Date: April 3, 1986
Author: Mike Askins
Subject: Cortland Serial Ports ERS
Document Version Number: 00:40

Revision History

Ver	Date	Author	Changes or Additions
00:00	14-Jun-85	FAB	Initial Release
00:10	05-Nov-85	MSA	Updated command description (deleted delay commands) Added handshaking and compatibility discussion
00:20	21-Feb-86	MSA	Added sections on buffering, background printing, extended interface Updated command description Added terminal mode description
00:30	27-Mar-86	MSA	Major changes to the extended interface; calls deleted, altered, added Recharge routine is called with DBR=$00 Added sections 'Using the Serial Firmware', 'Error Handling', and 'Interrupt Notification' Minor textual changes for clarification purposes
00:40	03-Apr-86	MSA	Mode Bits image corrected (btm half of page 14), Pascal Interface Note added (btm of page 4), 1st paragraph of page 9 changed.

Overview

The serial ports in Cortland use a two channel Zilog Serial Communications Controller chip (SCC 8530) and RS422 drivers. The driver firmware emulates the functionality of the Super Serial Card, and the Apple //c serial firmware, supports input/output buffering, as well as background printing. The firmware also implements a number of calls that the application can make to control the new features.

Input/output buffering and background printing are done on an interrupt basis and can use any buffer (size up to 65K, any location) that the application wishes. I/O buffering is transparent for BASIC and PASCAL. Background printing is started by an application, and the firmware transfers control back to the application when all the data has been sent to the printer.

Note that AppleTalk when active requires the use of one of the serial channels. Therefore only two of the three (AppleTalk, serial port 1, and serial port 2) are allowed to be active at any one time. (The Control Panel program ensures that at least one serial port is made inactive when AppleTalk has been selected). An attempt to initialize the serial firmware when the channel is being used by AppleTalk will fail. Both port 1 and 2 can be configured as either a printer or a communication (modem) port.

Compatibility

Though the commands used to communicate with the serial firmware are the same as the Super Serial Card and //c ports, we can expect that many existing programs using these ports *will not* be compatible with the Cortland serial ports. The reason is that a lot of programs, particularly communications packages (Access //, flaming example) go to the hardware directly, and the hardware is no longer a 6551. Print programs are more likely to work, as well as applications written in BASIC and PASCAL. Both AppleWorks and MousePaint are examples of programs which use the firmware and are compatible.

A source of quasi-incompatibility will be experienced by those who communicated with the firmware by modifying the screenholes. None of these screenhole pokes are guaranteed to work (although some might), and will almost certainly not work with any future serial firmware. The situations which caused programmers to have to resort to altering screenholes have all (hopefully) been eliminated by the implementation of the extended interface call.

Applications should make correct use of the BASIC interface. $CN00 is the initialization call (and also happens to output the character in the accumulator), $CN05 is the call to get a character, and $CN07 is the call to output a character. Note that the practice of sending characters to the firmware by calling $CN00 repeatedly is to be avoided. This will currently work (to a degree), but applications which do this are living on borrowed time, since it is almost certain that future firmware will not permit this practice.

One last difference between the Cortland serial firmware and the others is in the handling of errors. If a character with an error is received by either the SSC or the //c firmware, it is not deleted from the input stream. The Cortland firmware will delete the character from the input stream, and set a bit in the mode bytes to record the fact that an error was encountered. Applications therefore can be aware that trouble is being experienced on the serial line, but be spared the reception of errant data.

Using the Serial Firmware

The serial firmware interface really consists of two separate interfaces, one for BASIC, and one that adheres to the Pascal 1.1 firmware protocol.

Please Note: All calls to the serial firmware must have the 65816 Data Bank Register set to $00. Additionally, the processor must be in 6502 emulation mode at the time of the call (E bit = 1). All entry points are in the $Cn00 space in bank $00 (This note applies to all calls to port firmware.).

The BASIC interface has the following entry points ('n' represents the slot number, 1 or 2) :

$Cn00	BASIC Initialization	(Also outputs character in Acc)
$Cn05	BASIC Read Character	(Character returned in Acc; X,Y preserved)
$Cn07	BASIC Write Character	(Character passed in Acc; X,Y preserved)

These entry points are used by BASIC. When the user types 'IN#n', or 'PR#n', BASIC makes a call to $Cn00 after setting either the KSWL or CSWL hooks to $Cn00. When the serial firmware gets control it alters the hooks so that they point to the Read and Write routines.

The 'Pascal 1.1' interface is a much more flexible interface and it is recommended that machine language programmers use this interface to communicate with the serial firmware. The Pascal 1.1 protocol uses a branch table in the $Cn00 page indicating where each of the service routines begin. This table contains bytes specifying where in the $Cn00 page each of the service routines begins. The branch table locations are:

Address	Contains
$Cn0D	Initialization routine offset
$Cn0E	Read routine offset
$Cn0F	Write routine offset
$Cn10	Status routine offset
$Cn12	Control routine (extended interface) offset

To reach the Read routine for example, an application reads the value found at $Cn0E. For purposes of illustration suppose that this value was found to be $18. To reach the Read routine in this case, the application would have to do a JSR instruction to the address $Cn18. Please see the Super Serial Card p.50 for a specification of the init, read, write and status routines, and the section in this document for the spec on the control routine.

Note: The system defaults for the Pascal interface assume that the application supplies a linefeed after carriage return. If the application does not wish to supply the linefeeds, it should send the 'LE' linefeed generation call described in the next section.

Standard Serial Commands

Cortland, as well as it's predecessors the SSC and //c, supports control commands embedded in the serial output flow, whether through the BASIC or PASCAL output routine. These are distinguished from normal output by the detection of the "command character". The firmware accepts commands in the following command sequence:

<CommandChar> <CommandString>

When a port is in printer mode, CommandChar is a control-I, and in communication (modem) mode it is a control-A. The command character can be changed at any time by sending the current command character followed by a control character. The CommandString is a letter command sometimes prefixed by a number or suffixed with an E or D. Commands can be sent directly from BASIC by typing the commands from the keyboard after issuing a PR#n. They can also be sent through the Pascal WriteChar call.

Except where noted, these commands function just as the equivalents in the SSC and Apple //c serial firmware. For a further description of the commands on the following page, please refer to the description in the Super Serial Card documentation.

Command Strings:

<n>B — **Set the baud rate to value corresponding to n**

0B=	(use default)	6B=	300 baud	12B=	4800 baud
1B=	50 baud	7B=	600 baud	13B=	7200 baud
2B=	75 baud	8B=	1200 baud	14B=	9600 baud
3B=	110 baud	9B=	1800 baud	15B=	19200 baud
4B=	134.5 baud	10B=	2400 baud		
5B=	150 baud	11B=	3600 baud		

<n>D — **Set data format to values per n (data bits, stop bits)**

0D=	8 data, 1 stop	4D=	8 data, 2 stop
1D=	7 data, 1 stop	5D=	7 data, 2 stop
2D=	6 data, 1 stop	6D=	6 data, 2 stop
3D=	5 data, 1 stop	7D=	5 data, 2 stop

<n>P — **Set parity to n**

0P=	none	2P=	none
1P=	odd	3P=	even

(Note that MARK and SPACE parity are not supported)

<n>N — **Set Line Length to n**
This call does not enable line formatting; use the C command to do this.
Note that formatting is disabled when n = 0.

C<E/D> — **Enable Line Formatting**
The firmware issues a carriage return at the end of a line. The length of the line is set by the N command

T<E/D> — **Enable/Disable BASIC tabbing**

B<E/D> — **Enable/Disable I/O buffering**

X<E/D> — **Enable/Disable XON/XOFF handshaking protocol**

XE=	When an XOFF is detected XOFF, await XON before transmitting
XD=	Ignore XOFF

F<E/D> — **Enable/Disable keyboard input**

FE=	Insert keystrokes into serial input stream
FD=	Disable

E<E/D> — **Echo input to screen**

M<E/D> — **Enable/Disable filtering of linefeeds after carriage returns**

L<E/D> — **Add linefeed after carriage return**

R — **Reset the SCC and input/output hooks**

S — **Transmit a 233 millisecond break (all zeros)**

T — **Enter terminal mode**

Q — **Exit terminal mode**

Z — **Zap control character interpretation**
Tell firmware to stop interpreting command sequences. This effects of this command can only be reversed by an extended call (described below).

Terminal Mode

The Cortland serial firmware supports a minimum feature terminal emulation program which behaves just as its counterpart in the //c firmware. Terminal mode is a "bare bones" terminal routine which can be used when a full feature communications package is unavailable. While in terminal mode, all of the characters typed are passed to the serial output (except the command strings), and all serial input goes directly to the screen.

Terminal mode is entered through the BASIC interface. Initialize the firmware by typing **IN#n**. Then type the current command character followed by a **'T'**. The prompt character will change to a flashing '_' indicating that terminal mode is active. Exit terminal mode by typing the current command character followed by a **'Q'**. The Apple //c's 'remote' mode is also identically supported.

Terminal mode can be used with buffering enabled to minimize character loss at higher baud rates due to the overhead incurred in screen scrolling. Invoke buffering with the **'BE'** serial command.

Handshaking

The history of RS232 handshaking at Apple has been contorted to say the least. Limitations in the number of connector pins as well as compatibility between machines are the major reasons for the confusion. ACIA hardware constraints added to the confusion (for example, the 6551 shuts down the transmitter if the input handshake line is disasserted).

Due to the variety of ways that data communcations equipment manufacturers have defined the handshaking lines, the handshaking is designed to be very flexible. You can select any combination of the following options:

DTR/DSR Characters will be transmitted only when the input handshake line (DSR) is asserted (hi), and the output handshake line (DTR) will be used to tell the device when the host is ready to accept data. If this option is not enabled, the input handshake line will not be checked on transmit, and the DTR line won't be toggled. This mode could be referred to as 'hardware' handshake. (The DSR line is pin 2 of the serial connector, and the DTR line is pin 1.)

DCD Characters will only be transmitted when the DCD line (GPI line, pin 7 on the serial connector) is asserted. It has no direct effect on receiving characters. This mode is provided for compatibility with the Super Serial Card which handshakes this line.

XON/XOFF This mode could be referred to as the 'software' handshake. If an XOFF character ($13) is ever received, no characters will be transmitted until an XON character ($11) is received. It operates independently from the hardware handshake.

Error Handling

Every time the firmware gets a character from the hardware it checks the error status register. If this character is found to have a framing or parity error (assuming that the parity option is not set to 'none') it is deleted from the input stream and the appropriate mode bit is set. An application can make the GetModeBits call to read these two bits (one for framing errors and one for parity errors) to determine that at least one receive error has occurred. After the application has read these bits it should clear them (using SetModeBits) so that future errors can be detected. It is recommended that error checks be done periodically and the user notified if the receive data is being corrupted so that he can do something about it.

Buffering

The firmware supports transparent input and output buffering as well as background printing. Each port has two buffers, one for input and one for output. The firmware tries to allocate 2048 bytes for these buffers, and if there is not enough memory to do this, 128 bytes are allocated. If the application wishes to use a buffer larger than this, it must pass the firmware its address and length via the SetInBuffer or SetOutBuffer commands in the extended interface.

Buffering can be turned on by the user from the control panel, from the keyboard after a PR#n command from BASIC, or by the application using the 'BE' command through the output flow. In this mode, characters sent to the firmware are placed in a FIFO queue in the output buffer space, and are only sent out on an interrupt basis whenever the hardware is ready to send another character. The XON and XOFF characters are not queued; they are sent immediately through the channel so that the desired effect of shutting off characters being received is immediate. Characters received in buffering mode are placed in the input queue and all read calls return characters from the queue. As with transmit, any XON and XOFF characters received are not queued; they are absorbed by the firmware and cause the output flow to be halted or resumed.

In order to make input buffering work transparently, the firmware takes control of the handshake. When the input queue becomes more than 3/4 full the firmware disasserts the handshake. This may mean sending an XOFF character (if XON/XOFF handshaking is enabled) or disasserting the DTR line (if DSR/DTR handshaking is enabled). The application can determine that the handshake has been disasserted by inspecting the "input flow" mode bit using the GetModeBits call in the extended interface. The firmware reasserts the handshake as soon as the receive queue fills less than 1/4 of the input buffer.

It is possible for the application to determine the amount of characters in the input queue, or the amount of room left in the output queue through the InQStatus and OutQStatus commands in the extended interface. In addition, the InQStatus call also returns the amount of time elapsed since the last character was queued. This permits the application to keep track of the activity level of the input stream even though it is not involved in the interrupt process.

Note that certain items are not buffered. In general all characters except those involved in command strings are buffered. If XON/XOFF handshaking is enabled, XON characters (ASCII $11) and XOFF characters (ASCII $13) are not buffered and are sent immediately into the output flow. The immediate (non-buffered) execution of command sequences requires that applications do command sequences before (and not during) the output of the data, which is the way most applications want to behave anyway.

Interrupt Notification

When a channel has buffering enabled the firmware sevices all interrupts that occur on that channel. If an application wishes to service interrupts for a given channel itself, it should disable buffering using the SetModeBits command in the extended interface. If the buffering mode bit is off, the serial firmware will not process any interrupts; the system interrupt handler will transfer control to the user's interrupt vector at $3FE in bank $00 (This is the ProDOS user interrupt vector.). The user's interrupt handler is then completely responsible for all SCC interrupt service.

If the application does not wish to disable buffering, but does wish to be *notified* that a certain type of serial interrupt has occurred, it can instruct the firmware to pass control to the $3FE vector after it has serviced the interrupt. The application tells the firmware when it wishes to be notified by using the SetIntInfo call. This call guarantees that the user interrupt handler will get control when a specific type of interrupt occurs, but only after the serial firmware has processed and cleared the interrupt. The application then uses the GetIntInfo call to determine which interrupt condition occurred.

As an example of when interrupt notification would be desirable, imagine a typical terminal emulator program. It probably wishes to do input and output character buffering, handshaking and the like. It would like the firmware to handle all of these details, but might also need to get interrupted when a break character is received. The application sets the break interrupt enable through the SetIntInfo Call, and whenever a break character is received the firmware SCC interrupt handler records and clears the interrupt, finally passing control to the user interrupt handler. The user's interrupt handler then calls GetIntInfo, and if the 'break' bit is set the interrupt handler knows that a break interrupt was serviced.

It is important to realize that all of the interrupt sources (except receive and transmit) cause an interrupt on the *transition* of a given signal; any user's interrupt handler will get control passed on both positive and negative transitions of the signals of interest. For example, a break character sequence will cause two interrupts, one at the beginning of the sequence and one at the end. The user's interrupt handling routine should take this into account. A routine can always determine the *current* state of the bits of interest using the the GetPortStat command.

Background Printing

The firmware has the capability of sending a block of characters out a serial channel on an interrupt basis in a manner essentially transparent to a running application. Background printing is really just output buffering as described in the section on buffering, the major difference being that the firmware is handed a large number of characters to transmit at one time rather than getting them one at a time.

To launch a background printing process, an application needs to perform the following steps:

1) Make sure the firmware/hardware is active by doing an Init call through the Pascal interface. The hardware characteristics (baud rate, data format, etc.) will be as specified in the Control Panel.

2 Use GetModeBits and SetModeBits to disable buffering in case the user has set the control panel to enable it.

3) If it is desired to change the port characteristics, do so at this point; use either the SetModeBits call or send commands through the output flow.

4) Set the output buffer using SetOutBuffer. If the default buffer is the one to be used, make a call to GetOutBuffer to ascertain its location.

5) Load the data into the buffer.

6) Launch the process with SendQueue, passing the length of the data in the buffer and the address of the 'Recharge' routine.

The 'Recharge' Routine

After the SendQueue call characters will be sent periodically in the background until the buffer is exhausted. As the last character is removed from the buffer, a JSL is made to the 'Recharge' address passed when the SendQueue call was made. This application supplied 'Recharge' routine should do whatever is necessary to reload the buffer with the next set of data to be output. This might involve some disk activity if the application is background printing from disk. Finally the routine loads the number of bytes in the new block of data to be sent into the X and Y registers (these will both be zero in the case that background printing is completed), and does an RTL. The complete requirements for the Recharge routine are as follows:

On Entry:
System Speed = Fast
DBR = $00
Native mode, 8 bit M,X (E=0)

On Exit:
System Speed = Fast
DBR = $00
Native mode, 8 bit M,X (E=0)
X reg = data size (lo)
Y reg = data size (hi)

Note that the Recharge routine is called at interrupt time; it should be regarded as an interrupt handler in the sense that anything it changes it must restore. Also realize that since interrupts are disabled during the time that the Recharge routine executes, spending a lot of time in this routine will cause performance degradation of 'interrupt critical' processes (those that have stringent interrupt response requirements, like AppleTalk).

Extended Interface

To support the various firmware capabilities that are not present in either the SSC or the //c, the Cortland has a call made through the $CN00 space called the extended interface call.

To make a call through the extended interface, first ascertain the dispatch address by taking the value at $CN12 and adding it to $CN00. This byte is the 'optional control routine offset' of the Pascal 1.1 protocol (discussed in the Super Serial Card manual). Do an emulation mode JSR to this dispatch address with the registers loaded with the address of the command list:

Register	Contents
A	Address of cmdlist (lo)
X	Address of cmdlist (med)
Y	Address of cmdlist (hi)

The format of the command list is fairly regular. Every command list starts with a one byte parameter count (not a byte count), a command code, and space for a result code. The possible result codes returned are specified in the 'Errors' section. The calls fall into three main groups, calls associated with the hardware, mode control and buffering.

In the description of the calls, a DFB is an assembler directive producing a single byte, a DW produces a double byte (16 bits- low byte, high byte), and a DL produces a double word (32 bits- low, med low, med high, and high).

Compatibility note: An application writer affords him/herself a higher level of confidence that their application will work on future systems by limiting the use of the hardware control calls. If future systems use hardware other than the current serial chip (SCC 8530), the hardware control calls are the ones most likely to have to be changed, and applications using these calls could be rendered incompatible.

Mode Control Calls

GetModeBits — Returns the current mode bit settings

```
CmdList   DFB   $03      ;Parameter Count
          DFB   $00      ;Command Code
          DW    $00      ;Result Code (output)
          DL    $00      ;Mode Bit Image (output)
```

This call allows the application to determine the status of various operating modes of the firmware. Four bytes (32 bits) of mode information are returned. To change any of these bits, use this call to get the current settings, alter the bits of interest, and then use the SetModeBits call to do the actual modification. (To avoid race conditions in this process, be sure to disable interrupts [SEI] before the GetModebits call, and reenable [CLI] them after the SetModeBits call.). The meaning of each bit is described below.

SetModeBits — Sets the mode bits

```
CmdList   DFB   $03          ;Parameter Count
          DFB   $01          ;Command Code
          DW    $00          ;Result Code (output)
          DL    ModeBitImage ;(input)
```

Use this call to alter any of the mode bits whose function is described above. First read in the bits using GetModeBits, alter the bits of interest, and then write the bits by using this call, noting the note about interrupts in the GetModeBits description. The bits marked 'preserve' should not be changed.; they are informational only. Altering these bits will confuse the firmware.

ModeBitImage: (4 bytes, bit 0 is the lsb of the lowest addressed byte, bit 31 is the msb of the highest)

[31]	1 = Ignore commands in the output flow
[30]	1 = Framing error has occurred
[29]	(preserve)
[28]	1 = Parity error has occurred
[27..24]	(preserve)
[23..16]	(preserve)
[15]	(preserve]
[14]	(preserve) 1 = I/O buffering enabled
[13]	1 = DCD handshaking enabled
[12]	(preserve)
[11]	1 = Generate CR at end of line
[10]	(preserve) 1 = Input flow halted
[9]	(preserve) 1 = Output flow halted
[8]	(preserve) 1 = Background Printing in progress
[7]	1 = Echo input to the video screen
[6]	1 = Generate LF after CR
[5]	1 = XON/XOFF handshaking enabled
[4]	1 = Accept keyboard input
[3]	0 = Delete LF after CR
[2]	1 = DTR/DSR handshaking enabled
[1]	(preserve) 1 = awaiting XON character
[0]	(preserve) 1 = communication mode, 0 = printer mode

Buffer Management Calls

GetInBuffer

Return the address and length of the input buffer

```
CmdList   DFB   $04          ;Parameter Count
          DFB   $10          ;Command Code
          DW    $00          ;Result Code (output)
          DL    $00          ;BufferAddress (output)
          DW    $00          ;BufferLength (output)
```

This call and the one which follows are used to determine the addresses and lengths of the current input and output buffers. If background printing is to be invoked and the application wants to use the default buffer, its address can be retrieved by these calls.

GetOutBuffer

Return the address and length of the output buffer

```
CmdList   DFB   $04          ;Parameter Count
          DFB   $11          ;Command Code
          DW    $00          ;Result Code (output)
          DL    $00          ;BufferAddress (output)
          DW    $00          ;BufferLength (output)
```

SetInBuffer

Specify the buffer to contain the input queue

```
CmdList   DFB   $04          ;Parameter Count
          DFB   $12          ;Command Code
          DW    $00          ;Result Code (output)
          DL    BufferAddress ;(input)
          DW    BufferLength  ;(input)
```

This call and the one following allow the application to change the location and/or length of the input or output buffers. A queue buffer can cross bank boundaries but must be fixed in memory while buffering is active.

SetOutBuffer

Specify the buffer to contain the output queue

```
CmdList   DFB   $04          ;Parameter Count
          DFB   $13          ;Command Code
          DW    $00          ;Result Code (output)
          DL    BufferAddress ;(input)
          DW    BufferLength  ;(input)
```

FlushInQueue Discard all the characters in the input queue

```
CmdList   DFB   $02      ;Parameter Count
          DFB   $14      ;Command Code
          DW    $00      ;Result Code (output)
```

These two calls allow the application to flush unwanted data from either queue.

FlushOutQueue Discard all the characters in the output queue

```
CmdList   DFB   $02      ;Parameter Count
          DFB   $15      ;Command Code
          DW    $00      ;Result Code (output)
```

InQStatus Returns info about the input queue

```
CmdList   DFB   $04      ;Parameter Count
          DFB   $16      ;Command Code
          DW    $00      ;Result Code (output)
          DW    $00      ;# chars in receive queue (output)
          DW    $00      ;Time since last receive char queued (output)
```

These calls return information about the input or output queues. The InQStatus call additionally returns the number of heartbeat ticks (1 tick = 1/30 second) between the time of the queueing of the last character and the time of the call. Note that for this number to be valid the application must have turned on the heartbeat system by making a tool call. See the miscellaneous tool manager ERS for information about how to do this.

OutQStatus Returns info about the output queue

```
CmdList   DFB   $04      ;Parameter Count
          DFB   $17      ;Command Code
          DW    $00      ;Result Code (output)
          DW    $00      ;# chars 'til transmit queue overflow (output)
          DW    $00      ;Reserved (output)
```

SendQueue Launch background printing

```
CmdList   DFB   $04          ;Parameter Count
          DFB   $18          ;Command Code
          DW    $00          ;Result Code (output)
          DW    DataLength
          DL    RechargeAddress
```

This call begins the background printing process. The application must first set the output buffer address (or use the default buffer) load the data that it wishes to be output into the buffer starting at the buffer base address. Then the data is placed into the buffer, and the call to SendQueue is made specifying the length of the data in the buffer and the four byte address of a subroutine (the 'recharge' routine) which will be called by the interrupt firmware when the all the characters have been sent. (See the description of background printing elsewhere in this document for a description of the Recharge routine.)

Hardware Control Calls

(Please read the compatibility note at the beginning of this section.)

GetPortStat

Returns the port hardware status

```
CmdList   DFB   $03      ;Parameter Count
          DFB   $06      ;Command Code
          DW    $00      ;Result Code (output)
          DW    $00      ;Port Status Info (output)
```

This call is used to get the current status of the serial channel at the hardware level. There are 16 bits of result, and the meaning of these bits is outlined below.

[15..8]	(reserved)	
[7]	Break/Abort	Set to 1 when a break sequence is detected
[6]	Tx Underrun	Set to 1 when a transmit underrun occurs
[5]	DSR	State of the input handshake line
[4]	(reserved)	
[3]	DCD	State of the General Purpose Input line
[2]	Tx Buff Empty	Set to 1 when ready to transmit next character
[1]	(reserved)	
[0]	Rx Char Avail	Set to 1 when a character is available to be read

GetSCC

Return the value of the specified SCC register

```
CmdList   DFB   $03      ;Parameter Count
          DFB   $08      ;Command Code
          DW    $00      ;Result Code (output)
          DFB   Register ;SCC register number (input)
          DFB   $00      ;Value of SCC register (output)
```

The GetSCC returns the value in a specified SCC register. The Get/SetSCC calls are provided to allow direct access to the serial hardware, when this is deemed necessary. See the SCC 8530 technical manual for a description of the registers in the serial controller chip. The serial firmware does not need to be initialized for these calls to work; in fact it is suggested that these calls only be used if the application is handling *all* serial tasks itself, and not using the firmware at all.

SetSCC

Write a value into the SCC

```
CmdList   DFB   $03      ;Parameter Count
          DFB   $09      ;Command Code
          DW    $00      ;Result Code (output)
          DFB   Register ;SCC register to write (input)
          DFB   Value    ;Value to write to Register (input)
```

This call allows the writing of a register in the SCC.

GetDTR

Return the value of the output handshake line

```
CmdList   DFB   $03       ;Parameter Count
          DFB   $0A       ;Command Code
          DW    $00       ;Result Code (output)
          DW    $00       ;Bit 7 is the state of DTR (output)
```

Use this call to find out the current setting of the output handshake line. The state of this line is returned in the msb of the returned byte. The line may be set by the following call.

SetDTR

Set the value of the output handshake line

```
CmdList   DFB   $03       ;Parameter Count
          DFB   $0B       ;Command Code
          DW    $00       ;Result Code (output)
          DW    DTRState  ;Bit 7 is the state of DTR (input)
```

GetIntInfo

Return the informational interrupt setting

```
CmdList   DFB   $03       ;Parameter Count
          DFB   $0C       ;Command Code
          DW    $00       ;Result Code (output)
          DW    $00       ;(output)
```

This call allows the application to read the byte written by the SetIntInfo call.

SetIntInfo

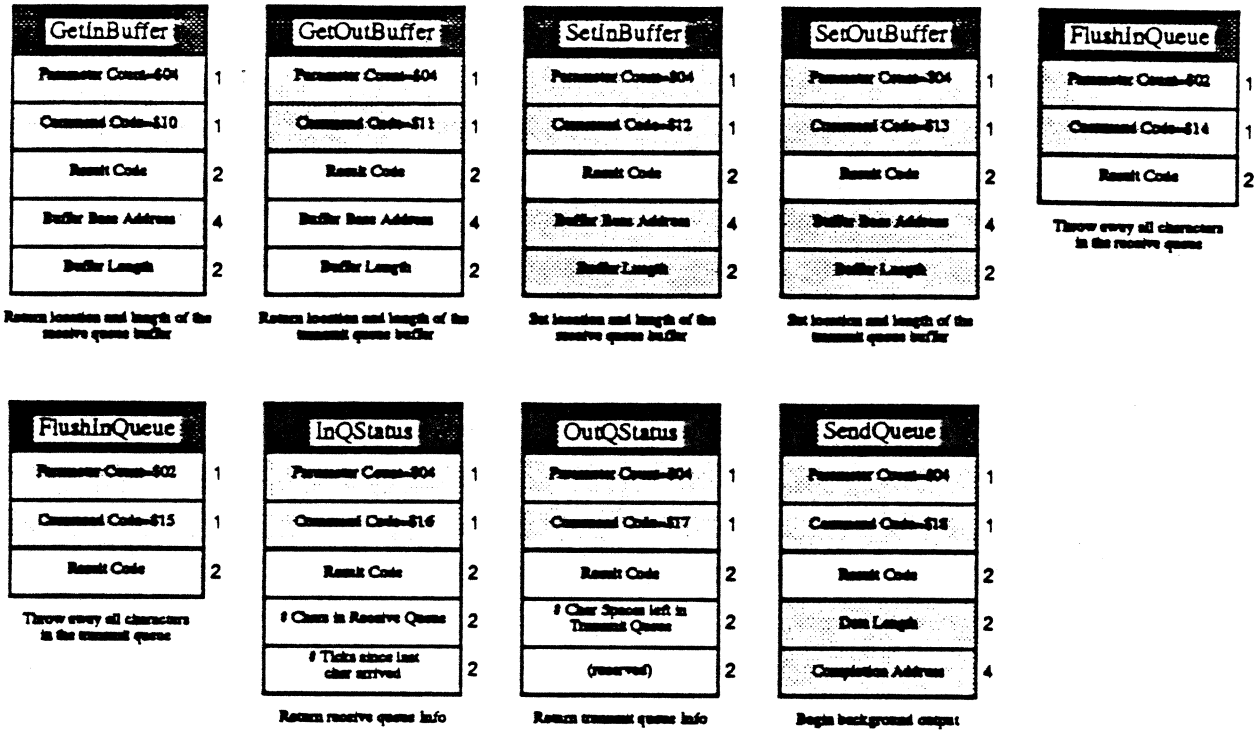
Set up informational interrupt handling

```
CmdList   DFB   $03              ;Parameter Count
          DFB   $0D              ;Command Code
          DW    $00              ;Result Code (output)
          DW    InterruptSetting ;(input)
```

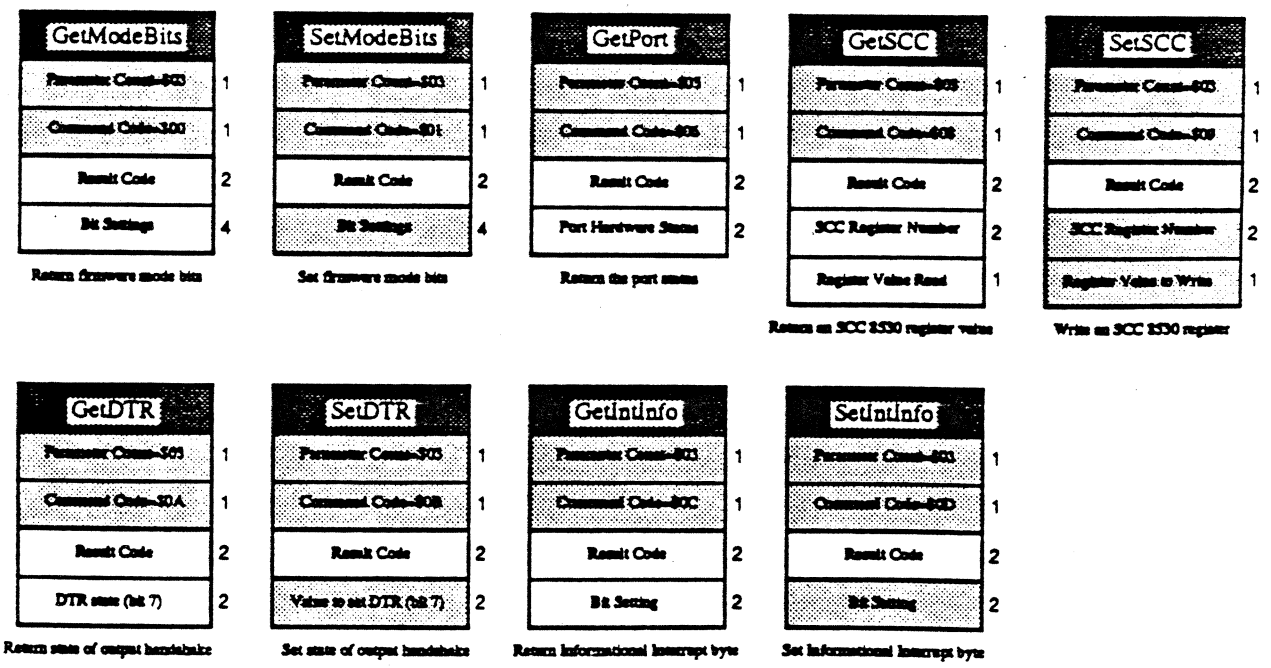
This call allows the application to specify the types of interrupts, which when they occur will be passed to the application's interrupt routine. The firmware should have been enabled and buffering turned on by the time this call is made. The types of interrupts and the bits used to enable them are:

Bit	Condition	Description
[15..8]	(reserved)	Set these to zero
[7]	Break/Abort	Break sequence detect
[6]	Tx Underrun	Transmit underrun detect
[5]	CTS	Transition on input handshake line
[4]	0	(reserved)
[3]	DCD	Transition on General Purpose line
[2]	Tx	Transmit register empty
[1]	0	(reserved)
[0]	Rx	Character available

Extended Serial Port Commands



Buffer Commands



Mode and Hardware Control Commands