SwiftLink-232 Application Notes (version 1.0b)

This information is made available from a paper document published by CMD, with CMD's express written permission. [This version includes a couple of grammatical corrections and minor changes, plus, the source code has been debugged and extended by Craig Bruce <csbruce@ccnga.uwaterloo.ca>.]

1. 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 the low cost and compatibility with other Commodore (MOS) integrated circuits. Commodore used the 6551 as a model for the Kernal 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 function as their software counterparts. You may need to make a few changes in your program to accommodate the differences.

2. BUFFERS

Bytes received are placed in "circular" or "ring" buffers by the sample 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 Kernal routines free many zero-page locations, which could improve performance of pointers to large 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 "maximum 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 into the transmit buffer follows the sample interrupt handler.

Using a transmit buffer allows your main program to perform other takes 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 the 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 waste's 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 US by Schnedler Systems).

3. ACIA REGISTERS

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

3.1. 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 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 be placed in this register.

3.2. STATUS REGISTER ($DE01)

This register has dual functionality: it shows the 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, 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 ACIAs, 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 echos of commands would not appear on the user's screen. We wanted the receiver active at all times. We also wanted to give the 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 an null-modem cable possibility).

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 i sset (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 Kernal software routines do, so adding features to your code might be a good idea.

3.3. COMMAND REGISTER ($DE02)

The Command Register control 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 on 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 insure that 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 the figures on the 6551 data sheet, there are small charts at the bottom of each of the labelled "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).

3.4. 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 various parameters are. See the figure in the data sheet for a complete list of 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 values given on the data sheet are doubled [this is how they are shown below] (the minimum speed is 100 bps and the maximum speed is 38,400 bps).

4. 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(R) PC/AT(TM) 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 state it 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 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 will 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 Kbaud, because we are using a double-speed crystal. If possible, test your program at 38.4 Kbaud as well as lower baud rates. Users may find this helpful for local file transfers using the C-64/C-128 as a dumb terminal on larger systems. And, after all, low-cost 28.8 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 U2 (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.

5. SAMPLE CODE

This section has been translated into ACE-assembler format. Cut on the dotted lines to extract the code, and assemble it using the ACE assembler (ACE is a public-domain program). This program will work on both the C64 and C128. To use from BASIC:

```
LOAD"SAMPLE",8,1
SYS8192
```

It is a very simple terminal program. Press the STOP key to exit from it.

%%%---8<---cut-here---8<---%%%

```
;Sample NMI interrupt handler for 6551 ACIA on Commodore 64/128

;(c) 1990 by Noel Nyman, Kent Sullivan, Brian Minugh,
;Geoduck Development 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
;Kernel 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 turn 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
OLDVEC   =   $03fe   ;innocuous location to store old NMI vector (two bytes)

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

;Call the following code as part of system initialization.

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

     org   $2000     ;change to suit your needs
     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   ;1010 == 4800 baud, change to what you want
;           :::: :::.-- 0-3
    lda   #%0001_1010
    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   #%0000_1001
    sta   command

;Besides initialization, also call the following code whenever the user
;changes parity of 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   #%1111_0000 ;mask off interrupt bits, keep parity/echo bits
    ora   #%0000_0101 ;and set bits to enable both transmit and
                ;receive interrupts
    sta   xmiton      ;store also for future use

;The standard NMI routine tests th <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 RESTOR 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 Kernal tests.

;BE SURE THE "NEWNMI" ROUTINE IS IN PLACE BEFORE EXITING 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 teh receive
;buffer, unless you don't have NEWNMI in place.

NEWVEC:
    sei           ;A stray IRQ shouldn't cause any 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

;continue initializing your program

;    :::   ::::::     ;program initialization continues
     jmp   TERMINAL   ;go to the example dumb-terminal subroutine

;Save two bytes to store the old vector only if you need it


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

;The code below is a simple 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  xxxx_xxxx

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         ;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 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   #%0000_0011   ;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   #%0001_1000   ;mask out all but transmit and
                        ;receive interrupt indicators

;If you don't use a transmit buffer you can use
;
;   and   #%0000_1000
;
;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 instructions.

RECEIVE:              ;process received byte
    and   #%0000_1000   ;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 on 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   TEST_DCD_DSR  ;skip if transmitter still busy

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

                  ;point to next character in buffer
    inc   thead        ;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

    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 teh work for you.  The cost is the added time required to
;execute the DCD_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

;and 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 you 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 (C64 only):

;    rti             ;return from interrupt instruction

;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, Brian 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       ;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

;Alternative routine to transmit a character from main program when not using
;a transmit buffer.
;
;(c) 1990 by Noel Nyman, Kent Sullivan, Brian 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   #%0001_0000
;    beq   TESTACIA      ;wait for bit #4 to be set
;    sty   data          ;give byte to ACIA
;    rts

;Sample routine to fetch a character that has been received, from the
;receive buffer.
```

```
;by Craig Bruce, 1995, adapted from above

;Will return the character in the 'A' register and the carry flag cleared if
;a character was available.  If no character was available, will return with
;the carry flag set.  Destroys the 'Y' register.

RECVBYTE:               ;fetches a byte from the receive buffer.
                        ;there is no need to fiddle with any interrupts

    lda   recvcount     ;count of bytes in receive buffer
    beq   RECVEMPTY     ;buffer is empty, indicate to caller

    ldy   rhead         ;pointer to start of buffer
    lda   (rbuff),y     ;fetch byte out of buffer into 'A' register
    inc   rhead         ;point to next slot in buffer
    dec   recvcount     ;and add one to count of bytes in buffer

    clc                 ;indicate that we have a character
    rts                 ;return to your program

RECVEMPTY:
    sec                 ;or whatever flag your program uses to tell that the
                        ;receive buffer was empty
    rts                 ;and return

;---------------------------------------------------------------------------
;Dumb -- very dumb -- terminal emulator.  Simply polls the receive buffer and
;the keyboard and puts received data to the screen and typed data to the send
;buffer (thus, it assumes a full-duplex, echoing link).  There is no
;PETSCII->ASCII conversion, no cursor, nor any other fancy features.  Press
;STOP to exit.
;
;by Craig Bruce, 1995.

TERMINAL:
    jsr   RECVBYTE      ;see if there is a received byte in the recv buffer
    bcs   TERMTRYSEND   ;if not, continue
    jsr   $FFD2         ;if received byte, print it to the screen (CHROUT)
TERMTRYSEND:
    jsr   $FFE4         ;try to get a character from the keyboard (GETIN)
    cmp   #$00          ;was there a keystroke available?
    beq   TERMINAL      ;no--go back to top of polling loop
    cmp   #$03          ;check for STOP key
    beq   TERMEXIT      ;  exit if pressed
    jsr   SENDBYTE      ;have char--put it into the transmit buffer and then
    jmp   TERMINAL      ;  go back to top of polling loop
TERMEXIT:
    lda   #%0000_0010   ;disable transmitter and receiver and all interrupts
    sta   command
    sei
    lda   OLDVEC        ;restore the NMI vector to its original value
    sta   NMINV
    lda   OLDVEC+1
    sta   NMINV+1
```

```
    cli
    rts             ;exit

TRANSMIT_BUFFER = *+0
RECEIVE_BUFFER  = *+256
%%%---8<---cut-here---8<---%%%
```

APPENDIX: 6551 ACIA HARDWARE SPECS (DATA SHEET)

C= Commodore Semiconductor Group
a division of Commodore Business Machines, Inc.
950 Rittenhouse Road, Nornstown, PA 19400 * 215/666-7950 * TWX 510-660-4168
(July 1987)

6551 ASYNCHRONOUS COMMUNICATION INTERFACE ADAPTER

CONCEPT:

The 6551 is an Asynchronous Communication Adapter (ACIA) intended to provide for interfacing the 6500/6800 microprocessor families to serial communication data sets and modems. A unique feature is the inclusion of an on-chip programmable baud-rate generator, with a crystal being the only external component required.

FEATURES:

* On-chip baud-rate generator: 15 programmable baud rates derived from a standard standard 1.8432 MHz external crystal (50 to 19,200 baud) [these rates are doubled in the SwiftLink].

* Programmable interrupt and status register to simplify software design.

* Single +5 volt power supply.

* Serial echo mode.

* False start bit detection.

* 8-bit bi-directional data bus for direct communication with the microprocessor.

* External 16x clock input for non-standard baud rates (up to 125 Kbaud).

* Programmable: word lengths; number of stop bits; and parity-bit generation and detection.

* Data set and modem control signals provided.

* Parity: (odd, even, none, mark, space).

* Full-duplex or half-duplex operation.

* 5,6,7 and 8-bit transmission.

* 1-MHz, 2-MHz, and 3-MHz operation.

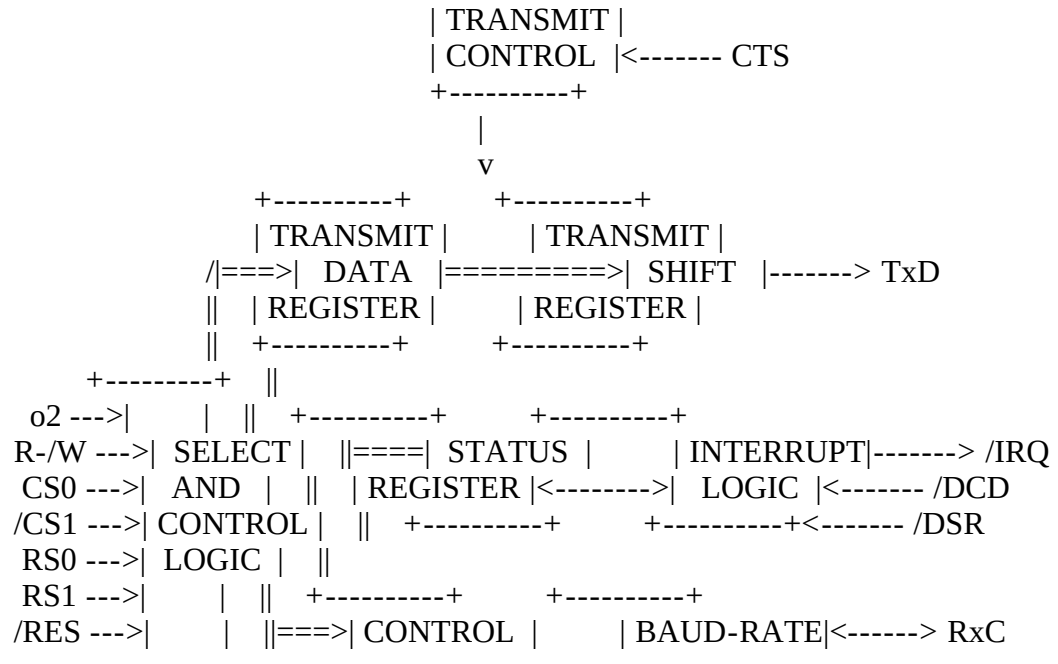
ORDER NUMBER

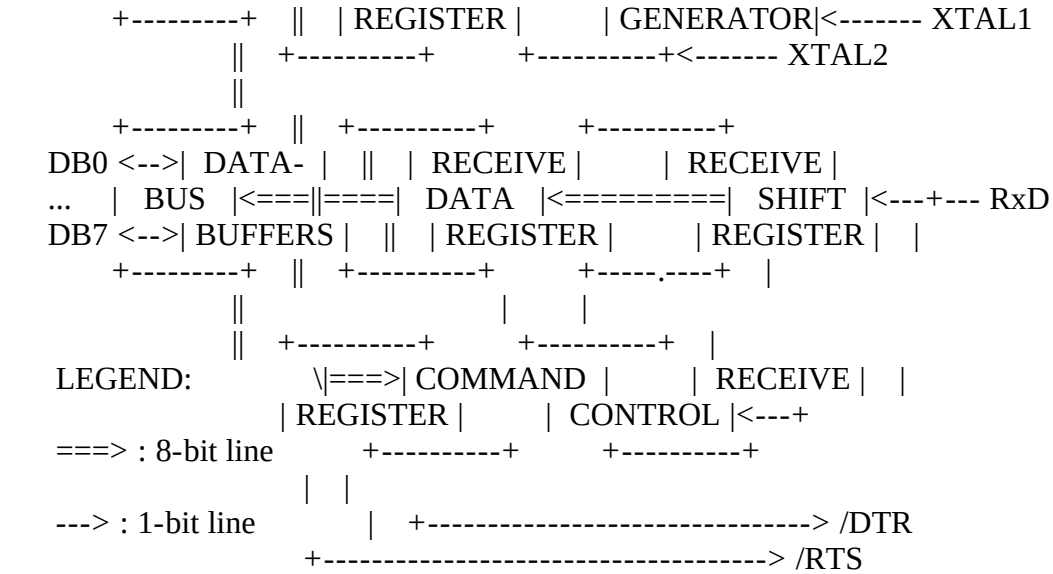
```
MXS 6551 ___
 -       |
 |       +---- Frequency range
 |              Plain = 1 MHz
 |                 A = 2 MHz
 |                 B = 3 MHz
 |
 +------------ Package Designator
               C = Ceramic
               P = Plastic
```

6551 PIN CONFIGURATION

```
           +---------------+
        GND --| 1         28 |-- R-/W
        CS0 --| 2         27 |-- o2
       /CS1 --| 3         26 |-- /IRQ
       /RES --| 4         25 |-- DB7
        RxC --| 5         24 |-- DB6
      XTAL1 --| 6         23 |-- DB5
      XTAL2 --| 7         22 |-- DB4
       /RTS --| 8         21 |-- DB3
       /CTS --| 9         20 |-- DB2
        TxD --| 10        19 |-- DB1
       /DTR --| 11        18 |-- DB0
        RxD --| 12        17 |-- /DSR
        RS0 --| 13        16 |-- /DCD
        RS1 --| 14        15 |-- Vcc
           +---------------+
```

BLOCK DIAGRAM





MAXIMUM RATINGS

<not included here>

ELECTRICAL CHARACTERISTICS

<not included here>

POWER DISSIPATION vs TEMPERATURE

<not included here>

TIMING CHARACTERISTICS

<not included here>

INTERFACE SIGNAL DESCRIPTION

/RES (Reset)

During system initialization a low on the /RES input will cause internal registers to be cleared.

o2 (Input Clock)

The input clock is the system o2 clock and is used to trigger all data transfers between the system microprocessor and the 6551.

R-/W (Read/Write)

The R-/W is generated by the microprocessor and is used to control the direction of data transfers. A high on the R-/W pin allows the processor to read the data supplied by the 6551. A low on the R-/W pin allows a write to the 6551.

/IRQ (Interrupt Request)

The /IRQ pin is an interrupt signal from the interrupt-control logic. It is an open drain output, permitting several devices to be connected to the common /IRQ microprocessor input. Normally a high level, /IRQ goes low when an interrupt occurs.

DB0--DB7 (Data Bus)

The DB0--DB7 pins are the eight data lines used for transfer of data between the processor and the 6551. These lines are bi-directional and are normally high-impedance except during Read cycles when selected.

CS0, /CS1 (Chip Selects)

The two chip-select inputs are normally connected to the processor-address lines either directly or through decoders. The 6551 is selected when CS0 is high and /CS1 is low.

RS0, RS1 (Register Selects)

The two register-select lines are normally connected to the processor-address lines to allow the processor to select the various 6551 internal registers. The following table indicates the internal register-select coding:

RS1	RS0	WRITE	READ	SL-Addr
0	0	Transmit Data Register	Receive Data Register	$DE00
0	1	Programmed Reset*	Status Register	$DE01
1	0	Command Register	Command Register	$DE02
1	1	Control Register	Control Register	$DE03

* for programmed reset, data is "don't care".

The table shows that only the Command and Control registers are read/write. The Programmed Reset operation does not cause any data transfer, but is used to clear the 6551 registers. The Programmed Reset is slightly different from the Hardware Reset (/RES) and these differences are described in the individual register definitions.

ACIA/MODEM INTERFACE SIGNAL DESCRIPTION

XTAL1, XTAL2 (Crystal Pins)

These pins are normally directly connected to the external crystal (1.8432 MHz) used to derive the various baud rates. Alternatively, an externally generated clock may be used to drive the XTAL1 pin, in which case the XTAL2 pin must float. XTAL1 is the input pin for the transmit clock.

TxD (Transmit Data)

The TxD output line is used to transfer serial NRZ (non-return-to-zero) data to the modem. The LSB (least-significant bit) of the Transmit Data Register is the first data bit transmitted and the reate of data transmission is determined by the baud rate selected.

RxD (Receive Data)

The RxD input line is used to transfer serial NRZ data into the ACIA from the modem, LSB first. The receiver data rate is either the programmed baud rate or the rate of an externally generated receiver clock. This selection is made by programming the Control Register.

RxC (Receive Clock)

The RxC is a bi-directional pin which serves as either the receiver 16x clock input or the receiver 16x clock output. The latter mode results if the internal baud rate generator is selected for receiver data clocking.

/RTS (Request to Send)

The /RTS output pin is used to control the modem from the processor. The state of the /RTS pin is determined by the contents of the Command Register.

/CTS (Clear to Send)

The /CTS input pin is used to control the transmitter operation. The enable state is with /CTS low. The transmitter is automatically disabled if /CTS is high.

/DTR (Data Terminal Ready)

The output pin is used to indicate the status of the 6551 to the modem. A low of /DTR indicates the 6551 is enabled and a high indicates it is disabled. The processor controls this pin via bit 0 of the Command Register.

/DSR (Data Set Ready)

The /DSR input pin is used to indicate to the 6551 the status of the modem. A low indicates the "ready" state and a high, "not-ready". /DSR is a high-impedance input and must not be a no-connect. If unused, it should be driven high or low, but not switched.

Note: If Command Register Bit #0 = 1 and a change of state on /DSR occurs, /IRQ will be set and Status Register Bit #[5] will reflect the new level. The state of /DSR does not affect Transmitter operation [but must be low for the Receiver to operate]. [This statement reflects the SwiftLink implementation].

/DCD (Data Carrier Detect)

The /DCD input pin is used to indicate to the 6551 the status of the carrier-detect output of the modem. A low indicates that the modem carrier signal is present and a high, that it is not. /DCD, like /DSR, is a high-impedance input and must not be a no-connect.

Note: If Command Register Bit #0 = 1 and a change of state on /DSR occurs, /IRQ will be set and Status Register Bit #[6] will reflect the new level. The state of /DCD does not affect either Transmitter or Receiver operation.

INTERNAL ORGANIZATION

<not included here>

TRANSMIT AND RECEIVE DATA REGISTERS (SL-Addr: $DE00 / 56832)

These registers are used as temporary data storage for the 6551 Transmit and Receive circuits. The Transmit Data Register is characterized as follows:

* Bit 0 is the leading bit to be transmitted.

* Unused data bits are the high-order bits and are "don't care" for transmission.

The Receive Data Register is characterized in a similar fashion:

* Bit 0 is the leading bit received.

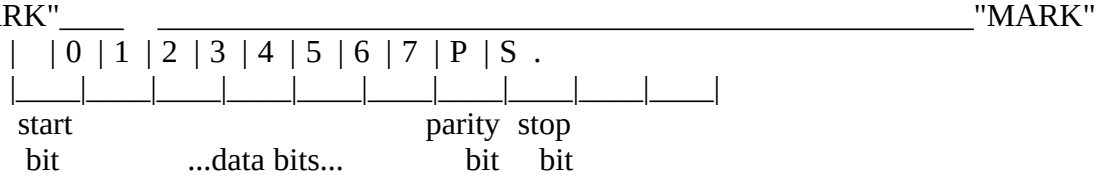
* Unused data bits are the high-order bits and are "0" for the receiver.

* Parity bits are not contained in the Receive Data Register, but are stripped off after being used for external parity checking. Parity and all unused high-order bits are "0".

Transmit / Receive Data Register

7	6	5	4	3	2	1	0
data							

The following figure illustrates a single transmitted or received data word, for the example of 8 data bits, parity, and 1 stop bit:



STATUS REGISTER (SL-Addr: $DE01 / 56833)

The Status Register is used to indicate to the processor the status of various 6551 functions and is outlined here:

Command Register

7	6	5	4	3	2	1	0
irq	dcd	dsr	txr	rxr	ovr	fe	pe

7	/IRQ*** : cleared by reading status register
0	No interrupt

1 Interrupt

+---+
| 6 | /DCD : non-resetable, indicates /DCD status
+---+ ----------

0 /DCD low
1 /DCD high

+---+
| 5 | /DSR : non-resetable, indicates /DSR status
+---+ ----------

0 /DSR low
1 /DSR high

+---+
| 4 | Transmit Data Register Empty: Cleared by write to Tx Data reg
+---+ ----------

0 Not empty
1 Empty

+---+
| 3 | Receive Data Register Full: Cleared by read from Rx Data reg
+---+ ----------

0 Not full
1 Full

+---+
| 2 | Overrun*: Self-clearing**
+---+ ----------

0 No error
1 Error

+---+
| 1 | Framing Error*: Self-clearing**
+---+ ----------

0 No error
1 Error

+---+
| 0 | Parity Error*: Self-clearing**
+---+ ----------

0 No error
1 Error

Notes: * No interrupt generated for these conditions
** Cleared automatically after a read of RDR and the next error-free receipt of data
*** Reading status reg. will clear the /IRQ bit except when transmit intr. enabled

7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+
| 0 | x | x | 1 | 0 | 0 | 0 | 0 | After Hardware reset
+---+---+---+---+---+---+---+---+

| x | x | x | x | x | 0 | x | x | After Software reset
+---+---+---+---+---+---+---+---+

COMMAND REGISTER (SL-Addr: $DE02 / 56834)

The Command Register is used to control specific Transmit/Receive functions and is shown here:

Command Register

7	6	5	4	3	2	1	0
parity			echo	tx ctrl		rxi	dtr

7	6	5	PARITY CHECK CONTROLS
x	x	0	parity disabled--no parity bit generated or received
0	0	1	odd parity receiver and transmitter
0	1	1	even parity receiver and transmitter
1	0	1	mark parity transmitted, parity check disabled
1	1	1	space parity transmitted, parity check disabled

4	NORMAL/ECHO MODE FOR RECEIVER
0	Normal
1	Echo (bits 2 and 3 must be "0")

3	2	Tx INTERRUPT	RTS LEVEL	TRANSMITTER
0	0	Disabled	High	Off
0	1	Enabled	Low	On
1	0	Disabled	Low	On
1	1	Disabled	Low	Transmit BRK

1	RECEIVE INTERRUPT ENABLE
0	/IRQ interrupt Enabled from bit 3 of Status Register
1	/IRQ interrupt Disabled

0	DATA TERMINAL READY
0	Disable Receiver and all interrupts (/DTR high)
1	Enable Receiver and all interrupts (/DTR low)

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	After Hardware reset

| x | x | x | 0 | 0 | 0 | 0 | 0 | After Software reset
+---+---+---+---+---+---+---+---+

CONTROL REGISTER (SL-Addr: $DE03 / 56835 / cpm: 0001xxxx)

The Control Register is used to select the desired mode for the 6551. The word length, number of stop bits, and clock controls are all determined by the Control Register, which is shown here:

Control Register

7	6	5	4	3	2	1	0
stops	word len	src	baud rate

7	STOP BITS
0	1 stop bit
1	2 stop bits
1	1 stop bit if word length== 8 bits and parity this allows for 9-bit transmission (8 data bits plus parity)
1	1.5 stop bits if word length== 5 bits and no parity

6	5	WORD LENGTH
0	0	8 bits
0	1	7 bits
1	0	6 bits
1	1	5 bits

4	RECEIVER CLOCK SOURCE
0	external receiver clock
1	baud rate generator

3	2	1	0	BAUD RATE GENERATOR
0	0	0	0	16x external clock
0	0	0	1	100 baud
0	0	1	0	150 baud
0	0	1	1	219.84 baud
0	1	0	0	269.16 baud
0	1	0	1	300 baud
0	1	1	0	600 baud
0	1	1	1	1200 baud
1	0	0	0	2400 baud
1	0	0	1	3600 baud
1	0	1	0	4800 baud
1	0	1	1	7200 baud

```
  1   1   0   0    9600 baud
  1   1   0   1    14400 baud
  1   1   1   0    19200 baud
  1   1   1   1    38400 baud

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |  After Hardware reset
+---+---+---+---+---+---+---+---+
| x | x | x | x | x | x | x |  After Software reset
+---+---+---+---+---+---+---+---+
```

==========

Design and Implementation of a Simple/Efficient Upload/Download Protocol
by Craig Bruce <csbruce@ccnga.uwaterloo.ca>

1. INTRODUCTION

If you use your Commodore for telecommunications, then you are basically interested in two things: using your C= to emulate a terminal for interactive stuff, and using modem-file-transfer protocols to upload and download files from and to your Commodore.

This document describes a custom upload/download protocol that was designed for use with the ACE-128/64 system and is freely available to anyone who wants it (well, when I finish with the Release #14 of ACE). While this protocol non-standard, it blows the doors off of all other protocols available for Commodore computers, even though it uses a simple "stop-and-wait" acknowledgement scheme. There are two reasons for its speed: the fast device drivers available with ACE, and its large packet size, up to about 18K (although this could be significantly larger is ACE's memory usage were reorganized).

The name of the protocol is "Craig's File eXchange Protocol", or just "FX" for short. It is "file exchange" rather than "upload" or "download" because you will use the same activation of the program to both upload and download all of the files you name.

2. USAGE

The current implementation of FX consists of a "client" program for you to run on your Commodore computer and a "server" program that you run on your Unix host. There is currently no server program for any other platform, but the necessary changes to the C-language program wouldn't be too hard. The client program is written in 6502 assembler, of course (for the ACE-assembler to be specific).

FX is an external program from the terminal program, so (for now) to activate FX, you have to exit from the terminal program and enter the FX command line, exchange the files, and then re-enter the terminal program from the command line.

When you run FX, you will activate the Server program first on your Unix host and then exit the terminal program and run the Client program on your

Commodore. You run the command "fx" on both the client and server machines, which may be a little confusing (but I think you'll get used to it), and name the files that you want to have transferred as arguments to the command on the machine that you want to transfer the files FROM. The usage of the "fx" command is as follows:

fx [-dlvV7] [-m maximums] [-f argfile] [[-b] binfile ...] [-t textfile ...]

-d = debug mode
-l = write to log file ("fx.log")
-v = verbose log/debug mode
-V = extremely verbose log/debug mode
-7 = use seven-bit encoding
-m = set maximum packet sizes; maximums = ulbin/ultxt/dlbin/dltxt (bytes)
-f = take arguments one-per-line from given argfile
-b = binary files prefix
-t = text files prefix
-help = help

well, for the server, anyway. The client program doesn't have the more exotic options. The "-d", "-l", "-v", and "-V" options are available only on the Server program, and are for debugging purposes only.

The "-7" option tells the protocol to use only 7-bit data. I.e., it tells it to not use the 8th bit position in the data is transmitted. This is useful if you are forced into the humiliation of only being able to use a 7-bit channel to your Unix host. You need only need to give this option on either the client or the host command line and the other side will be informed. It may be useful to create an alias for this command with all of your options set to what you want them to be.

The protocol has the capacity to use different packet sizes for four types of file-transfer situations: uploading binary data, uploading text, downloading binary data, and downloading text. These are useful distinctions, since your host may or may not be able to handle the larger packet sizes without losing bytes (your Commodore, of course, can handle the larger packet sizes with no problems).

In determining which packet size to use for a file transfer (where the type of transfer is known), the protocol finds that largest packet size that both the client and the server can handle and then take the minimum of these two values. The defaults for the client are all the same: the maximum amount of program-area memory that it can use, about 18K. For the server program, I have programmed in default maximum uploading packet sizes of 1K and maximum downloading packet sizes of 64K-1. You can change these defaults in the C program easily by changing some "#define"s.

The "-m" option allows you to manually set the default packet sizes for a transfer. The argument following the "-m" flag should have four numbers with slashes between them, which give the maximum ulbin/ultxt/dlbin/dltxt packet sizes, respectively. Note that the packet sizes only include the size of the user data encoded into packets and not the control or quoting information (below).

The "-f" option on the server allows you to read arguments from a file rather than the command line. This is useful if want to generate and edit the list of files to download before you run the FX command. It's also useful if you don't want other users to see the names of the files that you are downloading. The name of the file comes in the first argument following the "-f" flag and the arguments are put into this file one-per-line. You can put in "-" options in addition to filenames if you wish (like "-t" and "-b"). This option is not supported on the client program.

Finally come the "-b", "-t", and filename arguments. The "-b" argument tells FX that all of the following filenames (until the next "-t" option) are binary files and the "-t" argument says that the following filenames are all of text files. You can use as many "-b" and "-t" arguments as you want. If you don't use any, then all of the files you name will be assumed to be binary files.

For each filename you give on a command line, that file will be transferred from that machine to the other machine. On both Unix and ACE, you can use wildcards in your filenames, of course, to transfer groups of files.

The client program controls the file exchange, and it uploads all of its files first and then asks the server if the server has any files to be downloaded. When the exchange is completed, both the client and server FX programs will exit and you will find yourself back on the command lines in both environments. Re-enter the terminal program to continue with your online session. If something goes very wrong during a transfer or if you decide that you don't really want to transfer any files after activating the server program, you can type three Ctrl-X's to abort the server. This is the same as for the X-modem protocol.

3. DESIGN DECISIONS

There are a number of design decisions to be made about our protocol. But first, we want to recognize and appreciate that since we have a license to design a completely new protocol, we are not bound, shackled, gagged, and tortured by the "hysterical raisins" and bad design decisions of existing compromised and bloated standard protocols... such as Z-modem.

We want the protocol to understand whether a file is text or binary data and to translate them appropriately during downloading. We want the protocol to be aware of filenames, dates, permissions, and we do not want our file contents to get mangled like they do with X-modem (it pads them with Ctrl-Z's, since it was designed for CP/M), and we want it to translate to/from PETSCII if the file is text. We will require that the user tell us whether the file is binary or text (although we may be able to statistically determine this from snooping through the file), and we will use a "canonical form" for encoding the text data during transfer. A convenient canonical form to use is Unix-ASCII (ASCII-LF).

We want our protocol to be simultaneously simple and fast. To make it simple, we will use a stop-and-wait acknowledgement scheme. This means that after each packet is uploaded or downloaded, the transfer will pause and wait for the receiving host to acknowledge that the packet has been transferred correctly, and only then will the protocol continue to transfer more data.

In fact, this scheme fits well with the Commodore hardware, since it is not possible to send or receive serial data while doing disk I/O (in the general case), so we would have to stop listening anyway; the protocol makes it so that there will be no bytes that we end up ignoring while doing I/O.

To make the protocol be fast even though we are using a stop-and-wait acknowledgement scheme, we will use the largest data-packet sizes that we possibly can. In the (current) ACE environment, this means about 18K. This will maximize the amount of time of transferring data over the modem between pauses to do I/O. If the I/O is to the ACE ramdisk, then the length of this pause will be very short and we will achieve a very high link utilization. (The ACE ramdisk can process an 18K read/write request in about 20 milliseconds on a Fast-mode C128 using an REU --- RAMDOS in the same environment would require about 9 _seconds_ (450x slower)).

To allow for future use with other platforms, we will make the protocol define the packet sizes using 32-bit fields. There isn't much data overhead, and this allows us to change implementations to be able to transfer entire files in one large packet. Also, the size of an individual packet should be flexible: be from one to N bytes. This eliminates the X-modem padding problem and the Y-modem crufty hack of using the small packet size when less than 1K of user data remains to be transferred.

We also want our data to be well protected against corruption. Detecting transmission errors efficiently on Commodore computers is already a well solved problem: we will use a table-driven CRC-32 algorithm, the same one that ZMODEM, PKZIP, and CRC32 use. To hide the computation costs of the CRC even more (the cost is very low anyway), we will compute it WHILE sending or receiving packets. Oh, actually, I guess that I forgot to mention an a-priori design decision: we will be using a packet-oriented approach for transferring data (described below); packetization offers so many advantages that this decision is really a no-brainer.

Also, to make the process interaction as straightforward as possible, we want to use the Client/Server programming paradigm. This paradigm combines well with the stop-and-wait acknowledgement scheme to produce a Remote Procedure Call (RPC) type of interaction between the machines. For those not familiar with this Interprocess Communication (IPC) scheme, you can read a couple issues of C= hacking ago where I talked about it for use with a multitasking operation system. RPC is a very useful, powerful, simple, and widely applicable IPC scheme.

To recover from packet corruption, we will be using a timeout+retransmission scheme, and to be consistent with the RPC scheme, the client will do all timeouts and retransmissions. This means that after sending a request RPC packet out, if we don't receive the reply within a certain period of time, we will timeout and send the request again. Or, to be more precise, since we will be working with large packet sizes, we will timeout if we don't receive any bytes from the server for a certain period of time, say 5 seconds, while we are expecting more bytes from him.

The way that corrupted packets are dealt with is very simple: they are ignored. The server could possibly send back a negative acknowledgement, but we won't try that for now.

In order to make retransmissions work out correctly, we will be using sequence numbers and internal-state variables inside of the server to insure that requests aren't carried out more than once. We need these mechanisms because when an RPC fails, we won't know if we got no response because the original request was lost and the operations wasn't carried out, or whether the request was received and carried out but the reply message was lost.

For example, if we request that packet #123 be downloaded and the server carries out that request but the reply message is lost, then the client will time out and retransmit the request. The server remembers the last request number that the client sent it (123 here), so if the client asks for packet #123 again, the server will simply retransmit the reply that it gave last time. If, on the other hand, the client were to request packet #124 (or simply "not 123"), then the server reads the next chunk of data from the file and sends it as the reply. Our protocol will use an 8-bit sequence number even though it only needs a 1-bit sequence number (since eight bits will allow for the future expansion of having multiple requests being processed concurrently: asynchronous RPC).

We also want to be able to both upload and download as conveniently as possible. To me, this means doing both operations by calling only one command (as described in the previous section). This arrangement also allows for the future expansion of uploading and downloading files _simultaneously_ (the protocol as designed places no restrictions on this possibility).

We also want to make use of an eight-bit clean link between the Unix host and your Commodore, but this may not always be possible. Sometimes you may have only a 7-bit connection, and even if you do have an 8-bit connection, there may still be some software-flow-control problems with intermediate devices between your Commodore and your Unix host. So, we want our protocol to not make use of the X-on and X-off characters, and to use only 7-bit characters if it cannot use eight. The way to achieve this is called "escaping", "quoting" or "byte stuffing", and will be discussed in the next section. It turns out that supporting 7-bit characters is pretty simple and the mechanism is required by other aspects of the packetization.

There, that should take care of most of the major design decisions.

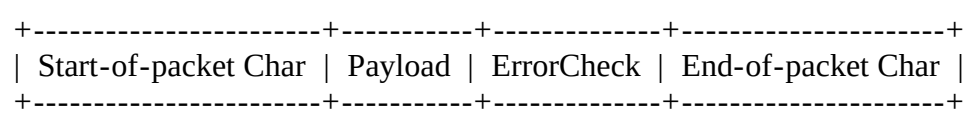
4. PACKETIZATION

Packetization refers to the process of taking a stream of data and breaking it up into discrete chunks of data. Each packet is easily identified and is processed as a single unit. There are many general advantages to using packets. If there is a transmission error, then only a single packet is corrupted, and the recovery will be easier since the packet is well identified, and only it needs to be recovered. Packetization also means that a link can be shared between multiple (logical) communication streams fairly and efficiently, and means that a single communication stream can utilize multiple physical links where facilities exist.

Packets also integrate well with many IPC schemes, including Remote Procedure Calls. In fact, you end up emulating a packet-oriented scheme even if you are using RPC over a stream-oriented transport system. Packets also take into

account the limited buffering capacity of both end systems and intermediate systems, and allow for the convenient implementation of flow control (even if said flow control consists of simply dropping packets on the floor). Packets are very useful things indeed! And just think that back in the early 1970s packets were dismissed as being infeasible and unusable.

Each packet used in the FX system has four parts to it: the start character, the user data (payload), the error-check characters, and the end character. Graphically, a packet has the following format:



The payload can be arbitrarily long, up to whatever limit the two computers involved in the transfer can handle.

The error check is a 32-bit (4-byte) Cyclic-Redundancy-Check value that occupies the last four bytes before the End-of-packet character. The implementation, which is based on a table-lookup method, is so efficient that it is as fast as a simple add-up checksum, except much more reliable. Using this error check, there will be approximately a one-in-4,000,000,000 chance that a packet with an error in it will be accepted has being error-free. These are pretty good odds for our purposes. The CRC is calculated exclusively on the raw payload data.

The following special characters used by packets are defined:

NAME	HEX	DEC	Control	Meaning
CHR_START	0x01	1	Ctrl-A	Packet-start indicator
CHR_END	0x19	25	Ctrl-Y	Packet-end indicator
CHR_ESC	0x05	5	Ctrl-E	Escape character for next code
CHR_ABORT	0x18	24	Ctrl-X	Abort transfer if repeated three times
CHR_XON	0x11	17	Ctrl-Q	Software flow-start: avoided
CHR_XOFF	0x13	19	Ctrl-S	Software flow-stop: avoided
CHR_QUOTE8	0x14	20	Ctrl-T	Quote-8 the next 7-bit sequence

CHR_START is used to signify the start of a new packet. This character is not allowed to be used anywhere else for any other purpose.

CHR_END is used to signify the end of the current packet, and cannot be used anywhere else. The reason for using special characters to mark the beginning and the ending of a packet is to allow for easy error recovery after a communication failure. All you do is search for the next CHR_START character after you toss away a garbled packet and you're back in business. I am unaware of any reasonable alternatives to framing packets with a CHR_START character. Using a CHR_END special character is a convenience.

CHR_ESC is used to "escape" the next character. Since there are special character codes that cannot be used in any other way than their intended function (including CHR_START and CHR_ESC itself), this character is needed. The character following the CHR_ESC character must be between "@" and "_" (0x40 and 0x5f) in the ASCII chart, or be the character "?" (0x3f). The

character following the CHR_ESC is then "and"ed with the value 0x1f to mask off the "letter" bits and turn it into a control character in the range of 0x00 to 0x1f (the same range as the special control characters) and the "escape sequence" is treated as a single character of user data. If the character following the CHR_ESC is a "?", then a code of 0x7f is interpreted instead. Using a character following the escape that is different from the character being represented allows for greater resiliance of the protocol in the presence of bits being garbled or bytes being dropped. All special characters in a packet except for the starting and ending characters are escaped as described above.

CHR_ABORT can be typed by the user into a terminal program at any time to shut down the server.

CHR_XON and CHR_XOFF can cause problems with intermediate devices on some systems, so the FX protocol does not use these character codes at all; it purposely avoids them and uses escape sequences (CHR_ESC) for them instead.

CHR_QUOTE8 is used to re-generate 8-bit data over a 7-bit link. Kermit uses this same technique. When this character is encountered in the receive stream, the next character is extracted and is "or"ed with a value of 0x80 to give it a "1" in the high-bit position. The CHR_QUOTE8 character can also be followed by a CHR_ESC code, which is interpreted as above and then "or"ed with the 0x80 value.

One of the disadvantages of using this scheme is that each byte in the range of 0x80 and 0xff takes at least two bytes to transmit and some of them three. If fact, for many binary files it may be faster to uuencode the file and transfer the resulting text, since uucode has a static encoding overhead of 33% whereas this quoting scheme has an expected overhead of 50% (plus the CHR_ESC overhead). Of course, this feature is intended to be used as a last resort if you cannot get an 8-bit connection.

So there you have it. Every message sent between the client and the server is encapsulated in a packet as specified above. Packetization allows for convenient error detection and recovery and works well with our interprocess communication scheme.

One implementation note about the packetization has to do with buffering. On the Unix host, it is advantageous to encode a packet into a memory buffer and then send out that buffer in a single "write" operation. This less operating-system overhead (which may or may not be significant) but more importantly, it means that the packet will be sent between intermediate communication devices as efficiently as possible. On my local Unix system, I connect to a terminal server and to my Unix host through that. Performing single-byte writes on the Unix host means that the bytes are sent in individual Ethernet packets between the Unix host and the terminal server, and encounter more overhead and communication delays. When I changed the program to send the FX packet in a single operation, a significant performance gain was realized.

For receiving data on the Unix host, there isn't much you can do other than reading one byte at a time, since the receiver doesn't know when a packet is going to end. However, the same problem is not encountered here that was encountered with sending data because data that is received by the Unix host

but not "read" by the user program are buffered and collected, smoothing out the system overhead, which is insignificant compared to the modem speed. The Unix program used the "stdin" and "stdout" file streams for receiving and transmitting data, and sets the tty driver to turn off all line-editing features to get at the raw bytes.

On the Commodore end, it is advantageous to read data from the modem driver in chunks, since the system overhead is significant compared to the modem speed. These are small computers that we are driving to the max, you know. Data is read from the modem in chunks of up to 255 bytes (whatever is available at the time) and processed a byte at a time from the read buffer. The CRC is calculated during processing, to avoid doing this on the critical path. The CRC calculation is performed as an operation by itself since the overhead is very small on fast processors. The character-set translation for text files will be performed on the critical path (on the Commodore) since it is more convenient to do it at a higher layer in the IPC scheme. The packet- handling software is logically at a distinct layer that doesn't have to worry about higher layers. The next layer up is logically the RPC layer and then the file-transfer layer.

5. CLIENT/SERVER OPERATION

As discussed previously, the client/server interaction is based on a Remote Procedure Call paradigm. Thus, for each operation, the client sends a request packet (message) to the server, and the server performs the requested operation and sends back a reply (acknowledgement) message to the client.

There are eight request/ack interactions that are defined for the protocol: two for connection management, three for uploading files, and three for downloading files. The client is in charge of the file-exchange session and of the error handling.

4.1. CONNECTION MANAGEMENT

When the client starts up, the first thing that it does is connect to the server. The format of the message that it sends is as follows:

OFF	SIZ	DESC
0	1	code: REQ_CONNECT ('C')
1	1	protocol version := 0x01
2	1	transmit byte size: '7' or '8' bits
3	-	SIZE

This is what gets put into the the "payload" portion of the packet. All of the messages used in the protocol have an ASCII letter in the first byte that identifies what the message type is. Each request has an uppercase letter and each acknowledgement has the corresponding lowercase letter.

The connection-request message is fairly simple: it includes the protocol version number and the number of bits wide that the client thinks that the communication channel is. The version number is currently always 0x01 and is included for cross-compatibility with future versions of the protocol. The channel width is encoded into either a '7' or an '8' ASCII character. The

client will think that the channel width is seven bits only if you tell it this on the command line.

When the server receives the connection request, it replies with the following message:

OFF	SIZ	DESC
0	1	code: ACK_CONNECT ('c')
1	1	protocol version := 0x01
2	1	transmit byte size: '7' or '8' bits
3	1	recommended request byte size: '7' or '8' bits
4	4	server maximum text-upload data size: H/M/M/L word
8	4	server maximum binary-upload data size: H/M/M/L word
12	4	server maximum text-download data size: H/M/M/L word
16	4	server maximum binary-download data size: H/M/M/L word
20	-	SIZE

The "protocol version" is what the server is using, currently always 0x01. The "transmit byte size" is the size that the user has specified on the command line that activated the server, and the "recommended request byte size" is a '7' if either the "transmit byte size" of the either the client or server is seven bits, or '8' otherwise. This is what should be used for the all subsequent messages that are exchanged.

The server's reply also includes the maximum packet sizes that it can handle for uploading and downloading binary and text files. The client then takes the "min" of the server's maximum packet sizes and its own, and uses the resulting maximum packet sizes for the rest of the file exchange session. The maximum packet sizes in the server's reply are all 32-bit unsigned integers that are stored from most-significant to least-significant bytes (big endian order). I picked big-endian order because that is the order used most commonly in inter-machine protocols.

The reason that the client doesn't have to inform the server of the client's maximum packet sizes in its connection message is that the maximum packet size to use is included with each request to get the next packet of a download file. It is sufficient that the client knows the full max-packet information. Really, the "transmit byte size" field isn't needed in the server reply message either, but I wanted the packet-size fields to be size-aligned.

After all of the file exchanging is completed, the client sends the following message to terminate the connection and return the server back to its command-line mode:

OFF	SIZ	DESC
0	1	code: REQ_DISCONNECT ('Q')
1	-	SIZE

When the server receives this request, it replies with:

OFF	SIZ	DESC

0	1	code: ACK_DISCONNECT ('q')
1	-	SIZE

And then exits like it should. Note that once the server exits, it cannot accept any more packets, since they would be sent to whatever command shell you use on your Unix system, and wouldn't do anything useful, so if the client sends the disconnect message but doesn't receive any reply, it will time out and tell the user that it couldn't disconnect cleanly from the server. This should be a rare occurrence. Anyway, what the user would do then is re-enter his terminal program and send Ctrl-X's at the server until it exits like it should have.

This arrangement allows us to avoid the famous(?) "two armies" problem that is inherent in disconnecting two connected processes: there is no "clean" way to do it. What systems like Z-Modem and Berkeley Sockets do is to have the server wait for a period of time that is longer than N times the timeout period of the client so that if there is a retransmission of the disconnection request, it likely that it will be received and processed correctly by the server. This is the reason (presumably) that Z-Modem does an annoying pause of 15 seconds or so after you finish transferring files. I think that my solution is much nicer, since the server can exit immediately (even though my server delays for 1 second, just so that your shell prompt will be cleanly in your modem's ARQ buffer when you re-enter your terminal program, if you have a hardware-flow-control modem).

4.2. FILE UPLOADING

Okay, so between connecting to and disconnecting from the server, actual

useful stuff happens, including uploading and downloading files. The uploading and downloading requests operate much like the regular file operations of open, close, read, and write. Really, the FX protocol makes the server program a special kind of file server.

When the client decides that it wants to upload a file, it first informs the server about this by sending the following message:

OFF	SIZ	DESC
0	1	code: REQ_UPLOAD_OPEN ('U')
1	1	data type: 't'=text file, 'b'=binary file: 'd'=directory
2	4	estimated file size: H/M/M/L word
6	2	permissions ("-----sgr:wxrwxrwx"), like Unix, H:L
8	12	modified date: BCD format: <YY:YY:MM:DD:hh:mm:ss:tt:tw:GG:gg:aa>
20	n	filename, null-terminated
20+n	-	SIZE

The "data type" field tells whether a text or binary file will be uploaded. There is a provision for "uploading" a directory entry (as part of uploading and downloading entire directory hierarchies), but support for this is not implemented yet. Also, it makes no difference to a Unix system whether a file contains text or binary data, but it may make a difference to other operating systems (like Mess-DOS). The "estimated file size" field isn't really used either, but it allows the server to make intelligent decisions about

pre-allocating space, buffering, etc., if it needed to. However, it is currently not filled in by the client, since file-size information is difficult to extract from Commodore-DOS. The file size is an unsigned 32-bit quantity.

The permissions field is currently not supported by the server, but it is intended to allow file permissions to be preserved when passing files from one system to another. The interpretation of the 16 bits of this field is like it is with the Unix operating system: "rwx" bits for the owner, group, and other, and execute-as-owner, execute-as-group bits. The owner-id and group-id fields aren't included since they are generally not portable across systems, and even if they were, we usually want to receive files as our own owner-id and our own group-id.

The "modification date" field is not currently filled in either, since this information is even harder to come across with Commodore-DOS, but when it is, it will have a 12-byte BCD format. The "YY:YY:MM:DD:hh:mm:ss" sub-fields should be easy enough to figure out, and the "tt:t" fields contain thousandths of seconds. The "w" field contains the day of the week, coded as 0-6 for Sunday to Saturday, and 7 for "unknown". The "GG:gg" fields contain the number of hours and minutes that your time zone is off from GMT. If the number is negative (in the western hemisphere), then the regular positive number of hours will be used, execept that the 0x80 bit of the hours byte will be set. Finally, the "aa" sub-field is used to encode the accuracy of the timestamp. The way that it is interpreted is that the time value is accurate to plus/minus 2^aa milliseconds. For example, if my clock were accurate to within one second, then this field would be set to 10 in BCD (2^10 == 1024ms). A value of 99 means "unknown" (or that the clock could be off by many billions of billions of years).

I decided to go all out in defining the date field so that it will be useful in the future when "world consciousness" will be much more important than it is today.

And last but certainly not least, the filename is encoded in ASCII with a trailing zero byte.

Upon receiving this request, the server will attempt to create a file according to your specifications, and will send back a reply of the form:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_OPEN ('u')
1	1	error code: 'y'=successful, 'n'=open unsuccessful
2	-	SIZE

The "error code" field tells whether the open operation was successful or not. If it was, then the client can continue with uploading its file; if not, then that file cannot be uploaded (and that the upload channel doesn't need to be closed). It's up to the client whether to go on to the next file, abort, or ask the user for help. The client will currently report an error to the user and then go onto the next file. Of course, it's likely that whatever caused the error in creating the current file will also cause an error in creating subsequent files (insufficient access permissions on the current

directory, disk full, etc.). The server will overwrite any existing file with the same name (since asking permission, etc., would require extra mechanism, and would probably be a nuisance anyway).

If the upload channel is opened successfully, then the packets of upload data should be sent to the server one at a time, until all of the data is uploaded. The client sends the following message to the server to upload a packet of data:

OFF	SIZ	DESC
0	1	code; REQ_UPLOAD_PACKET ('R')
1	1	upload sequence number
2	4	data length: H/M/M/L word
6	n	data
6+n	-	SIZE

The "upload sequence number", which was described before, is used to make sure that retransmissions of packets are detected and handled properly, so that each packet of data only appears in the file once. The "data length" field tells the number of user data bytes that follow in the packet, and then the actual user data bytes appear. The "data length" field is actually redundant, but I figured that it would make programming a little easier, and allows additional error checking. Normally, each upload-data packet will contain the maximum-packet-size number of bytes of user data (according to whether text or binary data is being uploaded), except for the last packet, which will contain the number of data bytes that are left in the file. However, each packet is allowed to contain anywhere from 1 to the maximum-packet-size number of bytes: whatever the client wishes to use. Variable-sized packets are a Good Thing (TM, Pat. Pend.). You will note that the data-size values are also what will be used for the "read" and "write" system calls on the client and server, respectively. I/O will be done in big, efficient chunks.

Upon receiving each upload packet, the server replies with the following acknowledgement message:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_PACKET ('r')
1	1	upload sequence number
2	-	SIZE

I don't think that the "sequence number" field is actually necessary here, but it is included to allow for future expansion and to provide redundancy for protocol-error checking.

When the client has uploaded all of the packets of the file currently being uploaded, it then sends the following message:

OFF	SIZ	DESC
0	1	code: REQ_UPLOAD_CLOSE ('V')
1	-	SIZE

This will close the upload channel and will finish writing the uploaded file to the Unix file system. The server will then respond with the following message to acknowledge the request:

OFF	SIZ	DESC
0	1	code: ACK_UPLOAD_CLOSE ('v')
1	4	number of bytes uploaded: H/M/M/L word
5	-	SIZE

The "number of bytes" field is actually redundant, but is used for additional error checking.

4.3. FILE DOWNLOADING

Downloading files is analogous to uploading them: first we open the download channel/file, then we download the packets, and then we close the download channel.

To open the download channel, the client sends the following request to the server:

OFF	SIZ	DESC
0	1	code: REQ_DOWNLOAD_OPEN ('D')
1	-	SIZE

To which the server replies with:

OFF	SIZ	DESC
0	1	code: ACK_DOWNLOAD_OPEN ('d')
1	1	data type: '0'=no more files (eom),'t'=text,'b'=bin,'e'=err,'d'=dir
2	4	estimated file size: H/M/M/L word
6	2	permissions ("-----sgr:wxrwxrwx"), like Unix, H:L
8	12	modified date: BCD format: <YY:YY:MM:DD:hh:mm:ss:tt:tw:GG:gg:aa>
20	n	filename, null-terminated
20+n	-	SIZE

The file information is the same as for opening an upload file, except that there are more possible return conditions, and all of the "meta data" fields are actually filled in by the Unix host (since this information is actually conveniently available via the "stat" system call).

If the server replies with a '0' "data type" code, then this means that the server has no more files to offer for downloading. The filenames to download are taken one at a time, from left to right, from the command line that was used to start the server. When the server runs out, then the downloading session is complete and the client disconnects (since the client uploads its files first).

Alternatively, the server could reply with a 'e' code, which means that it could not open the next filename given on its command line. An error

return is generated so that the client can inform the user that the file could not be downloaded. This will normally result from the user giving a bad filename on the command line. The client will continue the downloading process by closing the download channel (below) asking for the next file by re-opening the download channel. The download channel needs to be closed on this condition since otherwise there would be no way of distinguishing retransmissions from new requests at the server.

Finally, the server can reply with a 't' or 'b' code ('d' for directories is not currently implemented) indicating that the file was correctly opened and is either text or binary (as specified on the server's command line). Of the meta information about the file, only the filename and file size are currently used: the file is named according to the given name, translated to PETSCII and truncated to 16 characters, and the file size is reported to the user so that he can monitor downloading progress. I am not sure what to do yet about name collisions on the Commodore end: either ask the user whether to overwrite the file, automatically overwrite the file anyway, or automatically give the file a slightly different name and download normally. I think that for the time being, I will just overwrite the existing file. This will mean that you'll want to be extra careful in putting the filenames onto the correct command line (the client's or the server's), although there won't be a problem if the file doesn't exist on the machine whose command line you put the name on.

When the file handling is all squared away and the download channel is opened, the client then sucks packets out of the file until the end of the file is reached. The packets are sucked out with the following request:

OFF	SIZ	DESC
0	1	code: REQ_DOWNLOAD_PACKET ('S')
1	1	download sequence number
2	4	maximum acceptable data length: H/M/M/L word
6	-	SIZE

The "download sequence number" is used to distinguish retransmissions from requests for new packets, and the client tells the server the "maximum acceptable data length" for the reply packet. Although the max-packet information is actually static during the connection, I included it here in every "read" request since I didn't really want the server to keep that particular bit of "state" internally.

The server replies to the download-packet request with the following message:

OFF	SIZ	DESC
0	1	code: ACK_DOWNLOAD_PACKET ('s')
1	1	download sequence number
2	4	data length: H/M/M/L word, 0==EOF
6	n	data
6+n	-	SIZE

This is the only "large" message that the server can produce. It includes the sequence number, the number of bytes that are actually included, and the user data. The number of data bytes in the packet is allowed to be smaller than

the number of bytes requested, but this is normally only the case for the last packet of the file.

To indicate that the end of file has been reached and that no more user data is available, the server will return a download packet with zero bytes of user data in it. Upon receiving this, the client will close the download channel with the following message:

OFF	SIZ	DESC
0	1	code: REQ_DOWNLOAD_CLOSE ('E')
1	-	SIZE

And the server will reply with:

OFF	SIZ	DESC
0	1	code: ACK_DOWNLOAD_CLOSE ('e')
1	4	number of file bytes downloaded: H/M/M/L word
5	-	SIZE

The "number of file bytes downloaded" field is redundant but included for additional error checking. After closing a file, the client will then ask for the next file, or will disconnect if the last file to download was just closed.

4.4. ERROR HANDLING

With all of the server calls except for disconnecting (discussed earlier), the is the possibility that either the request message from the client or the reply message from the server will become garbled and be dropped by the packet-delivery layer of the software. To recover from this, if the client detects an extended period of inactivity on the serial line for received data (where "extended period" is defined as being "about five seconds"), then the client will assume that something went wrong and it will retransmit the request.

As pointed out way above, there are two possible reasons for a retransmission being needed: either the request packet was corrupted and dropped, or the reply packet was corrupted and dropped. In the format case, the request wasn't processed by the server, but in the latter case, it was. Since we don't want the server to perform an file operation twice (this is really what the six file-transfer client operations really boil down to from the server's perspective), the server must keep four pieces of internal state: the last upload sequence number, the last download sequence number, whether the upload file is open, and whether the download file is open.

If an upload-open request is received and the file to be uploaded is not open, the the request must be a new one and the server processes it and sends back a reply like normal. If an upload-open request is receive and the upload file IS currently open, then it must be the case that the current request is a retransmission, so all theat the server needs to do is to give a positive reply without performing any internal file operations. The same holds true for the download-open call and for both of the close calls (except that the

operation has already been processed if the file is CLOSED).

For the packet-upload and packet-download requests, sequence numbers are used to detect duplicates. You will note that these sequence numbers are distinct from one another, and, in fact, that the entire upload and download file-transfer channels are distinct and independent from each another. This is to allow for the future possibility of simultaneous file uploading and downloading. In fact, if stream numbers (file descriptors) were added to the open/read/write/close requests, then we could have us a full-blown remote-host over-the-phone interactive file server. But anywho, sequence numbers start from 0x00 for the first packet transferred and increment modulo 256 from there.

Note that for high-speed data-compression modems (like I have) that already include error detection and recovery at a level hidden from the user, the FX protocol will work particularly well: there will never be an error, never be a timeout delay, and never be a retransmission. And, really, the CRC-32 error computation and checking is pretty much a zero cost. But, if something does go wrong, outside of the modem-to-modem connection, the FX protocol is right there to pick up the pieces and carry on.

6. CONCLUSION

You'll have to wait to get your hands on the program. The Unix Server program is almost 100% (except for a few design changes that I made while writing this document), and the ACE program is implemented except for the error handling and text conversion. Both programs will be released with the next release of ACE, which will be Real Soon Now (TM).

Here is my performance testing so far, using my USR Sportster modem over a 14.4-kbps phone connection, with a 38.4-kbps link to my modem from my C128, to my usual Unix host:

Using FX to/from the ACE ramdisk, REU:

Download 156,260 bytes of ~text: time= 54.1 sec, rate=2888 cps.
Download 151,267 bytes of tabular text: time= 45.9 sec, rate=3296 cps.
Download 141,299 bytes of JPEG image: time= 92.5 sec, rate=1528 cps.
Upload 156,260 bytes of ~text: time= 57.4 sec, rate=2722 cps.
Upload 151,267 bytes of tabular text: time= 45.3 sec, rate=3339 cps.
Upload 141,299 bytes of JPEG image: time= 95.0 sec, rate=1487 cps.

Using FX to/from my CMD Hard Drive:

Download 156,260 bytes of ~text: time= 83.4 sec, rate=1874 cps.
Download 151,267 bytes of tabular text: time= 75.4 sec, rate=2006 cps.
Download 141,299 bytes of JPEG image: time=118.2 sec, rate=1195 cps.
Upload 156,260 bytes of ~text: time= 77.9 sec, rate=2006 cps.
Upload 151,267 bytes of tabular text: time= 66.2 sec, rate=2285 cps.
Upload 141,299 bytes of JPEG image: time=114.2 sec, rate=1237 cps.

Using DesTerm-128 v2.00 to/from my CMD Hard Drive, Y-Modem:

Download 156,260 bytes of ~text: time=189.5 sec, rate= 824 cps.

Download 151,267 bytes of tabular text: time=180.4 sec, rate= 839 cps.
Download 141,299 bytes of JPEG image: time=199.9 sec, rate= 707 cps.
Upload 156,260 bytes of ~text: time=255.1 sec, rate= 611 cps.
Upload 151,267 bytes of tabular text: time=238.6 sec, rate= 634 cps.
Upload 141,299 bytes of JPEG image: time=233.0 sec, rate= 606 cps.

Using NovaTerm-64 v9.5 to my CMD Hard Drive, Z-Modem, C64 mode:

Download 156,260 bytes of ~text: time=245.8 sec, rate= 636 cps.
Download 151,267 bytes of tabular text: time=230.0 sec, rate= 658 cps.
Download 141,299 bytes of JPEG image: time=262.6 sec, rate= 538 cps.

(There is no Z-Modem uploading support)

So there you have it: my simple protocol blows the others away. QED.