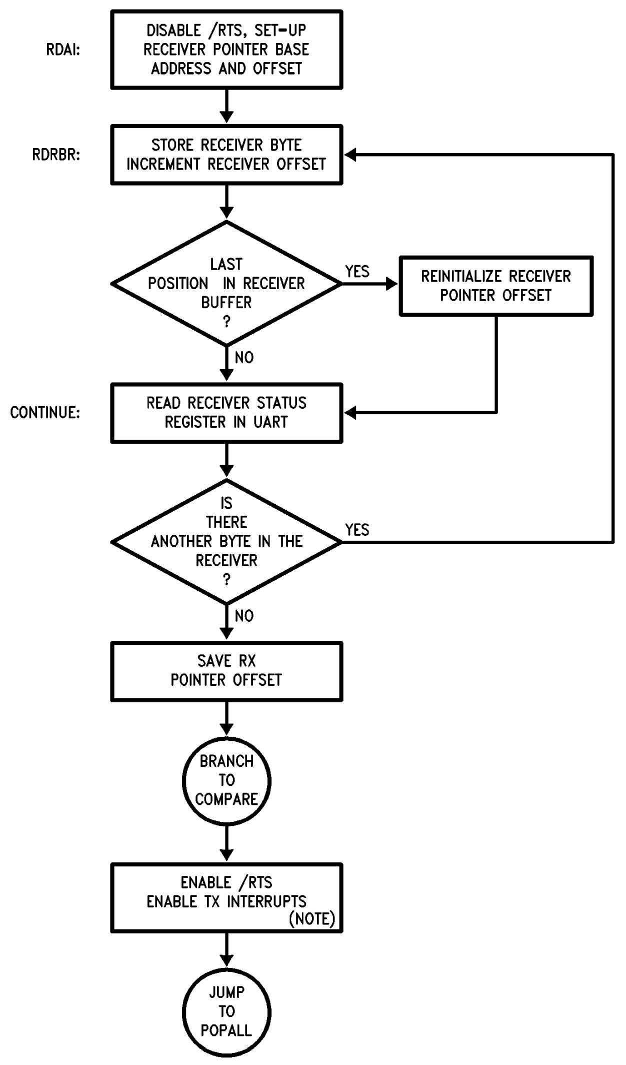
Note: This part of the software differs slightly in D2APPS.ASM


ISR:
SAVE CPU GENERAL PURPOSE REGISTERS
INTERRUPT SERVICE ROUTINE
READ UART INTERRUPT STATUS REGISTER
IS IT A LINE STATUS INTERRUPT ?
YES
JUMP TO LSINT:
NO
IS IT A RECEIVER DATA AVAILABLE INTERRUPT ?
YES
JUMP TO RDAI:
NO
IS IT A RECEIVER TIMEOUT INTERRUPT ?
YES
JUMP TO RTMOUT:
NO
IS IT A TRANSMITTER HOLDING REGISTER EMPTY INTERRUPT ?
YES
JUMP TO THREI:
NO
IS IT A MODEM STATUS INTERRUPT
JUMP TO MSINT:
PRINT INVALID INTERRUPT
STOP:

TL/C/9313–8

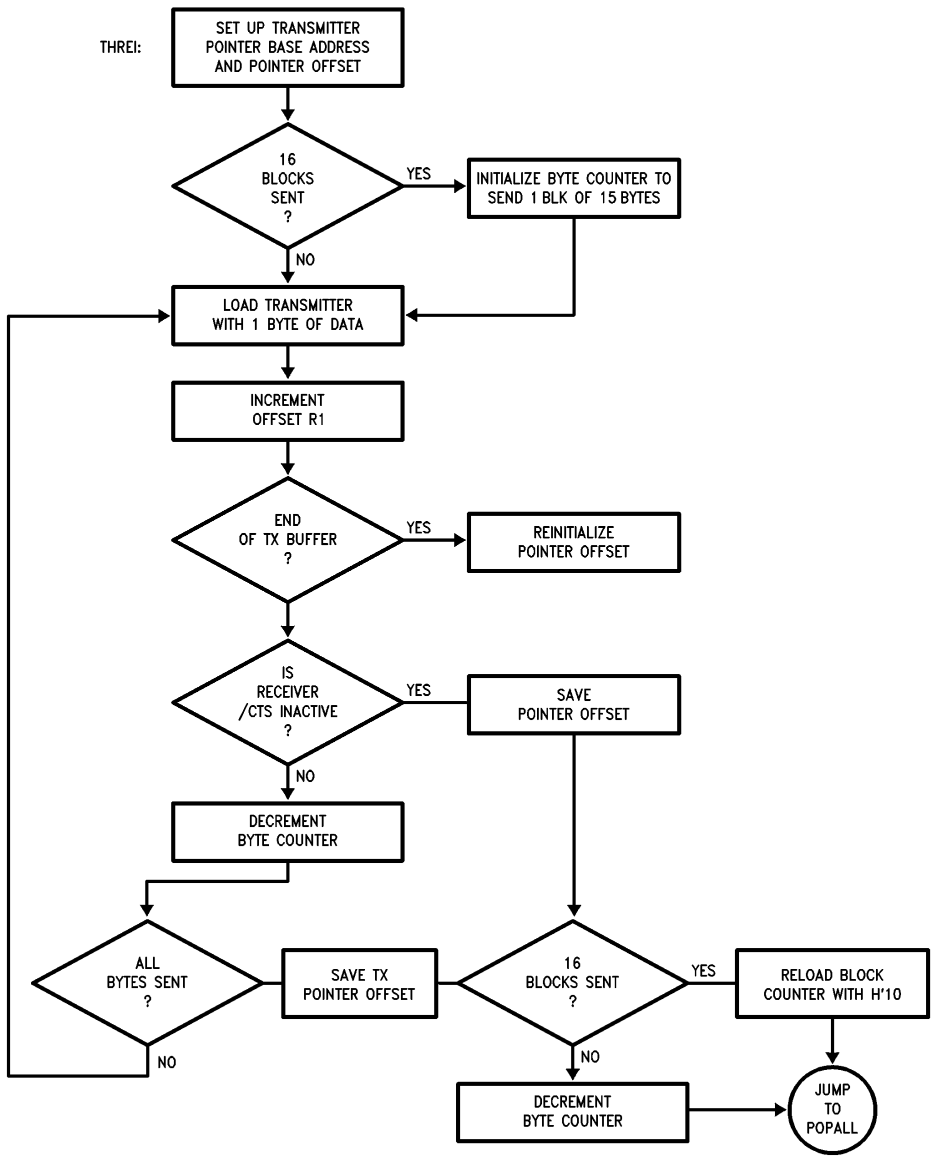

LSINT:
SEND MESSAGE "LINE STATUS INTERRUPT"
SAVE RECEIVER STATUS REGISTER
JUMP TO RDAI:
RTMOUT:
SEND MESSAGE "LINE STATUS INTERRUPT"
JUMP TO RDAI:
MSINT:
SEND MESSAGE "MODEM STATUS INTERRUPT"
JUMP TO POPALL:
POPALL:
RESTORE CPU GENERAL PURPOSE REGISTERS
RETURN FROM INTERRUPT

TL/C/9313–9

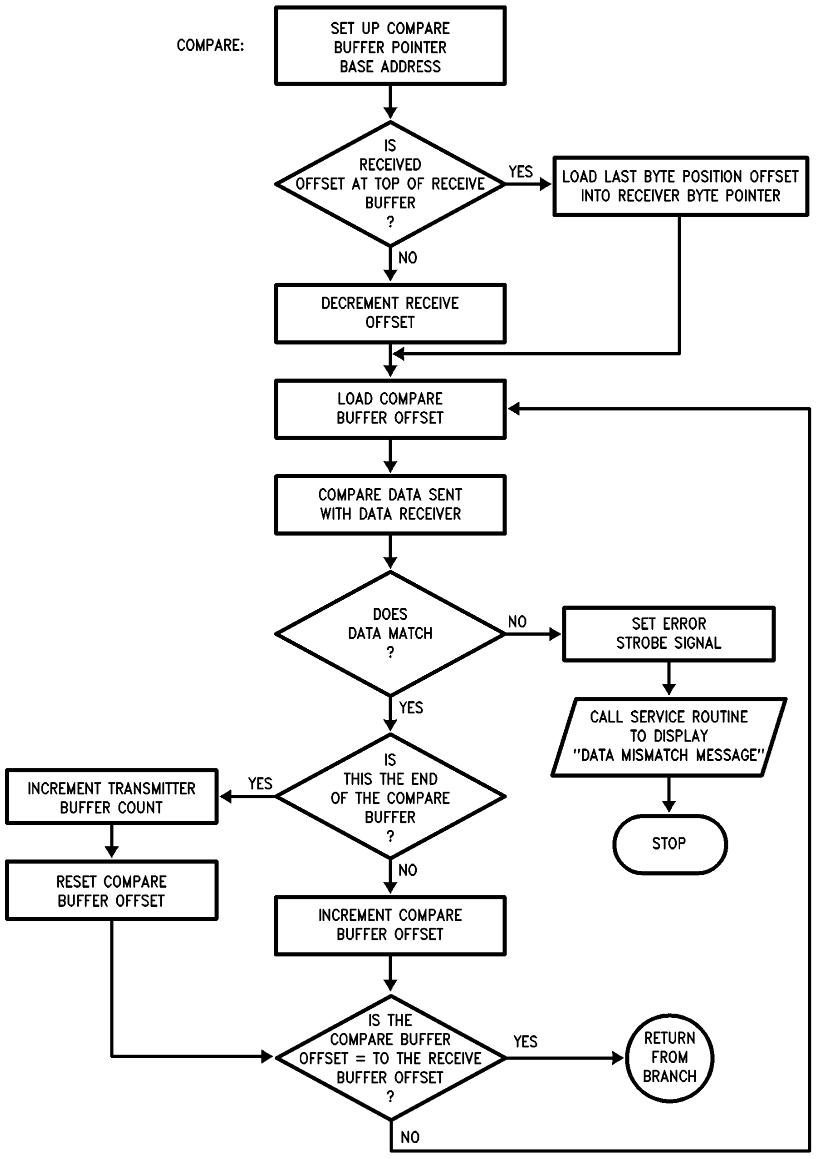

RDAI:
DISABLE /RTS, SET-UP RECEIVER POINTER BASE ADDRESS AND OFFSET
RDRBR:
STORE RECEIVER BYTE INCREMENT RECEIVER OFFSET
LAST POSITION IN RECEIVER BUFFER ?
YES
REINITIALIZE RECEIVER POINTER OFFSET
NO
CONTINUE:
READ RECEIVER STATUS REGISTER IN UART
IS THERE ANOTHER BYTE IN THE RECEIVER ?
YES
NO
SAVE RX POINTER OFFSET
BRANCH TO COMPARE
ENABLE /RTS ENABLE TX INTERRUPTS (NOTE)
JUMP TO POPALL

TL/C/9313–10

Note: This part of the software differs slightly in D2APPS.ASM


THREI:
SET UP TRANSMITTER POINTER BASE ADDRESS AND POINTER OFFSET
16 BLOCKS SENT ?
YES
INITIALIZE BYTE COUNTER TO SEND 1 BLK OF 15 BYTES
NO
LOAD TRANSMITTER WITH 1 BYTE OF DATA
INCREMENT OFFSET R1
END OF TX BUFFER ?
YES
REINITIALIZE POINTER OFFSET
IS RECEIVER /CTS INACTIVE ?
YES
SAVE POINTER OFFSET
NO
DECREMENT BYTE COUNTER
ALL BYTES SENT ?
NO
SAVE TX POINTER OFFSET
16 BLOCKS SENT ?
YES
RELOAD BLOCK COUNTER WITH H'10
NO
DECREMENT BYTE COUNTER
JUMP TO POPALL

TL/C/9313–11


COMPARE:
SET UP COMPARE BUFFER POINTER BASE ADDRESS
IS RECEIVED OFFSET AT TOP OF RECEIVE BUFFER ?
YES
LOAD LAST BYTE POSITION OFFSET INTO RECEIVER BYTE POINTER
NO
DECREMENT RECEIVE OFFSET
LOAD COMPARE BUFFER OFFSET
COMPARE DATA SENT WITH DATA RECEIVER
DOES DATA MATCH ?
NO
SET ERROR STROBE SIGNAL
CALL SERVICE ROUTINE TO DISPLAY "DATA MISMATCH MESSAGE"
STOP
YES
IS THIS THE END OF THE COMPARE BUFFER ?
YES
INCREMENT TRANSMITTER BUFFER COUNT
RESET COMPARE BUFFER OFFSET
NO
INCREMENT COMPARE BUFFER OFFSET
IS THE COMPARE BUFFER OFFSET = TO THE RECEIVE BUFFER OFFSET ?
YES
RETURN FROM BRANCH
NO

TL/C/9313–12

```
#3/30/87.....D1APPS.ASM.........ADAPTED ORIGINALLY FROM D1RON56K.ASM
#
#THIS PROGRAM RUNS USING 2 DB32000 BOARDS WITH 16550As ENABLED AT ADDRESS 0d00000
#WIRE-WRAPPED ON THE BOARDS. THIS SOFTWARE TRANSMITS THE DATA 00 THROUGH FF
#REPEATEDLY TO THE REMOTE UART AND EXPECTS TO REPEATEDLY RECEIVE THE DATA FF
#THROUGH 00 FROM THE REMOTE UART. IT SHOULD BE RUN IN CONJUNCTION WITH THE
#PROGRAM D2APPSC.ASM RUNNING ON THE OTHER DB32000 BOARD. THE TX PIN OF
#THIS 16550A SHOULD CONNECT TO THE RX PIN OF THE 16550A ON THE OTHER BOARD AND
#VICE VERSA. ALSO, THE CTS PIN OF THIS 16550A SHOULD BE CONNECTED TO THE RTS PIN
#OF THE 16550A ON THE OTHER BOARD AND VICE VERSA. THIS WILL ENABLE THE
# APPROPRIATE HANDSHAKES TO OCCUR.
#
#TO RUN THIS PROGRAM YOU MUST:
#
#       1. CONNECT THE RX & TX OF THE 2 16550As ON THE 2 DB32000 BOARDS
#       2. CONNECT THE CTS & RTS OF THE 2 16550As ON THE 2 DB32000 BOARDS
#       3. DOWNLOAD D1APPS.EXE TO THIS BOARD VIA THE GNX DEBUGGER [REV 1.02]
#       4. DOWNLOAD D2APPS.EXE TO OTHER BOARD VIA THE GNX DEBUGGER [REV 1.02]
#       5. START D1APPS.EXE RUNNING ON THIS DB32000 BOARD
#       6. START D2APPS.EXE RUNNING ON THE OTHER DB32000 BOARD
#
#PROGRAM DETAILS:
#
#
# ISR contains the TX SERVICE ROUTINE
#
# TX OVERWRITES are PREVENTED by the ICU
#
# TX FIFO is CLEARED before a transmission
#
# DATA SENT  00 ------ FF
#
# DATA RECEIVED  and COMPARED FF ------ 00
#
# BAUDRATE 128k WITH A 8.0 MHZ XTAL INPUT TO THE 16550A
                                        #
#********************** ESTABLISH 16550A REGISTER ADDRESSES ********************
                                        #
        .globl          isr             #
           .set rxd,     0x0d00000      #Equate registers to their addresses
           .set txd,     0x0d00000      #
           .set ier,     0x0d00004      #
           .set iir,     0x0d00008      #
           .set fcr,     0x0d00008      #
           .set lcr,     0x0d0000c      #
           .set mcr,     0x0d00010      #
           .set lsr,     0x0d00014      #
           .set msreg,   0x0d00018      #
           .set scr,     0x0d0001c      #
                                        #
#****************** ESTABLISH ADDRESSES FOR THE 32202 (ICU) *******************
                                        #
           .set a0,4                    #Establish address alignment
                                        #between CPU and ICU
           .set icu_hvct,0              #ICU register addresses
           .set icu_svct,1 *a0          #
           .set icu_elgt,2 *a0          #
           .set icu_tpl,4  *a0          #
           .set icu_ipnd,6 *a0          #
           .set icu_isrv,8 *a0          #
```

TL/C/9313–13

```
           .set icu_imsk,10 *a0         #
           .set icu_csrc,12 *a0         #
           .set icu_fprt,14 *a0         #
           .set icu_mctl,16 *a0         #
           .set icu_ciptr,18 *a0        #
           .set icu_pdat,19  *a0        #
           .set icu_ips,20   *a0        #
           .set icu_pdir,21  *a0        #
           .set icu_cctl,22  *a0        #
           .set icu_cictl,23 *a0        #
                                        #
                                        #First ICU register address
                                        #
           .set icu_addr,0xfffe00       #
                                        #
#************************* STATIC BASE STARTING LOCATIONS *********************
                                        #
           .set irl_mod, 17*4           #
           .set irl_off, 17*4+2         #
           .set start2, 0x0             #The following are static base variables
           .set start1, 0x0a            #used as base pointers. Start1/2 = flags
           .set txflag, 0x14            #txflaf = flag, sbuf = area used to
           .set sbuf, 0x1e              #store data to be transmitted, rbuf =
           .set rbuf, 0x41e             #area used to store received data,
           .set cbuf, 0x61e             #cbuf = area used to store compare
           .set intable, 0x81e          #buffer, intable = base pointer to the
                                        #interrupt table
                                        #
#********************** SET UP DISPATCH TABLE FOR THE 32032 *******************
                                        #
start::         bicpsrw $(0x100)        #Clear intr's
                movd $0x0c,r0           #Set for monitor svc to move intbase
                movd $0x055555555,r1    #from ROM to ram because you have
                addr intable(sb),r2     #to change the address for the
                movd $0x0c,r3           #interrupt service routine.
                svc                     #Actual svc for move
                sprd intbase,r2         #Put base addr of intbase in r2
                movd isrent,irl_mod(r2) #Put offset of isr into 1st location
                                        #of dispatch table
                                        #
#********************* LOAD TRANSMITTER BUFFER (00 to FF) *********************
                                        #
senddat:        addr  sbuf(sb),r0       #R0 contains string buffer ptr.
                movd  $0,r1             #R1 contains offset
                movb  $0,r2             #Init data reg.
sbufloop:       movb  r2,0(r0)[r1:b]    #Load char. to string buffer
                addqw 1,r1              #Increment offset ptr.
                addqw 1,r2              #Increment data
                cmpw  r1,$256           #Check for 256 chars. loaded
                bne   sbufloop          #Jump back if not done
                                        #
#********************** LOAD COMPARISON BUFFER (FF TO 00)*********************
                                        #
compdat:        addr  cbuf(sb),r0       #R0 contains pointer
                movd  $0,r1             #R1 contains offset
                movb  $0x0ff,r2         #Init data reg.
cbufloop:       movb  r2,0(r0)[r1:b]    #Load char. to compare buffer
                addqw 1,r1              #Increment ptr. offset
                subb  $1,r2             #Decrement data
                cmpw  r1,$256           #Check for 256 chars. loaded
```

TL/C/9313–14

```
                bne    cbufloop           #Jump back if not done
                                          #
#*************** SET UP INTERRUPT SERVICE ROUTINE PARAMETERS *******************
                                          #
                movd $0x0ff,start2(sb)    #Initialize compare
                movd $0x0ff,start1(sb)    #Initialize receiver data intr
                movd $16,blk16cnt         #Initialize 16 byte block counter
                movd $0,sbufcnt           #Initialize string bufffer transmitted
                                          #count
                                          #
#*************************** 16550A INITIALIZATION ******************************
                                          #
                movb  $0x080,lcr          #Set dlab = 1 for divisor latch access
                movb  $4,txd              #Low divisor latch 128k w/8.0 MHz xtal
                movb  $0,ier              #Upper divisor latch
                movb  $0x003,lcr          #Dlab = 0, 8 bits, no parity, 1 stop
                movb $0,ier               #Disable UART interrupts
                movb  $0x0c7,fcr          #Fifo=> trigger = 14, reset & enable
                                          #
#**************************** INITIALIZE 32202 (ICU) ***************************
                                          #
           movd    $icu_addr,r0           #R0 = icu address
           movb    $0xca,icu_mctl(r0)     #Set mode : 8 bit bus mode,
                                          #           freeze counters,
                                          #           disable interrupts,
                                          #           fixed priority.
           movqb 0,icu_cctl(r0)           #Halt the counters
           movqb -1,icu_ips(r0)           #Set all pins to interrupt source
           movqb 0,icu_csrc(r0)           #No cascaded interrupts (low reg)
           movqb 0,icu_csrc+a0(r0)        # (high reg)
           movb $0x10,icu_svct(r0)        #Set interrupt base vector
           movqb -1,icu_elgt(r0)          #Set level triggering mode (low reg)
           movqb -1,icu_elgt+a0(r0)       #(high reg)
           movqb $2,icu_tpl(r0)           #Set level triggering mode (low reg)
           movqb 0,icu_tpl+a0(r0)         #(high reg)
           movqb 0,icu_fprt(r0)           #Set highest priority to 0 (low reg)
           movqb 0,icu_fprt+a0            #(high reg)
           movqb 0,icu_isrv(r0)           #Clear intr in-service regs (low reg)
           movqb 0,icu_isrv+a0(r0)        #(high reg)
           movqb -1,icu_imsk(r0)          #Mask all intr (low reg)
           movqb -1,icu_imsk+a0(r0)       #(high reg)H
           setcfg [i]                     #Enable vectored intrp (I=1)
           movd $icu_addr,r0              #
           movb $0x02,icu_mctl(r0)        #Fixed mode, 8 bit bus mode
           movb $0x010,icu_cctl(r0)       #Set to internal sampling
           movb $0xfd,icu_imsk(r0)        #Enable ir1
           movb $0xff,icu_imsk+a0(r0)     #Mask all other interrupts
           bispsrw $(0x800)               #Enable cpu intr's
                                          #
           movd $0,r1                     #Initialize transmitter buffer offset
                                          #
#*************************** ENABLE 16550A INTERRUPTS ***************************
                                          #
                movb $2,mcr               #Clear out1, out2 and enable rts
endinit:        movb $0x05,ier            #Enable all but modem status interrupts
                                          #and the THRE so the boards can be
                                          #started.
                                          #
#********************** ENDLESS LOOP WAITING FOR INTERRUPTS ********************
                                          #
```

TL/C/9313–15

```
holdloop:       nop                     #
                br holdloop             #
                                        #
#***************************** INTERRUPT HANDLER ******************************
                                        #
isr:            save [r0,r1,r2,r3,r4,r5,r6,r7]
                movb iir,r0             #R0- contains iir
                cmpb r0,$0x0c6          #
                beq lsint               #Line status interrupt
                cmpb r0,$0x0c4          #
                beq rdai                #Receiver interrupt
                cmpb r0,$0x0cc          #
                beq rtmout              #Rec timeout interrupt
                cmpb r0,$0x0c2          #
                beq threi               #THRE interrupt
                cmpb r0,$0x0c0          #
                beq msint               #Modem status interrupt
                                        #
                                        #
#************************* INVALID INTERRUPT ROUTINE *************************
                                        #
                save [r0,r1,r2,r3]      #
                movd $4,r0              #
                addr message2,r1        #
                movd $21,r2             #
                movd $0,r3              #
                svc                     #
                restore [r0,r1,r2,r3]   #
                                        #
                                        #
                jump stop               #Restore all registers
                                        #
                                        #
#******************** RECEIVER TIMEOUT INTERRUPT ROUTINE **********************
                                        #
rtmout:         jump rdai
                                        #
#************************** RECEIVER INTERRUPT ROUTINE ************************
                                        #
#This portion of the program is reached by a positive test for the received data
#available interrupt. Once in this routine each byte is removed from the FIFO,
#placed in a designated static base memory location and the LSR is tested to see
#if the data ready (DR) bit is still set. Data is removed from the FIFO and
#placed in memory until the DR bit is no longer set. The data sent will be
#compared to known data, located in another designated static base location, by
#calling the compare subroutine.
                                        #
rdai:           movb $0,mcr             #Disable RTS; stop transmission
                addr rbuf(sb),r4        #r4 contains rbuf base address
                movd rbufoff,r6         #Put rbuf offset runner into r6
rdrbr:          movb rxd,0(r4)[r6:b]    #Store a byte in the receiver buffer
                cmpb $0x00,0(r4)[r6:b]  #Is it the last character
                addqw 1,r6              #Increment offset ptr.
                addqw 1,rbufoff         #Track r6
                bne continue            #
                movw $0,r6              #Reset pointer offset
                movw $0,rbufoff         #Reset rbufoff
continue:       movb lsr,r3             #Read lsr
                andb $01,r3             #Mask all but bit 0
                cmpb $01,r3             #
```

TL/C/9313–16

```
                beq rdrbr               #Read rbr again if set
                movd r6,rbufoff         #Put result of r6 back into rbufoff
                bsr compare             #
                movb $7,ier             #Turn on transmitter interrupts
                movb $2,mcr             #Enable rts
                jump popall             #
                                        #
#****************************** TRANSMIT ROUTINE ******************************
                                        #
#Before the transmitter sends data, the data has been loaded into static base
#memory for transmission. The transmitter routine is called to send data. (ie
#THREI is set) Data is sent as 16 blocks of 16 bytes and 1 block of 15 bytes
#continuously. NOTE: Before transmission occurs /CTS is checked to ensure that
#the receiver is ready.
                                        #
threi:          addr sbuf(sb),r0        #R0 contains base pointer
                movw xmitoff,r1         #setup xmit ptr offset
                cmpd $0,blk16cnt        #Check to see if it is the 16th block *
                beq send15              #Yes, send only 15 bytes instead of 16 *
                movd $0x10,r7           #No, send 16 bytes *
                jump sendnext           #Jump around 15 byte load *
send15:         movd $0x0f,r7           #Load counter for 15 byte load *
sendnext:       movb 0(r0)[r1:b],txd    #Load a byte into the transmitter
                addqw 1,r1              #
                cmpw r1,$256            #Are we one address past end of table
                beq reload              #Yes, reload ptr
finish:         save [r7]
                movb msreg,r7           #Read modem status reg
                andb $0x10,r7           #Mask all bits except CTS (MSR4)
                cmpb $0,r7              #Check for disabled CTS
                restore [r7]
                beq abort               #Wait for active CTS (MSR4=1)
                subb $1,r7              #No, decrement counter and continue
                cmpb $0,r7              #Is byte counter 0?
                bne sendnext            #No, send next byte
abort:          movw r1,xmitoff         #save xmit ptr offset in ram
                cmpd $0,blk16cnt        #Check to see if it is 16th block *
                beq setsnd16            #Yes, reload block counter *
                subb $1,blk16cnt        #Decrement block counter *
                jump popall             #Finished sending 16 bytes
setsnd16:       movd $16,blk16cnt       #Reload block counter *
                jump popall             #Finished sending 15 bytes *
reload:         movd $0,r1              #Reset offset
                jump finish             #Go back and finish
                                        #
#************************ LINE STATUS INTERRUPT ROUTINE ************************
                                        #
lsint:          save [r0,r1,r2,r3]      #
                movd $4,r0              #
                addr message6,r1        #
                movd $25,r2             #
                movd $0,r3              #
                svc                     #
                restore [r0,r1,r2,r3]   #
                movb lsr,r3             #Read lsr
                                        #
                                        #
                jump rdai               #
                                        #
#************************ MODEM STATUS INTERRUPT ROUTINE ***********************
```

TL/C/9313–17

```
                                                #
msint:          save [r0,r1,r2,r3]              #
                movd $4,r0                      #
                addr message7,r1                #
                movd $26,r2                     #
                movd $0,r3                      #
                svc                             #
                movb 0x0d00018,r0               #
                restore [r0,r1,r2,r3]           #
                jump popall                     #
#****************************** COMPARE DATA ROUTINE ****************************
                                                #
#This subroutine is called by the receiver interrupt routine which has set the
#receiver offset (rbufoff) to point at the last byte received. This subroutine
#uses the compare offset (compoff) pointer as the pointer for both receive
#buffer data and compare buffer data. Each location is compared to ensure data
#sent is identical to data received. This is done until compoff equals rbufoff
#stopping the process and returning from the interrupt. NOTE: Data being
#received is known data and an exact copy is loaded into memory prior to any
#transmission.
                                                #
compare:        addr cbuf(sb),r1                #R1- base address of cbuf base
                cmpd $0,r6                      #Check for potential invalid subtraction
                beq zeror6                      #Jump around subtraction
                subd $1,r6                      #
                jump compbyte                   #Jump around subtraction fix
zeror6:         movd $0xff,r6                   #
compbyte:       movd compoff,r5                 #
                cmpb 0(r1)[r5:b],0(r4)[r5:b] #Compare data sent to data received
                bne wrong                       #Branch and set out1 if wrong
                                                #
                cmpb $0x00,0(r4)[r5:b]          #Check for end of buffer
                bne notend                      #Branch and increment pointers
                jump reload1                    #Test for having compared all bytes
                                                #
notend:         addd $1,compoff                 #Increment pointer
notend1:        cmpd r5,r6                      #
                beq bye                         #
                jump compbyte                   #
                                                #
reload1:        addd $1,sbufcnt                 #Increment transmiter cnt
                movd $0,compoff                 #Reload offset of pointer
                jump notend1                    #
                                                #
wrong:          nop                             #
                movb $0x0c,mcr                  #Set out 2, for error strobe
                                                #
#************************ DATA MISMATCH MESSAGE ******************************
                                                #
                save [r0,r1,r2,r3]              #Save register for supervisor call
                movd $4,r0                      #Value required by svc call
                addr message8,r1                #Mover address of message into r1
                movd $17,r2                     #Number of characters into r2
                movd $0,r3                      #Value required by svc call
                svc                             #Actual call
                restore [r0,r1,r2,r3]           #Restore registers
                                                #
stop:           nop                             #
                jump stop                       #Test point
                                                #
```

TL/C/9313–18

```
bye:            ret 0                   #
                                        #
#*************************** RETURN FROM INTERRUPT ***************************
                                        #
popall:         restore [r0,r1,r2,r3,r4,r5,r6,r7]
                reti                    #
                                        #
#********************************** Messages **********************************
                                        #
                message1: .byte 13,10,"Compare Complete",13,10
                message2: .byte 13,10,"Invalid Interrupt",13,10
                message3: .byte 13,10,"Receiver Timeout",13,10
                message4: .byte 13,10,"Receive data available Interrupt",13,10
                message5: .byte 13,10,"THRE Interrupt",13,10
                message6: .byte 13,10,"Line Status Interrupt",13,10
                message7: .byte 13,10,"Modem Status Interrupt",13,10
                message8: .byte 13,10,"Data Mismatch",13,10
                xmitoff: .double 0
                compoff: .double 0
                blk16cnt: .double 0
                sbufcnt: .double 0
                rbufoff: .double 0
isrent: .word   0x9020                  #Mod table
        .word   isr-start               #Offset of service routine for
                                        #Dispatch table.
```

TL/C/9313–19

```
#3/30/87.....D2APPS.ASM.........ADAPTED ORIGINALLY FROM D1RON56K.ASM
#
#THIS PROGRAM RUNS USING 2 DB32000 BOARDS WITH 16550As ENABLED AT ADDRESS
#0d00000 WIRE-WRAPPED ON THE BOARDS. THIS SOFTWARE TRANSMITS THE DATA FF
#THROUGH 00 REPEATEDLY TO THE REMOTE UART AND EXPECTS TO REPEATEDLY RECEIVE
#THE DATA 00 THROUGH FF FROM THE REMOTE UART. IT SHOULD BE RUN IN CONJUNCTION
#WITH THE PROGRAM D1APPS.ASM RUNNING ON THE OTHER DB32000 BOARD. THE TX PIN OF
#THIS 16550A SHOULD CONNECT TO THE RX PIN OF THE 16550A ON THE OTHER BOARD AND
#VICE VERSA. ALSO, THE CTS PIN OF THIS 16550A SHOULD BE CONNECTED TO THE RTS PIN
#OF THE 16550A ON THE OTHER BOARD AND VICE VERSA. THIS WILL ENABLE THE
# APPROPRIATE HANDSHAKES TO OCCUR.
#
#TO RUN THIS PROGRAM YOU MUST:
#
#       1. CONNECT THE RX & TX OF THE 2 16550As ON THE 2 DB32000 BOARDS
#       2. CONNECT THE CTS & RTS OF THE 2 16550As ON THE 2 DB32000 BOARDS
#       3. DOWNLOAD D2APPS.EXE TO THIS BOARD VIA THE GNX DEBUGGER [REV 1.02]
#       4. DOWNLOAD D1APPS.EXE TO OTHER BOARD VIA THE GNX DEBUGGER [REV 1.02]
#       5. START D1APPS.EXE RUNNING ON THE OTHER DB32000 BOARD
#       6. START D2APPS.EXE RUNNING ON THIS DB32000 BOARD
#
#PROGRAM DETAILS:
#
#
# ISR contains the TX SERVICE ROUTINE
#
# TX FIFO is CLEARED before a transmission
#
# DATA SENT  FF ------ 00
#
# DATA RECEIVED  and COMPARED 00 ------ FF
#
# BAUDRATE 128k WITH A 8.0 MHZ XTAL INPUT TO THE 16550A
                                          #
#*********************** ESTABLISH 16550A REGISTER ADDRESSES *******************
                                          #
        .globl          isr               #
           .set rxd,     0x0d00000        #Equate registers to their addresses
           .set txd,     0x0d00000        #
           .set ier,     0x0d00004        #
           .set iir,     0x0d00008        #
           .set fcr,     0x0d00008        #
           .set lcr,     0x0d0000c        #
           .set mcr,     0x0d00010        #
           .set lsr,     0x0d00014        #
           .set msreg,   0x0d00018        #
           .set scr,     0x0d0001c        #
                                          #
#****************** ESTABLISH ADDRESSES FOR THE 32202 (ICU) ******************
                                          #
           .set a0,4                      #Establish address alignment
                                          #between CPU and ICU
           .set icu_hvct,0                #ICU register addresses
           .set icu_svct,1 *a0            #
           .set icu_elgt,2 *a0            #
           .set icu_tpl,4  *a0            #
           .set icu_ipnd,6 *a0            #
           .set icu_isrv,8 *a0            #
           .set icu_imsk,10 *a0           #
           .set icu_csrc,12 *a0           #
```

TL/C/9313-20

```
          .set icu_fprt,14 *a0         #
          .set icu_mctl,16 *a0         #
          .set icu_ciptr,18 *a0        #
          .set icu_pdat,19  *a0        #
          .set icu_ips,20   *a0        #
          .set icu_pdir,21  *a0        #
          .set icu_cctl,22  *a0        #
          .set icu_cictl,23 *a0        #
                                       #
                                       #First ICU register address
                                       #
          .set icu_addr,0xfffe00       #
                                       #
#************************ STATIC BASE STARTING LOCATIONS *********************
                                       #
          .set irl_mod, 17*4           #Dispatch table offset for IRl entry
          .set sbuf, 0x1e              #sbuf = area used to
          .set rbuf, 0x41e             #store data to be transmitted, rbuf =
          .set cbuf, 0x61e             #area used to store received data,
          .set intable, 0x81e          #cbuf = area used to store compare
                                       #buffer, intable = base pointer to the
                                       #interrupt table
                                       #
#********************* SET UP DISPATCH TABLE FOR THE 32032 *******************
                                       #
start::        bicpsrw $(0x100)        #Clear intr's
               movd $0x0c,r0           #Set for monitor svc to move intbase
               movd $0x055555555,rl    #from ROM to ram because you have
               addr intable(sb),r2     #to change the address for the
               movd $0x0c,r3           #interrupt service routine.
               svc                     #Actual svc for move
               sprd intbase,r2         #Put base addr of intbase in r2
               movd isrent,irl_mod(r2) #Put offset of isr into lst location
                                       #of dispatch table
                                       #
#******************** LOAD TRANSMITTER BUFFER (FF to 00) *********************
                                       #
senddat:       addr  sbuf(sb),r0       #R0 contains string buffer ptr.
               movd  $0,rl             #Rl contains offset
               movb  $0x0ff,r2         #Init data reg.
sbufloop:      movb  r2,0(r0)[rl:b]    #Load char. to string buffer
               addqw l,rl              #Increment offset ptr.
               subb  $l,r2             #Increment data
               cmpw  rl,$256           #Check for 256 chars. loaded
               bne   sbufloop          #Jump back if not done
                                       #
#********************** LOAD COMPARISON BUFFER (00 TO FF) ********************
                                       #
compdat:       addr  cbuf(sb),r0       #R0 contains pointer
               movd  $0,rl             #Rl contains offset
               movb  $0,r2             #Init data reg.
cbufloop:      movb  r2,0(r0)[rl:b]    #Load char. to compare buffer
               addqw l,rl              #Increment ptr. offset
               addqw l,r2              #Decrement data
               cmpw  rl,$256           #Check for 256 chars. loaded
               bne   cbufloop          #Jump back if not done
                                       #
#************** SET UP INTERRUPT SERVICE ROUTINE PARAMETERS ******************
                                       #
               movd $16,blkl6cnt       #Initialize 16 byte block counter
```

TL/C/9313–21

```
                                        #
#*************************** 16550A INITIALIZATION *****************************
                                        #
                movb  $0x080,lcr        #Set dlab = 1 for divisor latch access
                movb  $4,txd            #Low divisor latch 56k w/8.0 xtal
                movb  $0,ier            #Upper divisor latch
                movb  $0x003,lcr        #Dlab = 0, 8 bits, no parity, 1 stop
                movb  $0,ier            #Disable UART interrupts
                movb  $0x0c7,fcr        #Fifo=> trigger = 14, reset & enable
                                        #
#**************************** INITIALIZE 32202 (ICU) ***************************
                                        #
                                        #
           movd   $icu_addr,r0          #R0 = icu address
           movb   $0xca,icu_mctl(r0)    #Set mode : 8 bit bus mode,
                                        #           freeze counters,
                                        #           disable interrupts,
                                        #           fixed priority.
           movqb 0,icu_cctl(r0)         #Halt the counters
           movqb -1,icu_ips(r0)         #Set all pins to interrupt source
           movqb 0,icu_csrc(r0)         #No cascaded interrupts (low reg)
           movqb 0,icu_csrc+a0(r0)      # (high reg)
           movb $0x10,icu_svct(r0)      #Set interrupt base vector
           movqb -1,icu_elgt(r0)        #Set level triggering (low reg)
           movqb -1,icu_elgt+a0(r0)     #(high reg)
           movqb $2,icu_tpl(r0)         #Set high polarity mode (low reg)
           movqb 0,icu_tpl+a0(r0)       #(high reg)
           movqb 0,icu_fprt(r0)         #Set highest priority to 0 (low reg)
           movqb 0,icu_fprt+a0          #(high reg)
           movqb 0,icu_isrv(r0)         #Clear intr in-service regs (low reg)
           movqb 0,icu_isrv+a0(r0)      #(high reg)
           movqb -1,icu_imsk(r0)        #Mask all intr (low reg)
           movqb -1,icu_imsk+a0(r0)     #(high reg)H
           setcfg [i]                   #Enable vectored intrp (I=1)
           movd $icu_addr,r0            #
           movb $0x02,icu_mctl(r0)      #Fixed mode, 8 bit bus mode
           movb $0x010,icu_cctl(r0)     #Set to internal sampling
           movb $0xfd,icu_imsk(r0)      #Enable ir1
           movb $0xff,icu_imsk+a0(r0)   #Mask all other interrupts
           bispsrw $(0x800)             #Enable cpu intr's
                                        #
#*************************** ENABLE 16550A INTERRUPTS **************************
                                        #
                movb $2,mcr             #Clear out1, out2 and enable rts
endinit:        movb $0x07,ier          #Enable all but modem status interrupts
                                        #
#********************* ENDLESS LOOP WAITING FOR INTERRUPTS *********************
                                        #
holdloop:       nop                     #
                br holdloop             #
                                        #
#****************************** INTERRUPT HANDLER ******************************
                                        #
isr:            save [r0,r1,r2,r3,r4,r5,r6,r7]
                movb iir,r0             #R0- contains iir
                cmpb r0,$0x0c6          #
                beq lsint               #Line status interrupt
                cmpb r0,$0x0c4          #
                beq rdai                #Receiver interrupt
                cmpb r0,$0x0cc          #
```

TL/C/9313–22

```
                beq rtmout              #Rec timeout interrupt
                cmpb r0,$0x0c2          #
                beq threi               #THRE interrupt
                cmpb r0,$0x0c0          #
                beq msint               #Modem status interrupt
                                        #
                                        #
#************************** INVALID INTERRUPT ROUTINE **************************
                                        #
                save [r0,r1,r2,r3]      #
                movd $4,r0              #
                addr message2,r1        #
                movd $21,r2             #
                movd $0,r3              #
                svc                     #
                restore [r0,r1,r2,r3]   #
                                        #
                                        #
                jump stop               #Restore all registers
                                        #
                                        #
#********************* RECEIVER TIMEOUT INTERRUPT ROUTINE **********************
                                        #
rtmout:         jump rdai
                                        #
#*************************** RECEIVER INTERRUPT ROUTINE ************************
                                        #
#This portion of the program is reached when the received data available
#interrupt is active. Once in this routine each byte removed from the FIFO
#is placed in the designated static base memory location (labelled rbuf).
#The data ready bit (DR) in the LSR is checked before each byte is removed
#from the FIFO. Data sent will be compared to known data in another designated
#static base area (labelled cbuf) by calling the compare subroutine.
                                        #
rdai:           movb $0,mcr             #Disable RTS; stop transmission
                addr rbuf(sb),r4        #r4 contains rbuf base address
                movd rbufoff,r6         #Put rbuf offset runner into r6
rdrbr:          movb rxd,0(r4)[r6:b]    #Store a byte in the receive buffer
                cmpb $0xff,0(r4)[r6:b]  #Is it the last character
                addqw 1,r6              #Increment offset ptr.
                addqw 1,rbufoff         #Track r6
                bne continue            #
                movw $0,r6              #Reset pointer offset
                movw $0,rbufoff         #Reset rbufoff
continue:       movb lsr,r3             #Read lsr
                andb $01,r3             #Mask all but bit 0
                cmpb $01,r3             #
                beq rdrbr               #Read rbr again if set
                movd r6,rbufoff         #Put result of r6 back into rbufoff
                bsr compare             #
                movb $2,mcr             #Enable rts
                jump popall             #
                                        #
#****************************** TRANSMIT ROUTINE *******************************
                                        #
#The transmitter sends data previously loaded into the static base memory area
#labelled sbuf. Thids routine sends data as 16 blocks of 16 bytes and 1 block
#of 15 bytes, continuously. NOTE: Before each block transmission occurs /CTS
#is checked to ensure that the receiver ready.
                                        #
```

TL/C/9313–23

```
threi:          addr sbuf(sb),r0        #R0 contains base pointer
                movw xmitoff,r1         #setup xmit ptr offset
                cmpd $0,blk16cnt        #Check to see if it is the 16th block
                beq send15              #Yes, send only 15 bytes instead of 16
                movd $0x10,r7           #No, send 16 bytes
                jump sendnext           #Jump around 15 byte load
send15:         movd $0x0f,r7           #Load counter for 15 byte load
sendnext:       movb 0(r0)[r1:b],txd    #Load a byte into the transmitter
                addqw 1,r1              #
                cmpw r1,$256            #Are we one address past end of table
                beq reload              #Yes, reload ptr
finish:         save [r7]
                movb msreg,r7           #Read modem status reg
                andb $0x10,r7           #Mask all bits except CTS (MSR4)
                cmpb $0,r7              #Check for disabled CTS
                restore [r7]
                beq abort               #Leave on inactive CTS (MSR4=0)
                subb $1,r7              #No, decrement counter and continue
                cmpb $0,r7              #Is byte counter 0?
                bne sendnext            #No, send next byte
abort:          movw r1,xmitoff         #save xmit ptr offset in ram
                cmpd $0,blk16cnt        #Check to see if it is 16th block
                beq setsnd16            #Yes, reload block counter
                subb $1,blk16cnt        #Decrement block counter
                jump popall             #Finished sending 16 bytes
setsnd16:       movd $16,blk16cnt       #Reload block counter
                jump popall             #Finished sending 15 bytes
reload:         movd $0,r1              #Reset offset
                jump finish             #Go back and finish
                                        #
#*********************** LINE STATUS INTERRUPT ROUTINE ***********************
                                        #
lsint:          save [r0,r1,r2,r3]      #
                movd $4,r0              #
                addr message6,r1        #
                movd $25,r2             #
                movd $0,r3              #
                svc                     #
                restore [r0,r1,r2,r3]   #
                movb lsr,r3             #Read lsr
                jump rdai               #
                                        #
#*********************** MODEM STATUS INTERRUPT ROUTINE **********************
                                        #
msint:          save [r0,r1,r2,r3]      #
                movd $4,r0              #
                addr message7,r1        #
                movd $26,r2             #
                movd $0,r3              #
                svc                     #
                movb 0x0d00018,r0       #
                restore [r0,r1,r2,r3]   #
                jump popall             #
#**************************** COMPARE DATA ROUTINE ***************************
                                        #
#The receiver subroutine branches to this subroutine after it has removed all of
#the data from the Rx FIFO. The receive offset (rbufoff) is changed to point to
#the last byte received in rbuf. The compare offset (compoff) points to each
#byte in the receive buffer and its associated byte in the compare register.
#Compoff is incremented after each successful comparison and the comparisons
```

TL/C/9313–24

```
#end when compoff equals rbufoff. NOTE: Data being received by this test program
#is known data and a copy of it is loaded into cbuf before transmissions begin.
                                            #
compare:        addr cbuf(sb),r1            #R1- base address of cbuf base
                cmpd $0,r6                  #Check for potential invalid subtraction
                beq zeror6                  #Jump around subtraction
                subd $1,r6                  #
                jump compbyte               #Jump around subtraction fix
zeror6:         movd $0xff,r6               #
compbyte:       movd compoff,r5             #
                cmpb 0(r1)[r5:b],0(r4)[r5:b] #Compare data sent to data received
                bne wrong                   #Branch and set out1 if wrong
                                            #
                                            #
                cmpb $0xff,0(r4)[r5:b]      #Check for end of buffer
                bne notend                  #Branch and increment pointers
                jump reload1                #Test for having compared all bytes
                                            #
                                            #
notend:         addd $1,compoff             #Increment pointer
notend1:        cmpd r5,r6                  #
                beq bye                     #
                jump compbyte               #
                                            #
reload1:        addd $1,sbufcnt             #Increment transmiter cnt
                movd $0,compoff             #Reload offset of pointer
                jump notend1                #
                                            #
wrong:          movb $0x0c,mcr              #Set out 2, for error strobe
                                            #
#************************ DATA MISMATCH MESSAGE *******************************
                                            #
                save [r0,r1,r2,r3]          #Save register for supervisor call
                movd $4,r0                  #Value required by svc call
                addr message8,r1            #Mover address of message into r1
                movd $17,r2                 #Number of characters into r2
                movd $0,r3                  #Value required by svc call
                svc                         #Actual call
                restore [r0,r1,r2,r3]       #Restore registers
                                            #
stop:           nop                         #
                jump stop                   #Test point
                                            #
bye:            ret 0                       #
                                            #
#*************************** RETURN FROM INTERRUPT ****************************
                                            #
popall:         restore [r0,r1,r2,r3,r4,r5,r6,r7]
                reti                        #
                                            #
#********************************** Messages *********************************
                                            #
                message1: .byte 13,10,"Compare Complete",13,10
                message2: .byte 13,10,"Invalid Interrupt",13,10
                message3: .byte 13,10,"Receiver Timeout",13,10
                message4: .byte 13,10,"Receive data available Interrupt",13,10
                message5: .byte 13,10,"THRE Interrupt",13,10
                message6: .byte 13,10,"Line Status Interrupt",13,10
                message7: .byte 13,10,"Modem Status Interrupt",13,10
                message8: .byte 13,10,"Data Mismatch",13,10
```

TL/C/9313–25

```
                    xmitoff: .double 0
                    compoff: .double 0
                    blk16cnt: .double 0
                    sbufcnt: .double 0
                    rbufoff: .double 0
isrent: .word       0x9020                        #Mod table
        .word       isr-start                     #Offset of service routine for
                                                  #Dispatch table.
```

TL/C/9313–26

Lit. # 100491

National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: 1(800) 272-9959
Fax: 1(800) 737-7018

National Semiconductor Europe
Fax: (+49) 0-180-530 85 86
Email: cnjwge@tevm2.nsc.com
Deutsch Tel: (+49) 0-180-530 85 85
English Tel: (+49) 0-180-532 78 32
Français Tel: (+49) 0-180-532 93 58
Italiano Tel: (+49) 0-180-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-043-299-2309
Fax: 81-043-299-2408