Devices

COLLABORATORS			
	TITLE : Devices		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Devices

1.1 Amiga® RKM Devices: 12 Serial Device

```
The serial device provides a hardware-independent interface to the Amiga's
built-in RS-232C compatible serial port.  Serial ports have a wide range
of uses, including communication with modems, printers, MIDI devices, and
other computers.  The same device interface can be used for additional
"byte stream oriented devices" - usually more serial ports.  The serial
device is based on the conventions of Exec device I/O, with extensions for
parameter setting and control.

                   Serial Device Characteristics
                   -----------------------------
                   MODES           Exclusive
                                   Shared Access

                   BAUD RATES      110-292,000

                   HANDSHAKING     Three-Wire
                                   Seven-Wire
```


1.2 12 Serial Device / Serial Device Commands and Functions

```
Device Command   Operation
--------------   ---------
```

```
CMD_CLEAR         Reset the serial port's read buffer pointers.

CMD_FLUSH         Purge all queued requests for the serial device (does not
                  affect active requests).

CMD_READ          Read a stream of characters from the serial port buffer.
                  The number of characters can be specified or a
                  termination character(s) used.

CMD_RESET         Reset the serial port to its initialized state.  All
                  active and queued I/O requests will be aborted and the
                  current buffer will be released.

CMD_START         Restart all paused I/O over the serial port.  Also sends
                  an "xON".

CMD_STOP          Pause all active I/O over the serial port.  Also sends an
                  "xOFF".

CMD_WRITE         Write out a stream of characters to the serial port.  The
                  number of characters can be specified or a
                  NULL-terminated string can be sent.

SDCMD_BREAK       Send a break signal out the serial port.  May be done
                  immediately or queued. Duration of the break (in
                  microseconds) can be set by the application.

SDCMD_QUERY       Return the status of the serial port lines and registers,
                  and the number of bytes in the serial port's read buffer.

SDCMD_SETPARAMS   Set the parameters of the serial port.  This ranges from
                  baud rate to number of microseconds a break will last.


Exec Functions as Used in This Chapter
--------------------------------------
AbortIO()         Abort a command to the serial device. If the command is
                  in progress, it is stopped immediately.  If it is queued,
                  it is removed from the queue.

BeginIO()          Initiate a command and return immediately (asynchronous
                  request).  This is used to minimize the amount of system
                  overhead.

CheckIO()         Determine the current state of an I/O request.

CloseDevice()     Relinquish use of the serial device.  All requests must
                  be complete.

DoIO()            Initiate a command and wait for completion (synchronous
                  request).

OpenDevice()      Obtain use of the serial device.

SendIO()          Initiate a command and return immediately (asynchronous
                  request).
```

```
WaitIO()          Wait for the completion of an asynchronous request. When
                  the request is complete the message will be removed from
                  your reply port.


Exec Support Functions as Used in This Chapter
----------------------------------------------
CreateExtIO()     Create an extended I/O request structure of type
                  IOExtSer.  This structure will be used to communicate
                  commands to the serial device.

CreatePort()      Create a signal message port for reply messages from the
                  serial device.  Exec will signal a task when a message
                  arrives at the port.

DeleteExtIO()     Delete an extended I/O request structure created by
                  CreateExtIO().

DeletePort()      Delete the message port created by CreatePort().
```

1.3 12 Serial Device / Device Interface

The serial device operates like the other Amiga devices. To use it, you must first open the serial device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the serial device is called IOExtSer.

```
    struct IOExtSer
       {
        struct   IOStdReq IOSer;
        ULONG    io_CtlChar;    /* control characters */
        ULONG    io_RBufLen;    /* length in bytes of serial read buffer */
        ULONG    io_ExtFlags;   /* additional serial flags */
        ULONG    io_Baud;       /* baud rate */
        ULONG    io_BrkTime;    /* duration of break in microseconds */
        struct   iOTArray  io_TermArray;  /* termination character array */
        UBYTE    io_ReadLen;    /* number of bits per read character */
        UBYTE    io_WriteLen;   /* number of bits per write character */
        UBYTE    io_StopBits;   /* number of stopbits for read */
        UBYTE    io_SerFlags;   /* serial device flags */
        UWORD    io_Status;     /* status of serial port and lines */
    };
```

See the include file devices/serial.h for the complete structure definition.

Opening The Serial Device   Writing To The Serial Device
Reading From The Serial Device   Closing The Serial Device

1.4 12 / Device Interface / Opening The Serial Device

Three primary steps are required to open the serial device:

* Create a message port using CreatePort(). Reply messages from the device must be directed to a message port.

* Create an extended I/O request structure of type IOExtSer using CreateExtIO(). CreateExtIO() will initialize the I/O request to point to your reply port.

* Open the serial device. Call OpenDevice(), passing the I/O request.

```
struct MsgPort  *SerialMP;          /* Define storage for one pointer */
struct IOExtSer *SerialIO;         /* Define storage for one pointer */

if (SerialMP=CreatePort(0,0) )
    if (SerialIO=(struct IOExtSer *)
           CreateExtIO(SerialMP,sizeof(struct IOExtSer)) )
        SerialIO->io_SerFlags=SERF_SHARED;  /* Turn on SHARED mode */
        if (OpenDevice(SERIALNAME,0L,(struct IORequest *)SerialIO,0) )
            printf("%s did not open\n",SERIALNAME);
```

During the open, the serial device pays attention to a subset of the flags in the io_SerFlags field. The flag bits, SERF_SHARED and SERF_7WIRE, must be set before open. For consistency, the other flag bits should also be properly set. Full descriptions of all flags will be given later.

The serial device automatically fills in default settings for all parameters - stop bits, parity, baud rate, etc. For the default unit, the settings will come from Preferences. You may need to change certain parameters, such as the baud rate, to match your requirements. Once the serial device is opened, all characters received will be buffered, even if there is no current request for them.

1.5 12 / Device Interface / Reading From The Serial Device

You read from the serial device by passing an IOExtSer to the device with CMD_READ set in io_Command, the number of bytes to be read set in io_Length and the address of the read buffer set in io_Data.

```
#define READ_BUFFER_SIZE 256
char SerialReadBuffer[READ_BUFFER_SIZE]; /* Reserve SIZE bytes */

SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
SerialIO->IOSer.io_Data     = (APTR)&SerialReadBuffer[0];
SerialIO->IOSer.io_Command  = CMD_READ;
DoIO((struct IORequest *)SerialIO);
```

If you use this example, your task will be put to sleep waiting until the serial device reads 256 bytes (or terminates early). Early termination can be caused by error conditions such as a break. The number of characters actually received will be recorded in the io_Actual field of the IOExtSer structure you passed to the serial device.

1.6 12 / Device Interface / Writing To The Serial Device

You write to the serial device by passing an IOExtSer to the device with CMD_WRITE set in io_Command, the number of bytes to be written set in io_Length and the address of the write buffer set in io_Data.

To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters and transmits a value of zero (0x00).

```
   SerialIO->IOSer.io_Length   = -1;
   SerialIO->IOSer.io_Data     = (APTR)"Life is but a dream. ";
   SerialIO->IOSer.io_Command  = CMD_WRITE;
   DoIO((struct IORequest *)SerialIO);             /* execute write */
```

The length of the request is -1, meaning we are writing a NULL-terminated string. The number of characters sent can be found in io_Actual.

1.7 12 / Device Interface / Closing The Serial Device

Each OpenDevice() must eventually be matched by a call to CloseDevice(). When the last close is performed, the device will deallocate all resources and buffers.

All IORequests must be complete before CloseDevice(). Abort any pending requests with AbortIO().

```
   if (!(CheckIO(SerialIO)))
       {
       AbortIO((struct IORequest *)SerialIO); /* Ask device to abort */
       }                      /*   request, if pending */
   WaitIO((struct IORequest *)SerialIO);      /* Wait for abort, then */
   CloseDevice((struct IORequest *)SerialIO); /*  clean up */
```

1.8 12 Serial Device / Alternative Modes for Serial Input or Output

As an alternative to DoIO() you can use an asynchronous I/O request to transmit the command. Asynchronous requests are initiated with SendIO(). Your task can continue to execute while the device processes the command. You can occasionally do a CheckIO() to see if the I/O has completed. The write request in this example will be processed while the example continues to run:

```
    SerialIO->IOSer.io_Length   = -1;
    SerialIO->IOSer.io_Data     = (APTR)"Save the whales! ";
    SerialIO->IOSer.io_Command  = CMD_WRITE;
    SendIO((struct IORequest *)SerialIO);

    printf("CheckIO %lx\n",CheckIO((struct IORequest *)SerialIO));
    printf("The device will process the request in the background\n");
    printf("CheckIO %lx\n",CheckIO((struct IORequest *)SerialIO));
    WaitIO((struct IORequest *)SerialIO);  /* Remove message and cleanup */
```

Most applications will want to wait on multiple signals. A typical application will wait for menu messages from Intuition at the same time as replies from the serial device. The following fragment demonstrates waiting for one of three signals. The Wait() will wake up if the read request ever finishes, or if the user presses Ctrl-C or Ctrl-F from the Shell. This fragment may be inserted into the above complete example.

```
    /* Precalculate a wait mask for the CTRL-C, CTRL-F and message
     * port signals.  When one or more signals are received,
     * Wait() will return.  Press CTRL-C to exit the example.
     * Press CTRL-F to wake up the example without doing anything.
     * NOTE: A signal may show up without an associated message!
     */

    WaitMask = SIGBREAKF_CTRL_C|
                SIGBREAKF_CTRL_F|
                 1L << SerialMP->mp_SigBit;

    SerialIO->IOSer.io_Command  = CMD_READ;
    SerialIO->IOSer.io_Length   = READ_BUFFER_SIZE;
    SerialIO->IOSer.io_Data     = (APTR)&SerialReadBuffer[0];
    SendIO(SerialIO);

    printf("Sleeping until CTRL-C, CTRL-F, or serial input\n");

    while (1)
           {
           Temp = Wait(WaitMask);
           printf("Just woke up (YAWN!)\n");

           if (SIGBREAKF_CTRL_C & Temp)
               break;

           if (CheckIO(SerialIO) ) /* If request is complete... */
               {
               WaitIO(SerialIO);   /* clean up and remove reply */
               printf("%ld bytes received\n",SerialIO->IOSer.io_Actual);
               break;
               }
           }
    AbortIO(SerialIO);  /* Ask device to abort request, if pending */
    WaitIO(SerialIO);   /* Wait for abort, then clean up */
```

WaitIO() vs. Remove().

The WaitIO() function is used above, even if the request is already known to be complete. WaitIO() on a completed request simply removes the reply and cleans up. The Remove() function is not acceptable for clearing the reply port; other messages may arrive while the function is executing.

High Speed Operation
Use Of BeginIO() With The Serial Device
Ending A Read Or Write Using Termination Characters
Using Separate Read And Write Tasks

1.9 12 / Alternative Modes for Serial Input or Output / High Speed Operation

The more characters that are processed in each I/O request, the higher the total throughput of the device. The following technique will minimize device overhead for reads:

* Use the SDCMD_QUERY command to get the number of characters currently in the buffer (see the devices/serial.h Autodocs for information on SDCMD_QUERY).

* Use DoIO() to read all available characters (or the maximum size of your buffer). In this case, DoIO() is guaranteed to return without waiting.

* If zero characters are in the buffer, post an asynchronous request (SendIO()) for 1 character. When at least one is ready, the device will return it. Now go back to the first step.

* If the user decides to quit the program, AbortIO() any pending requests.

1.10 12 / / Use Of BeginIO() With The Serial Device

Instead of transmitting the read command with either DoIO() or SendIO(), you might elect to use the low level BeginIO() interface to a device.

BeginIO() works much like SendIO(), i.e., asynchronously, except it gives you control over the quick I/O bit (IOB_QUICK) in the io_Flags field. Quick I/O saves the overhead of a reply message, and perhaps the overhead of a task switch. If a quick I/O request is actually completed quickly, the entire command will execute in the context of the caller. See the "Exec: Device Input/Output" chapter of the Amiga ROM Kernel Reference Manual: Libraries for more detailed information on quick I/O.

The device will determine if a quick I/O request will be handled quickly. Most non-I/O commands will execute quickly; read and write commands may or may not finish quickly.

```
SerialIO.IOSer.io_Flags |= IOF_QUICK;  /* Set QuickIO Flag */

BeginIO((struct IORequest *)SerialIO);
if (SerialIO->IOSer.io_Flags & IOF_QUICK )
    /* If flag is still set, I/O was synchronous and is now finished.
     * The IORequest was NOT appended a reply port.  There is no
     * need to remove or WaitIO() for the message.
     */
    printf("QuickIO\n");
else
     /* The device cleared the QuickIO bit.  QuickIO could not happen
      * for some reason; the device processed the command normally.
      * In this case BeginIO() acted exactly like SendIO().
      */
     printf("Regular I/O\n");
WaitIO(SerialIO);
```

The way you read from the device depends on your need for processing speed. Generally the BeginIO() route provides the lowest system overhead when quick I/O is possible. However, if quick I/O does not work, the same reply message overhead still exists.

1.11 12 / / Ending A Read Or Write Using Termination Characters

Reads and writes from the serial device may terminate early if an error occurs or if an end-of-file (EOF) is sensed. For example, if a break is detected on the line, any current read request will be returned with the error SerErr_DetectedBreak. The count of characters read to that point will be in the io_Actual field of the request.

You can specify a set of possible end-of-file characters that the serial device is to look for in the input stream or output using the SDCDMD_SETPARAMS command. These are contained in an io_TermArray that you provide. io_TermArray is used only when the SERF_EOFMODE flag is selected (see the Serial Flags section below).

If EOF mode is selected, each input data character read into or written from the user's data block is compared against those in io_TermArray. If a match is found, the IOExtSer is terminated as complete, and the count of characters transferred (including the termination character) is stored in io_Actual.

To keep this search overhead as efficient as possible, the serial device requires that the array of characters be in descending order. The array has eight bytes and all must be valid (that is, do not pad with zeros unless zero is a valid EOF character). Fill to the end of the array with the lowest value termination character. When making an arbitrary choice of EOF character(s), you will get the quickest response from the lowest value(s) available.

Terminate_Serial.c

The read will terminate before the io_Length number of characters is read if a "Q", "E", ETX, or EOT is detected in the serial input stream.

1.12 12 / / Using Separate Read And Write Tasks

In some cases there are advantages to creating a separate IOExtSer for reading and writing. This allows simultaneous operation of both reading and writing. Some users of the device have separate tasks for read and write operations. The sample code below creates a separate reply port and request for writing to the serial device.

```
struct IOExtSer *SerialWriteIO;
struct MsgPort  *SerialWriteMP;

/*
```

```
 * If two tasks will use the same device at the same time, it is
 * preferred use two OpenDevice() calls and SHARED mode.  If exclusive
 * access mode is required, then you will need to copy an existing
 * IORequest.
 * Remember that two separate tasks will require two message ports.
 */

SerialWriteMP = CreatePort(0,0);
SerialWriteIO = (struct IOExtSer *)
                CreateExtIO( SerialWriteMP,sizeof(struct IOExtSer) );

if (SerialWriteMP && SerialWriteIO )
    {

    /* Copy over the entire old IO request, then stuff the
     * new Message port pointer.
     */

    CopyMem( SerialIO, SerialWriteIO, sizeof(struct IOExtSer) );
    SerialWriteIO->IOSer.io_Message.mn_ReplyPort = SerialWriteMP;

    SerialWriteIO->IOSer.io_Command  = CMD_WRITE;
    SerialWriteIO->IOSer.io_Length = -1;
    SerialWriteIO->IOSer.io_Data = (APTR)"A poet's food is love and fame";
    DoIO(SerialWriteIO);
    }


   Where's OpenDevice()?
   ---------------------
   This code assumes that the OpenDevice() function has already been
   called.  The initialized read request block is copied onto the new
   write request block.
```

1.13 12 Serial Device / Setting Serial Parameters (SDCMD_SETPARAMS)

```
When the serial device is opened, default values for baud rate and other
parameters are automatically filled in from the serial settings in
Preferences.  The parameters may be changed by using the
SDCMD_SETPARAMS command.  The flags are defined in the include file
devices/serial.h.

                   SERIAL DEVICE PARAMETERS (IOExtSer)

   IOExtSer
   Field Name   Serial Device Parameter It Controls
   ---------    -----------------------------------
   io_CtlChar   Control characters to use for xON, xOFF,
                INQ, ACK respectively.  Positioned within an unsigned
                longword in the sequence from low address to high as
                listed.  INQ and ACK handshaking is not currently
                supported.

   io_RBufLen   Recommended size of the buffer that the serial device
                should allocate for incoming data.  For some hardware the
```

buffer size will not be adjustable. Changing the value may cause the device to allocate a new buffer, which might fail due to lack of memory. In this case the old buffer will continue to be used.

For the built-in unit, the minimum size is 64 bytes. Out-of-range numbers will be truncated by the device. When you do an SDCMD_SETPARAMS command, the driver senses the difference between its current value and the value of buffer size you request. All characters that may already be in the old buffer will be discarded. Thus it is wise to make sure that you do not attempt buffer size changes (or any change to the serial device, for that matter) while any I/O is actually taking place.

io_ExtFlags An unsigned long that contains the flags SEXTF_MSPON and SEXTF_MARK. SEXTF_MSPON enables either mark or space parity. SEXTF_MARK selects mark parity (instead of space parity). Unused bits are reserved.

io_Baud The real baud rate you request. This is an unsigned long value in the range of 1 to 4,294,967,295. The device will reject your baud request if the hardware is unable to support it.

For the built-in driver, any baud rate in the range of 110 to about 1 megabaud is acceptable. The built-in driver may round 110 baud requests to 112 baud. Although baud rates above 19,200 are supported by the hardware, software overhead will limit your ability to "catch" every single character that should be received. Output data rate, however, is not software-dependent.

io_BrkTime If you issue a break command, this variable specifies how long, in microseconds, the break condition lasts. This value controls the break time for all future break commands until modified by another SDCMD_SETPARAMS.

io_TermArray A byte-array of eight termination characters, must be in descending order. If the EOFMODE bit is set in the serial flags, this array specifies eight possible choices of character to use as an end of file mark. See the section above "Ending a Read Or Write Using Termination Characters" and the SDCMD_SETPARAMS summary page in the Autodocs.

io_ReadLen How many bits per read character; typically a value of 7 or 8. Generally must be the same as io_WriteLen.

io_WriteLen How many bits per write character; typically a value of 7 or 8. Generally must be the same as io_ReadLen.

io_StopBits How many stop bits are to be expected when reading a character and to be produced when writing a character; typically 1 or 2. The built-in driver does not allow values above 1 if io_WriteLen is larger than 7.

io_SerFlags See the "Serial Flags" section below.

```
 io_Status      Contains status information filled in by the SDCMD_QUERY
                command.  Break status is cleared by the execution of
                SDCMD_QUERY.
```

You set the serial parameters by passing an IOExtSer to the device with SDCMD_SETPARAMS set in io_Command and with the flags and parameters set to the values you want.

```
  SerialIO->io_SerFlags      &= ~SERF_PARTY_ON; /* set parity off */
  SerialIO->io_SerFlags      |= SERF_XDISABLED; /* set xON/xOFF disabled */
  SerialIO->io_Baud           = 9600;           /* set 9600 baud */
  SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS;/* Set params command */
  if (DoIO((struct IORequest *)SerialIO))
      printf("Error setting parameters!\n");
```

The above fragment modifies two bits in io_SerFlags and changes the baud rate. If the parameters you request are unacceptable or out of range, the SDCMD_SETPARAMS command will fail. You are responsible for checking the error code and informing the user.

```
   Proper Time for Parameter Changes.
   ----------------------------------
   A parameter change should not be performed while an I/O request is
   actually being processed because it might invalidate the request
   handling already in progress.  To avoid this, you should use
   SDCMD_SETPARAMS only when you have no serial I/O requests pending.
```

Serial Flags (Bit Definitions For io_SerFlags)

1.14 12 / Setting Serial Parameters / Serial Flags (Bits For io_SerFlags)

There are additional serial device parameters which are controlled by flags set in the io_SerFlags field of the IOExtSer structure. The default state of all of these flags is zero. SERF_SHARED and SERF_7WIRE must always be set before OpenDevice(). The flags are defined in the include file serial.h.

```
                        SERIAL FLAGS (IO_SERFLAGS)

   Flag Name            Effect on Device Operation
   --------             --------------------------
   SERF_XDISABLED       Disable the XON/XOFF feature.  XON/XOFF must be
                        disabled during XModem transfers.

   SERF_EOFMODE         Set this bit if you want the serial device to
                        check input characters against io_TermArray and to
                        terminate the read immediately if an end-of-file
                        character has been encountered. Note:  this bit
                        may be set and reset directly in the user's
                        IOExtSer without a call to SDCMD_SETPARAMS.

   SERF_SHARED          Set this bit if you want to allow other tasks to
                        simultaneously access the serial port.  The
                        default is exclusive-access. Any number of tasks
```

```
                         may have shared access. Only one task may have
                         exclusive access.  If someone already has the port
                         for exclusive access, your OpenDevice() call will
                         fail. This flag must be set before OpenDevice().

   SERF_RAD_BOOGIE       If set, this bit activates high-speed mode.
                         Certain peripheral devices (MIDI, for example)
                         require high serial throughput.  Setting this bit
                         high causes the serial device to skip certain of
                         its internal checking code to speed throughput.
                         Use SERF_RAD_BOOGIE only when you have:

                         *  Disabled parity checking
                         *  Disabled XON/XOFF handling
                         *  Use 8-bit character length
                         *  Do not wish a test for a break signal

                         Note that the Amiga is a multitasking system and
                         has immediate processing of software interrupts.
                         If there are other tasks running, it is possible
                         that the serial driver may be unable to keep up
                         with high data transfer rates, even with this bit
                         set.

   SERF_QUEUEDBRK        If set, every break command that you transmit will
                         be enqueued. This means that all commands will be
                         executed on a FIFO (first in, first out) basis.

                         If this bit is cleared (the default), a break
                         command takes immediate precedence over any serial
                         output already enqueued. When the break command
                         has finished, the interrupted request will
                         continue (if not aborted by the user).

   SERF_7WIRE            If set at OpenDevice() time, the serial device
                         will use seven-wire handshaking for RS-232-C
                         communications. Default is three-wire (pins 2, 3,
                         and 7).

   SERF_PARTY_ODD        If set, selects odd parity.  If clear, selects
                         even parity.

   SERF_PARTY_ON         If set, parity usage and checking is enabled. Also
                         see the SERF_MSPON bit described under io_ExtFlags
                         above.
```

1.15 12 Serial Device / Querying the Serial Device

You query the serial device by passing an IOExtSer to the device with SDCMD_QUERY set in io_Command. The serial device will respond with the status of the serial port lines and registers, and the number of unread characters in the read buffer.

```
   UWORD Serial_Status;
   ULONG Unread_Chars;
```

```
   SerialIO->IOSer.io_Command  = SDCMD_QUERY; /* indicate query */
   SendIO((struct IORequest *)SerialIO);

   Serial_Status = SerialIO->io_Status; /* store returned status */
   Unread_Chars = SerialIO->IOSer.io_Actual; /* store unread count */
```

The 16 status bits of the serial device are returned in io_Status; the number of unread characters is returned in io_Actual.

```
                        SERIAL DEVICE STATUS BITS

         Bit     Active     Symbol      Function
         ---     ------     ------      --------
          0        -                    Reserved
          1        -                    Reserved
          2      high       (RI)        Parallel Select on the A1000.  On
                                        the A500 and A2000, Select is also
                                        connected to the serial port's
                                        Ring Indicator.  (Be cautious when
                                        making cables.)

          3      low        (DSR)       Data set ready
          4      low        (CTS)       Clear to send
          5      low        (CD)        Carrier detect
          6      low        (RTS)       Ready to send
          7      low        (DTR)       Data terminal ready
          8      high                   Read overrun
          9      high                   Break sent
          10     high                   Break received
          11     high                   Transmit x-OFFed
          12     high                   Receive x-OFFed
          13-15    -                    (reserved)
```

1.16 12 Serial Device / Sending the Break Command

You send a break through the serial device by passing an IOExtSer to the device with SDCMD_BREAK set in io_Command. The break may be immediate or queued. The choice is determined by the state of flag SERF_QUEUEDBRK in io_SerFlags.

```
   SerialIO->IOSer.io_Command  = SDCMD_BREAK; /* send break */
   SendIO((struct IORequest *)SerialIO);
```

The duration of the break (in microseconds) can be set in io_BrkTime. The default is 250,000 microseconds (.25 seconds).

1.17 12 Serial Device / Error Codes from the Serial Device

The serial device returns error codes whenever an operation is attempted.

```
   SerialIO->IOSer.io_Command  = SDCMD_SETPARAMS; /* Set parameters */
```

```
    if (DoIO((struct IORequest *)SerialIO))
        printf("Set Params failed. Error: %ld ",SerialIO->IOSer.io_Error);
```

```
The error is returned in the io_Error field of the IOExtSer structure.

                    SERIAL DEVICE ERROR CODES

    Error                   Value    Explanation
    -----                   -----    ----------
    SerErr_DevBusy          1        Device in use
    SerErr_BaudMismatch     2        Baud rate not supported by hardware
    SerErr_BufErr           4        Failed to allocate new read buffer
    SerErr_InvParam         5        Bad parameter
    SerErr_LineErr          6        Hardware data overrun
    SerErr_ParityErr        9        Parity error
    SerErr_TimerErr         11       Timeout (if using 7-wire handshaking)
    SerErr_BufOverflow      12       Read buffer overflowed
    SerErr_NoDSR            13       No Data Set Ready
    SerErr_DetectedBreak    15       Break detected
    SerErr_UnitBusy         16       Selected unit already in use
```

1.18 12 Serial Device / Multiple Serial Port Support

```
Applications that use the serial port should provide the user with a means
to select the name and unit number of the driver.  The defaults will be
"serial.device" and unit number 0.  Typically unit 0 refers to the
user-selected default.  Unit 1 refers to the built-in serial port.
Numbers above 1 are for extended units.  The physically lowest connector
on a board will always have the lowest unit number.

Careful attention to error handling is required to survive in a multiple
port environment.  Differing serial hardware will have different
capabilities.  The device will refuse to open non-existent unit numbers
(symbolic name mapping of unit numbers is not provided at the device
level). The SDCMD_SETPARAMS command will fail if the underlying hardware
cannot support your parameters.  Some devices may use quick I/O for read
or write requests, others will not.  Watch out for partially completed
read requests; io_Actual may not match your requested read length.

If the Tool Types mechanism is used for selecting the device and unit, the
defaults of "DEVICE=serial.device" and "UNIT=0" should be provided.
The user should be able to permanently set the device and unit in a
configuration file.
```

1.19 12 Serial Device / Taking Over the Hardware

```
For some applications use of the device driver interface is not possible.
By following the established rules, applications may take over the serial
interface at the hardware level.  This extreme step is not, however,
encouraged.  Taking over means losing the ability to work with additional
serial ports, and will limit future compatibility.
```

```
Access to the hardware registers is controlled by the misc.resource. See
the "Resources" chapter, and exec/misc.i for details.  The MR_SERIALBITS
and MR_SERIALPORT units control the serial registers.

One additional complication exists.  The current serial device will not
release the misc.resource bits until after an expunge.  This code provides
a work around:

   /*
   * A safe way to expunge ONLY a certain device.
   * This code attempts to flush ONLY the named device out of memory and
   * nothing else.  If it fails, no status is returned (the information
   * would have no valid use after the Permit().
   */
   #include <exec/types.h>
   #include <exec/execbase.h>

   void FlushDevice(char *);

   extern struct ExecBase *SysBase;

   void FlushDevice(name)
   char  *name;
   {
   struct Device *devpoint;

   Forbid();   /* ugly */
   if (devpoint = (struct Device *)FindName(&SysBase->DeviceList,name) )
       RemDevice(devpoint);
   Permit();
   }
```

1.20 12 Serial Device / Additional Information on the Serial Device

```
Additional programming information on the serial device can be found in
the include files and the Autodocs for the serial device.  Both are
contained in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

                         Serial Device Information
                     --------------------------------
                     INCLUDES        devices/serial.h
                                     devices/serial.i

                     AUTODOCS        serial.doc
```