SwiftLink-232 Application Notes (revised)

Introduction

The *SwiftLink-232* ACIA cartridge replaces the Commodore Kernal RS-232 routines with a hardware chip. The chip handles all the bit-level processing now done in software by the Commodore Kernal. The ACIA may be accessed by polling certain memory locations in the I/O block ($D000 - $DFFF) or through interrupts. The ACIA may be programmed to generate interrupts in the following situations:

1) when a byte of data is received
2) when a byte of data may be transmitted (i. e. the data register is empty)
3) both (1) and (2)
4) never

The sample code below sets up the ACIA to generate an interrupt each time a byte of data is received. For transmitting, two techniques are shown. The first technique consists of an interrupt handler which enables transmit interrupts when there are bytes ready to be sent from a transmit buffer. There is a separate routine given that manages the transmit buffer. In the second technique, which can be found at the very end of the sample code, neither a transmit buffer or transmit interrupts are used. Instead, bytes of data are sent to the ACIA directly as they are generated by the terminal program.

Note: The ACIA will always generate an interrupt when a change of state occurs on either the DCD or DSR line (unless the lines are not connected in the device's cable).

The 6551 ACIA was chosen for several reasons, including low cost and compatibility with other Commodore (MOS) integrated circuits. Commodore used the 6551 as a model for the Kemal software. Control, Command, and Status registers in the Kernal routines partially mimic their hardware counterparts in the ACIA.

Note: If you're using the Kernal software registers in your program, be sure to review the enclosed 6551 data sheet carefully. Several of the hardware register locations do not perform the same functions as their software counterparts. You may need to make a few changes in your program to accommodate the differences.

Buffers

Bytes received are placed in "circular" or "ring" buffers by the sample receive routine below, and also by the first sample transmit routine. To keep things similar to the Kernal RS-232 implementation, we've shown 256 byte buffers. You may want to use larger buffers for file transfers or to allow more screen processing time. Bypassing the Kemal routines frees many zero page locations, which could improve performance of pointers to larger buffers.

If your program already directly manipulates the Kernal RS-232 buffers, you'll find it very easy to adapt to the ACIA. If you use calls to the Kernal RS-232 file routines instead, you'll need to implement lower level code to get and store buffer data.

Briefly, each buffer has a "head" and "tail" pointer. The head points to the next byte to be removed from the buffer. The tail points to the next free location in which to store a byte. If the head and tail both point to the same location, the buffer is empty. If (tail + 1) = head, the buffer is full.

The interrupt handler described below will place received bytes at the tail of the receive buffer. Your program should monitor the buffer, either by comparing the head and tail pointers (as the Commodore Kernal routines do), or by maintaining a byte count through the interrupt handler (as the attached sample does). When bytes are available,

your program can process them, move the head pointer to the next character, and decrement the counter if you use one.

You should send a "Ctrl-S" (ASCII 19) to the host when the buffer is nearing capacity. At higher baud rates, this "maximum size" point may need to be lowered. We found 50 to 100 bytes worked fairly well at 9600 baud. You can probably do things more efficiently (we were using a very rough implementation) and set a higher maximum size. At some "minimum size", a "Ctrl-Q" (ASCII 17) can be sent to the host to resume transmission.

To transmit a byte using the logic of the first transmit routine below, first make sure that the transmit buffer isn't full. Then store the byte at the tail of the transmit buffer, point the tail to the next available location, and increment the transmit buffer counter (if used).

The 6551 transmit interrupt occurs when the transmit register is empty and available for transmitting another byte. Unless there are bytes to transmit, this creates unnecessary interrupts and wastes a lot of time. So, when the last byte is removed from the buffer, the interrupt handler in the first transmit routine below disables transmit interrupts.

Your program's code that stuffs new bytes into the transmit buffer must re-enable transmit interrupts, or your bytes may never be sent. A model for a main code routine for placing bytes in the transmit buffer follows the sample interrupt handler.

Using a transmit buffer allows your main program to perform other tasks while the NMI interrupt routine takes care of sending bytes to the ACIA. If the buffer has more than a few characters, however, you may find that most of the processor time is spent servicing the interrupt. Since the ACIA generates NMI interrupts, you can't "mask" them from the processor, and you may have timing difficulties in your program.

One solution is to eliminate the transmit buffer completely. Your program can decide when to send each byte and perform any other necessary tasks in between bytes as needed. A model for a main code routine for transmitting bytes without a transmit buffer is also shown following the sample interrupt handler code. Feedback from developers to date is that many of them have better luck not using transmit interrupts or a transmit buffer.

Although it's possible to eliminate the receive buffer also, we strongly advise that you don't. The host computer, not your program, decides when a new byte will arrive. Polling the ACIA for received bytes instead of using an interrupt-driven buffer just wastes your program's time and risks missing data.

For a thorough discussion of the use of buffers, the Kernal RS-232 routines, and the Commodore NMI handler, see *COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: Kernal*, by Dan Heeb (COMPUTE! Books) and *What's Really Inside the Commodore 64*, by Milton Bathurst (distributed in the U.S. by Schnedler Systems).

ACIA Registers

The four ACIA registers are explained in detail in the enclosed data sheets. The default location for them in our cartridge is addresses $DE00 - $DE03 (56832 - 56836).

Data Register ($DE00)

This register has dual functionality: it is used to receive and transmit all data bytes (i.e. it is a read/write register).

Data received by the ACIA is placed in this register. If receive interrupts are enabled, an interrupt will be generated when all the bits for a received byte have been assembled and the byte is ready to read.

Transmit interrupts, if enabled, are generated when this register is empty (available for transmitting). A byte to be transmitted can then be placed in this register.

Status Register ($DE01)

This register has dual functionality: it show the status of various ACIA settings when read, but when written to (data = anything [i.e. don't care]), this register triggers a reset of the chip.

As the enclosed data sheet shows (Figure 8), the ACIA uses bits in this register to indicate data flow and errors. If the ACIA generates an interrupt, bit #7 is set. There are four possible sources of interrupts:

1) receive (if programmed)
2) transmit (if programmed)
3) if a connected device changes the state of the DCD line
4) if a connected device changes the state of the DSR line

Some programmers have reported problems with using bit #7 to verify ACIA interrupts. At 9600 bps and higher, the ACIA generates interrupts properly, and bits #3-#6 (described below) are set to reflect the cause of the interrupt, as they should. But, bit #7 is not consistently set. At speeds under 9600 bps, bit #7 seems to work as designed. To avoid any difficulties, the sample code below ignores bit #7 and tests the four interrupt source bits directly.

Bit #5 indicates the status of the DSR line connected to the RS-232 device (modem, printer, etc.), while bit #6 indicates the status of the DCD line. **Note**: The function of these two bits is reversed from the standard implementation. Unlike many ACIA's, the 6551 was designed to use the DCD (Data Carrier Detect) signal from the modem to activate the receiver section of the chip. If DCD is inactive (no carrier detected), the modem messages and echoes of commands would not appear on your user's screen. We wanted the receiver active at all times. We also wanted to give you access to the DCD signal from the modem. So, we exchanged the DCD and DSR signals at the ACIA. Both lines are pulled active internally by the cartridge if left unconnected by the user (i.e., in a null-modem cable possibly).

Bit #4 is set if the transmit register is empty. Your program must monitor this bit and not write to the data register until the bit is set (see the sample XMIT code below).

Bit #3 is set if the receive register is full.

Bits #2, #1, and #0, when set, indicate overrun, framing, and parity errors in received bytes. The next data byte received erases the error information for the preceding byte. If you wish to use these bits, store them for processing by your program. The sample code below does not implement any error checking, but the Kemal software routines do, so adding these features to your code might be a good idea.

Command Register ($DE02)

The Command register (Figure 6) controls parity checking, echo mode, and transmit/receive interrupts. It is a read/write register, but reading the register simply tells you what the settings of the various parameters are. You use bits #7, #6, and #5 to choose the parity checking desired.

Bit #4 should normally be cleared (i.e. no echo).

Bits #3 and #2 should reflect whether or not you are using transmit interrupts, and if so, what kind. In the first sample transmit routine below, bit #3 is set and bit #2 is cleared to disable transmit interrupts (with RTS low [active]) on startup. However, when a byte is placed in the transmit buffer, bit #3 is cleared and bit #2 is set to enable transmit interrupts (with RTS low). When all bytes in the buffer have been transmitted, the interrupt handler disables transmit interrupts. **Note:** If you are connected to a RS-232 device that uses CTS/RTS handshaking, you can tell the device to stop temporarily by bringing RTS high (inactive): clear both bits #2 and #3.

Bit #1 should reflect whether or not you are using receive interrupts. In the sample code below, it is set to enable receive interrupts.

Bit #0 acts as a "master control switch" for all interrupts and the chip itself. It must be set to enable any interrupts-- if it is cleared, all interrupts are turned off and the receiver section of the chip is disabled. This bit also pulls the DTR line low to enable communication with the connected RS-232 device. Clearing this bit causes most Hayes-compatible modems to hang up (by bringing DTR high). This bit should be cleared when a session is over and the user exits the terminal program to ensure no spurious interrupts are generated. One fairly elegant way to do this is to perform a software reset of the chip (writing any value to the Status register).

Note: In Figures 6, 7, and 8 on the 6551 data sheet, there are small charts at the bottom of each that are labeled "Hardware Reset/Program Reset". These charts indicate what values the bits of these registers contain after a hardware reset (like toggling the computer's power) and a program reset (a write to the Status register).

Control Register ($DE03)

You use this register to control the number of stop bits, the word length, switch on the internal baud rate generator, and set the baud rate. It is a read/write register, but reading the register simply tells you what the settings of the various parameters are. See Figure 7 of the data sheet for a complete list of the parameters.

Be sure that bit #4, the "clock source" bit, is always set to use the on-chip crystal-controlled baud rate generator. You use the other bits to choose the baud rate, word length, and number of stop bits. Note that our cartridge uses a double-speed crystal, so the values given in Figure 7 should be doubled (i.e. the minimum speed is 100 bps and the maximum speed is 38,400 bps).

ACIA Hardware Interfacing

The ACIA is mounted on a circuit board designed to plug into the expansion (cartridge) port. The board is housed in a cartridge shell with a male DB-9 connector at the rear. The "IBM® PC/AT™ standard" DB-9 RS-232 pinout is implemented. Commercial DB-9 to DB-25 patch cords are readily available, and are sold by us as well.

Eight of the nine lines from the AT serial port are implemented: TxD, RxD, DTR, DSR, RTS, CTS, DCD, & GND. RI (Ring Indicator) is not implemented because the 6551 does not have a pin to handle it. CTS and RTS are not normally used by 2400 bps or slower Hayes-compatible modems, but these lines are being used by several newer, faster modems (MNP modems in particular). Note that although CTS is connected to the 6551, there is no way to monitor what its state is-- the value does not appear in any register. The 6551 handles CTS automatically: if it is pulled high (inactive) by the connected RS-232 device, the 6551 stops transmitting (clears the "transmit data register empty" bit [#4] in the status register).

The output signals are standard RS-232 level compatible. We've tested units with several commercial modems and with various computers using null modem cables at up to 38,400 bps without difficulties. In addition, there are pull-up resistors on three of the four input lines (DCD, DSR, CTS) so that if these pins are not connected in a cable, those three lines will pull to the active state. For example, if you happen to use a cable that is missing the DCD line, the pull-up resistor would pull the line active, so that bit #6 in the status register would be cleared (DCD is active low).

An on-board crystal provides the baud rate clock signal, with a maximum of 38.4 K baud, because we are using a double-speed crystal. If possible, test your program at 38.4Kb as well as lower baud rates. Users may find this helpful for local file transfers or using the C-64/C-128 as a dumb terminal on larger systems. And, after all, low cost 19.2 Kb modems for the masses are just around the corner.

Default decoding for the ACIA addresses is done by the I/O #1 line (pin 7) on the cartridge port. This line is infrequently used on either the C-64 or C-128 and should allow compatibility with most other cartridge products,

including the REU. The circuit board also has pads for users with special needs to change the decoding to I/O #2 (pin 10). This change moves the ACIA base address to $DF00, making it incompatible with the REU.

C-128 users may also elect to decode the ACIA at $D700 (this is a SID chip mirror on the C-64). Since a $D700 decoding line is not available at the expansion port, the user would need to run a clip lead into the computer and connect to pin 12 of U3 (a 74LS138). We have tried this and it works. $D700 is an especially attractive location for C-128 BBS authors, because putting the *SwiftLink* there will free up the other two memory slots for devices that many BBS sysops use: IEEE and hard drive interfaces.

Although we anticipate relatively few people changing ACIA decoding, you should allow your software to work with a *SwiftLink* at any of the three locations. You could either (1) test for the ACIA automatically by writing a value to the control register and then attempting to read it back or (2) provide a user-configurable switch/ poke/menu option.

The Z80 CPU used for CP/M mode in the C-128 is not connected to the NMI line, which poses a problem since the cleanest software interface for C-64/C-128 mode programming is with this interrupt. We have added a switch to allow the ACIA interrupt to be connected to either NMI or IRQ, which the Z80 does use. The user can move this switch without opening the cartridge.

Sample Code

```
;Sample NMI interrupt handler for 6551 ACIA on Commodore 64/128.
;(c) 1990 by Noel Nyman, Kent Sullivan, Bryan Minugh,
;Geoduck Developmental Systems, and Dr. Evil Labs.


;     ------ EQUATES ======

base      =    $DE00    ;base ACIA address
data      =    base
status    =    base+1
command   =    base+2
control   =    base+3


;Using the ACIA frees many addresses in zero page normally used by
;Kernal RS-232 routines.  The addresses for the buffer pointers were
;chosen arbitrarily.  The buffer vector addresses are those used by
;the Kernal routines.

rhead     =    $A7      ;pointer to next byte to be removed from
                        ;receive buffer
rtail     =    $A8      ;pointer to location to store next byte received
rbuff     =    $F7      ;receive buffer vector

thead     =    $A9      ;pointer to next byte to be removed from
                        ;transmit buffer
ttail     =    $AA      ;pointer to location to store next byte
                        ;in transmit buffer
tbuff     =    $F9      ;transmit buffer
```

```
xmitcount =    $AB      ;count of bytes remaining in transmit (xmit) buffer
recvcount =    $B4      ;count of bytes remaining in receive buffer

errors    =    $B5      ;DSR, DCD, and received data errors information

xmiton    =    $B6      ;storage location for model of command register
                        ;which turns both receive and transmit interrupts on
xmitoff   =    $BD      ;storage location for model of command register
                        ;which turns the receive interrupt on and the
                        ;transmit interrupts off

NMINV     =    $0318    ;Commodore Non-Maskable Interrupt vector


;     ------ INITIALIZATION ======


;Call the following code as part of system initialization.

;clear all buffer pointers, buffer counters, and errors location

      lda    #$00
      sta    rhead
      sta    rtail
      sta    thead
      sta    ttail

      sta    xmitcount
      sta    recvcount

      sta    errors

;store the addresses of the buffers in the zero page vectors

      lda    #<TRANSMIT.BUFFER
      sta    tbuff
      lda    #>TRANSMIT.BUFFER
      sta    tbuff + 1

      lda    #<RECEIVE.BUFFER
      sta    rbuff
      lda    #>RECEIVE.BUFFER
      sta    rbuff + 1

;The next four instructions initialize the ACIA to arbitrary values.
;These could be program defaults, or replaced by code that picks up
;the user's requirements for baud rate, parity, etc.


;The ACIA "control" register controls stop bits, word length, the
;choice of internal or external baud rate generator, and the baud
;rate when the internal generator is used.  The value below sets the
;ACIA for one stop bit, eight bit word length, and 4800 baud using the
;internal generator.
```

```
;               .------------------------ 0 = one stop bit
;               :
;               :.-------------------- word length, bits 6-7
;               ::.------------------- 00 = eight bit word
;               :::
;               :::.-------------- clock source, 1 = internal generator
;               ::::
;               ::::.----- baud
;               ::::::.---- rate
;               :::::::.--- bits
;               ::::::::.-- 0-3
      lda    #%00011010
      sta    control


;The ACIA "command" register controls the parity, echo mode, transmit and
;receive interrupt enabling, hardware "BRK", and (indirectly) the "RTS"
;and "DTR" lines.  The value below sets the ACIA for no parity check,
;no echo, disables transmit interrupts, and enables receive interrupts
;(RTS and DTR low).
;               .------------------------ parity control,
;               :.----------------------- bits 5-7
;               ::.---------------------- 000 = no parity
;               :::
;               :::.----------------- echo mode, 0 = normal (no echo)
;               ::::
;               ::::.------------ transmit interrupt control, bits 2-3
;               ::::::.----------- 10 = xmit interrupt off, RTS low
;               ::::::
;               :::::::.------- receive interrupt control, 0 = enabled
;               :::::::
;               ::::::::.--- DTR control, 1 = DTR low
      lda    #%00001001
      sta    command

;Besides initialization, also call the following code whenever the user
;changes parity or echo mode.

;It creates the "xmitoff" and "xmiton" models used by the interrupt
;handler and main program transmit routine to control the ACIA
;interrupt enabling.  If you don't change the models' parity bits,
;you'll revert to "default" parity on the next NMI.

                           ;initialize with transmit interrupts off since
                           ;buffer will be empty

      sta    xmitoff       ;store as a model for future use
      and    #%11110000    ;mask off interrupt bits, keep parity/echo bits
      ora    #%00000101    ;and set bits to enable both transmit and
                           ;receive interrupts
      sta    xmiton        ;store also for future use


;The standard NMI routine tests the <RESTORE> key, CIA #2, and checks
```

```
;for the presence of an autostart cartridge.

;You can safely bypass the normal routine unless:
;       * you want to keep the user port active
;       * you want to use the TOD clock in CIA #2
;       * you want to detect an autostart cartridge
;       * you want to detect the RESTORE key

;If you need any of these functions, you can wedge the ACIA
;interrupt handler in ahead of the Kernal routines.  It's probably
;safer to replicate in your own program only the Kernal NMI functions
;that you need.  We'll illustrate bypassing all the Kernal tests.

;BE SURE THE "NEWNMI" ROUTINE IS IN PLACE BEFORE EXECUTING THIS CODE!
;A "stray" NMI that occurs after the vector is changed to NEWNMI's address
;will probably cause a system crash if NEWNMI isn't there.  Also, it would
;be best to initialize the ACIA to a "no interrupts" state until the
;new vector is stored.  Although a power-on reset should disable all
;ACIA interrupts, it pays to be sure.

;If the user turns the modem off and on, an interrupt will probably be
;generated.  At worst, this may leave a stray character in the receive
;buffer, unless you don't have NEWNMI in place.

NEWVEC
      sei                   ;A stray IRQ shouldn't cause problems
                            ;while we're changing the NMI vector, but
                            ;why take chances?

;If you want all the normal NMI tests to occur after the ACIA check,
;save the old vector.  If you don't need the regular stuff, you can
;skip the next four lines.  Note that the Kernal NMI routine pushes
;the CPU registers to the stack.  If you call it at the normal address,
;you should pop the registers first (see EXITINT below).

;     lda    NMINV          ;get low byte of present vector
;     sta    OLDVEC         ;and store it for future use
;     lda    NMINV+1        ;do the same
;     sta    OLDVEC+1       ;with the high byte

                            ;come here from the SEI if you're not saving
                            ;the old vector
      lda    #$<NEWNMI      ;get low byte of new NMI routine
      sta    NMINV          ;store in vector
      lda    #$>NEWNMI      ;and do the same with
      sta    NMINV+1        ;the high byte

      cli                   ;allow IRQs again


;Reserve two bytes to store the old vector only if you saved it

;OLDVEC
;     nop                   ;low byte of original NMI stored here by NEWVEC
```

```
;     nop                 ;high byte of original NMI stored here by NEWVEC

;continue initializing your program

      :::    :::::::      ;program initialization continues


;     ---=== New NMI Routine Starts Here ===---


;The code below is a sample interrupt patch to control the ACIA.  When
;the ACIA generates an interrupt, this routine examines the status
;register which contains the following data.

;              .------------------------------ high if ACIA caused interrupt--
;              :                               not used in code below
;              :
;              :.--------------------------- reflects state of DCD line
;              ::
;              ::.------------------------ reflects state of DSR line
;              :::
;              :::.--------------------- high if xmit data register is empty
;              ::::
;              ::::.------------------ high if receive data register full
;              :::::
;              ::::::.------------- high if overrun error
;              ::::::
;              :::::::.------- high if framing error
;              :::::::
;              ::::::::.--- high if parity error
;       status xxxxxxxx

NEWNMI
;     sei                 ;the Kernal routine already does this before
                          ;jumping through the NMINV vector

      pha                 ;save A register
      txa
      pha                 ;save X register
      tya
      pha                 ;and save Y register

;As discussed above, the ACIA can generate an interrupt from one of four
;different sources.  We'll first check to see if the interrupt was
;caused by the receive register being full (bit #3) or the transmit
;register being empty (bit #4) since these two activities should receive
;priority.  A BEQ (Branch if EQual) tests the status register and branches
;if the interrupt was not caused by the data register.

;Before testing for the source of the interrupt, we'll prevent more
;interrupts from the ACIA by disabling them at the chip.  This prevents
;another NMI from interrupting this one.  (SEI won't work because the
;CPU can't disable non-maskable interrupts.)
```

```
;At lower baud rates (2400 baud and lower) this may not be necessary.  But,
;it's safe and doesn't take much time, either.

;The same technique should be used in any parts of your program where timing
;is critical.  Disk access, for example, uses SEI to mask IRQ interrupts.
;You should turn off the ACIA interrupts during disk access also to prevent
;disk errors and system crashes.

;First, we'll load the status register which contains all the interrupt
;and any received data error information in the 'A' register.

      lda    status

;Now prevent any more NMIs from the ACIA

      ldx    #%00000011    ;disable all interrupts, bring RTS inactive, and
                           ;leave DTR active
      stx    command       ;send to ACIA-- code at end of interrupt handler
                           ;will re-enable interrupts

;Store the status register data only if needed for error checking.
;The next received byte will clear the error flags.

;     sta    errors        ;only if error checking implemented

      and    #%00011000    ;mask out all but transmit and
                           ;receive interrupt indicators

;If you don't use a transmit buffer you can use
;
;     and    #%00001000

;to test for receive interrupts only and skip the receive test shown
;below.

      beq    TEST_DCD_DSR

;If the 'A' register=0, either the interrupt was not caused by the
;ACIA, or the ACIA interrupt was caused by a change in the DCD or
;DSR lines, so we'll branch to check those sources.

;If your program ignores DCD and DSR, you can branch to the end of
;the interrupt handler instead:
;
;     beq    NMIEXIT
;


;Test the status register information to see if a received byte is ready.
;If you don't use a transmit buffer, skip the next two lines.

RECEIVE                    ;process received byte
      and    #%00001000    ;mask all but bit #3
      beq    XMITCHAR      ;if not set, no received byte - if you're using
```

```
                        ;a transmit buffer, the interrupt must have been
                        ;caused by transmit.  So, branch to handle.
      lda    data       ;get received byte
      ldy    rtail      ;index to buffer
      sta    (rbuff),y  ;and store it
      inc    rtail      ;move index to next slot
      inc    recvcount  ;increment count of bytes in receive buffer
                        ;(if used by your program)

;Skip the "XMIT" routines below if you decide not to use a transmit buffer.
;In that case, the next code executed starts at TEST DCD_DSR or NMIEXIT.


;After processing a received byte, this sample code tests for bytes
;in the transmit buffer, and sends one if present.  Note that, in this
;sample, receive interrupts take precedence.  You may want to reverse the
;order in your program.

;If the ACIA generated a transmit interrupt and no received byte was
;ready, status bit #4 is already set.  The ACIA is ready to accept
;the byte to be transmitted and we've branched directly to XMITCHAR below.

;If only bit #3 was set on entry to the interrupt handler, the ACIA may have
;been in the process of transmitting the last byte, and there may still be
;characters in the transmit buffer.  We'll check for that now, and send the
;next character if there is one.  Before sending a character to the ACIA to
;be transmitted, we must wait until bit #4 of the status register is set.

XMIT
      lda    xmitcount  ;if not zero, characters still in buffer
                        ;fall through to process xmit buffer
      beq    TEST_DCD_DSR  ;no characters in buffer-- go to next check
;or
;
;     beq    NMIEXIT
;
;if you don't check DCD or DSR in your program.

XMITBYTE
      lda    status     ;test bit #4
      and    #%00010000
      beq    XMITBYTE   ;wait for bit #4 to be set

XMITCHAR                ;transmit a character
      ldy    thead
      lda    (tbuff),y  ;get character at head of buffer
      sta    data       ;place in ACIA for transmit

      inc    thead      ;point to next character in buffer
                        ;and store new index
      dec    xmitcount  ;subtract one from count of bytes
                        ;in xmit buffer

      lda    xmitcount
```

```
      beq    TEST_DCD_DSR

;or
;     beq    NMIEXIT
;
;if you don't check DCD or DSR in your program.

;If xmitcount decrements to zero, there are no more characters to
;transmit.  The code at NMIEXIT turns ACIA transmit interrupts off.

;If there are more bytes in the buffer, set up the 'A' register with
;the model that turns both transmit and receive interrupts on.  We felt
;that was safer, and not much slower, than EORing bits #3 and #4.  Note
;that the status of the parity/echo bits is preserved in the way "xmiton"
;and "xmitoff" were initialized earlier.

      lda    xmiton        ;model to leave both interrupts enabled

;If you don't use DCD or DSR, you can skip the following test.

      bne    NMICOMMAND    ;branch always to store model in command register


;If your program uses DCD and/or DSR, you'll want to know when the state
;of those lines changes.  You can do that outside the interrupt handler
;by polling the ACIA status register, but if either of the lines changes,
;the chip will generate an NMI anyway.  So, you can let the interrupt
;handler do the work for you.  The cost is the added time required to
;execute the DCR_DSR code on each NMI.

TEST_DCD_DSR
;     pha                 ;only if you use a transmit buffer, 'A' holds
                          ;the proper mask to re-enable interrupts on
                          ;the ACIA

      ::
      ::                  ;appropriate code here to compare bit #6 (DCD)
      ::                  ;and/or bit #5 (DSR) with their previous states
      ::                  ;which you've already stored in memory and take
      ::                  ;appropriate action
      ::
      ::

;     pla                 ;only if you pushed it at the start of the
                          ;DCD/DSR routine
;     bne    NMICOMMAND   ;'A' holds the xmiton mask if it's not zero,
                          ;implying that we arrived here from xmit routine
                          ;not used if you're not using a transmit buffer.


;If the test for ACIA interrupt failed on entry to the handler, we branch
;directly to here.  If you don't use additional handlers, the RESTORE key
;(for example) will fall through here and have no effect on your program
;or the machine, except for some wasted cycles.
```

```
NMIEXIT
      lda    xmitoff       ;load model to turn transmit interrupts off


;This line sets the interrupt status to whatever is in the 'A' register.

NMICOMMAND
      sta    command

;That's all we need for the ACIA interrupt handler.  Since we've pushed the
;CPU registers to the stack, we need to pop them off.  Note that we must
;do this EVEN IF YOU JUMP TO THE KERNAL HANDLER NEXT, since it will push
;them again immediately.  You can skip this step only if you're proceeding
;to a custom handler.

EXITINT                    ;restore things and exit
      pla                  ;restore 'Y' register
      tay
      pla                  ;restore 'X' register
      tax
      pla                  ;restore 'A' register

;If you want to continue processing the interrupt with the Kernal routines:

;     jmp    (OLDVEC)      ;continue processing interrupt with Kernal
                           ;handler

;Or, if you add your own interrupt routine:

;     jmp    YOURCODE      ;continue with your own handler

;If you use your own routine, or if you don't add anything, BE SURE to do
;this last:

      rti                  ;return from interrupt

;to restore the flags register the CPU pushes to the stack before jumping
;to the Kernal code. It also returns you to the interrupted part of
;your program.
--------------------------------------------------------------------------
;Sample routine to store a character in the buffer to be transmitted
;by the ACIA.

;(c) 1990 by Noel Nyman, Kent Sullivan, Bryan Minugh,
;Geoduck Developmental Systems, and Dr. Evil Labs.


;Assumes the character is in the 'A' register on entry. Destroys 'Y'--
;push to stack if you need to preserve it.

SENDBYTE                   ;adds a byte to the xmit buffer and sets
                           ;the ACIA to enable transmit interrupts (the
                           ;interrupt handler will disable them again
```

```
                        ;when the buffer is empty)

      ldy    xmitcount     ;count of bytes in transmit buffer
      cpy    #255          ;max buffer size
      beq    NOTHING       ;buffer is full, don't add byte

      ldy    ttail         ;pointer to end of buffer
      sta    (tbuff),y     ;store byte in 'A' at end of buffer
      inc    ttail         ;point to next slot in buffer
      inc    xmitcount     ;and add one to count of bytes in buffer

      lda    xmiton        ;get model for turning on transmit interrupts
      sta    command       ;and tell ACIA to do it

      rts                  ;return to your program

NOTHING
      lda    #$00          ;or whatever flag your program uses to tell that
                           ;the byte was not transmitted
      rts                  ;and return
--------------------------------------------------------------------------
;Sample routine to transmit a character from main program when not
;using a transmit buffer.

;(c) 1990 by Noel Nyman, Kent Sullivan, Bryan Minugh,
;Geoduck Developmental Systems, and Dr. Evil Labs.


;Assumes the character to be transmitted is in the 'A' register on entry.
;Destroys 'Y'-- push to stack if you need to preserve it.

SENDBYTE
      tay                  ;remember byte to be transmitted

TESTACIA
      lda    status        ;bit #4 of the status register is set if
                           ;the ACIA is ready to transmit another byte,
                           ;even if transmit interrupts are disabled
      and    #%00010000
      beq    TESTACIA      ;wait for bit #4 to be set

      sty    data          ;give byte to ACIA
      rts                  ;and exit
```