Implementing a Software UART on the TMS320C54x with the McBSP and DMA

Robert J. DeNardo *DSP Applications – Semiconductor Group*

ABSTRACT

This report discusses the implementation of a universal asynchronous receiver and transmitter (UART) on a TMS320C54x™ DSP using the McBSP and DMA and provides a software UART implementation in C-callable assembly code. In order to implement an asynchronous interface such as a UART using a serial device, software must be written to detect and generate the appropriate framing bits. The initialization of the McBSP and DMA and the timing at which each is enabled is critical to the correct operation of the UART. A thorough examination of these issues is given in this report as well as an explanation of the code.

Contents

TMS320C54x is a trademark of Texas Instruments.

List of Figures

List of Tables

1 Introduction

The TMS320C54x DSP provides a flexible synchronous serial interface through the McBSPs. However, interfacing the DSP to an asynchronous device, such as a UART, requires more than just correct initialization of the McBSP. Synchronous communication relies on three separate signals to transmit and receive data: data, frame sync and clock. Asynchronous communication, however, transmits the data on a single line without any clocking. For the receiver to know when the data begins and ends, start and stop bits must frame the data. The purpose of this report is to explain in detail how to hook up an asynchronous device to the McBSP of the DSP and how to correctly process this data using the DMA and software.

2 UART Functionality

A UART (universal asynchronous receiver and transmitter) is nothing more than a serial asynchronous interface. It is responsible for correctly formatting the data for transmission and decoding it on reception.

The data received or transmitted by the UART requires a start bit (logic low) at the front of the packet and a stop bit (logic high) at the end. The data packet is sent from least significant bit to most significant bit. For error checking purposes, a parity bit may also be added. The signal on the line is always high unless data is present. An example of such a packet is shown in Figure 1.

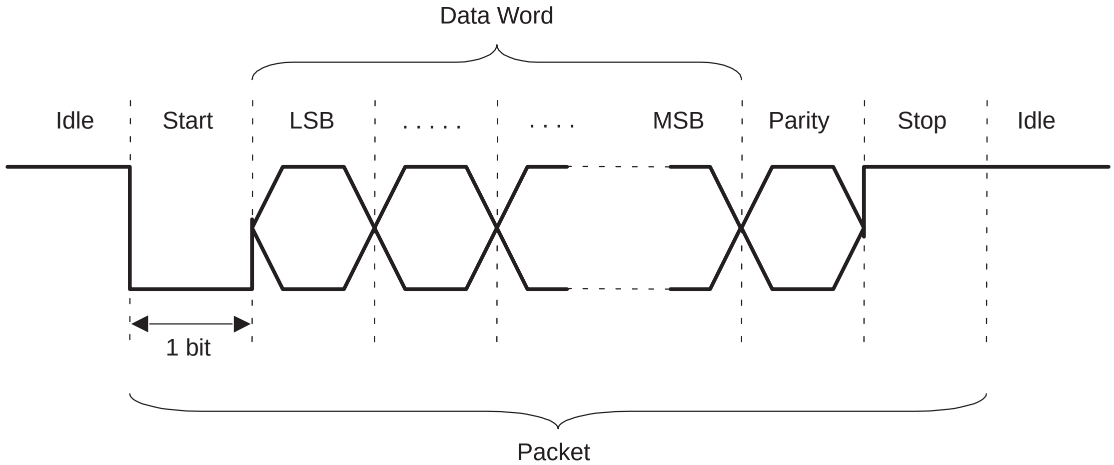

Figure 1. UART Data Packet

Typically, a UART has the following capabilities:

- Variable length data

  The character to be sent can be of length 5, 6, 7, or 8 bits.

- Variable number of stop bits

  There can be 1, 1.5, or 2 stop bits.

- Programmable baud rate

  A register is provided for a divisor that divides down the master clock to generate the intended baud rate.

- Autobaud detect

  This feature allows the UART to automatically detect the baud rate of the transmitter.

- Parity generation

  When sending a character, the UART has the ability to send a parity bit for error checking purposes. The parity settings are none, even, odd, space, or mark. If no parity is generated, the parity bit is omitted from the packet. Even parity ensures that the number of 1’s in the transmitted word is even. Odd parity ensures that the number of 1’s in the transmitted word is odd. Space parity always sets the parity bit to 0. Mark parity always sets the parity bit to 1.

- Parity detection

  When a character is received, the UART will check the parity bit and make sure it matches the parity setting of the connection. If the parity bit does not match, an error is set inside a status register.

- Set Break

  This will send a stream of 0s which is longer than the packet length (start bit + data bits + parity bit + stop bits). It provides a means of indicating a special event to the receiver (such as change in baud rate).

- Break indicator

  The UART has the ability to detect a break condition (when a stream of 0s longer than the packet length is received). This indicates a special condition to the UART and is reported in a status register.

- Framing Error detection

  Signals if an invalid stop bit was detected and reports it in a status register.

- Overrun detection

  Signals that another word was received before the prior word was read out. Reports the error in a status register.

- Interrupt-based or polling-based operation

  The UART can be serviced either when it generates an interrupt to the DSP for a receive, transmit, or error event, or it can be serviced by polling the status bits to determine when an event occurs.

- Character FIFO

  A FIFO is used to buffer the receive and transmit characters, relieving the host from servicing the UART for each new character.

- Modem Functionality

  This is provided through four inputs (CTS_, DSR_, DCD_, and RI_) and four outputs (DTR_, RTS_, OUT1_, and OUT2_). These signals allow the UART to setup hardware flow control with a device emulating a modem.

This implementation of a software UART provides all of these features except for Autobaud detect, FIFO mode, and Modem Functionality. It is modeled after the TL16C450 ACE[1]. The number of data and stop bits and the parity is selectable at compile time. If desired, INTERRUPT_BASED mode allows the user to handle receive, transmit, and error events within an ISR.

3 Implementation

In order to emulate this asynchronous interface, a way to generate and detect the framing bits must be devised. Because the serial port is not synchronized to the UART signals, we cannot guarantee the serial port clock will align perfectly with the edge of the start bit. This creates an offset between the asynchronous signal and the synchronous serial port. Also, the DSP serial port clock frequency will in almost all cases not be exactly matched to the baud rate of the asynchronous signal, causing rate skew in the signal. The best way to reduce the offset and rate skew is to oversample the bit stream. In this implementation an oversampling of 16 will be used, as this is optimum for a 16 bit DSP in terms of data storage and manipulation as well as for providing robustness to the process. The oversampling also gives the UART the ability to run at slower speeds.

For the receive process, the McBSP will oversample the data bits and the DMA will store them in a memory buffer for later handling. When a complete packet is read in, the DMA will interrupt the DSP so it can interpret the packet. The procedure is opposite for the transmit process; the DSP fills a buffer with the oversampled bit stream and then enables the DMA to begin transferring this data out the serial port.

A number of issues arise with this implementation which must be dealt with:

- How to interface the McBSP to the asynchronous data line
- How to initialize the DMA for receiving and transmitting the packets
- How to create and transmit a packet
- How to receive and decode a packet

4 McBSP

A serial port typically has three signals it either creates or receives for each direction of data: data, frame sync, clock. The asynchronous signal, however, is present on only one line, on which it has its own framing signals. To properly communicate between these interfaces, these signals must be properly mapped and interpreted.

The biggest challenge in interfacing a synchronous device to an asynchronous signal is not in the transmission but rather in the reception. Transmission is a simple process in terms of the timings of the signals; the serial port can transmit according to its clock and the receiver will correctly decode the signal, as long as the start and stop bits are appropriately placed and the sampling rate is appropriate. The receiver timings are more complicated. The asynchronous signal, by nature, can be received at any time, and most likely will not be aligned with the serial port clock. Also, there can be slight differences in the baud rate compared to the sample rate of the serial port, causing the received data to “slide”. Because of these issues, the serial port receive channel and software must be setup appropriately to recognize these constraints and to work around them.

The length of the packets (PKTBITS) are #start bits + #data bits + #parity bits + #stop bits. There is 1 start bit and 1, 1.5, or 2 stop bits. If parity generation and detection are enabled, an extra parity bit is added. The number of data bits can be 1-15 without parity, or 1-14 with parity. Each of these bits will be oversampled and represented by 16 bits on the DSP.

For transmission, the UART must be able to send half stop bits. Therefore, the McBSP transmit port is set for dual phase frames, with the first phase having 16 bit words and the second phase having 8 bit words. The length of the first phase is (#start bits + #data bits + #parity bits) words and the length of the second phase is (2*#stop bits) words. The total length of the transmit frame (TxPKTBITS) in words is the sum of these phases. The data transmit (DX) pin of the DSP is tied to the transmit data line of the interface. The transmit frame sync (FSX) and clock (CLKX) pins are not used.

From Figure 1 we can see that the asynchronous signal line is always high unless a data packet has been sent across. When a packet is sent, the start bit is sent first, so the signal will go low. This is similar to an active-low frame sync. The McBSP gives us the flexibility to choose the polarity of the frame sync signal as active-low. By tying the receive data line to the data receive (DR) and frame sync (FSR) pins of the McBSP receive channel, we can trigger the McBSP to start receiving the packet whenever the line goes low. To prevent the McBSP from re-triggering, it is set to ignore all frame syncs during the receive packet.

During decoding, the center of each oversampled bit is checked. Only the first half of the stop bit is received and checked, which gives more flexibility in the sampling rate, as will be seen below. The total number of bits the McBSP receives in a frame will be RxPKTBITS=(#start bits + #data bits + #parity bits + 0.5). Therefore, the McBSP receive port is set to have dual phase frames with the first phase of length (#start bits + #data bits + #parity bits) words and the second phase of length 1 word. The word size of the first phase is 16 and the second phase is 8. Since the start bit is part of the data packet, ideally there will be a 0-bit delay between the received frame sync and the data. However, on the 5410 errors are generated on receive (missed frame syncs) unless the delay is set to 1 bit. See Figure 2 for an example of how the McBSP receive frame aligns with the data packet.

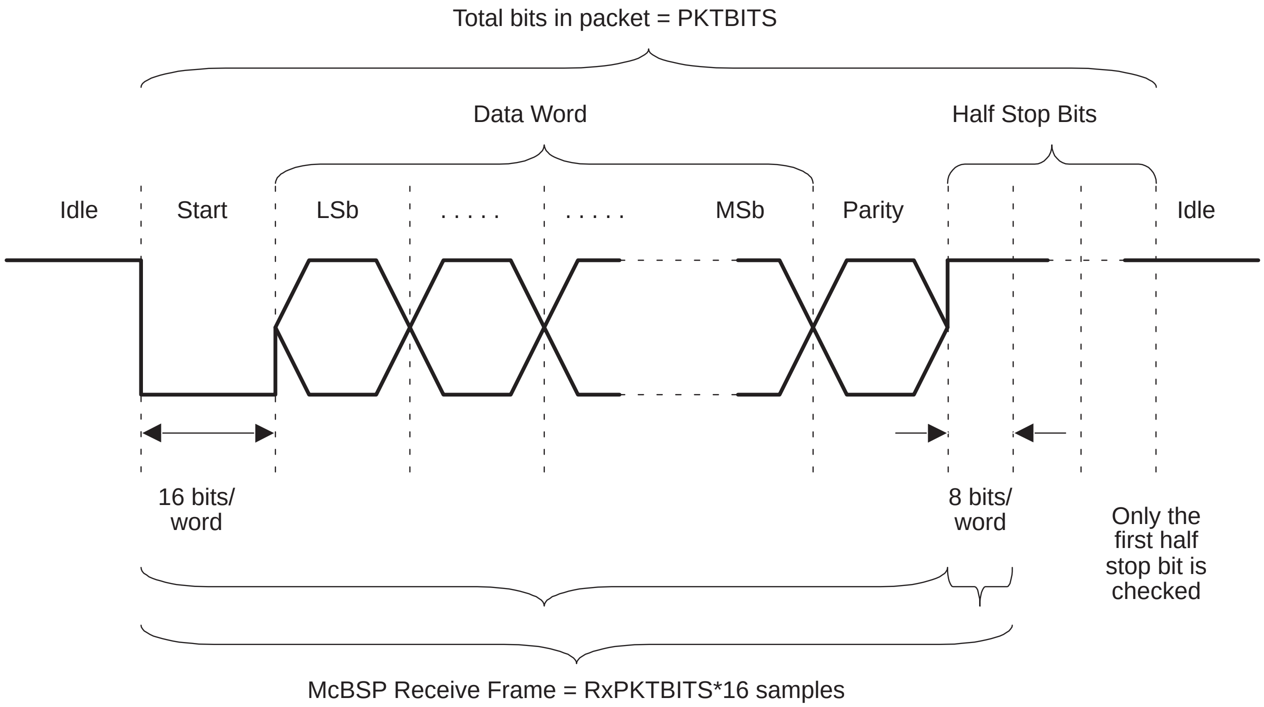

Figure 2. McBSP Receive Frame Structure

The sampling rate of the McBSP is critical to the correct operation of the software UART. The McBSP will ignore all subsequent frame syncs during the reception of the frame we have defined above. To get the maximum data rate, it must be able to detect the next start bit, which could immediately follow the stop bit. The frames syncs and receive data are latched on the falling edges of the serial port clock. For a frame sync to be detected, the signal must be high for at least one clock cycle before it goes low again. This resets the frame sync logic. Therefore, the McBSP must be finished reading in the first data packet before the transition from the stop bit to the next start bit occurs.

In an ideal case, the clock edges of the serial port line up with the bit edges of the data packet, there are exactly 16 clock periods for each bit in the packet, and the offset between the beginning of the start bit and the falling edge of the serial port clock is minimal. See Figure 3 for an example of this timing.

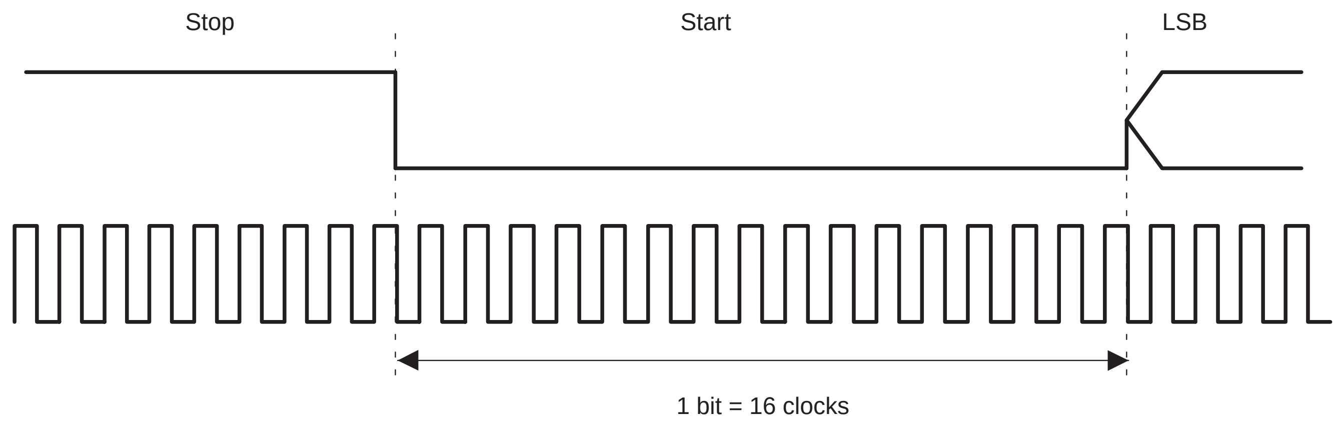

Figure 3. Timing of Signal Perfectly Synchronized to Serial Port Clock

In the practical case, an offset between the beginning of the start bit and the clock's falling edge exists. The serial port must also be set for a data delay of 1, causing another clock period of offset. This offset is the same no matter how many bits are in the packet. The serial port clock will not generate exactly 16 periods per data bit because the divisor used to create the clock rate has limited resolution (integer) and cannot produce a clock of exactly the baud rate times 16. Therefore, the serial port clock may be slower or faster than 16 times the baud rate. This rate skew causes a timing error that is added for each bit in the packet. If the clock is too slow, there are less than 16 samples per data bit, causing the McBSP to possibly sample past the end of the stop bit and into the next start bit. Because the McBSP was ignoring frame syncs during this time, it misses the transition to the next start bit. The next frame sync is generated whenever another high to low transition is encountered, most likely in the middle of the next data word. See Figure 4 for an example of this more practical signal timing.

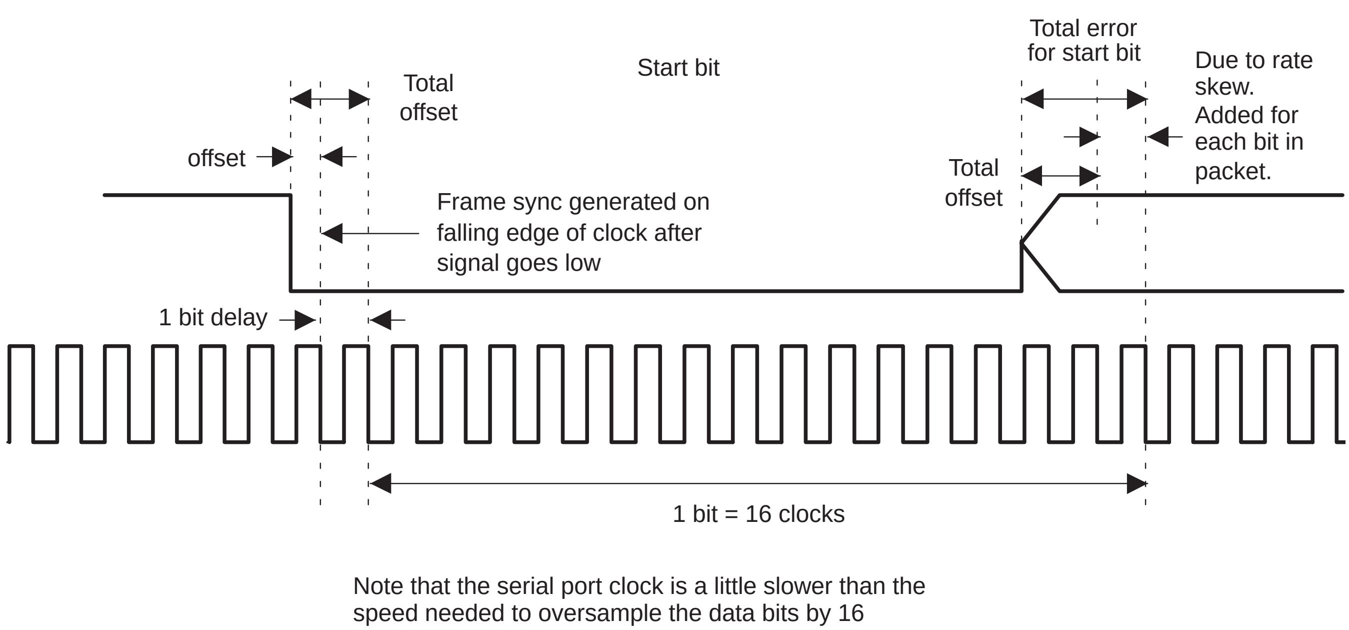

Figure 4. Timing of Signal with Offset and Rate Skew

To prevent start bits from being missed, the McBSP must run fast enough such that the last sample in the frame is confined to the stop bit. See Figure 5 for an example of the minimum speed of the McBSP.

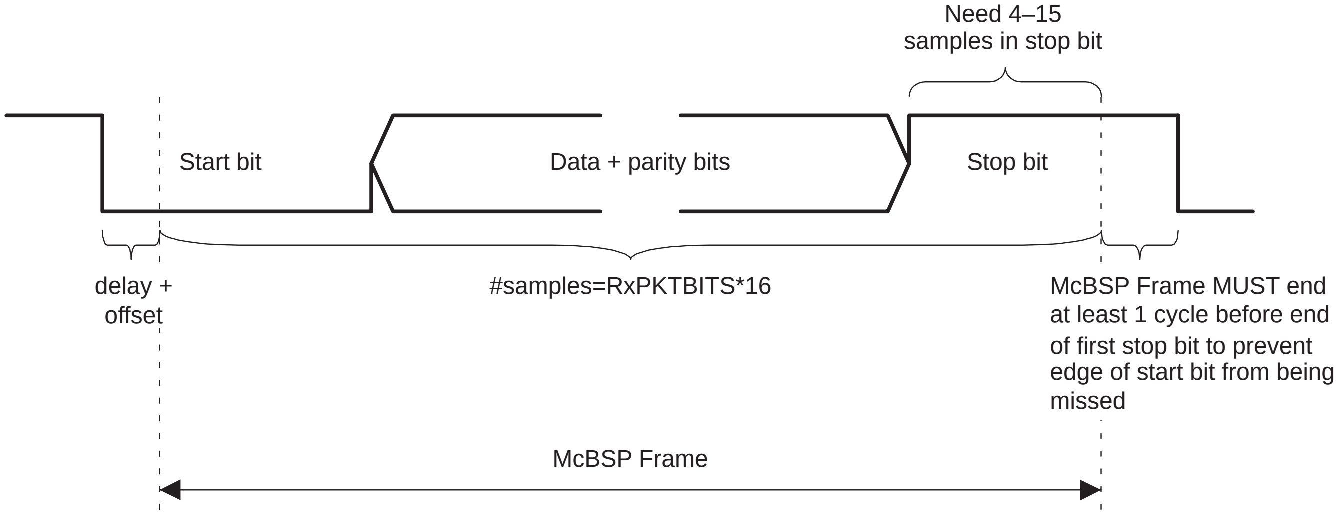

Figure 5. McBSP Frame Restrictions

The minimum speed of the serial port should account for the maximum offset between the beginning of the start bit and the frame sync, the delay between the frame sync and first data sample, and the minimum time between the end of the frame and the next start bit. Given 16 samples per data bit, the equation is:

$$FrameLength < PacketLength - offset - delay - extraFrameSyncSample \tag{1}$$

The units of measure are in seconds. This can be rearranged, as follows, with the delays and offsets now measured in number of serial port samples.

$$\left(16\,\frac{samples}{bit} * RxPKTBITS + offset + delay + extraFrameSyncSample\right) * samplelength < \frac{PKBITS}{baudrate} \tag{2}$$

The maximum offset is 1 sample, as is the delay. The extra frame sync sample in the left part of the equation is for the frame sync to reset before the next start bit edge, and is one sample in length. The sample length is the number of seconds per serial port sample, which is DIV/DSPCLK. We need to solve for the divisor (DIV).

$$DIV \le \frac{PKTBITS * DSPCLK}{baudrate\left(16\,\frac{samples}{bit} * RxPKTBITS + 3\right)} \tag{3}$$

To get the most stringent limit, the most number of bits should be assumed to be in a packet with 1 stop bit. For example, given a baud rate of 19200, a DSPCLK of 75MHz and 17 packet bits (1 start, 14 data, 1 parity, 1 stop: PKTBITS=17, RxPKTBITS=16.5), we get:

$$DIV \le 248.71 \tag{4}$$

Which can only be encoded as an integer, so

$$DIV \le 248 \tag{5}$$

The decoding scheme will check the middle 4 bits of the received words, except for the ½ stop bit, which has its last 4 samples, tested. The maximum speed must ensure that the last 4 samples of the frame lie within the stop bit. This is the same as saying that the McBSP frame minus those 4 samples must fit within the packet bits prior to the stop bit. The limiting case would have an offset of 0 (still a delay of 1, though), so the equation is:

$$\left(16 \frac{samples}{bit} * RxPKTBITS - 4 + delay\right) * \frac{DIV}{DSPCLK} \geq \frac{(PKTBITS - STOPBITS)}{baudrate} \tag{6}$$

or

$$DIV \geq \frac{(PKTBITS - STOPBITS) * DSPCLK}{baudrate * \left(16 \frac{samples}{bit} * RXPKTBITS - 3\right)} \tag{7}$$

For the most stringent limit, assume the maximum number of bits in a packet (1 start, 14 data, 1 parity, 2 stop bits: PKTBITS=18, STOPBITS=2, RxPKTBITS=16.5). Or for the example above,

$$DIV \geq 239.46 \tag{8}$$

Which must be an integer

$$DIV \geq 240 \tag{9}$$

Note that increasing the number of stop bits will increase the length of time between the end of the McBSP frame and the beginning of another start bit. This will raise the maximum DIV value because RxPKTBITS (only checks first half stop bit) remains the same, but PKTBITS increases. The maximum and minimum DIV values for common baud rates are listed in Table 1, as well as what the divisor should be to get the exact baud rate with 16 samples per bit. It is important to use a divisor as close to the exact baud rate as possible so that the UART transmits properly.

Table 1. Divisor Limits for Given Baud Rate and Clock Rate

	75-MHz DSP Clock			100-MHz DSP Clock		
Baud Rate	Divisor Minimum	Exact Divisor	Divisor Maximum	Divisor Minimum	Exact Divisor	Divisor Maximum
19200	240	244.14	248	320†	325.52†	331†
38400	120	122.07	124	160	162.76	165
57600	80	81.38	82	107	108.51	110
115200	40	40.69	41	54	54.25	55

† This divisor is too large to be used on the 54xx McBSP (max is 256)

In order to setup the McBSP, the transmit and receive channels, as well as the frame sync generator and clock generator, must be in reset when registers associated with that portion are written to. It is important to wait for at least 2 CLKG periods after putting that portion in reset or taking it out of reset. This allows time for internal synchronization of the McBSP. The McBSP registers are initialized to the settings in Table 2. See the *TMS320C54x DSP Enhanced Peripherals* guide for detailed information on the McBSP.

Table 2. McBSP Initialization

Register	Bit Field	Value	Comment
SRGR1	FWID	0	unused
	CLKGDV	DIV-1	Generate clock period according to above Equations
SRGR2	GSYNC	0	Sample rate clock free running
	CLKSP	0	Unused
	CLKSM	1	Sample rate clock derived from CPU clock
	FSGM	0	Transmit frame sync due to DXR-to-XSR copy
	FPER	0	unused
SPCR1	DLB	0	No loopback
	RJUST	00	Right justify data
	CLKSTP	00	Clock stop mode disabled
	DXENA	0	DX enabler off
	ABIS	0	A-bis mode disabled
	RINTM	00	RINT driven by RRDY
	RRST_	0	Receiver disabled
SPCR2	FREE	0	FREE mode disabled
	SOFT	1	SOFT mode enabled
	FRST_	0	Frame sync generator in reset
	GRST_	0	Clock generator in reset
	XINTM	00	XINT driven by XRDY
	XRST_	0	Transmitter disabled
PCR	XIOEN	0	No Tx pins used for GPIO
	RIOEN	0	No Rx pins used for GPIO
	FSXM	1	Transmit frame sync determined by FSGM
	FSRM	0	Receive frame sync generated by external device
	CLKXM	1	CLKX is output driven by sample rate generator
	CLKRM	1	CLKR is output driven by sample rate generator
	FSXP	1	FSX is active low
	FSRP	1	FSR is active low
	CLKXP	0	Transmit data sampled on rising edge of CLKX
	CLKRP	0	Receive data sampled on falling edge of CLKR
RCR1	RFRLEN1	RxPKTBITS-RxHSTOPBITS-1	Phase 1 of receive frame includes start bit, data bits, and parity bit (not half stop bit) (RxHSTOPBITS=1)
	RWDLEN1	010	Words in phase 1 are 16 bits

Table 2. McBSP Initialization (Continued)

Register	Bit Field	Value	Comment
RCR2	RPHASE	1	Dual phase frames
	RFRLEN2	RxHSTOPBITS-1	Phase 2 of receive frame includes 1 half stop bit
	RWDLEN2	000	Words in phase 2 are 8 bits
	RCOMPAND	00	No companding
	RFIG	1	Ignore receive frame syncs during receive frame
	RDATDLY	01	1-bit delay between FSR and data
XCR1	XFRLEN1	TxPKTBITS-TxHSTOPBITS-1	Phase 1 of transmit frame includes start bit, data bits, and parity bit (not half stop bits)
	XWDLEN1	010	Words in phase 1 are 16 bits
XCR2	XPHASE	1	Dual phase frames
	XFRLEN2	TxHSTOPBITS-1	Phase 2 of receive frame includes half stop bits
	XWDLEN2	000	Words in phase 2 are 8 bits
	XCOMPAND	00	No companding
	XFIG	1	Ignore transmit frame syncs during transmit frame
	XDATDLY	00	0-bit delay between FSX and data

5 DMA

The DMA is needed to shuttle data between memory and the McBSP without CPU intervention. Each time a complete packet is read in, the DMA will interrupt the DSP so it can process the received packet. Each time a packet is transmitted, the DMA will interrupt the DSP so it knows another packet may be sent out.

A new packet is received when the DMA has transferred RxPKTBITS words to a buffer from the McBSP DRR register, and a packet has been transmitted when the DMA moves TxPKTBITS words to the McBSP DXR register. The DMA can be set to transfer data from/to the McBSP when the RRDY/XRDY signal becomes active. Interrupts can be generated by the DMA either by setting the DMA to interrupt at the end of a block transfer (use autoinit or manually re-init the DMA) or by using ABU mode. We use ABU mode to preserve the autoinit registers for other uses. Each half of the receive and transmit buffers will hold one packet, and the DMA will interrupt when its pointer passes into each half.

There is one caveat to using ABU mode. On the 5410, if the pointer into the buffer is post-incremented, the interrupts will NOT occur on the half buffer points. Instead, when the pointer moves into the second half of the buffer the interrupt is not generated until it reaches the second word of that half. This is unlike the other TMS320C54xx processors, where the interrupt occurs when the pointer gets to the first word in each half. However, if the pointer is post-decremented, the interrupts do occur in the correct location. So, our buffers will be filled and emptied from back to front. See Figure 6 for a depiction of the buffer orientation. For the transmitter, HSTOPBITS (number of half-stop bits) is set by the user. For the receiver HSTOPBITS is always 1. Note that in autobuffering mode, the DMA buffers must be aligned on power-of-2 boundaries greater than the buffer size (i.e. buffer size of 16-31 must be on a 32 word boundary).

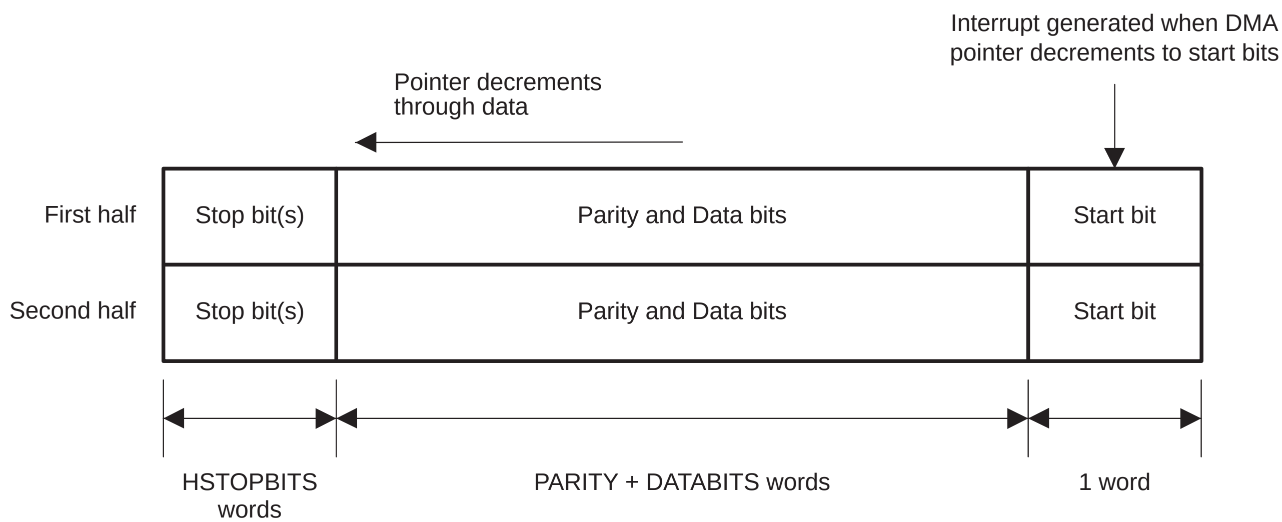

Figure 6. DMA Circular Buffers

Another issue arises due to the DMA triggering on XRDY and RRDY events. The DMA only recognizes the event if the DMA channel is enabled when that event occurs. If a DMA channel is disabled when the event occurs, and the DMA channel is subsequently enabled, it will not recognize that the event has occurred and it will need to be “kickstarted”. The DMA receive channel is never disabled, so it will never need to be kickstarted. Kickstarting the DMA for transmit consists of manually writing a word to the McBSP and then enabling the DMA channel. When the McBSP writes out that word, it will generate the XRDY event again, this time with the DMA enabled. Note that the McBSP generates the XRDY and RRDY events when it is brought out of reset. Therefore, when the UART is started, the DMA should be enabled before the McBSP is brought out of reset.

On the receive side, the McBSP is constantly running and always has the DMA enabled. The DMA interrupts the DSP when a packet is read in and the DSP then decodes the oversampled packet and performs error checking. Note that the receive DMA channel only moves data into the buffer when the McBSP gets new frame syncs, which only occurs when a new packet is received. Because the DMA buffer has 2 halves, the receive data is double buffered.

The transmit side is a little more complicated. When the McBSP is out of reset, it is constantly clocking out data sent to it. If the DMA is enabled, it will continuously move data to the McBSP when XEVT (XRDY) occurs. Because a packet should only be transmitted when it is valid, either the DMA or the McBSP must be stopped until valid data is in the buffer.

Though stopping the McBSP seems like the best choice (i.e. no point in running the McBSP when not using it), latency issues in starting and stopping as well as issues with ensuring the entire data packet has been transmitted make it less desirable. For instance, when taking the McBSP transmitter out of reset, 2 serial port bit clocks must go by before it is running properly. The same is true when it is halted. If these times are not adhered to, the XRDY event is not properly generated, which causes the DMA to fail. Depending on the baud rate, 4 bit clocks between reset and running can be a very long time (i.e. almost 1000 DSP cycles for 19200 baud).

Also, a DMA transmit channel interrupt means the DMA has moved TxPKTBITS words to the McBSP. It does not mean the McBSP is finished transmitting all of the bits. When the DMA pointer moves into the next half of the buffer (generating the interrupt), it has just moved the last bit of the packet into the McBSP DXR (note the last 2 bits in the packet are always two 8 sample long stop bits). That means the second to last word in the packet has just moved into the XSR register. Therefore, there are 16 samples to transmit before the McBSP is done. We must wait until the McBSP DXR and XSR registers are empty before halting the McBSP, or these bits will be corrupted. That is 4000 cycles at a 19200 baud rate. These delays make this approach less desirable.

The approach we use is to halt the DMA transmit channel when there is no valid data. When the DMA transmit channel interrupts the DSP, it has just moved the last bit in the packet to the McBSP, and is therefore done with the packet. By halting the DMA now, the McBSP can still clock out the remaining bits without making the DSP wait. The only issue with this method is when to restart the DMA.

The DMA transmit buffer has 2 halves, allowing double buffering of the transmit data. As long as there is an empty half, more transmit data can be written into the buffer. If the DMA transmit channel generates an interrupt but there is still another word in the buffer to transmit, the DMA need not be disabled. However, if the DMA has just transmitted the last valid word in the buffer, the DMA must be halted and then restarted when a new transmission is desired.

If the DMA has been halted and a new packet is to be transmitted, we must restart the DMA, making sure it correctly catches the XRDY events. To sync the DMA to the XRDY event, the DMA must be kickstarted by manually writing the first data bit to the McBSP and then enabling the DMA. Because it is possible that the last packet still has a bit in the DXR register, we cannot write to DXR until we are sure it is empty. This is done by waiting until XRDY=1.

It may be possible to speed this process up by only kickstarting the DMA if XRDY=1 when we want to transmit (i.e. the DMA has missed the XRDY event). But, there is always the possibility that XRDY may toggle between our read of it and the enabling of the DMA, so it is safer to always kickstart it. For example, if XRDY=0, we may figure enabling the DMA now will ensure that it catches the XRDY event. However, if XRDY goes from 0 to 1 right after we check it and before we enable the DMA, we can miss the event anyway.

The DMA is initialized as in Table 3. See the *TMS320C54x DSP Enhanced Peripherals* guide for detailed information on the DMA.

Table 3. DMA Initialization

Register	Bit Field	Value	Comment
DMSRC (Rx)	DMSRC	McBSPDRR1	Get data from McBSP DRR register
DMDST (Rx)	DMDST	RxBuffer+RxPKTBITS-1	Start pointer at end of first half of buffer
DMCTR (Rx)	DMCTR	2*RxPKTBITS	Buffer length is twice the packet length
DMSFC (Rx)	DSYN	REVT	Sync on RRDY for the McBSP (depends on which McBSP using)
	DBLW	0	16 bit words
	Frame Count	00000000	Unused
DMMCR (Rx)	AUTOINIT	0	Autoinit disabled
	DINM	1	Interrupt generated based on IMOD bit
	IMOD	1	Interrupt at buffer full and half-full
	CTMOD	1	ABU mode
	SIND	000	Source address not modified
	DMS	01	Source address in data space
	DIND	010	Destination address post-decremented
	DMD	01	Destination address in data space
DMSRC (Tx)	DMSRC	TxBuffer+TxPKTBITS-1	Start pointer at end of first half of buffer
DMDST (Tx)	DMDST	McBSPDXR1	Put data into McBSP DXR register
DMCTR (Tx)	DMCTR	2*TxPKTBITS	Buffer length is twice the packet length
DMSFC (Tx)	DSYN	XEVT	Sync on XRDY for the McBSP (depends on which McBSP using)
	DBLW	0	16 bit words
	Frame Count	00000000	unused
DMMCR (Tx)	AUTOINIT	0	Autoinit disabled
	DINM	1	Interrupt generated based on IMOD bit
	IMOD	1	Interrupt at buffer full and half-full
	CTMOD	1	ABU mode
	SIND	010	Source address post-decremented
	DMS	01	Source address in data space
	DIND	000	Destination address not modified
	DMD	01	Destination address in data space

When the UART is started, the McBSP and DMA are initialized as in Table 2 and Table 3. To start the UART, the DMA must have the receive channel enabled and make sure the correct interrupts are selected from the multiplexed interrupts. This is done in the DMPREC register. Depending upon which DMA channel is used for reception, its enable bit must be set and the proper interrupt selection must be made so that the DMA receive and transmit channels have their interrupts generated. For example, DMA channel 4 can be used for receive and DMA channel 5 for transmit, which means DPREC must be set to 0x0010 to enable the receive channel and INTOSEL set to 00 (selects DMA channels 4 and 5 to have interrupts). The interrupt vector table must of course be appropriately setup to handle these interrupts.

The DMA channels should be disabled before they are setup. See Figure 7 for a depiction of the overall initialization process.

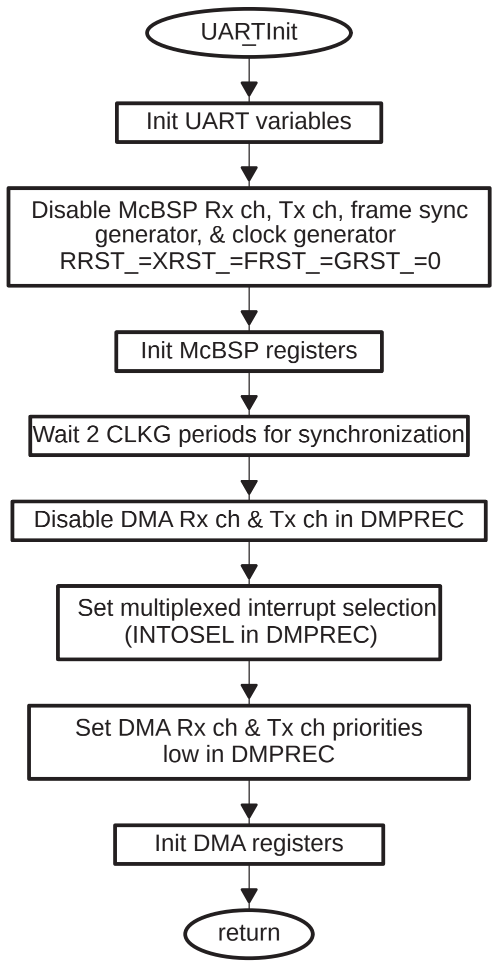

Figure 7. UART Initialization

6 Transmit Process

There are two portions to the transmit process: how to place a new packet in the transmit buffer, and what to do when a packet has just been transmitted. The first case is taken care of whenever the user calls the routine to perform a transmit (_UARTTxChar). The second occurs in the DMA transmit channel ISR (_UARTDMATxISR).

6.1 Procedure at Start of Transmission

The flow to create a new packet for transmission is depicted in Figure 8.

The transmission routine should not be entered into unless there is space in the transmit buffer, verified by polling THRE (Transmit Holding Register Empty) for a 1. Entering the routine with THRE equal to 0 will disrupt the transmission process, causing the DMA pointer to be aligned improperly.

To put a new packet in the buffer, the next available half of the buffer is first checked from a flag (*txbufhalf*).

If parity is to be generated, a routine that calculates the parity bit is called. The parity bit is generated based on a successive approximation scheme. This method is detailed in the Texas Instruments Application Brief, *Parity Generation on the TMS320C54x* by David Nerge. The parity bit is added above the data bits.

Next, each bit in the packet is encoded and placed in the transmit buffer. The encoding of the bits is a simple process; 0xFFFF replaces a 1, and 0x0000 replaces a 0. The start bit is written to the end of the DMA buffer half, followed by the data bits and the parity bit (if used). Finally, the stop bits (in 8 bit words) are added. So a packet of 1 start bit, 8 data bits, and 1 stop bit will require 11 words in the buffer half (stop bit has two 8 bit halves).

If there is another packet in the buffer which has not yet been fully transmitted (i.e. with the addition of the new packet, there are now 2 packets), then the DMA transmit ISR has not disabled the DMA. In this case, nothing else needs to be done. The DMA will continue to output the new packet when it has finished outputting the old packet. Because there are 2 packets in the buffer, the THRE flag is cleared to indicate that it is full. This flag should be consulted before every call to the transmit routine to ensure space exists in the buffer.

If there is not another older packet to transmit in the buffer, then the DMA transmit ISR has shut down the DMA, and it will need to be kickstarted by writing to DXR. The DXR register cannot be written to until all bits of the previous packet still in the DXR register are shifted out. This can be checked by waiting until XRDY=1. When that occurs, the start bit is written to DXR, the DMA pointer is decremented to the first data bit, and the transmit DMA channel is enabled.

Note that the DMA transmit interrupt was disabled during this routine to prevent it from occurring between the increment of the number of packets in the transmit buffer and the check of the number of packets. If the number of packets was incremented to 2 (full buffer) and then a DMA transmit interrupt occurred, the ISR would decrement the number of packets but would not disable the DMA because it still sees another packet in the buffer. When the ISR returns, the transmit routine sees only 1 packet in the buffer, so it thinks it needs to restart the DMA, even though it was never disabled. This will cause a problem because the DMA will start moving the new packet to the McBSP, but the kickstart of the DMA would overwrite the data. Disabling the DMA transmit interrupt prevents this from happening.

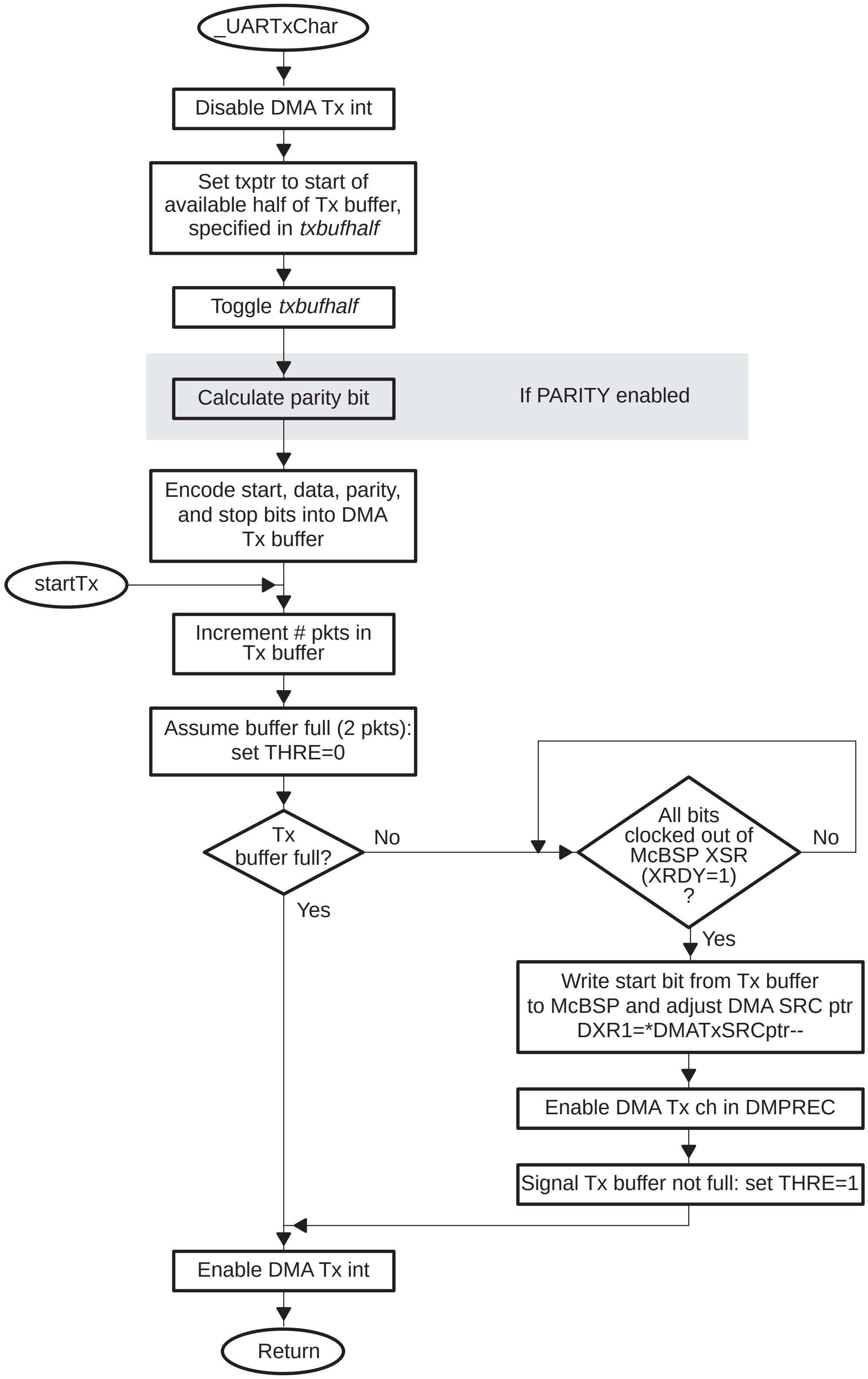

Figure 8. Procedure at Start of Transmission

6.2 Procedure at End of Transmission

The second transmission case occurs when a packet has been completely read out of the DMA transmit buffer, freeing up space for another transmission. When this occurs, the DMA transmit channel ISR is entered. See Figure 9 for a depiction of the flow of this routine.

When the DMA transmit channel interrupts the DSP, a packet has been completely read out by the DMA. The interrupt service routine that is subsequently entered must check if any more packets are ready for transmission, and if not, disable the DMA.

First, the number of transmit packets in the DMA buffer is decremented.

Then, the status of the transmitter is always set to THRE equals 1, as there must be an available space in the transmit buffer now that a packet has been removed.

If the number of transmit packets is now 0, there is no more data to transmit, so the DMA is disabled in DMPREC.

Finally, if the INTERRUPT_BASED mode is enabled, the _UARTTBEint routine is called, allowing the user to process a transmit event during the ISR.

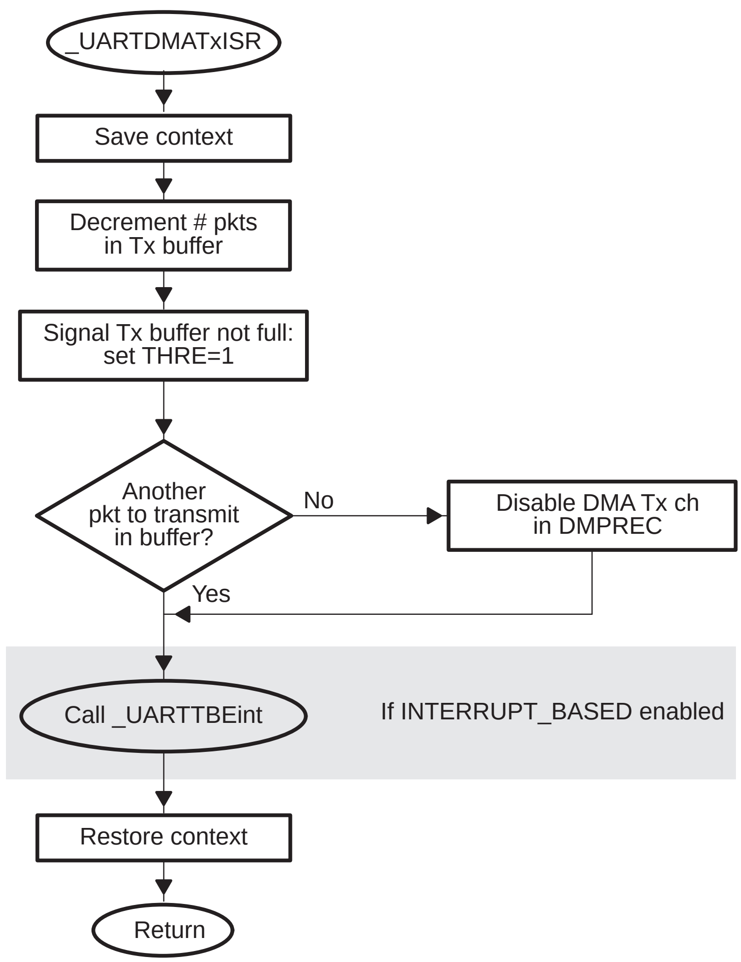

Figure 9. Procedure at End of Transmission

7 Receive Process

As with the transmit process, there are two cases for reception: what to do when a new packet has just been received, and how to read out a packet from the buffer. The first case is taken care of by the DMA receive channel interrupt service routine whenever a new packet is received (_UARTDMARxISR). The second case takes place whenever the user calls a routine to read a packet from the buffer (_UARTRxChar).

7.1 Procedure When Packet Received

When the DMA has moved an entire packet of data into its receive buffer, it interrupts the DSP. The ISR associated with this must decode the received packet from the oversampled version into one word of data and perform any necessary error checking. This is depicted in Figure 10.

First, the buffer half with the newly received packet is determined by checking *rxbufhalf*.

Next, the bits are decoded. Only the lower (last) 4 samples of the half stop bit are checked to decode it. The other bits in the packet have their middle 4 samples checked.

The decoding routine must account for the rate skew in the signal. A 16-bit word may contain only a portion of a data bit, so the decoder needs to manage this shifting of the samples. The word is decoded to a 1 if the middle four samples are 1100b, 1110b, 1111b, or 0111b. This accounts for the shifting of the data. Also, to simplify the decoder 1101b and 1011b are decoded to a 1. These patterns should not occur, as there should be approximately 16 samples in a row with the same value. Unless noise on the line corrupts a sample, 1101b and 1011b will never be received. Any other pattern is decoded as a 0.

The decoded bits are compiled into a decoded word and are stored in the *rxchar* variable. The data bits are in the lowest part of the word, followed by the parity bit and stop bit.

Next, error checking is performed. The different conditions checked are framing errors, break indications, parity errors and overrun.

A framing error is detected if the stop bit was not present (i.e. not decoded as a 1). The framing error (FE) flag is set if this is the case.

A break is detected if all of the bits in the packet are zero. This includes the data, parity and stop bits. The start bit is not checked, as it must be zero for the word to be received. The break indicator (BI) flag is set when a break is detected.

A parity error is detected if the parity of the received word does not match the parity setting of the UART. If there is a parity error (PE), a flag is set in the status register.

Overrun occurs if the last packet received has not been read out and another packet is received and decoded. If an overrun error occurs (OE), a flag is set in the status register.

The data ready flag (DR) is set to note that there is a new packet ready that has not been read by the user yet.

Finally, If the INTERRUPT_BASED mode is enabled, the _UARTRBFint routine is called, allowing the user to process a receive event during an ISR. If any of the status bits were set during reception (OE|FE|PE|BI), the _UARTLSIint routine is called so that the user can perform any error handling routines in the receive ISR. None of the status bits are cleared (except DR and THRE) by the code. It is the user’s responsibility to handle these error conditions.

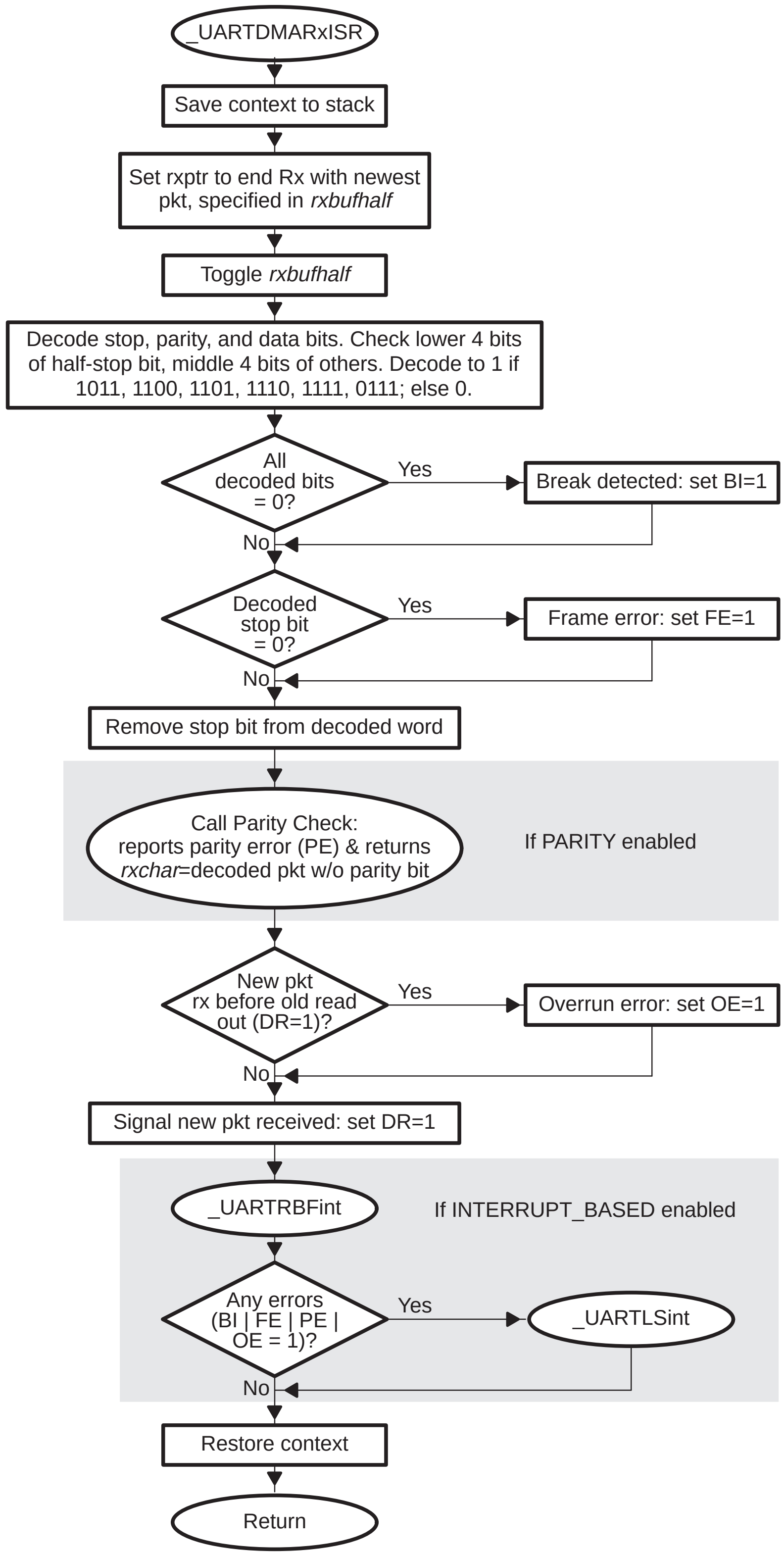

Figure 10. Procedure When Packet Received

7.2 Procedure To Read Received Packet

The receive routine simply reads the character from *rxchar* and resets the DR flag. This routine should only be entered if DR is a 1. See Figure 11 for a graphical depiction.

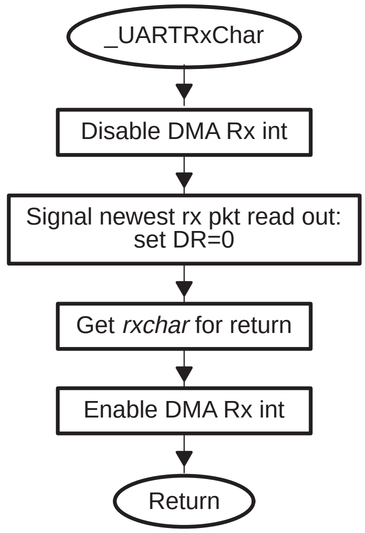

Figure 11. Procedure to Read Received Packet

The DMA receive interrupt is disabled during the receive routine to prevent this interrupt from occurring between the clearing of DR and the return of the character. If a new receive interrupt did occur at that point, the receive data would be overwritten but an overrun error would not be noted because DR was 0. Since the DR flag would be set by the ISR, the same data would then be read out a second time once this routine returned.

8 Overview of Code

The code supporting this report implements the UART in the manner described by the document, except for a couple exceptions.

Though this report details the case where the DMA pointers are post-decremented, the code provides a way to use the DMA with post-incremented pointers, if desired. This is selected with the DMA_PTR_MOD equate in the *UARTsetup.inc* file. Note that if the 5410 is used, the pointers must be post-decremented in order for the UART to work properly.

Also, the DMA in ABU mode on the 5402 works differently than that of the 5410. It generates early DMA interrupts if the DMA pointer is started in the second half of the DMA buffer. For this reason, the code provides a selectable workaround, which ensures the DMA is only restarted with the pointer in the first half of the buffer. This workaround is selected with the DMA_ABU_FIX equate in the *UARTsetup.inc* file.

There are 11 equates, 1 public variable, 10 private variables, 12 public routines, and 2 private routines in the code. The public routines are all C-callable. All routines are written in C54x assembly code.

9 Equates

9.1 MCBSP_CHOICE

McBSP to use for UART (0-2, depending on 54xx choice).

9.2 DMA_RX_CHOICE

DMA channel to use for receive (0-5).

9.3 DMA_TX_CHOICE

DMA channel to use for transmit (0-5). Must be different than DMA_RX_CHOICE.

9.4 INTOSEL

Selection of DMA/McBSP multiplexed interrupts (0-3). The choices are device dependent and specified in the *TMS320C54x DSP Enhanced Peripherals*(2) guide.

9.5 PARITY

Specifies the type of parity checked and generated (0=no parity, 1=even, 2=odd, 3=mark, 4=space).

9.6 HSTOPBITS

Number of 1/2 stop bits (2,3 or 4) used in transmission. Gives either 1,1.5, or 2 stop bits.

9.7 DATABITS

Number of data bits (1-14 with parity, or 1-15 w/o parity) in each packet.

9.8 BAUDRATE

Baud rate divisor used to divide down CPU clock to get McBSP clock. Should be approximately DSPCLK/(16*baudrate). See equations 3 and 7.

9.9 INTERRUPT_BASED

Gives the user the option to process the UART events within the DMA receive and transmit ISRs (0=only use polling to check status of UART, 1=run ISR's for the interrupt events on UART).

9.10 DMA_PTR_MOD

Direction for DMA pointer modification (0=post-decrement, 1=post-increment). 5410 can only use post-decrement in order for each character to be properly processed.

9.11 DMA_ABU_FIX

Adds workaround for ABU difference in 5402 (0=no fix, 1=add fix). Difference occurs when DMA ABU started with DMA pointer in second half of buffer. The workaround ensures the DMA pointer is never started in the second half.

10 Public Variables

10.1 _UARTLSR

The software UART reports status to the DSP through the UART Line Status Register. This register is organized as in Figure 12.

	15–6	5	4	3	2	1	0
_UARTLSR	Reserved	THRE	BI	FE	PE	OE	DR

Figure 12. UART Status Register

The bit definitions are:

- DR - Data Ready

  Set when a new packet is received and decoded by the UART. Cleared when the _UARTRxChar routine is called. This bit should be polled prior to calling the _UARTRxChar routine to determine if a new character is available.

- OE - Overrun Error

  Set when another packet is received by the UART before the previous packet was read out. This bit must be manually cleared.

- PE - Parity Error

  Set when the parity of the received packet does not match the settings in the UART. This bit must be manually cleared.

- FE - Framing Error

  Set when an invalid stop bit is detected. This bit must be manually cleared.

- BI - Break Indicator

  Set when a break is detected. This bit must be manually cleared.

- THRE - Transmit Holding Register Empty

  Set when space is available in the transmit buffer for another packet. Cleared by the _UARTTxChar routine if the transmit buffer is filled (2 packets). This bit should be polled prior to calling the _UARTTxChar routine to determine if a new packet can be sent.

11 Private Variables

11.1 rxchar

Holds the last character received by the UART.

11.2 rxbufhalf

A flag signaling the valid half of the DMA receive buffer (the half with the newest packet). It is 0 for the first half and 1 for the second half. For post-decremented DMA pointers, the first half is the half at the higher addresses. For post-incremented pointers, the first half is the half at the lower addresses.

11.3 txbufhalf

A flag signaling the valid half of the DMA transmit buffer (the next half to write in). It is 0 for the first half and1 for the second half. For post-decremented DMA pointers, the first half is the half at the higher addresses. For post-incremented pointers, the first half is the half at the lower addresses.

11.4 numTxPkts

Holds the number of packets currently in the transmit buffer. The maximum number is 2. Used by the transmit routine to determine when to disable the DMA transmit channel (when *numTxPkts* decrements to 0) and when the buffer is full (when *numTxPkts* increments to 2). THRE is cleared to 0 whenever *numTxPkts* becomes 2.

11.5 TxBuffer[2*TxPKTBITS]

The DMA transmit buffer. It has a size of 2*TxPKTBITS and must be aligned on a 2^n word boundary greater than 2*TxPKTBITS.

11.6 RxBuffer[2*RxPKTBITS]

The DMA receive buffer. It has a size of 2*RxPKTBITS and must be aligned on a 2^n word boundary greater than 2*RxPKTBITS.

11.7 decoderMask

Used in decoding routine to mask out center samples for testing. Saves cycles to use a variable rather than an immediate value.

11.8 mask1011b

Used in the decoding routine to determine if bit is 1 or 0. Saves cycles to use a variable rather than an immediate value.

11.9 mask0100b

Used in the decoding routine to determine if bit is 1 or 0. Saves cycles to use a variable rather than an immediate value.

11.10 one

Used in the decoding routine to create decoded character. Saves cycles to use a variable rather than an immediate value.

12 Public Routines

12.1 _UARTInit(inputs: none; outputs: none)

Initializes the software UART, specifically the McBSP, DMA, and decoder variables. Should be run once prior to running _UARTStart for the first time. See Figure 7.

12.2 _UARTStart(inputs: A<0:start Rx, A==0:start Rx & Tx, A>0:start Tx; outputs: none)

Starts the software UART. Takes A as an input to determine if transmit (A>0), receive (A<0) or both (A=0) should be started. Initializes the status register and enables global interrupts. For transmit or receive, enables the McBSP clock generator, initializes the valid half of the DMA buffer, sets the DMA pointer to the end of the first half of the buffer, enables the DMA transmit/receive channel interrupt in the IMR register, and takes the McBSP transmit/receive channels out of reset. For transmit, also zeros out the number of packets in the transmit buffer. For receive, also enables the DMA Rx channel in DMPREC. This routine may be run whenever the UART is to be restarted (i.e. after the _UARTStop or _UARTInit routines). See Figure A–1 in Appendix A.

12.3 _UARTStop(inputs: A<0(stop Rx), A==0(stop Rx & Tx), A>0(stop Tx); outputs: none)

Stops the UART. Takes A as an input to determine if transmit (A>0), receive (A<0) or both (A=0) should be stopped. For transmit, waits until all packets in the transmit buffer are transmitted. For transmit or receive, disables the DMA transmit/receive interrupt and channel and puts the McBSP transmit/receive ports in reset. If both receive and transmit are stopped, it disables the McBSP clock generator. Use _UARTStart to restart the UART. This routine should not be called from within an ISR, as it requires interrupts to be enabled in order to run properly. See Figure A–2 in Appendix A.

12.4 _UARTSetBaudRate(inputs: A=clock divisor; outputs: none)

Sets a new baud rate for the UART. Divisor must be in accumulator A on entry and must conform to equations 3 and 7. The UART (receive and transmit) must be stopped using _UARTStop before this routine can be run. See Figure A–3 in Appendix A.

12.5 _UARTSetBreak(inputs: A!=0:send break, A==0:end break; outputs: none)

Sends a break to the receiver. With a non-zero input, sends a packet of all 0's with the stop bit set to 0. With an input of zero, sends a string of 1's. To send a long break, loop over this routine with a non-zero input for as many packet lengths as the break is desired to be, then call again with an input of zero to end the break. It is necessary to end the break by sending a string of 1's so that the line goes high before a new character is sent. If the line didn't go high, the next character will be misinterpreted because its start bit will be missed. This routine should only be called when THRE is 1, indicating space available for another transmit packet. See Figure A–4 in Appendix A.

12.6 _UARTTxChar(inputs: A=char to transmit; outputs: none)

Transmits a character given in accumulator A. Adds start, stop and parity bits and stores the oversampled data in the valid half of the DMA transmit buffer. The DMA transmit channel is appropriately enabled, as specified in Figure 8. This routine should only be called when THRE is 1, indicating space available for another transmit packet.

12.7 _UARTRxChar(inputs: none; outputs: A=last received char)

Receives the newest character. Returns the last character read and resets the DR flag. This routine should only be called when DR is 1, indicating a new packet has been received. See Figure 11 for a description of the process.

12.8 _UARTDMATxISR(inputs: none; outputs: none)

Must be branched to from the DMA channel interrupt vector used for the transmit channel. Entered when the DMA sends a packet. Sets the THRE flag and disables the DMA transmit channel if no more packets are in the transmit buffer. See Figure 9 for a description of the process.

12.9 _UARTDMARxISR(inputs: none; outputs: none)

Must be branched to from the DMA channel interrupt vector used for the receive channel. Entered when the DMA receives a new packet. Decodes the received bits, checks for error conditions, stores the received character to *rxchar*, and sets the DR flag. See Figure 10 for a description of the process.

12.10 _UARTRBFint(inputs: none; outputs: none)

Called from _UARTDMARxISR if INTERRUPT_BASED mode is enabled, allowing receive event processing within an ISR. _UARTRxChar may be called from within this routine to receive a new character. Because it is run from within an ISR, the context must be saved and restored within this routine.

12.11 _UARTTBEint(inputs: none; outputs: none)

Called from _UARTDMATxISR if INTERRUPT_BASED mode is enabled, allowing transmit event processing within an ISR. _UARTTxChar or _UARTSetBreak may be called from within this routine to send a new character. Because it is run from within an ISR, the context must be saved and restored within this routine.

12.12 _UARTLSIint(inputs: none; outputs: none)

Called from _UARTDMARxISR if INTERRUPT_BASED mode is enabled, allowing error condition processing within an ISR. Because it is run from within an ISR, the context must be saved and restored within this routine.

13 Private Routines

13.1 ParityCalc(inputs: A=received char (data & parity bits only); outputs: TC=0 (even parity), TC=1(odd parity))

Performs parity calculation using the successive approximation technique described in the Texas Instruments Application Brief, *Parity Generation on the TMS320C54x*(3) by David Nerge. The TC bit will equal 1 if the current parity is odd, and 0 if the current parity is even.

13.2 ParityCheck(inputs: A=received char (data & parity bits only); outputs: AL=received char (data bits only))

Checks the parity of the received word against the parity setting of the UART and strips off the parity bit. Invalid parity is reported in the PE status bit.

14 Usage of UART Code

The UART routines are contained in *uart.asm*, located in Appendix B. The routines are all C callable.

There is an include file with the UART code (*UARTsetup.inc* in Appendix C) in which parameters of the code must be set. These include choices of the McBSP to use, the DMA channels to use, the type of parity to generate and detect (if any), the baud rate divisor, the number of stop and data bits, and whether to use polling or interrupt mode.

An example of using the software UART is included in the files *ExampleC.asm* in Appendix D (C code interface) and *ExampleASM.asm* in Appendix E (ASM code interface). An example interrupt vector table is included in *vectors.asm*, in Appendix F. An example command file is included in *uart.cmd,* in Appendix G. The example files perform a loopback of received data and were tested by hooking up an RS232 interface to a PC running HyperTerminal (see the RS232 Connections section for more information on the hookup). If any error is received, the UART generates a break back to the host.

The PLL and processor are initialized before starting the UART. See the *TMS320C54x DSP CPU and Peripherals*(4) guide for information on the PLL and interrupts. See the *TMS320C54x DSP Enhanced Peripherals*(2) guide for detailed information on choosing which DMA channel and McBSP is available for your particular device, as well as which interrupts are multiplexed and how they are selected in INTOSEL.

Then, the _UARTInit routine is called. Once this is done, calling _UARTStart with A=0 will enable the UART to start receiving and transmitting packets.

There are two types of examples in the code. One uses a polling method to transmit and receive characters; the other uses an interrupt method.

For the polling method, to transmit a character, first check that the THRE bit in *_UARTLSR* equals 1. If so, load the character to accumulator A and call _UARTTxChar.

To receive a new character, check that the DR bit in *_UARTLSR* equals 1 and then call _UARTRxChar. The new character is returned in accumulator A.

For the INTERRUPT_BASED mode, to transmit a block of characters, the first two characters in the buffer are manually written by calling the transmit routine twice (fills the transmit buffer). The ISR writes a new character to the buffer whenever space opens up, keeping the UART at the maximum transmit rate (consecutive characters). When the buffer is emptied, the ISR exits without transmitting a new character. No more interrupts will occur after that, which is why the first characters must be manually written.

The receive ISR writes the characters into a buffer automatically. A variable is used to check how many characters are in the buffer and perform different routines based on this number.

To halt the UART, call _UARTStop with A=0.

15 Performance

15.1 Memory

The memory used by the UART code depends on the number of total bits in a packet and especially whether parity generation and detection is enabled, as these are compile-time options. The approximate size in words, assuming the DMA ABU Fix is not used, is given in Table 4.

Table 4. UART Memory Consumption

	Not Interrupt Based		Interrupt Based	
Memory Type	**No Parity**	**Parity**	**No Parity**	**Parity**
Program	414	439	422	447
Data	15+4D+4P+4S			

D=#data bits, S=#stop bits, P=1 if parity enabled
Memory consumption is measured in words (1 word=2 bytes)

15.2 Cycle Count

The number of cycles each UART routine consumes, with no DMA ABU fix, is listed in Table 5.

Table 5. UART Routine Cycle Counts

Routine	Not Interrupt Based: Maximum Data Rate	Not Interrupt Based: Individual Char Sent
_UARTInit	759	759
_UARTStart (rx & tx)	1096	1096
_UARTStop (rx & tx)	583	583
_UARTSetBaudRate	10	10
_UARTSetBreak	45+D+P+2S	72+D+P+2S*
_UARTTxChar	43+7D+28P+2S	71+7D+28P+2S*
_UARTRxChar	15	15
_UARTDMATxISR	22	22
_UARTDMARxISR	74+10D+42P	74+10D+42P
ParityCalc	14	14
ParityCheck	14	14

D=#data bits, S=#stop bits, P=1 if parity enabled
* Will take longer if bits of prev char still being transmitted by McBSP

In Table 5, the Maximum Data Rate is achieved when characters are transmitted consecutively, with no idle time on the data lines. The extra savings in cycle count is achieved due to the fact that the UART does not need to shut down and restart the DMA transmit channel.

From Table 5, it can be seen that to transmit consecutive packets with 8 data bits, 1 stop bit and with parity enabled (total of 11 bits with start bit) will cost 129+22=151 cycles per packet, or 151/11=13.7 cycles/bit. At 19200 baud rate that is 13.7*19200=0.26 MIPS. At 115200 baud rate that is 1.58 MIPS.

To receive a packet of 8 data bits, 1 stop bit and with parity enabled will cost 15+196=211 cycles per packet, or 211/11=19.2 cycles/bit. At 19200 baud rate that is 18.7*19200=0.37 MIPS. At 115200 baud rate that is 2.21 MIPS.

The cycles and MIPS for the UART transmit and receive processes are displayed in Table 6.

Table 6. Performance of Software UART

#Data Bits	Parity Enabled	#Stop Bits	Cycles: Transmit	Cycles: Receive	MIPS 19200 baud: Transmit	MIPS 19200 baud: Receive	MIPS 115200 baud: Transmit	MIPS 115200 baud: Receive
5	N	1	102	139	0.28	0.38	1.68	2.29
5	Y	1	130	181	0.31	0.43	1.87	2.61
8	N	1	123	169	0.24	0.32	1.42	1.95
8	Y	1	151	211	0.26	0.37	1.58	2.21

These numbers do not reflect the cycles waited while polling the status bits or the cycles used in ISR's for an interrupt-based routine. Note that as the number of bits increase, the MIPS go down due to the low overhead for additional bit processing.

The rates at which the UART can run are limited by the clock divider in the McBSP. The divider ranges from 1-256, so given a DSP clock frequency of 75MHz, the McBSP can run between 293kHz and 37.5MHz (can't go above 50 MHz). The oversampling of 16 leads to bit rates of approximately 18.31kbps to 2344kbps. If the equations concerning the McBSP sampling rate (3 and 7) are not satisfied such that an integer value of the divisor results, then the UART will misinterpret the asynchronous data, causing errors in the transfer.

16 Verification

This code has been verified at 19200bps and 115200bps running on a 5410 EVM and 5402 DSK (at 75 MHz) using three different test environments. First, it was tested using HyperTerminal on a PC attached over an RS232 cable to the EVM/DSK. Data was looped back by the UART to the PC, where it was displayed and compared to the transmitted data. It was also tested by tying the RS232 transmit line to the RS232 receive line and comparing the looped back data on the DSP. Finally, it was tested by connecting to a TL16C550 UART running in non-FIFO mode.

17 RS232 Connections

The signal coming across the RS232 interface has a level of 10 volts for a 0 and –10 volts for a 1. When no signal is present, it is pulled to –10 volts. The DSP I/O level is 3.3 volts when no signal is present or a 1 is sent, and 0 volts when a 0 is sent. In order to interface to the RS232 cable, a level shifter circuit was built as described in Figure 13. A Max232 RS232 Transceiver chip was used for the interface.

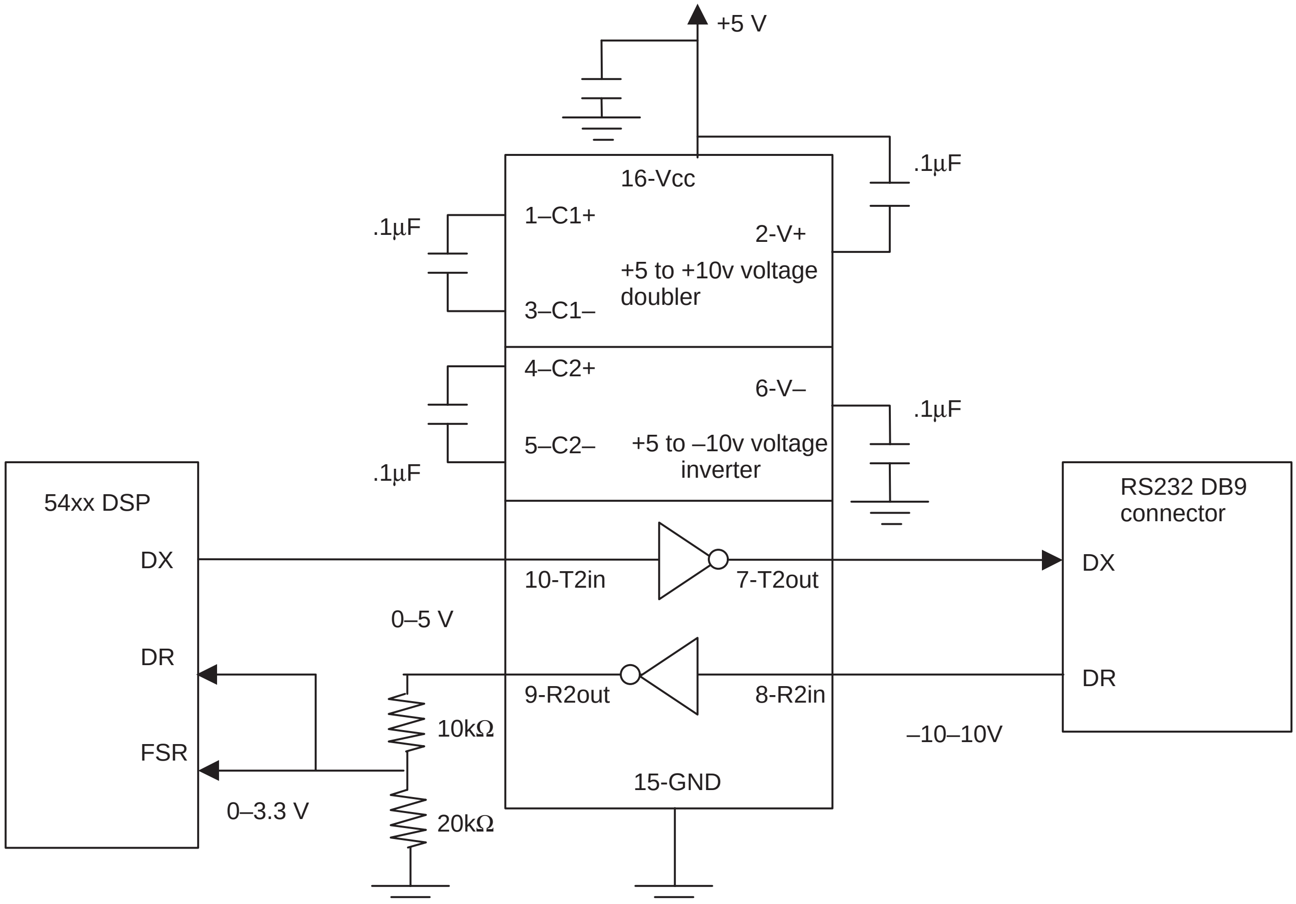

Figure 13. RS232 Interface Circuit

18 References

1. *TL16C450 Asynchronous Communications Element* data sheet (SLLS037B).
2. *TMS320C54x DSP Enhanced Peripherals Reference Set Volume 5* (SPRU302) .
3. Nerge, David. *Parity Generation on the TMS320C54x,* TMS320 DSP Designer’s Notebook, Application Brief (SPRA266).
4. *TMS320C54x DSP CPU and Peripherals Reference Set Volume 1* (SPRU131).

Appendix A Flowcharts for Routines

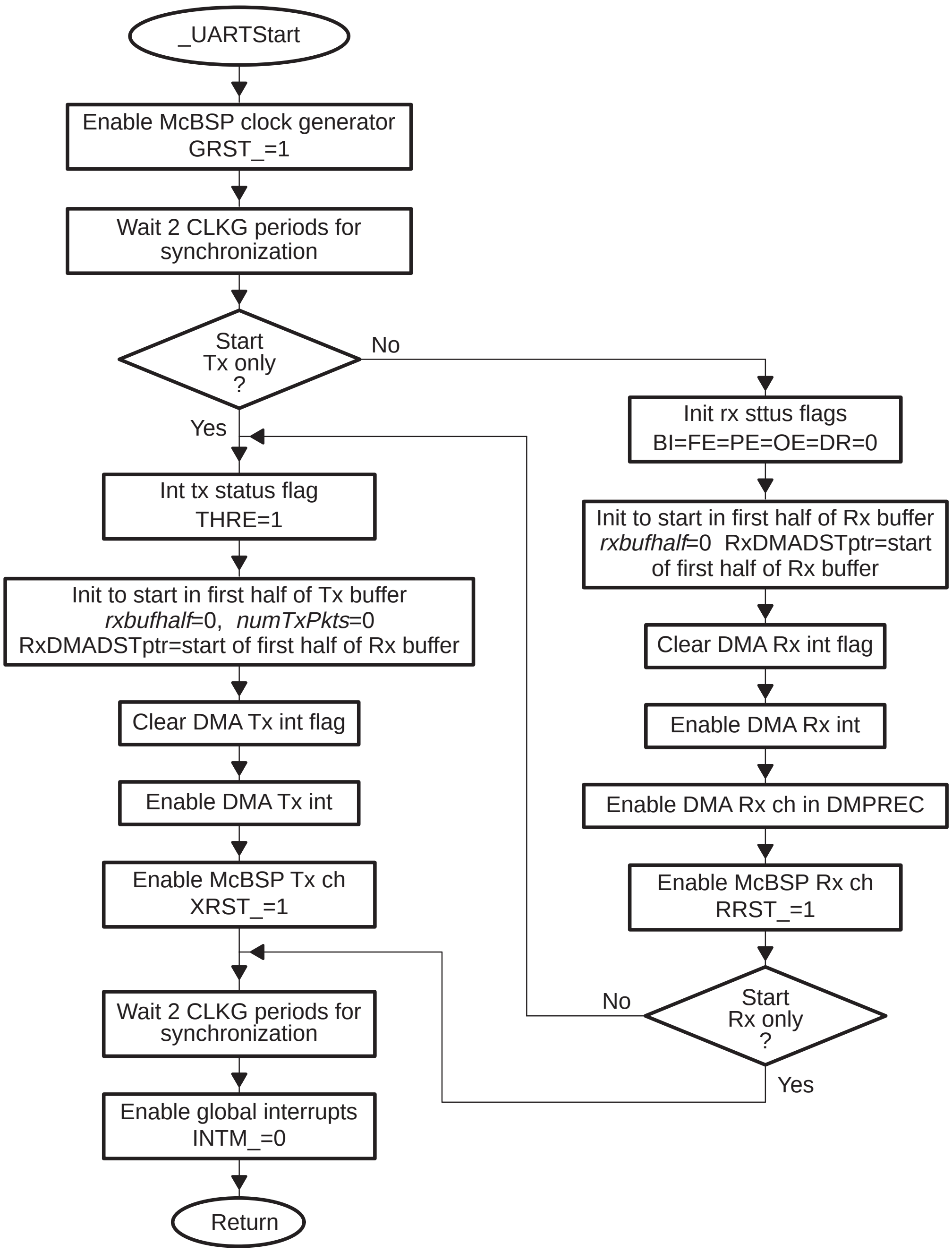

Figure A–1. Procedure to Start UART

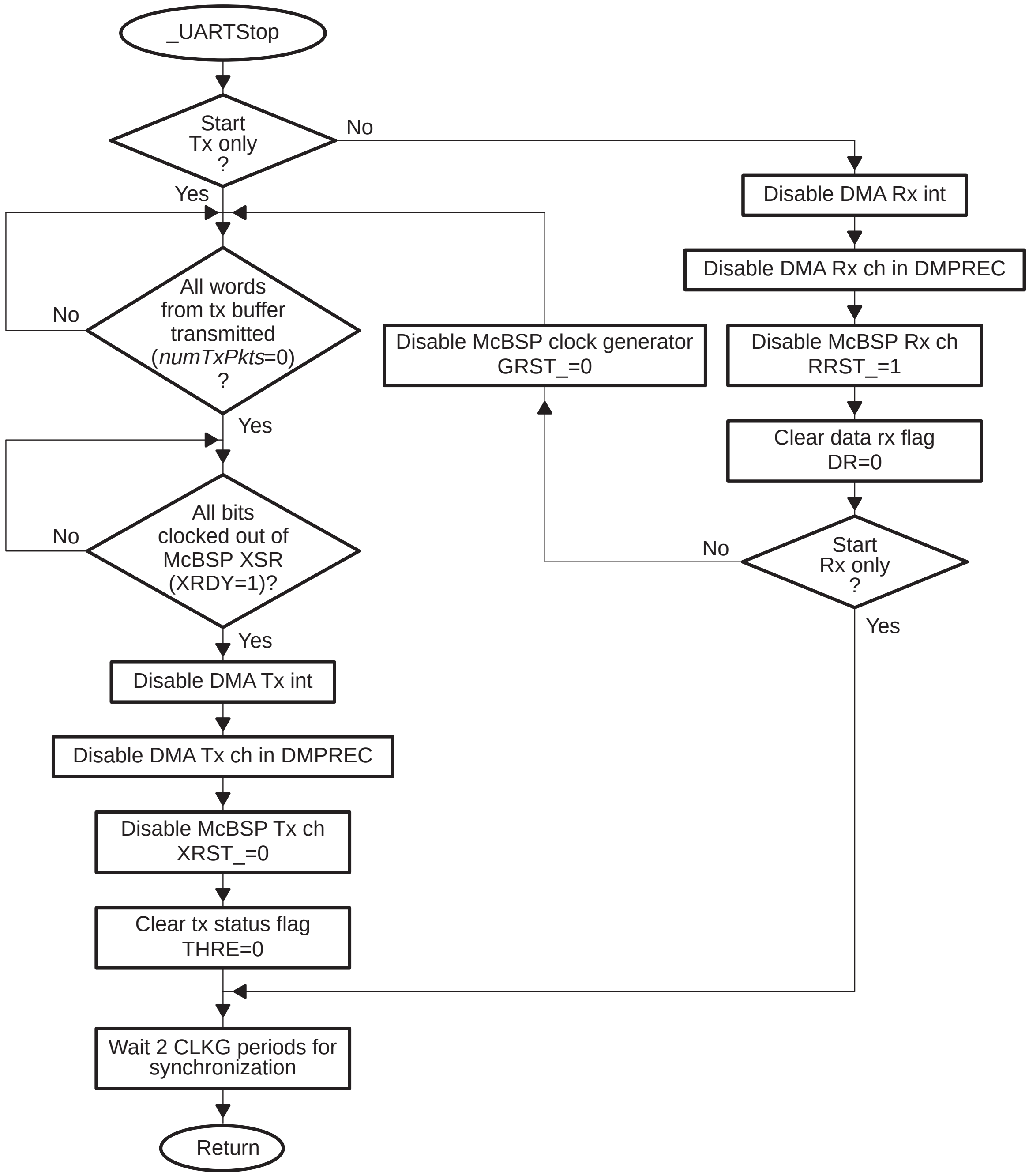

Figure A–2. Procedure to Stop UART

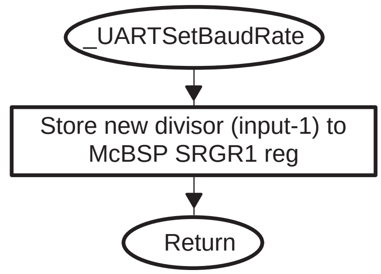

Figure A–3. Procedure to Change Baud Rate

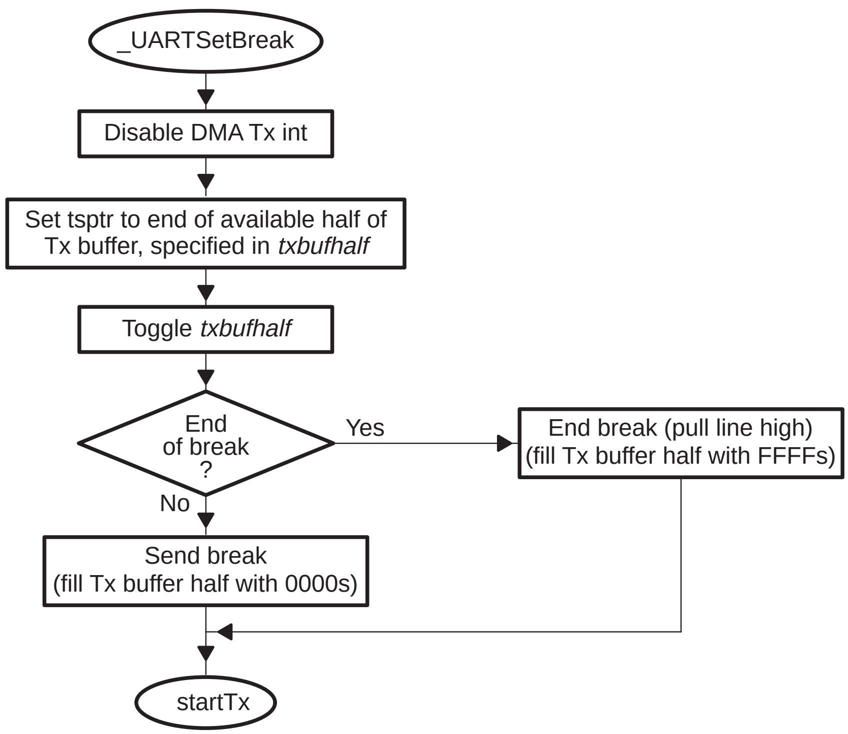

Figure A–4. Procedure to Send a Break

Appendix B UART Code (uart.asm)

```
*******************************************************************
* Filename:  uart.asm                                              *
* Function:  Software UART                                         *
* Author:    Robert J. DeNardo                                     *
*            Texas Instruments, Inc                                *
* Revision History:                                                *
* 11/12/99  Original Code                        Robert DeNardo    *
* 12/21/99  Increased data bits to 1-15.         Robert DeNardo    *
*           Disabled ints during RxChar &                          *
*           TxChar. Break routine rewritten.                       *
*           Removed numTxPkts check from TxChar                    *
*           routine. Removed 'ssbx CPL' in                         *
*           RxISR. Added latency after McBSP                       *
*           reset. Disabled DMA chs and McBSP                      *
*           in Init. Moved INTOSEL init to Init.                   *
*           Disable and enable GRST during McBSP                   *
*           reg writes. Removed UART stop and                      *
*           start from SetBaud routine. Fixed                      *
*           break detect to check stop bit=0 too.                  *
*           Check lower 4 bits of stop bit, not                    *
*           upper.                                                 *
* 1/07/00   Added workaround for 5402 DMA ABU    Robert DeNardo    *
*           Added (inc/dec)rement option for                       *
*           DMA pointers.                                          *
* 09/05/00  Fixed _UARTDMATxISR to conditionally Robert DeNardo    *
*           turn off DMA BEFORE calling                            *
*           _UARTTBEint. Keeps DMA from turning                    *
*           off too late or from turning off even                  *
*           if new packet in buffer.                               *
*                                                                  *
* Notes:                                                           *
* This code implements a software UART using a McBSP and 2 DMA     *
* channels, with the following features:                           *
*                                                                  *
*   Generation of start and stop bits                              *
*   Compile-time selectable data length (1-15 bits)                *
*   Compile-time selectable stop bit length (1, 1.5, 2 bits)       *
*   Compile-time selectable parity (none,even,odd,mark,space)      *
*   Compile- and Run-time selectable baud rate                     *
*   Break detection and generation                                 *
*   Parity detection and generation                                *
*   Overrun detection                                              *
*   Framing error detection                                        *
*   Double buffered receive and transmit signals                   *
*   Routines to transmit and receive characters                    *
*   Error condition ISR                                            *
*                                                                  *
* Parameters:                                                      *
* All parameters which the user can set are defined in the         *
* UARTsetup.inc file.                                              *
*                                                                  *
* Public Routines:                                                 *
* All public routines are C callable. Routines which the           *
* user may access include:                                         *
```

```
*   _UARTDMARxISR - must be branched to from DMA Rx channel        *
*                   interrupt vector.                              *
*   _UARTDMATxISR - must be branched to from DMA Tx channel        *
*                   interrupt vector.                              *
*   _UARTTxChar - transmits a character                            *
*   _UARTRxChar - reads last received character                    *
*   _UARTStart - starts the UART                                   *
*   _UARTStop - stops the UART                                     *
*   _UARTInit - initializes the McBSP and DMA for the UART         *
*   _UARTSetBaudRate - changes the baud rate                       *
*   _UARTSetBreak - sends a break                                  *
*                                                                  *
* If using INTERRUPT_BASED mode, user must define these funcs:     *
*   _UARTLSIint - handles error conditions                         *
*   _UARTRBFint - handles received chars                           *
*   _UARTTBEint - handles new transmission                         *
*                                                                  *
* Public Registers:                                                *
*   _UARTLSR - contains status bits. See register definition in    *
*             UARTsetup.inc for bit information.                   *
*                                                                  *
*                                                                  *
* This code uses the DMA ABU mode. It was tested with an           *
* interface to a PC COM port through an RS232 level shifter        *
* on a 5410 EVM and 5402 DSK, and with an interface to a           *
* HW UART.                                                         *
*                                                                  *
* McBSP is selectable for the interface (default=0)                *
* DMA Channel for Receive is selectable (default=4)                *
* DMA Channel for Transmit is selectable (default=5)               *
*                                                                  *
* Hardware setup:                                                  *
* Data receive line must be tied to FSR and DR of the McBSP.       *
* The data transmit line must be tied to the DX pin of the         *
* McBSP.                                                           *
*                                                                  *
* See the 'Implementing a Software UART on the                     *
* TMS320C54xx with the McBSP and DMA' Application Note             *
* for detailed information on how this code works and              *
* how it can be used (SPRA661).                                    *
********************************************************************
      .version 548
      .mmregs

*********** public routines defined in this file ****************
      .def   _UARTDMARxISR
      .def   _UARTDMATxISR
      .def   _UARTTxChar
      .def   _UARTRxChar
      .def   _UARTStart
      .def   _UARTStop
      .def   _UARTInit
      .def   _UARTSetBaudRate
      .def   _UARTSetBreak
*********** public variables defined in this file ***************
      .def   _UARTLSR
```

```
***************** Equates used in this file *********************
; parity choices
NO             .set   0
EVEN           .set   1
ODD            .set   2
MARK           .set   3
SPACE          .set   4

; decoder equates
DECODER_MASK   .set   0xF<<6    ; only test middle 4 bits of 16 bit word
MASK1011b      .set   1011b<<6  ; used to test for a 1
MASK0100b      .set   0100b<<6  ; used to test for a 1
ONE            .set   1         ;
STOPBITSHFT    .set   6         ; amount to left shift half stop bit before decoding
                                ; puts lower 4 bits into center of 16 bit word

; McBSP register addresses
SPSA0          .set   38h       ; address of McBSP0 subaddress register
DRR10          .set   21h       ; address of McBSP0 DRR1 register
DXR10          .set   23h       ; address of McBSP0 DXR1 register
SPSA1          .set   48h       ; address of McBSP1 subaddress register
DRR11          .set   41h       ; address of McBSP1 DRR1 register
DXR11          .set   43h       ; address of McBSP1 DXR1 register
SPSA2          .set   34h       ; address of McBSP2 subaddress register
DRR12          .set   31h       ; address of McBSP2 DRR1 register
DXR12          .set   33h       ; address of McBSP2 DXR1 register

; offsets for McBSP sub-addressed registers
SPCR1          .set   0         ; Serial Port Control Register 1
SPCR2          .set   1         ; Serial Port Control Register 2
RCR1           .set   2         ; Receive Control Register 1
RCR2           .set   3         ; Receive Control Register 2
XCR1           .set   4         ; Transmit Control Register 1
XCR2           .set   5         ; Transmit Control Register 2
SRGR1          .set   6         ; Sample Rate Generator Register 1
SRGR2          .set   7         ; Sample Rate Generator Register 2
PCR            .set   14        ; Pin Control Register

; SPCR1 -- bit definitions
RRST           .set   1<<0      ; RRST_ bit

; SPCR2 -- bit definitions
XRST           .set   1<<0      ; XRST_ bit mask
XRDY           .set   1<<1      ; XRDY bit mask
GRST           .set   1<<6      ; GRST_ bit mask
FRST           .set   1<<7      ; FRST_ bit mask

; Reset Latency -- amount of time needed after McBSP is put in or out of reset
RESET_LATENCY .set   2*256     ; max time is 2 bit clocks for max divisor (256)

; DMA register addresses
DMPREC         .set   54h       ; channel PRiority and Enable Control register
DMSA           .set   55h       ; Sub-Bank Access Register
DMSDI          .set   56h       ; Sub-Bank Access Register with Auto-Increment
DMSDN          .set   57h       ; Sub-Bank Access Register without Auto-Increment

; offsets for DMA sub-addressed registers
```

```
DMSRC0        .set     0      ; Channel 0 Source Address Register (first reg for ch 0)
DMSRC1        .set   5        ; Channel 1 Source Address Register (first reg for ch 1)
DMSRC2        .set   10       ; Channel 2 Source Address Register (first reg for ch 2)
DMSRC3        .set   15       ; Channel 3 Source Address Register (first reg for ch 3)
DMSRC4        .set   20       ; Channel 4 Source Address Register (first reg for ch 4)
DMSRC5        .set   25       ; Channel 5 Source Address Register (first reg for ch 5)

; DMPREC:DE -- DMA channel enable definitions
DMAch0        .set   1<<0
DMAch1        .set   1<<1
DMAch2        .set   1<<2
DMAch3        .set   1<<3
DMAch4        .set   1<<4
DMAch5        .set   1<<5

; DMSFC:DSYN -- DMA sync event definitions
REVT0         .set   0001b      ; McBSP0 receive event
XEVT0         .set   0010b      ; McBSP0 transmit event
REVT2         .set   0011b      ; McBSP2 receive event
XEVT2         .set   0100b      ; McBSP2 transmit event
REVT1         .set   0101b      ; McBSP1 receive event
XEVT1         .set   0110b      ; McBSP1 transmit event

;IMR/IFR bit definitions
DMAC0int      .set   1<<6
DMAC1int      .set   1<<7
DMAC2int      .set   1<<10
DMAC3int      .set   1<<11
DMAC4int      .set   1<<12
DMAC5int      .set   1<<13

       .copy ”UARTsetup.inc”     ; contains USER specifications for UART

***************** Config Error Checking *****************
 .if ((MCBSP_CHOICE>2)|(MCBSP_CHOICE<0))
       .emsg ”CONFIG ERROR: MCBSP_CHOICE limited to 0, 1, or 2”
 .endif
 .if ((DMA_RX_CHOICE>5)|(DMA_RX_CHOICE<0))
       .emsg ”CONFIG ERROR: DMA_RX_CHOICE limited to 0-5”
 .endif
 .if ((DMA_TX_CHOICE>5)|(DMA_TX_CHOICE<0))
       .emsg ”CONFIG ERROR: DMA_TX_CHOICE limited to 0-5”
 .endif
 .if (DMA_RX_CHOICE==DMA_TX_CHOICE)
       .emsg ”CONFIG ERROR: DMA_RX_CHOICE and DMA_TX_CHOICE must be different”
 .endif
 .if (INTOSEL<0)|(INTOSEL>3)
       .emsg ”CONFIG ERROR: INTOSEL limited to 0-3”
 .endif
 .if (PARITY<NO)|(PARITY>SPACE)
       .emsg ”CONFIG ERROR: PARITY limited to NO, EVEN, ODD, MARK or SPACE”
 .endif
 .if (PARITY!=NO)&((DATABITS>14)|(DATABITS<1))
       .emsg ”CONFIG ERROR: With PARITY, DATABITS limited to 1-14”
 .endif
 .if (PARITY==NO)&((DATABITS>15)|(DATABITS<1))
       .emsg ”CONFIG ERROR: Without PARITY, DATABITS limited to 1-15”
```

```
 .endif
 .if ((HSTOPBITS>4)|(HSTOPBITS<2))
        .emsg ”CONFIG ERROR: HSTOPBITS limited to 2, 3, or 4”
 .endif
 .if ((DMA_PTR_MOD<0)|(DMA_PTR_MOD>1))
        .emsg ”CONFIG ERROR: DMA_PTR_MOD limited to 0 or 1”
 .endif
 .if ((DMA_ABU_FIX<0)|(DMA_ABU_FIX>1))
        .emsg ”CONFIG ERROR: DMA_ABU_FIX limited to 0 or 1”
 .endif

**************** Config Determinations ******************
 .if PARITY==NO
PARITYBITS    .set   0
 .else
PARITYBITS    .set   1
 .endif

 .if PARITY!=NO                  ; set position of calculated parity bit for ParityCalc
routine
  .if DATABITS+PARITYBITS>16
PARITYCHECK   .set   1<<31
  .elseif DATABITS+PARITYBITS>8
PARITYCHECK   .set   1<<15
  .elseif DATABITS+PARITYBITS>4
PARITYCHECK   .set   1<<7
  .elseif DATABITS+PARITYBITS>2
PARITYCHECK   .set   1<<3
  .elseif DATABITS+PARITYBITS>1
PARITYCHECK   .set   1<<1
  .endif
 .endif

STARTBITS     .set   1           ; always 1 start bit
RxHSTOPBITS   .set   1           ; Receiver only checks first 1/2 stop bit
TxHSTOPBITS   .set   HSTOPBITS   ; Transmitter sends # half stop bits defined by user
STOPBIT       .set   1<<DATABITS+PARITYBITS   ; define the stop bit position
TxPKTBITS     .set   STARTBITS+DATABITS+PARITYBITS+TxHSTOPBITS ; total number of bits
                                                           ; in each word
RxPKTBITS     .set   STARTBITS+DATABITS+PARITYBITS+RxHSTOPBITS ; total number of bits
                                                           ; in each word

 .if MCBSP_CHOICE==0
SPSA          .set   SPSA0
DRR1reg       .set   DRR10  ; need to use ’reg’ because DRR1 already defined in .mmregs
DXR1reg       .set   DXR10  ; need to use ’reg’ because DXR1 already defined in .mmregs
REVT          .set   REVT0
XEVT          .set   XEVT0
 .elseif MCBSP_CHOICE==1
SPSA          .set   SPSA1
DRR1reg       .set   DRR11
DXR1reg       .set   DXR11
REVT          .set   REVT1
XEVT          .set   XEVT1
 .elseif MCBSP_CHOICE==2
SPSA          .set   SPSA2
DRR1reg       .set   DRR12
```

```
DXR1reg       .set   DXR12
REVT          .set   REVT2
XEVT          .set   XEVT2
 .endif
McBSPDataReg  .set   SPSA+1  ; McBSP register to write/read data values

 .if DMA_RX_CHOICE==0
RxDMAptr      .set   DMSRC0   ; point to first DMA regsiter for this channel
RxDMACh       .set   DMAch0   ; define bit to enable this DMA channel in DPREC:DE
RxDMAInt      .set   DMAC0int ; define bit mask for this DMA channel interrupt in
                              ; IMR/IFR

 .elseif DMA_RX_CHOICE==1
RxDMAptr      .set   DMSRC1
RxDMACh       .set   DMAch1
RxDMAInt      .set   DMAC1int
 .elseif DMA_RX_CHOICE==2
RxDMAptr      .set   DMSRC2
RxDMACh       .set   DMAch2
RxDMAInt      .set   DMAC2int
 .elseif DMA_RX_CHOICE==3
RxDMAptr      .set   DMSRC3
RxDMACh       .set   DMAch3
RxDMAInt      .set   DMAC3int
 .elseif DMA_RX_CHOICE==4
RxDMAptr      .set   DMSRC4
RxDMACh       .set   DMAch4
RxDMAInt      .set   DMAC4int
 .elseif DMA_RX_CHOICE==5
RxDMAptr      .set   DMSRC5
RxDMACh       .set   DMAch5
RxDMAInt      .set   DMAC5int
 .endif

 .if DMA_TX_CHOICE==0
TxDMAptr      .set   DMSRC0
TxDMACh       .set   DMAch0
TxDMAInt      .set   DMAC0int
 .elseif DMA_TX_CHOICE==1
TxDMAptr      .set   DMSRC1
TxDMACh       .set   DMAch1
TxDMAInt      .set   DMAC1int
 .elseif DMA_TX_CHOICE==2
TxDMAptr      .set   DMSRC2
TxDMACh       .set   DMAch2
TxDMAInt      .set   DMAC2int
 .elseif DMA_TX_CHOICE==3
TxDMAptr      .set   DMSRC3
TxDMACh       .set   DMAch3
TxDMAInt      .set   DMAC3int
 .elseif DMA_TX_CHOICE==4
TxDMAptr      .set   DMSRC4
TxDMACh       .set   DMAch4
TxDMAInt      .set   DMAC4int
 .elseif DMA_TX_CHOICE==5
TxDMAptr      .set   DMSRC5
```

```
TxDMACh       .set   DMAch5
TxDMAInt      .set   DMAC5int
 .endif

********************* variable definitions ********************
UARTvars     .usect   "UART_vars",9,1 ; create a section for the UART vars. Keep in
                                      ; same data page
_UARTLSR     .set   UARTvars+0        ; Line Status Register (LSR) holds information on
                                      ; errors and ready status
rxchar       .set   UARTvars+1        ; holds last character received
rxbufhalf    .set   UARTvars+2        ; defines which half of raw receive buffer will
                                      ; have next character
txbufhalf    .set   UARTvars+3        ; defines which half of raw transmit buffer is
                                      ; open for write
numTxPkts    .set   UARTvars+4        ; holds # of pkts in tx buffer
mask1011b    .set   UARTvars+5        ; used in decoder
mask0100b    .set   UARTvars+6        ; used in decoder
decodeMask   .set   UARTvars+7        ; used in decoder
one          .set   UARTvars+8        ; used in decoder

; This section must be aligned on 2^n boundary greater than TxPKTBITS*2:
TxBuffer     .usect "UARTTxBuffer",2*TxPKTBITS     ; Holds coded bits for transmission.

; This section must be aligned on 2^n boundary greater than RxPKTBITS*2:
RxBuffer     .usect "UARTRxBuffer",2*RxPKTBITS     ; Holds coded bits upon reception.

******** Config Determinations Dependent on Variables ***********
; DMA ptr modification (5410 needs to decrement pointers, while 5402
; and others can increment)
 .if DMA_PTR_MOD==0   ; (decrement) on 5410 DMA buffers must fill from high to low
                      ; (start bit at higher addr than stop bits)
             .asg   ar2-,ar2fwd          ; fills from high to low addr, so fwd is a
                                         ; decrement (fwd)
             .asg   ar2+,ar2bwd          ; backward direction is an increment (bwd)
Tx1stStart   .set   TxBuffer+2*TxPKTBITS-1   ; start of buffer halves are where start
                                             ; bits are
Tx2ndStart   .set   TxBuffer+TxPKTBITS-1
Rx1stEnd     .set   RxBuffer+RxPKTBITS   ; end of buffer halves are where stop bits are
Rx2ndEnd     .set   RxBuffer
Rx1stStart   .set   RxBuffer+2*RxPKTBITS-1
DMAptrMod    .set   010b              ; post decrement the DMA pointers
 .else   ; (DMA_PTR_MOD==1, increment) other 54xx DSPs can fill from low to high if
         ; desired
             .asg   ar2+,ar2fwd       ; fills from low to high addr, so fwd is an
                                      ; increment (fwd)
             .asg   ar2-,ar2bwd       ; backward direction is a decrement (bwd)
Tx1stStart   .set   TxBuffer          ; start of buffer halves are where start bits are
Tx2ndStart   .set   TxBuffer+TxPKTBITS
Rx1stEnd     .set   RxBuffer+RxPKTBITS-1; end of buffer halves are where stop bits are
Rx2ndEnd     .set   RxBuffer+2*RxPKTBITS-1
Rx1stStart   .set   RxBuffer
DMAptrMod    .set   001b         ; post increment the DMA pointers
 .endif

   .sect "uart"
*******************************************************************
*   USER DEFINED INTERRUPT FUNCTIONS                              *
```

```
*                                                                  *
*   These are run whenever an interrupt event occurs on            *
*   the UART.                                                      *
*   If it is desired to perform processing during an               *
*   interrupt, these can be used, otherwise a polling              *
*   scheme will work as well.                                      *
*   To include these in the code, INTERRUPT_BASED must be 1.       *
*   These routines should be defined in your code and must         *
*   follow the information given in their headers.                 *
********************************************************************
 .if INTERRUPT_BASED
    .ref   _UARTLSIint
    .ref   _UARTRBFint
    .ref   _UARTTBEint
  .if 0   ; Copy these into your own code or create your own C functions

********************************************************************
*   Function:  _UARTLSIint                                         *
*   Called By: _UARTDMARxISR                                       *
*   Purpose:  Run when a Line Status Interrupt event occurs.       *
*             This is when a Parity Error, Overrun Error,          *
*             Break Interrupt Error, or Framing Error occurs.      *
*             The registers, ar2, a, b, st0, st1, brc, rsa,        *
*             rea are saved and restored by _UARTDMARXISR.         *
*             Any other registers used MUST be saved and           *
*             restored by this routine.                            *
*   Inputs:   none                                                 *
*   Outputs:  none                                                 *
*   Modified: none                                                 *
********************************************************************
_UARTLSIint:
   ret

********************************************************************
*   Function:  _UARTRBFint                                         *
*   Called By: _UARTDMARxISR                                       *
*   Purpose:  Run when a new char is received into rxchar.         *
*             The registers, ar2, a, b, st0, st1, brc, rsa,        *
*             rea are saved and restored by _UARTDMARXISR.         *
*             Any other registers used MUST be saved and           *
*             restored during this routine.                        *
*   Inputs:   CPL = 0 (data page rel. direct addressing)           *
*   Outputs:  none                                                 *
*   Modified: none                                                 *
********************************************************************
_UARTRBFint:
   ret

********************************************************************
*   Function:  _UARTTBEint                                         *
*   Called By: _UARTDMATxISR                                       *
*   Purpose:  Run when a char has just been transmitted            *
*             and the UART is ready to transmit another.           *
*             The st0 register is saved and restored by            *
*             _UARTDMATxISR. Any other registers used MUST         *
*             be saved and restored during this routine.           *
```

```
*   Inputs:   none                                                 *
*   Outputs:  none                                                 *
*   Modified: none                                                 *
********************************************************************
_UARTTBEint:
   ret
  .endif
 .endif    ; End INTERRUPT_BASED

********************************************************************
*   Function: _UARTDMARxISR                                        *
*   Purpose:  ISR which runs whenever a new coded character        *
*             is received. This occurs when each half of           *
*             buffer is filled by the DMA Rx channel.              *
*             Reads raw character from Rx buffer and               *
*             decodes it from MSb to LSb. A majority rule          *
*             scheme decodes each bit. Start bits                  *
*             are ignored. Parity and stop bits are                *
*             checked and stripped. Any error is reported          *
*             in _UARTLSR register.                                *
*   Inputs:   rxbufhalf - contains half of raw rx buffer           *
*             where data is located                                *
*   Outputs:  rxchar - holds value of last rx char                 *
*             _UARTLSR - holds status of line and errors           *
*   Modified: none                                                 *
********************************************************************
_UARTDMARxISR:
      pshm     al                        ; save the registers used in ISR
      pshm     ah                        ;
      pshm     ag                        ;
      pshm     bl                        ;
      pshm     bh                        ;
      pshm     bg                        ;
      pshm     st0                       ;
      pshm     st1                       ;
      pshm     ar2                       ;
      rsbx     cpl                       ; set to data-page relative direct addressing
      pshm     brc                       ;
      pshm     rsa                       ;
      pshm     rea                       ;
      ld       #UARTvars,DP              ; load the UART variable data page
      bitf     @rxbufhalf,#1             ; check if in 2nd half
      xorm     #1,@rxbufhalf             ; update current half to receive into
      stm      #Rx2ndEnd,ar2             ; assume in 2nd half of buffer and point to
                                         ; end of it
      xc     2,ntc                       ; if not,
        stm  #Rx1stEnd,ar2               ; point ar2 to the end of first half of buffer
      stm    #RxPKTBITS-STARTBITS-1,brc  ; loop over the data+parity+stop bit
                                         ; (not start bit)
      rptbd  DecodeLoop-1                ;
        ld  *ar2bwd,#STOPBITSHFT,b       ; shift 1/2 stop bit so upper 4 bits are in
                                         ; middle of BL
        ld  #0,a                         ; init the decoded word to 0000h
      and   @decodeMask,b                ; only look at bits 6-9
      sub   @mask1011b,b                 ; check if equal to 11-15, (i.e. will decode
                                         ; to a 1)
```

```
      bcd   DecodeOne,bgeq              ; if it is a 1 instead, branch to decode of 1
        add @mask0100b,b                ; check if equal to -4 (i.e. if was
                                        ; originally a 7)
        sftl a,1                        ; assume it is a 0 by shifting in 0 into the
                                        ; LSb of decoded word
      nop                               ; need 2 cycles latency for xc
      xc    1,beq                       ; if it is equal to 7, decode to 1
DecodeOne:                              ;
        or  @one,a                      ; or the 1 into the LSb of decoded word
      ld    *ar2bwd,b                   ; load the next coded bit
DecodeLoop:                             ; at end, A holds decoded character
      and   #STOPBIT,a,b                ; check the decoded stop bit
      xc    2,aeq                       ; if stop bit and all data/parity bits are 0,
        orm #BI,@_UARTLSR               ; note a break detected
      xc    2,beq                       ; if stop bit is invalid (i.e. 0),
        orm #FE,@_UARTLSR               ; note the framing error
      and   #(~STOPBIT)&0xffff,a        ; strip the stop bit
 .if PARITY!=NO
      call  ParityCheck                 ; check if the parity is valid and strip
                                        ; parity bit (char output in AL)
 .endif                                 ;
      stl   a,@rxchar                   ; store the char
      bitf  @_UARTLSR,#DR               ; check if previous character read yet
      orm   #DR,@_UARTLSR               ; indicate character is ready in rxchar
      xc    2,tc                        ; if it was not read (DR=1 before we just set
it),
        orm #OE,@_UARTLSR               ; note the overrun error
 .if INTERRUPT_BASED        ; only do this if user wants to run ISRs for status changes
      call  _UARTRBFint                 ; call ISR to process received data
      bitf  @_UARTLSR,#(BI|FE|PE|OE)    ; check for any line status interrupt events
      cc    _UARTLSIint,tc              ; if any events set, call the ISR
 .endif
      popm   rea                        ; restore context
      popm   rsa                        ;
      popm   brc                        ;
      popm   ar2                        ;
      popm   st1                        ;
      popm   st0                        ;
      popm   bg                         ;
      popm   bh                         ;
      popm   bl                         ;
      popm   ag                         ;
      popm   ah                         ;
      popm   al                         ;
      rete                              ; return and enable global interrupts

*******************************************************************
*   Function: _UARTDMATxISR                                       *
*   Purpose:  ISR which runs whenever DMA has moved last          *
*             ”bit” of coded character to DXR.                    *
*             This occurs when each half of the raw Tx            *
*             buffer is emptied by the DMA Tx channel.            *
*             Resets a flag to indicate transmit process          *
*             is over and disables DMA Tx ch if no more           *
*             pkts in Tx buffer.                                  *
*   Inputs:   none                                                *
```

```
*   Outputs:  none                                                  *
*   Modified: none                                                  *
*********************************************************************
_UARTDMATxISR:
      pshm  st0
      addm  #-1,*(numTxPkts)         ; decrement # of Tx pkts in Tx buffer
      cmpm  *(numTxPkts),#1          ; check if another pkt is ready for transmit still
      orm   #THRE,*(_UARTLSR)        ; signal that another pkt can be put in Tx buffer
      bc    SkipDisableTx,tc         ; if another pkt is in buffer for Tx, don't
                                     ; disable DMA
      andm   #~TxDMACh,*(DMPREC)     ; disable DMA channel for Tx
SkipDisableTx:
 .if INTERRUPT_BASED
      call   _UARTTBEint             ; call routine to handle transmitter reg empty
 .endif
      popm   st0
      rete                           ; return and enable global interrupts

*********************************************************************
*   Function: _UARTTxChar                                           *
*   Purpose:  Adds parity, start, and stop bits to                  *
*             character to transmit. Puts coded character           *
*             into raw Tx buffer and resets the Transmit            *
*             Holding Register Empty flag. Only call when           *
*             THRE=1.                                               *
*   Inputs:   al = character to transmit                            *
*   Outputs:  none                                                  *
*   Modified: a,ar2,st0(tc,c),brc,rea,rsa                           *
*********************************************************************
_UARTTxChar:
      pshm  imr                    ; save the IMR state to restore Tx DMA int at end
      andm  #~TxDMAInt,*(imr)      ; disable the DMA Tx interrupt (so DMA ISR doesn't
                                   ; change state here)
 .if DMA_ABU_FIX   ; 5402 workaround (can't start DMA ABU in 2nd half of buffer)
      bitf  *(txbufhalf),#1        ; check if in 2nd half of buffer (txbufhalf=1)
      bc    NoAdjust,ntc           ; if not, nothing to worry about, so skip out
      cmpm  *(numTxPkts),#0        ; check if DMA is disabled (no packets to
                                   ; transmit)
      bc    NoAdjust,ntc           ; if DMA still on, nothing to worry about,
                                   ; so skip out
      xorm  #1,*(txbufhalf)        ; set txbufhalf to 0 to point to 1st half
      stm   #TxDMAptr,DMSA         ; set subaddress to Tx DMA Source register.
      stm   #Tx1stStart,DMSDN      ; Set Tx DMA pointer to start of 1st half
NoAdjust:
 .endif                            ; end 5402 workaround
      bitf  *(txbufhalf),#1        ; check if in 2nd half of buffer
      xorm  #1,*(txbufhalf)        ; toggle the buffer half which is available
      stm   #Tx2ndStart,ar2        ; assume in 2nd half of buffer (point to start of
                                   ; this half)
      xc    2,ntc                  ; if not,
        stm #Tx1stStart,ar2        ; point ar2 to the start of first half of buffer
 .if  (PARITY==EVEN)|(PARITY==ODD)
      call  ParityCalc             ; returns parity in TC: (0=even, 1=odd) and char
                                   ; in AL
   .if PARITY==EVEN
      xc   2,tc                    ; if need to add one to make even parity
```

```
        or   #1,DATABITS,a         ; add in the parity bit
   .endif
   .if PARITY==ODD
      xc   2,ntc                   ; if need to add one to make odd parity
        or #1,DATABITS,a           ; add in the parity bit
   .endif
 .elseif PARITY==MARK
      or    #1,DATABITS,a              ; MARK parity always adds in the parity bit
 .endif
      stm   #DATABITS+PARITYBITS-1,brc ; now translate the data and parity bits in the
                                       ; character
      rptbd CodeLoop-1                 ;
        st  #00000h,*ar2fwd            ; write the start bit first
      ror   a                          ; rotate LSb out of A into C (carry bit)
      st    #00000h,*ar2               ; assume it is a zero and write code for 0 into
                                       ; buffer
      xc    2,c                        ; if it was a 1 instead,
        st  #0ffffh,*ar2               ; write code for 1 into buffer
      mar   *ar2fwd                    ; increment buffer pointer
CodeLoop:
      rpt   #TxHSTOPBITS-1             ;
        st  #000ffh,*ar2fwd            ; write the stop bits into the buffer
startTx:
      addm  #1,*(numTxPkts)            ; increment number of pkts in Tx buffer
      cmpm  *(numTxPkts),#2            ; check if another pkt was already in buffer
                                       ; (now have 2)
      andm  #~THRE,*(_UARTLSR)         ; signal that no space is available in Tx
                                       ; buffer (assume it now has 2 words)
      bcd   SkipDMARestart,tc          ; if another pkt in buffer, skip the restart
                                       ; routine
        stm #SPSA,ar2                  ; point to McBSP subaddress register
      st    #SPCR2,*ar2+               ; write the SPCR2 subregister offset and point
                                       ; to access register
waitReady:
      bitf  *ar2,#XRDY                 ; check if XRDY==1
      bc    waitReady,ntc              ; wait until serial port has clocked out any
                                       ; bits in XSR
      stm   #TxDMAptr,DMSA             ; set subaddress to Tx DMA Source register
      mvmd  DMSDN,ar2                  ; get source address w/o autoincrement and
                                       ; put in ar2
      mvdk  *ar2fwd,DXR1reg            ; write first ”bit” to DXR
      mvdm  ar2,DMSDN                  ; write decremented address to DMA
      orm   #TxDMACh,*(DMPREC)         ; enable DMA channel for Tx data.
      orm   #THRE,*(_UARTLSR)          ; signal that one space is still available in
                                       ; Tx buffer
SkipDMARestart:
      popm  imr                        ; restore state of IMR (turn Tx DMA int back on)
      ret

********************************************************************
*   Function: _UARTRxChar                                          *
*   Purpose:  Returns last character read by UART and              *
*             resets the Data Ready (DR) flag.                     *
*   Inputs:   none                                                 *
*   Outputs:  al = received character                              *
*             intm = 0                                             *
```

```
*   Modified: a                                                    *
*******************************************************************
_UARTRxChar:
      pshm  imr                     ; save the IMR state to restore Rx DMA int at end
      andm  #~RxDMAInt,*(imr)       ; disable the DMA Rx interrupt
      andm  #~DR,*(_UARTLSR)        ; once read, reset the flag
      ld    *(rxchar),a             ; return the character in al
      popm  imr                     ; restore state of IMR (turn Rx DMA int back on)
      ret

*******************************************************************
*   Function: _UARTInit                                            *
*   Purpose:  Initializes variables as well as the                 *
*             McBSP and DMA registers. When this function          *
*             returns, the DMA and McBSP are initialized           *
*             but are not enabled.                                 *
*   Inputs:   none                                                 *
*   Outputs:  none                                                 *
*   Modified: ar2,a,brc,rea,rsa                                    *
*******************************************************************
_UARTInit:
      st    #MASK1011b,*(mask1011b)   ; init mask values
      st    #MASK0100b,*(mask0100b)
      st    #DECODER_MASK,*(decodeMask)
      st    #1,*(one)               ; init 1
      st    #0,*(_UARTLSR)          ; clear the status register
      stm   #SPCR1,SPSA             ; write the SPCR1 sub-address
      andm  #~RRST,*(McBSPDataReg)    ; write the SPCR1 register value to put rx
                                    ; in reset
      stm   #SPCR2,SPSA             ; write the SPCR2 sub-address
      andm  #~(FRST|GRST|XRST),*(McBSPDataReg)   ; write the SPCR2 register value to
                                               ; put tx in reset
      stm   #SPSA,ar2               ; point ar2 to McBSP subaddress register
      ldx   #McBSPInitTable,16,a      ; get the program address of the McBSP init table
      or    #McBSPInitTable,a         ; both high and low words
      stm   #(EndMcBSPInitTable-McBSPInitTable)/2-1,brc
      rptb  McBSPloop-1             ;
      reada *ar2+                   ; set the subaddress
      add   #1,a                    ; increment the table pointer
      reada *ar2-                   ; write the value
      add   #1,a                    ; increment the table pointer
McBSPloop:
      rpt    #RESET_LATENCY–1
        nop                         ; wait for McBSP to sync internally
      andm  #~(TxDMACh|RxDMACh),*(DMPREC)   ; disable DMA channels for Tx and Rx data.
      orm   #(INTOSEL<<6),*(DMPREC)   ; set multiplexed interrupt choices
      andm  #~(TxDMACh<<8|RxDMACh<<8),*(DMPREC); set DMA channel Priorities low (=0)
      ldx   #DMAInitTable,16,a        ; get the program address of the DMA init table
      or    #DMAInitTable,a         ; both high and low words
      stm   #DMSA,ar2               ; point ar2 to the DMA Rx channel subaddress
                                    ; register
      reada *ar2+                   ; store the first subaddress
      add   #1,a                    ; increment the table pointer
      rpt   #(EndDMARxInitTable-DMARxInitTable)-1
        reada *ar2                  ; write the values, autoincrementing
      add   #5,a                    ; update the table address
```

```
      stm   #DMSA,ar2                  ; point ar2 to the DMA Tx channel subaddress
                                       ; register
      reada *ar2+                      ; store the first subaddress
      add   #1,a                       ; increment the table pointer
      rpt   #(EndDMATxInitTable-DMATxInitTable)-1
        reada *ar2                     ; write the values, autoincrementing
      ret

********************************************************************
*   Function: _UARTStart                                           *
*   Purpose:  Enables UART for reception by enabling the           *
*             Rx and TX DMA channels and taking the                *
*             receiver of McBSP out of reset. The                  *
*             DMA is reinitialized in this routine to              *
*             ensure correct alignment of DMA pointers             *
*             in case DMA halted in mid-word last time.            *
*             Note that this routine will globally enable          *
*             all unmasked interrupts.                             *
*   Inputs:   a:  -1 = start Rx only                               *
*                  0 = start Rx and Tx                             *
*                  1 = start Tx only                               *
*   Outputs:  none                                                 *
*   Modified: imr,ifr,intm(st1),SPCR1,SPCR2,DMPREC                 *
********************************************************************
_UARTStart:
      stm   #SPCR2,SPSA                ; write the SPCR2 sub-address
      orm   #GRST,*(McBSPDataReg)      ; enable clk generator (if not already enabled)
      rpt   #RESET_LATENCY-1
        nop                            ; wait for 2 bit clocks
      bc    TxStartUp,agt              ; if input denotes Tx only startup, branch
RxStartUp:
      andm  #~(BI|FE|PE|OE|DR),*(_UARTLSR)   ; init the status reg bits for Rx to 0
      st    #0,*(rxbufhalf)            ; initialize the half of Rx buffer to get
                                       ; bits from
      stm   #(RxDMAptr+1),DMSA         ; set subaddress to Rx DMA Destination register.
      stm   #Rx1stStart,DMSDN          ; DMDST: Destination is Receive raw data buffer
      stm   #RxDMAInt,ifr              ; clear all pending Rx interrupts
      orm   #RxDMAInt,*(imr)           ; enable the DMA Rx interrupt
      orm   #RxDMACh,*(DMPREC)         ; enable DMA channel for Rx data.
      stm   #SPCR1,SPSA                ; write the SPCR1 sub-address
      orm   #RRST,*(McBSPDataReg)      ; write the SPCR1 register value to enable rx
      bc    SkipTxStartUp,alt          ; if input denotes Rx only startup, branch
TxStartUp:
      st    #0,*(txbufhalf)            ; initialize the half of Tx buffer to put bits in
      orm   #THRE,*(_UARTLSR)          ; init the status reg bits for Tx
                                       ; (available for tx)
      st    #0,*(numTxPkts)            ; init number of pkts in tx buffer to 0
      stm   #TxDMAptr,DMSA             ; set subaddress to Tx DMA Source register.
      stm   #Tx1stStart,DMSDN          ; DMSRC: Source is Transmit raw data buffer
      stm   #TxDMAInt,ifr              ; clear all pending Tx interrupts
      orm   #TxDMAInt,*(imr)           ; enable the DMA Tx interrupt
      stm   #SPCR2,SPSA                ; write the SPCR2 sub-address
      orm   #XRST,*(McBSPDataReg)      ; write the SPCR2 register value to enable tx
SkipTxStartUp:
      rpt   #RESET_LATENCY-1
        nop                            ; wait for McBSP to come out of reset
```

```
      rsbx  intm                       ; enable maskable interrupts
      ret

*******************************************************************
*   Function: _UARTStop                                           *
*   Purpose:  Disables UART for reception by disabling the        *
*             Rx and TX DMA channels and putting the              *
*             receiver and transmitter on McBSP in reset.         *
*             Waits until all data from Tx buffer                 *
*             has been transmitted before it halts Tx UART.       *
*   Inputs:   a:  -1 = stop Rx only                               *
*                  0 = stop Rx and Tx                             *
*                  1 = stop Tx only                               *
*   Outputs:  none                                                *
*   Modified: ar2,imr,st0(tc),SPCR1,SPCR2,DMPREC                  *
*******************************************************************
_UARTStop:
      bcd   ShutDownTx,agt            ; if input denotes Tx only shutdown, branch
        stm #SPSA,ar2               ; point to McBSP subaddress register
ShutDownRx:
      andm  #~RxDMAInt,*(imr)         ; disable the DMA Rx interrupt
      andm  #~RxDMACh,*(DMPREC)       ; disable DMA channel for Rx data.
      st    #SPCR1,*ar2+              ; write the SPCR1 sub-address and point to access
                                      ; register
      andm  #~RRST,*ar2-              ; write the SPCR1 register value to disable rx and
                                      ; move pointer back
      andm  #~DR,*(_UARTLSR)          ; clear data ready flag so no more data received
      bc    SkipShutDownTx,alt        ; if input denotes Rx only shutdown, branch
      st    #SPCR2,*ar2+              ; write the SPCR2 sub-address and point to access
                                      ; register
      andm  #~GRST,*ar2-              ; disable clk generator when both rx and tx
                                      ; shutdown
ShutDownTx:
      cmpm  *(numTxPkts),#0           ;
      bc    ShutDownTx,ntc            ; wait until all pkts have been sent
      st    #SPCR2,*ar2+              ; write the SPCR2 subregister offset and point to
                                      ; access register
waitDone:
      bitf  *ar2,#XRDY                ; check if XRDY==1
      bc    waitDone,ntc              ; wait until serial port has clocked out any bits
                                      ; in XSR
      mar   *ar2-                     ; point to subaddress register
      andm  #~TxDMAInt,*(imr)         ; disable the DMA Tx interrupt
      andm  #~TxDMACh,*(DMPREC)       ; disable DMA channel for Tx data.
      st    #SPCR2,*ar2+              ; write the SPCR2 sub-address and point to access
                                      ; register
      andm  #~XRST,*ar2-              ; write the SPCR2 register value to disable tx and
                                      ; move pointer back
      andm  #~THRE,*(_UARTLSR)        ; clear THRE flag so no more data sent
SkipShutDownTx:
      rpt   #RESET_LATENCY-1
        nop                           ; wait for McBSP to go into reset
      ret

*******************************************************************
*   Function: _UARTSetBaudRate                                    *
```

```
*   Purpose:  Sets new baud rate for UART. The UART MUST          *
*             be stopped before calling this routine. Use         *
*             _UARTStop to halt the UART (both tx and rx).        *
*   Inputs:   a = new baud divisor                                *
*   Outputs:  none                                                *
*   Modified: a, SRGR1                                            *
*******************************************************************
_UARTSetBaudRate:
      sub   #1,a                     ; reduce divisor by 1 for correct storage to McBSP
                                     ; register
      stm   #SRGR1,SPSA              ; write the SRGR1 sub-address to address register
      stlm  a,McBSPDataReg           ; write the register value to data register
      ret

*******************************************************************
*   Function: _UARTSetBreak                                       *
*   Purpose:  Sends a packet of all 0, including what             *
*             would normally be the stop & parity bits.           *
*             Must call with input of 0 to end the                *
*             break before can transmit another char.             *
*             Only call when THRE=1.                              *
*   Inputs:   a != 0 - send break                                 *
*             a  = 0 - end break                                  *
*   Outputs:  none                                                *
*   Modified: a,ar2,st0(tc)                                       *
*******************************************************************
_UARTSetBreak:
      pshm  imr                      ; save the IMR state to restore Tx DMA int at end
      andm  #~TxDMAInt,*(imr)        ; disable the DMA Tx interrupt (so DMA ISR doesn't
                                     ; change state here)
 .if DMA_ABU_FIX   ; 5402 workaround (can't start DMA ABU in 2nd half of buffer)
      bitf  *(txbufhalf),#1          ; check if in 2nd half of buffer (txbufhalf=1)
      bc    NoAdjust2,ntc            ; if not, nothing to worry about, so skip out
      cmpm  *(numTxPkts),#0          ; check if DMA is disabled (no packets to
                                     ; transmit)
      bc    NoAdjust2,ntc            ; if DMA still on, nothing to worry about, so
                                     ; skip out
      xorm  #1,*(txbufhalf)          ; set txbufhalf to 0 to point to 1st half
      stm   #TxDMAptr,DMSA           ; set subaddress to Tx DMA Source register.
      stm   #Tx1stStart,DMSDN        ; Set Tx DMA pointer to start of 1st half
NoAdjust2:
 .endif                              ; end 5402 workaround
      bitf  *(txbufhalf),#1          ; check if in 2nd half of buffer
      xorm  #1,*(txbufhalf)          ; toggle the buffer half which is available
      stm   #Tx2ndStart,ar2          ; assume in 2nd half of buffer (point to start of
                                     ; this half)
      xc    2,ntc                    ; if not,
        stm #Tx1stStart,ar2          ; point ar2 to the start of first half of buffer
      xc    2,aneq                   ; if sending break (a!=0)
        ld  #0xffff,a                ; init a to all 1's
      cmpl  a                        ; a=a_ (if input is 0, send all 1's, else
                                     ; send all 0's)
      rpt   #TxPKTBITS-1             ; create packet of all 0 or all 1
        stl a,*ar2fwd
      b     startTx                  ; branch into TxChar routine to start the
                                     ; transmission
```

```
********* PRIVATE ROUTINES ***********
 .if (PARITY==EVEN)|(PARITY==ODD)
*******************************************************************
*   Function: ParityCalc                                          *
*   Purpose:  Determines the parity of an input word              *
*             using a successive approximation scheme.            *
*             The number of iterations is determined              *
*             at assembly time by the number of bits              *
*             in each character.                                  *
*   Inputs:   al = character to determine parity of.              *
*   Outputs:  tc = 0 - parity is even                             *
*             tc = 1 - parity is odd                              *
*   Modified: ag,ah,st0(tc)                                       *
*******************************************************************
ParityCalc:
      pshm  al                        ; save character
   .if DATABITS+PARITYBITS>16         ;
      xor   a,16,a                    ;
   .endif                             ;
   .if DATABITS+PARITYBITS>8          ;
      xor   a,8,a                     ;
   .endif                             ;
   .if DATABITS+PARITYBITS>4          ;
      xor   a,4,a                     ;
   .endif                             ;
   .if DATABITS+PARITYBITS>2          ;
      xor   a,2,a                     ;
   .endif                             ;
   .if DATABITS+PARITYBITS>1          ;
      xor   a,1,a                     ;
   .endif                             ;
      bitf  *(al),#PARITYCHECK        ; test the calculated parity and return it in TC
      popm  al                        ; restore character
      ret
 .endif
 .if PARITY!=NO
*******************************************************************
*   Function: ParityCheck                                         *
*   Purpose:  Computes parity of input word and determines        *
*             if it is valid, according to settings.              *
*             The parity setting is made at assembly time.        *
*             The input is assumed to only contain data and       *
*             parity bits. No start or stop bits should           *
*             be in the word.                                     *
*   Inputs:   al = character to check parity of. Must             *
*             only contain data and parity bits.                  *
*             DP = UARTvars (uses data-page direct addressing)    *
*   Outputs:  al = character with parity bit stripped.            *
*             _UARTLSR - if parity is invalid, Parity             *
*             Error bit is set in _UARTLSR.                       *
*   Modified: ah,ag,_UARTLSR,st0(tc)                              *
*******************************************************************
ParityCheck:
   .if (PARITY==EVEN)|(PARITY==ODD)
      call  ParityCalc              ; find parity of received word (output tc==0-even,
                                    ; tc==1-odd)
```

```
   .else ; (PARITY==MARK)|(PARITY==SPACE)
      bitf  *(al),#(1<<DATABITS)      ; test the parity bit (tc==0-space, tc==1-mark)
   .endif
      and   #(1<<DATABITS)-1,a        ; mask out the parity bit (char was
                                      ; returned in AL)
   .if (PARITY==EVEN)|(PARITY==SPACE)
      xc    2,tc                      ; if parity is even/space skip error report
   .else ; (PARITY==ODD)|(PARITY==MARK)
      xc    2,ntc                     ; if parity is odd/mark skip error report
   .endif
        orm #PE,@_UARTLSR             ; set Parity Error flag in status register
      ret                             ;
  .endif ; if PARITY!=NO
********************************************************************
*   Table:   McBSPInitTable                                        *
*   Purpose: Contains all values to initialize the McBSP           *
*            used by the UART. Note that the McBSP which           *
*            will be used is defined at assembly time.             *
*            Specifically, the following major settings            *
*            will be made:                                         *
*                                                                  *
*   Receiver:                                                      *
*            - Dual phase frames                                   *
*              (1st phase = RxPKTBITS-RxHSTOPBITS words of         *
*               16 bits each)                                      *
*              (2nd phase = RxHSTOPBITS words of                   *
*               8 bits each)                                       *
*            - Enable frame ignore                                 *
*            - 1 bit delay between FSR and data                    *
*                                                                  *
*   Transmitter:                                                   *
*            - Dual phase frames                                   *
*              (1st phase = TxPKTBITS-TxHSTOPBITS words of         *
*               16 bits each)                                      *
*              (2nd phase = TxHSTOPBITS words of                   *
*               8 bits each)                                       *
*            - Enable frame ignore                                 *
*            - 0 bit delay between FSX and data                    *
*            - Generate CLKG and FSX using baud rate               *
*              divisor given in assembly-time conditions           *
********************************************************************
McBSPInitTable:
      .word  SRGR1             ; SRGR1 settings:
      .word  0000000000000000b | BAUDRATEDIV
;            00000000~~~~~~~~b     FWID: unused because FSGM=0
;            ~~~~~~~~xxxxxxxxb     CLKGDV: Sample rate generator clock
;                                          divider=(BAUDRATEDIV+1)
      .word  SRGR2            ; SRGR2 settings:
      .word  0010000000000000b
;            0~~~~~~~~~~~~~~~b     GSYNC:  sample rate gen clock (CLKG) is free running
;            ~0~~~~~~~~~~~~~~b     CLKSP:  unused
;            ~~1~~~~~~~~~~~~~b     CLKSM:  Sample rate gen clock derived from CPU clock
;            ~~~0~~~~~~~~~~~~b     FSGM:   Tx frame sync (FSX) due to DXR-to-XSR copy
;            ~~~~000000000000b     FPER:   unused because FSGM=0
      .word  PCR              ; PCR settings:
      .word  0000101100001100b
```

```
;             00~~~~~~~~~~~~~~b     reserved
;             ~~0~~~~~~~~~~~~~b     XIOEN:  DR,CLKS,DX,FSX,CLKX not GPIO
;             ~~~0~~~~~~~~~~~~b     RIOEN:  DR,CLKS,DX,FSR,CLKR not GPIO
;             ~~~~1~~~~~~~~~~~b     FSXM:   FSX determined by FSGM (in SRGR2)
;             ~~~~~0~~~~~~~~~~b     FSRM:   FSR generated by external device (is input)
;             ~~~~~~1~~~~~~~~~b     CLKXM:  CLKX is output driven by internal sample
;                                           rate generator
;             ~~~~~~~1~~~~~~~~b     CLKRM:  CLKR is output driven by internal sample
;                                           rate generator
;             ~~~~~~~~0~~~~~~~b     reserved
;             ~~~~~~~~~0~~~~~~b     CLKS_STAT: Read Only
;             ~~~~~~~~~~0~~~~~b     DX_STAT:Read Only
;             ~~~~~~~~~~~0~~~~b     DR_STAT:Read Only
;             ~~~~~~~~~~~~1~~~b     FSXP:   FSX is active low
;             ~~~~~~~~~~~~~1~~b     FSRP:   FSR is active low
;             ~~~~~~~~~~~~~~0~b     CLKXP:  Transmit data sampled on rising edge of CLKX
;             ~~~~~~~~~~~~~~~0b     CLKRP:  Receive data sampled on falling edge of CLKR
      .word  SPCR1             ; SPCR1 settings:
      .word  0000000000000000b
;             0~~~~~~~~~~~~~~~b     DLB:    Digital loopback mode is disabled
;             ~00~~~~~~~~~~~~~b     RJUST:  Right-justify and zero-fill MSbs in DRR(1/2)
;             ~~~00~~~~~~~~~~~b     CLKSTP: Clock Stop Mode disabled
;             ~~~~~000~~~~~~~~b     reserved
;             ~~~~~~~~0~~~~~~~b     DXENA:  DX enabler is off
;             ~~~~~~~~~0~~~~~~b     ABIS:   A-bis mode is disabled
;             ~~~~~~~~~~00~~~~b     RINTM:  RINT driven by RRDY
;             ~~~~~~~~~~~~0~~~b     RSYNCERR: Read Only
;             ~~~~~~~~~~~~~0~~b     RFULL:  Read Only
;             ~~~~~~~~~~~~~~0~b     RRDY:   Read Only
;             ~~~~~~~~~~~~~~~0b     RRST_:  Receiver is disabled and in reset state
      .word  SPCR2             ; SPCR2 settings:
      .word  0000000100000000b
;             000000~~~~~~~~~~b     reserved
;             ~~~~~~0~~~~~~~~~b     FREE:   Free running mode is disabled
;             ~~~~~~~1~~~~~~~~b     SOFT:   Soft mode enabled
;             ~~~~~~~~0~~~~~~~b     FRST_:  Frame sync generator is reset
;             ~~~~~~~~~0~~~~~~b     GRST_:  Sample rate generator is pulled out of reset
;             ~~~~~~~~~~00~~~~b     XINTM:  XINT driven by XRDY
;             ~~~~~~~~~~~~0~~~b     XSYNCERR: Read Only
;             ~~~~~~~~~~~~~0~~b     XEMPTY: Read Only
;             ~~~~~~~~~~~~~~0~b     XRDY:   Read Only
;             ~~~~~~~~~~~~~~~0b     XRST_:  Transmitter is disabled and in reset state
      .word  RCR1             ; RCR1 settings:
      .word  0000000001000000b|(((RxPKTBITS-RxHSTOPBITS) -1) <<8)
;             0~~~~~~~~~~~~~~~b     reserved
;             ~xxxxxxx~~~~~~~~b     RFRLEN1:Receive frame length for phase 1 is
;                                           RxPKTBITS-RxHSTOPBITS words
;             ~~~~~~~~010~~~~~b     RWDLEN1:Receive word length for phase 1 is 16 bits
;             ~~~~~~~~~~~00000b     reserved
      .word  RCR2              ; RCR2 settings:
      .word  1000000000000101b|((RxHSTOPBITS -1) <<8)
;             1~~~~~~~~~~~~~~~b     RPHASE: dual phase receive frame
;             ~xxxxxxx~~~~~~~~b     RFRLEN2:Receive frame length for phase 2 is
;                                           RxHSTOPBITS
;             ~~~~~~~~000~~~~~b     RWDLEN2:Receive word length for phase 2 is 8 bits
;             ~~~~~~~~~~~00~~~b     RCOMPAND:no companding, data transfer starts with
```

```
;                                          MSb first
;            ~~~~~~~~~~~~~1~~b     RFIG:   ignore receive frame syncs after first onez
;            ~~~~~~~~~~~~~~01b     RDATDLY: 1-bit delay between FSR and data
      .word  XCR1              ; XCR1 settings:
      .word  0000000001000000b|((TxPKTBITS - TxHSTOPBITS-1) <<8)
;            0~~~~~~~~~~~~~~~b     reserved
;            ~xxxxxxx~~~~~~~~b     XFRLEN1: Transmit frame length for phase 1 is
;                                           TxPKTBITS - TxHSTOPBITS words
;            ~~~~~~~~010~~~~~b     XWDLEN1: Transmit word length for phase 1 is 16
;                       bits
;            ~~~~~~~~~~~00000b     reserved
      .word  XCR2              ; XCR2 settings:
      .word  1000000000000100b|((TxHSTOPBITS - 1) <<8)
;            1~~~~~~~~~~~~~~~b     XPHASE:  dual phase transmit frame
;            ~xxxxxxx~~~~~~~~b     XFRLEN2: Transmit frame length for phase 2 is
;                                           TxHSTOPBITS words
;            ~~~~~~~~000~~~~~b     XWDLEN2: Transmit word length for phase 2 is 8 bits
;            ~~~~~~~~~~~00~~~b     XCOMPAND: no companding, data transfer starts with
;                                            MSb first
;            ~~~~~~~~~~~~~1~~b     XFIG:    ignore transmit frame syncs after first one
;            ~~~~~~~~~~~~~~00b     XDATDLY: 0-bit delay between FSX and data
EndMcBSPInitTable:

********************************************************************
*   Table:   DMAInitTable                                          *
*   Purpose: Contains all values to initialize the DMA             *
*            channels used by the UART. Note that the              *
*            DMA channels which are used are defined at            *
*            assembly-time, as is the McBSP. Specifically,         *
*            the following major settings will be made:            *
*                                                                  *
*   Rx Channel:                                                    *
*            - ABU mode (buffer size is RxPKTBITS*2)               *
*            - Interrupts are at each 1/2 buffer point             *
*            - Source is McBSP DRR1 register                       *
*            - Destination is Rx raw data buffer                   *
*            - Synchronized to McBSP Receive Event (REVT)          *
*                                                                  *
*   Tx Channel:                                                    *
*            - ABU mode (buffer size is TxPKTBITS*2)               *
*            - Interrupts are at each 1/2 buffer point             *
*            - Source is Tx raw data buffer                        *
*            - Destination is McBSP DXR1 register                  *
*            - Synchronized to McBSP Transmit Event (XEVT)         *
*                                                                  *
********************************************************************
DMAInitTable:
      .word  RxDMAptr ;;;;;;;;;;;;;;;;;;;;;;;  DMA Rx Channel settings. Use
                                              ;autoincrement of subaddress after 1st
                                              ;word
DMARxInitTable:
      .word  DRR1reg                    ; DMSRC: Source is McBSP DRR1 register
      .word  Rx1stStart                 ; DMDST: Destination is Receive raw data buffer
      .word  2*RxPKTBITS                ; DMCTR: Element count (words to transfer) is
;                                         2*total bits per character
      .word  0000000000000000b|(REVT<<12); DMSFC4:
```

```
;            xxxx~~~~~~~~~~~~b               DSYN: DMA sync event is one of the McBSP
;                                                  Receive Events
;            ~~~~0~~~~~~~~~~~b               DBLW: Single-word mode (each element is 16
;                                                  bits)
;            ~~~~~000~~~~~~~~b               reserved
;            ~~~~~~~~00000000b               Frame Count: not relevant in ABU mode
      .word  0111000001000001b|(DMAptrMod<<2); DMMCR:
;            0~~~~~~~~~~~~~~~b               AUTOINIT: Auto-initialization is disabled
;            ~1~~~~~~~~~~~~~~b               DINM: Interrupts generated based on IMOD
;                                                  bit
;            ~~1~~~~~~~~~~~~~b               IMOD: Interrupt at half buffer full and
;                                                  buffer full
;            ~~~1~~~~~~~~~~~~b               CTMOD: ABU Mode
;            ~~~~0~~~~~~~~~~~b               reserved
;            ~~~~~000~~~~~~~~b               SIND:  Source Address not modified
;            ~~~~~~~~01~~~~~~b               DMS:   Source Address in data space
;            ~~~~~~~~~~0~~~~~b               reserved
;            ~~~~~~~~~~~xxx~~b               DIND:  Destination Address post
;                                                  (dec/inc)remented
;            ~~~~~~~~~~~~~~01b               DMD:   Destination Address in data space
EndDMARxInitTable:
DMATxInitTable
    .word  TxDMAptr ;;;;;;;;;;;;;;;;;;;;;;;; DMA Tx channel settings. Use autoincrement
                                            ;of subaddress after 1st word
    .word  Tx1stStart             ; DMSRC: Source is Transmit raw data buffer
    .word  DXR1reg                ; DMDST: Destination is McBSP DXR1 register
    .word  2*TxPKTBITS            ; DMCTR: Element count (words to transfer)
;                                          is 2*total bits per character
    .word  0000000000000000b|(XEVT<<12); DMSFC5:
;          xxxx~~~~~~~~~~~~b                 DSYN: DMA sync event is one of the McBSP
;                                                  Transmit Events
;          ~~~~0~~~~~~~~~~~b                 DBLW: Single-word mode (each element is 16
;                                                  bits)
;          ~~~~~000~~~~~~~~b                 reserved
;          ~~~~~~~~00000000b                 Frame Count: not relevant in ABU mode
    .word  0111000001000001b|(DMAptrMod<<8); DMMCR:
;          0~~~~~~~~~~~~~~~b                 AUTOINIT: Auto-initialization is disabled
;          ~1~~~~~~~~~~~~~~b                 DINM: Interrupts generated based on IMOD
;                                                   bit
;          ~~1~~~~~~~~~~~~~b                 IMOD: Interrupt at half buffer full and
;                                                  buffer full
;          ~~~1~~~~~~~~~~~~b                 CTMOD: ABU Mode
;          ~~~~0~~~~~~~~~~~b                 reserved
;          ~~~~~xxx~~~~~~~~b                 SIND:  Source Address post
                                                    (dec/inc)remented
;          ~~~~~~~~01~~~~~~b                 DMS:   Source Address in data space
;          ~~~~~~~~~~0~~~~~b                 reserved
;          ~~~~~~~~~~~000~~b                 DIND:  Destination Address not modified
;          ~~~~~~~~~~~~~~01b                 DMD:   Destination Address in data space

EndDMATxInitTable:
```

Appendix C Include File (UARTSetup.inc)

```
********************************************************************
* Filename:  UARTsetup.inc                                         *
* Function:  Software UART parameter selection                     *
* Author:    Robert J. DeNardo                                     *
*            Texas Instruments, Inc                                *
* Revision History:                                                *
* 11/12/99  Original Code                     Robert DeNardo       *
* 12/21/99  Increased data bits to 1-15.      Robert DeNardo       *
* 01/07/00  Added DMA_PTR_MOD & DMA_ABU_FIX.  Robert DeNardo       *
********************************************************************
****** Conditional Choices ********
MCBSP_CHOICE    .set  0   ; # of McBSP to use for UART (0-2, depending on 54xx choice)
DMA_RX_CHOICE   .set  4   ; # of DMA channel to use for Rx (0-5)
DMA_TX_CHOICE   .set  5   ; # of DMA channel to use for Tx (0-5), different than above
INTOSEL         .set  0   ; Selection of DMA/McBSP multiplexed interrupts (0-3).
                          ; The choices are device dependent and specified in DMA
                          ; User’s Guide.
PARITY          .set  1   ; parity checked and generated (0=NO, 1=EVEN, 2=ODD, 3=MARK,
4=SPACE)
HSTOPBITS       .set  2   ; number of 1/2 stop bits (2,3 or 4) for Tx
DATABITS        .set  8   ; number of data bits (1-14 with parity, or 1-15 w/o parity)
BAUDRATEDIV     .set  244 ; Enter baud rate divisor (approx DSPCLK/(16*baudrate)
                          ; See app note for calculation.
INTERRUPT_BASED .set  0   ; 0=only use polling to check status of UART
                          ; 1=run ISR’s for the interrupt events on UART
DMA_PTR_MOD     .set  0   ; Direction for DMA ptr modification
                          ; (0=post-decrement, 1=post-increment)
                          ; 5410 can only use 0=post-decrement
DMA_ABU_FIX     .set  1   ; Adds workaround for ABU difference in 5402
                          ; (0=no fix, 1=add fix)
                          ; Difference occurs when DMA ABU started with DMA ptr in 2nd
                          ; half of buffer
***********************************

***********************************************
* This is the available public variable       *
* which can be used in your code              *
***********************************************
* _UARTLSR - Line Status Register bit definitions
* This register is used to monitor status of the line, including
* status for available receive data or status for transmit, as well
* as error bits.
DR      .set   1<<0    ; Data Ready: character is ready
OE      .set   1<<1    ; Overrun Error: before last char read, it was overwritten
PE      .set   1<<2    ; Parity Error: parity of received char doesn’t match setting of
                       ; UART
FE      .set   1<<3    ; Framing Error: received character has invalid stop bit
BI      .set   1<<4    ; Break Indicator: received data input was 0 longer than packet
                       ; length
THRE    .set   1<<5    ; Transmit Holding Register Empty: another char can be
                       ; transmitted
```

Appendix D Command File (uart.cmd)

```
/******************************************************************
* Filename:  uart.cmd                                             *
* Function:  Software UART example command file using 5410 EVM    *
* Author:    Robert J. DeNardo                                    *
*            Texas Instruments, Inc                               *
* Revision History:                                               *
* 11/12/99  Original Code                      Robert DeNardo     *
******************************************************************/
MEMORY
{
  PAGE 0: DARAM1: origin = 00080h  length = 00780h  /* Overlay memory */
          DARAM2: origin = 00800h  length = 00800h  /* Overlay memory */
          DARAM3: origin = 01000h  length = 00800h  /* Overlay memory */
          DARAM4: origin = 01800h  length = 00800h  /* Overlay memory */
          SARAM1: origin = 02000h  length = 02000h  /* Overlay memory */
          SARAM2: origin = 04000h  length = 02000h  /* Overlay memory */
          SARAM3: origin = 06000h  length = 02000h  /* Overlay memory */
  PAGE 1: SPRAM:  origin = 00060h  length = 00020h  /* Scratch Pad  */
          DARAM1: origin = 00080h  length = 00780h
          DARAM2: origin = 00800h  length = 00800h
          DARAM3: origin = 01000h  length = 00800h
          DARAM4: origin = 01800h  length = 00800h
          SARAM1: origin = 02000h  length = 02000h
          SARAM2: origin = 04000h  length = 02000h
          SARAM3: origin = 06000h  length = 02000h
}
SECTIONS
{
  .text        :> SARAM1 PAGE 0
  .cinit       :> SARAM1 PAGE 0
  .switch      :> SARAM1 PAGE 0
  vecs         :> SARAM2 PAGE 0     /* Vector table */
  .stack       :> DARAM1 PAGE 1
  .data        :> DARAM2 PAGE 1
  .bss         :> DARAM2 PAGE 1
  uart         :> SARAM1 PAGE 0
  UARTTxBuffer :> DARAM3 align(32) PAGE 1
  UARTRxBuffer :> DARAM3 align(32) PAGE 1
  UART_vars    :> DARAM3 PAGE 1
  ExampleRxBuf :> DARAM2 align(128) PAGE 1
  ExampleTxBuf :> DARAM2 align(128) PAGE 1
}
```

Appendix E Example Use C Code (ExampleC.c)

```
/*******************************************************************
* Filename:  ExampleC.c                                            *
* Function:  Software UART Access code in C                        *
* Author:    Robert J. DeNardo                                     *
*            Texas Instruments, Inc                                *
* Revision History:                                                *
* 11/12/99  Original Code                      Robert DeNardo      *
* 12/21/99  Modified example loop structure.   Robert DeNardo      *
* 09/05/00  Fixed address of PMST register.    Robert DeNardo      *
*           Setup for 5402DSK.                 Robert DeNardo      *
*                                                                  *
* This code performs basic accesses to the software UART           *
* using C code.                                                    *
*                                                                  *
* Defining the INTERRUPTBASED keyword below will setup the         *
* UART to receive a block of 10 chars and then transmit that       *
* block of 10 chars using the UART interrupts routines.            *
* If this is used, make sure the INTERRUPT_BASED definition        *
* in UARTsetup.inc is set to 1.                                    *
*                                                                  *
* Not defining the INTERRUPTBASED keyword will setup the           *
* UART to use a polling method to receive and transmit             *
* one char at a time. If this is used, make sure the               *
* INTERRUPT_BASED definition in UARTsetup.inc is set to 0.         *
* This code is set to run on 5402DSK. Make sure to set CPLD        *
* to use McBSP0 from daughterboard (0x4@io = 0xFF03).              *
*******************************************************************/
#if 0
#define INTERRUPTBASED
#endif
extern volatile struct StatusStruct{   /* structure to model the UARTLSR status bits */
   unsigned int reserved:10;
   unsigned int THRE:1;      /* Transmit Holding Register Empty */
   unsigned int BI:1;        /* Break Indicator */
   unsigned int FE:1;        /* Frame Error */
   unsigned int PE:1;        /* Parity Error */
   unsigned int OE:1;        /* Overrun Error */
   unsigned int DR:1;        /* Data Ready */
}UARTLSR;

#define TXNUM    10
#define BUFSIZE  20
volatile int RxCharCnt;
int *RxHeadPtr;
int *RxTailPtr;
int *TxHeadPtr;
int *TxTailPtr;
int RxCharBuf[50]={0};      /* create a received character buffer */
int TxCharBuf[50]={0};      /* create a transmit character buffer */
void InitPLL()              /* sets up PLL for a 3.75 multiplier */
{
   volatile unsigned int *CLKMD=(volatile unsigned int*)0x58;  /*set addr of CLKMD
reg*/
   *CLKMD=0;               /* set to DIV mode */
```

```
   while((*CLKMD&1)==1);  /* wait until PLLstatus reflects DIV mode */
   *CLKMD=0xfffb;         /* set to mult of 3.75 and max cnt cycles, turn on PLL */
   while((*CLKMD&1)==0);  /* wait until PLLstatus reflects in PLL mode */
}
#ifdef INTERRUPTBASED
interrupt void UARTRBFint()       /* called from the UART code when */
{                                 /* a character is received */
   *RxHeadPtr++=UARTRxChar();     /* call UART routine to receive char */
                                  /* & move character to receive buffer */
   if(RxHeadPtr>=(RxCharBuf+BUFSIZE))
      RxHeadPtr=RxCharBuf;        /* keep pointer in circular buffer */
   RxCharCnt++;                   /* increment the character count */
}

interrupt void UARTTBEint()       /* called from the UART code when */
{                                 /* another character can be transmitted */
   if(TxTailPtr!=TxHeadPtr)       /* if a character to transmit is in buffer, */
   {
      if((*TxTailPtr==0)||(*TxTailPtr==1))   /* send break if char=1 or 0 */
      {
         UARTSetBreak(*TxTailPtr++);    /* 1 sends break, 0 sends end of break */
         if(TxTailPtr>=(TxCharBuf+BUFSIZE))
            TxTailPtr=TxCharBuf;  /* keep pointer in circular buffer */
      }
      else                        /* send character */
      {
         UARTTxChar(*TxTailPtr++);/* call UART routine to transmit the character */
         if(TxTailPtr>=(TxCharBuf+BUFSIZE))
            TxTailPtr=TxCharBuf;  /* keep pointer in circular buffer */
      }
   }
}
interrupt void UARTLSIint()      /* called from UART code when a char */
{                                /* is received and an error is detected */
   UARTLSR.OE=0;                 /* clear error flags */
   UARTLSR.PE=0;
   UARTLSR.BI=0;
   UARTLSR.FE=0;
   *TxHeadPtr++=1;               /*put break into tx buffer */
   if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
      TxHeadPtr=TxCharBuf;       /* keep pointer in circular buffer */
   *TxHeadPtr++=0;               /*put end of break into buffer */
   if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
      TxHeadPtr=TxCharBuf;       /* keep pointer in circular buffer */
}
main()
{
   int i;
   volatile unsigned int *PMST=(volatile unsigned int*)0x1D;  /* define the PMST reg */

   *PMST=0x4020;            /* IPTR=0x4000, OVLY=1, DROM=0, MP/MC=0 */
   InitPLL();       /* initialize the DSP Clock to 75MHz (with a 20MHz crystal) */
   UARTInit();              /* initialize the McBSP and DMA for UART */
   RxCharCnt=0;             /* init received char count to 0 */
   RxHeadPtr=RxCharBuf;     /* init rx head pointer to start of buffer */
   RxTailPtr=RxCharBuf;     /* init rx tail pointer to start of buffer */
```

```
   TxHeadPtr=TxCharBuf;    /* init tx head pointer to start of buffer */
   TxTailPtr=TxCharBuf;    /* init tx tail pointer to start of buffer */

   UARTStart(0);       /* start UART Rx and Tx(begins receiving) */
   for(;;)             /* infinite loop */
   {
      while(RxCharCnt<TXNUM);         /* wait until TXNUM chars received */
      for(i=0;i<TXNUM;i++)
      {
         *TxHeadPtr++=*RxTailPtr++;   /* copy the TXNUM chars to transmit buffer */
         if(RxTailPtr>=(RxCharBuf+BUFSIZE))
            RxTailPtr=RxCharBuf;      /* keep pointer in circular buffer */
         if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
            TxHeadPtr=TxCharBuf;      /* keep pointer in circular buffer */
      }
      RxCharCnt-=TXNUM;               /* reduce number of received chars */
      UARTTxChar(*TxTailPtr++);       /* kickstart: call UART routine to tx the char */
      if(TxTailPtr>=(TxCharBuf+BUFSIZE))
         TxTailPtr=TxCharBuf;         /* keep pointer in circular buffer */
      UARTTxChar(*TxTailPtr++);     /* (put 2 chars in buf to get consecutive xmits) */
                                   /* It will xmit these TXNUM chars and then stop */
      if(TxTailPtr>=(TxCharBuf+BUFSIZE))
         TxTailPtr=TxCharBuf;         /* keep pointer in circular buffer */
   }
}
#else    /*polling method*/
main()
{
   volatile unsigned int *PMST=(volatile unsigned int*)0x1D;  /* define the PMST reg */
   int RxCharBuf[BUFSIZE]={0};      /* create a received character buffer */
   int TxCharBuf[BUFSIZE]={0};      /* create a transmit character buffer */

   *PMST=0x4020;                    /* IPTR=0x4000, OVLY=1, DROM=0, MP/MC=0 */
   InitPLL();       /* initialize the DSP Clock to 75MHz (with a 12MHz crystal) */
   UARTInit();      /* initialize the McBSP and DMA for UART */
   RxCharCnt=0;                     /* init received char count to 0 */
   RxHeadPtr=RxCharBuf;             /* init rx head pointer to start of buffer */
   RxTailPtr=RxCharBuf;             /* init rx tail pointer to start of buffer */
   TxHeadPtr=TxCharBuf;             /* init tx head pointer to start of buffer */
   TxTailPtr=TxCharBuf;             /* init tx tail pointer to start of buffer */

   UARTStart(0);                    /* start UART Rx and Tx(begins receiving) */
   for(;;)                          /* infinite loop */
   {
      if(UARTLSR.DR==1)             /* if new char is available */
      {
         *RxHeadPtr++=UARTRxChar(); /* get the new char */
            if(RxHeadPtr>=(RxCharBuf+BUFSIZE))
         RxHeadPtr=RxCharBuf;       /* keep pointer in circular buffer */
      }
      if(UARTLSR.THRE==1)           /* if able to transmit a char */
      {
         if(TxTailPtr!=TxHeadPtr)   /* if a character to transmit is in buffer, */
         {
            UARTTxChar(*TxTailPtr++);  /* call UART routine to xmit the character */
            if(TxTailPtr>=(TxCharBuf+BUFSIZE))
```

```
            TxTailPtr=TxCharBuf;        /* keep pointer in circular buffer */
         }
      }
      if((UARTLSR.BI|UARTLSR.FE|UARTLSR.PE|UARTLSR.OE)==1)   /* if any errors */
      {
         UARTLSR.OE=0;                       /* clear error flags.*/
         UARTLSR.PE=0;
         UARTLSR.BI=0;
         UARTLSR.FE=0;
         *TxHeadPtr++=1;                     /*put break into tx buffer */
         if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
            TxHeadPtr=TxCharBuf;             /* keep pointer in circular buffer */
         *TxHeadPtr++=0;                     /*put end of break into buffer */
         if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
            TxHeadPtr=TxCharBuf;             /* keep pointer in circular buffer */
      }
      if(RxTailPtr!=RxHeadPtr)               /* if new character is in rx buffer, */
      {
         *TxHeadPtr++=*RxTailPtr++;          /* put the char in the tx buffer */
         if(RxTailPtr>=(RxCharBuf+BUFSIZE))
            RxTailPtr=RxCharBuf;             /* keep pointer in circular buffer */
         if(TxHeadPtr>=(TxCharBuf+BUFSIZE))
            TxHeadPtr=TxCharBuf;             /* keep pointer in circular buffer */
      }
   }
}
#endif
```

Appendix F Example Use ASM Code (ExampleASM.asm)

```
********************************************************************
* Filename:  ExampleASM.asm                                        *
* Function:  Demonstrates usage of the SW UART                     *
* Author:    Robert J. DeNardo                                     *
*            Texas Instruments, Inc.                               *
* Revision History:                                                *
* 11/12/99   Original Code                      Robert DeNardo     *
* 12/21/99   Changed example loop structure.   Robert DeNardo     *
* 09/05/00   Setup to run on 5402DSK.           Robert DeNardo     *
*                                                                  *
* This code is set to run on 5402DSK. Make sure to set CPLD        *
* to use McBSP0 from daughterboard (0x4@io = 0xFF03).              *
********************************************************************
   .mmregs
   .include ”UARTsetup.inc”
********** routines called by this file ***********
   .ref   _UARTTxChar
   .ref   _UARTRxChar
   .ref   _UARTStart
   .ref   _UARTStop
   .ref   _UARTInit
   .ref   _UARTSetBaudRate
   .ref   _UARTSetBreak
   .ref   _UARTDMATxISR
   .ref   _UARTDMARxISR
********** routines defined in this file ***********
   .def   _main
   .def   _c_int00   ; use this label so we can share vectors.asm with C example
 .if INTERRUPT_BASED
   .def   _UARTLSIint
   .def   _UARTRBFint
   .def   _UARTTBEint
 .endif
*********** public variables referenced ************
         .ref   _UARTLSR

*************** variable definitions ***************
   .bss   RxHeadPtr,1
   .bss   RxTailPtr,1
   .bss   RxCharCnt,1
   .bss   TxHeadPtr,1
   .bss   TxTailPtr,1
****************** equates ***********************
DMPREC      .set   54h   ; DMA channel PRiority and Enable Control register
TXNUM       .set   10
STACKSIZE   .set   100
RxCharBuf   .usect   ”ExampleRxBuf”,TXNUM*2   ; address of the circular character
                                             ; receive buffer
TxCharBuf   .usect   ”ExampleTxBuf”,TXNUM*2   ; address of the circular character
                                             ; transmit buffer
stack       .usect  ”.stack”,STACKSIZE

       .text
 .if INTERRUPT_BASED
```

```
******************************************************************
*   Function:  _UARTLSIint                                       *
*   Called By: _UARTDMARxISR                                     *
*   Purpose:  Run when a Line Status Interrupt event occurs.     *
*             This is when a Parity Error, Overrun Error,        *
*             Break Interrupt Error, or Framing Error occurs     *
*             The registers, ar2, a, b, st0, st1, brc, rsa,      *
*             rea are saved and restored by _UARTDMARXISR.       *
*             Any other registers used MUST be saved and         *
*             restored by this routine.                          *
*   Inputs:   none                                               *
*   Outputs:  none                                               *
*   Modified: none                                               *
******************************************************************
_UARTLSIint:
   pshm   bk
   stm    #TXNUM*2,BK         ; set for circular addressing
   call   ErrRoutine
   popm   bk
   ret
******************************************************************
*   Function:  _UARTRBFint                                       *
*   Called By: _UARTDMARxISR                                     *
*   Purpose:  Run when a new char is received into rxchar.       *
*             The registers, ar2, a, b, st0, st1, brc, rsa,      *
*             rea are saved and restored by _UARTDMARXISR.       *
*             Any other registers used MUST be saved and         *
*             restored during this routine.                      *
*   Inputs:   CPL=0 (data page rel. direct addressing)           *
*   Outputs:  none                                               *
*   Modified: none                                               *
******************************************************************
_UARTRBFint:
   pshm   bk
   stm    #TXNUM*2,BK         ; set for circular addressing
   call   RxRoutine
   popm   bk
   ret

******************************************************************
*   Function:  _UARTTBEint                                       *
*   Called By: _UARTDMATxISR                                     *
*   Purpose:  Run when a char has just been transmitted          *
*             and the UART is ready to transmit another.         *
*             The st0 register is saved and restored by          *
*             _UARTDMATxISR. Any other registers used MUST       *
*             be saved and restored during this routine.         *
*   Inputs:   none                                               *
*   Outputs:  none                                               *
*   Modified: none                                               *
******************************************************************
_UARTTBEint:
   pshm   al
   pshm   ah
   pshm   ag
   pshm   st1
```

```
   pshm   ar2
   pshm   bk
   pshm   brc
   pshm   rea
   pshm   rsa
   stm    #TXNUM*2,BK         ; set for circular addressing
   call   TxRoutine
   popm   rsa
   popm   rea
   popm   brc
   popm   bk
   popm   ar2
   popm   st1
   popm   ag
   popm   ah
   popm   al
   ret
 .endif
********************************************************************
*   Function: _InitPLL                                             *
*   Purpose:  Sets the PLL for a 3.75 multiplier.                   *
*   Inputs:   none                                                 *
*   Outputs:  none                                                 *
*   Modified: clkmd,tc                                             *
********************************************************************
_InitPLL:
   stm   #0,CLKMD
waitDiv:
   bitf  *(CLKMD),#1       ; check the status bit to see if in DIV mode
   bc    waitDiv,tc        ; if not, keep looping
   stm   #0fffbh,CLKMD     ; set multiplier to 15/4=3.75 and set count to 0xff
waitPLL:
   bitf  *(CLKMD),#1       ; check the status bit to see if in PLL mode
   bc    waitPLL,ntc       ; if not, keep looping
   ret

********************************************************************
*   Function: _main                                                *
*   Purpose:  Example routine to use the UART.                     *
*             This demonstrates the calls to the                   *
*             initialization routines and how to read and          *
*             write characters with the UART.                      *
*   Inputs:   none                                                 *
*   Outputs:  none                                                 *
*   Modified: NA                                                   *
********************************************************************
_c_int00:
_main:
   stm   #0100000000100000b,PMST
         ;010000000~~~~~~~b      IPTR:   Vector table resides at 04000h
         ;~~~~~~~~~0~~~~~~b      MP/MC_: On-chip ROM is enabled
         ;~~~~~~~~~~1~~~~~b      OVLY:   On-chip RAM mapped to prog & data space
         ;~~~~~~~~~~~0~~~~b      AVIS:   Address visibility mode off
         ;~~~~~~~~~~~~0~~~b      DROM:   On-chip ROM not mapped into data space
         ;~~~~~~~~~~~~~0~~b      CLKOFF: CLOCKOUT not disabled
         ;~~~~~~~~~~~~~~0~b      SMUL:   Saturate on Multiply is disabled
```

```
        ;~~~~~~~~~~~~~~~0b      SST:    Saturate on Store is disabled
   stm   #stack + STACKSIZE,sp   ; init the stack pointer to top of stack
   stm   #0,imr                  ; disable all interrupts
   stm   #0ffffh,ifr             ; clear all pending interrupts
   st    #0,*(DMPREC)            ; disable all DMA channels.
   call  _InitPLL                ; init the PLL
   call  _UARTInit               ; init the UART
   st    #0,*(RxCharCnt)         ; init received character count to 0
   st    #RxCharBuf,*(RxHeadPtr) ; init the head ptr into received character buffer
   st    #RxCharBuf,*(RxTailPtr) ; init the tail ptr into received character buffer
   st    #TxCharBuf,*(TxHeadPtr) ; init the head ptr into transmit character buffer
   st    #TxCharBuf,*(TxTailPtr) ; init the tail ptr into transmit character buffer
   stm   #TXNUM*2,BK             ; set for circular addressing
   ld    #0,a                    ; start the Rx and Tx UART channels
   call  _UARTStart              ;
   stm   #1,ar0                  ; use for copy below
 .if INTERRUPT_BASED ; example for interrupt based servicing of UART
echoLoop:
   ld   *(RxCharCnt),a
   sub   #TXNUM,a                ; see if TXNUM or greater characters received
   bc    echoLoop,alt            ; if not, loop
   mvdk  *(RxTailPtr),ar2        ; load the receive tail pointer
                                 ; (oldest rx char) to ar2
   mvdk  *(TxHeadPtr),ar3        ; load the transmit head pointer to ar3
   rpt   #TXNUM-1
     mvdd   *ar2+0%,*ar3+0%      ; copy TXNUM received chars to transmit buffer
   mvkd  ar2,*(RxTailPtr)        ; update the receive tail pointer
   mvkd  ar3,*(TxHeadPtr)        ; update the transmit head pointer
   addm  #-TXNUM,*(RxCharCnt)    ; decrement the received character count by TXNUM
   call  TxRoutine               ; need to ”kickstart” UART for transmit.
   call  TxRoutine               ; need to ”kickstart” UART for transmit
                                 ; (put 2 chars in buf to get consecutive
                                 ; transmits).
                                 ; It will transmit these TXNUM chars and then
                                 ; stop.
   b    echoLoop
 .else ; example for polling based servicing of UART
echoLoop:
   bitf  *(_UARTLSR),#DR              ; Check if new character received
   cc    RxRoutine,tc                 ; if so, call routine to process it
   bitf  *(_UARTLSR),#THRE            ; Check if new character can be sent
   cc    TxRoutine,tc                 ; if so, call routine to process it
   bitf  *(_UARTLSR),#(BI|FE|PE|OE)   ; Check if any errors
   cc    ErrRoutine,tc                ; if so, process the errors
   ld    *(RxTailPtr),a               ; check if any received data is available
   sub   *(RxHeadPtr),a
   bc    echoLoop,aeq                 ; if head=tail ptr for rx, skip copy
   mvdk  *(RxTailPtr),ar2
   mvdk  *(TxHeadPtr),ar3
   mvdd  *ar2+0%,*ar3+0%              ; move rx char to tx buffer
   mvkd  ar2,*(RxTailPtr)             ; update pointers
   mvkd  ar3,*(TxHeadPtr)
   b     echoLoop
 .endif
RxRoutine:
   call  _UARTRxChar                  ; returns char in al
```

```
   mvdk  *(RxHeadPtr),ar2              ; load the receive head pointer into ar2
   stl   a,*ar2+%                      ; store the newly decoded character into the
                                       ; received character buffer
   mvkd  ar2,*(RxHeadPtr)           ; update the head ptr
   addm  #1,*(RxCharCnt)            ; increment received character count
   ret
TxRoutine:
   ld   *(TxTailPtr),a              ; check if any transmit data is available to be
                                    ; sent
   sub   *(TxHeadPtr),a
   rc    aeq                        ; if transmit head=tail pointer, no data, so exit
   mvdk  *(TxTailPtr),ar2           ; otherwise load the transmit tail pointer
   ld   *ar2+%,a                    ; get the next character to transmit
   mvkd  ar2,*(TxTailPtr)           ; update the tail pointer
   sub   #2,a
   bcd   sendBreak,alt              ; if buffer has 0 or 1, send break
     add   #2,a
   call  _UARTTxChar                ; format the character for transmit.
   ret
sendBreak:
   call  _UARTSetBreak              ; sending a 1 sends break, 0 sends end of break
   ret

ErrRoutine:                          ; this error routine puts break in tx buffer
                                     ; and clears flags
   andm  #~(BI|FE|PE|OE),*(_UARTLSR); clear error flags
   mvdk  *(TxHeadPtr),ar2            ; get current tx head ptr to write new data
   st   #1,*ar2+%                    ; add a break to transmit buffer
   st   #0,*ar2+%                    ; add end of break to transmit buffer
   mvkd  ar2,*(TxHeadPtr)            ; update transmit head ptr
   ret
```

Appendix G Example Interrupt Vectors Table (vectors.asm)

```
*********************************************************************
* Filename:  vectors.asm                                            *
* Function:  Software UART example vector table                     *
* Author:    Robert J. DeNardo                                      *
*            Texas Instruments, Inc                                 *
* Revision History:                                                 *
* 11/12/99  Original Code                     Robert DeNardo        *
*********************************************************************
   .mmregs
   .sect   ”vecs”
   .ref   _c_int00
   .ref   _UARTDMARxISR
   .ref   _UARTDMATxISR
ResetVector:     ; Reset Vector
   b    _c_int00
   nop
   nop

   b    $        ; NMI Vector
   nop
   nop

   b    $        ; SWI 17
   nop
   nop

   b    $        ; SWI 18
   nop
   nop

   b    $        ; SWI 19
   nop
   nop

   b    $        ; SWI 20
   nop
   nop

   b    $        ; SWI 21
   nop
   nop

   b    $        ; SWI 22
   nop
   nop

   b    $        ; SWI 23
   nop
   nop

   b    $        ; SWI 24
   nop
   nop

   b    $        ; SWI 25
   nop
   nop
```

```
b    $        ; SWI 26
nop
nop

b    $        ; SWI 27
nop
nop

b    $        ; SWI 28
nop
nop

b    $        ; SWI 29
nop
nop

b    $        ; SWI 30
nop
nop

b    $        ; Ext Int 0
nop
nop

b    $        ; Ext Int 1
nop
nop

b    $        ; Ext Int 2
nop
nop

b   $         ; Timer Int
nop
nop

b   $         ; McBSP0 Rx Int
nop
nop

b    $        ; McBSP0 Tx Int
nop
nop

b    $        ; McBSP2 Rx Int
nop
nop
b    $        ; McBSP2 Tx Int
nop
nop
b    $        ; Ext Int 3
nop
nop

b    $        ; HPI Int
nop
nop
```

```
   b    $        ; McBSP1 Rx Int
   nop
   nop

   b    $        ; McBSP1 Tx Int
   nop
   nop

DMAC4Vector:     ; DMA Ch 4 Int
   b    _UARTDMARxISR
   nop
   nop

DMAC5Vector:     ; DMA Ch 5 Int
   b   _UARTDMATxISR
   nop
   nop

   b    $        ; Reserved
   nop
   nop

   b    $        ; Reserved
   nop
   nop
```

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 acknowledgment, 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.

Customers are responsible for their applications using TI components.

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.