This chapter describes how you can use the Serial Driver to transfer data to a device connected to a Macintosh modem or printer port. The ***Serial Driver*** supports asynchronous serial data communication between applications and serial devices through these ports.

The Serial Driver provides low-level support for communicating with serial devices that cannot be accessed through the Communications Toolbox or Printing Manager. For example, a scientific instrument or a printer that does not support QuickDraw. Before you decide to use the Serial Driver, you should determine whether it is the appropriate solution for your communication needs.

The Communications Toolbox is the recommended method for integrating modems and other telecommunications devices into the Macintosh environment. The Communications Toolbox provides hardware-independent services and a standard interface that offers compatibility with all Macintosh models. To find out more about the Communications Toolbox, see *Inside the Macintosh Communications Toolbox*.

Likewise, the Printing Manager is the recommended interface for printers and similar output devices. Using the Printing Manager makes your hardware or software product compatible with every other device or application that supports this standard interface. Refer to *Inside Macintosh: Imaging With QuickDraw* for more information.

To use the Serial Driver, you should understand how to open, close, and communicate with device drivers using the Device Manager. You can find this information in the chapter "Device Manager" in this book. For information about the Macintosh serial port hardware, including circuit diagrams and signal descriptions, see *Guide to the Macintosh Family Hardware*, second edition.

This chapter begins with a brief summary of key concepts in serial data communication, then describes how you can use the Serial Driver to

- configure a Macintosh serial port
- specify a data transfer buffer
- send and receive data through a serial port
- interpret serial communication errors

Introduction to Serial Communication

Serial Communication, like any data transfer, requires coordination between the sender and receiver. For example, when to start the transmission and when to end it, when one particular bit or byte ends and another begins, when the receiver's capacity has been exceeded, and so on. A ***protocol*** defines the specific methods of coordinating transmission between a sender and receiver.

The scope of serial data transmission protocols is large and complex, encompassing everything from electrical connections to data encoding methods. This section summarizes the most important protocols and standards related to using the Serial Driver.

Asynchronous and Synchronous Communication

Serial data transfers depend on accurate timing in order to differentiate bits in the data stream. This timing can be handled in one of two ways: asynchronously or synchronously. In asynchronous communication, the scope of the timing is a single byte. In synchronous communication, the timing scope comprises one or more blocks of bytes. The terms asynchronous and synchronous are slightly misleading, because both kinds of communication require synchronization between the sender and receiver.

Asynchronous communication is the prevailing standard in the personal computer industry, both because it is easier to implement and because it has the unique advantage that bytes can be sent whenever they are ready, as opposed to waiting for blocks of data to accumulate.

IMPORTANT

Do not confuse asynchronous communication with asynchronous execution. Asynchronous communication is a protocol for coordinating serial data transfers. Asynchronous execution refers to the capability of a device driver to carry out background processing. The Serial Driver supports both asynchronous communication and asynchronous execution. ▲

The Serial Driver does not support synchronous communication protocols. However, it does support synchronous clocking supplied by an external device.

Duplex Communication

Another important characteristic of digital communication is the extent to which simultaneous two-way transfers of data can be achieved.

In a simple connection, the hardware configuration is such that only one-way communication is possible (for example, from a computer to a printer that cannot send status signals back to the computer). In a half-duplex connection, two-way transfer of data is possible, but only in one direction at a time. That is, the two parties to the connection take turns transmitting and receiving data. In a full-duplex connection, both parties can send and receive data simultaneously. The Serial Driver supports full-duplex operation.

Flow Control Methods

Because a sender and receiver can't always process data at the same rate, some method of negotiating when to start and stop transmission is required. The Serial Driver supports two methods of controlling serial data flow. One method relies on the serial port hardware, the other is implemented in software.

Hardware flow control uses two of the serial port signal lines to control data transmission. When the Serial Driver is ready to accept data from an external device, it asserts the Data Terminal Ready (DTR) signal on pin 1 of the serial port, which the external device receives through its Clear to Send (CTS) input. Likewise, the Macintosh receives the external device's DTR signal through the CTS input on pin 2 of the serial

port. When either the Macintosh or the external device is unable to receive data, it negates its DTR signal and the sender suspends transmission until the signal is asserted again.

Flow control can also be handled in software by using an agreed-upon set of characters as start and stop signals. The Serial Driver supports XON/XOFF flow control, which typically assigns the ASCII DC1 character (also known as control-Q) as the start signal and the DC3 character (control-S) as the stop signal, although you can choose different characters.

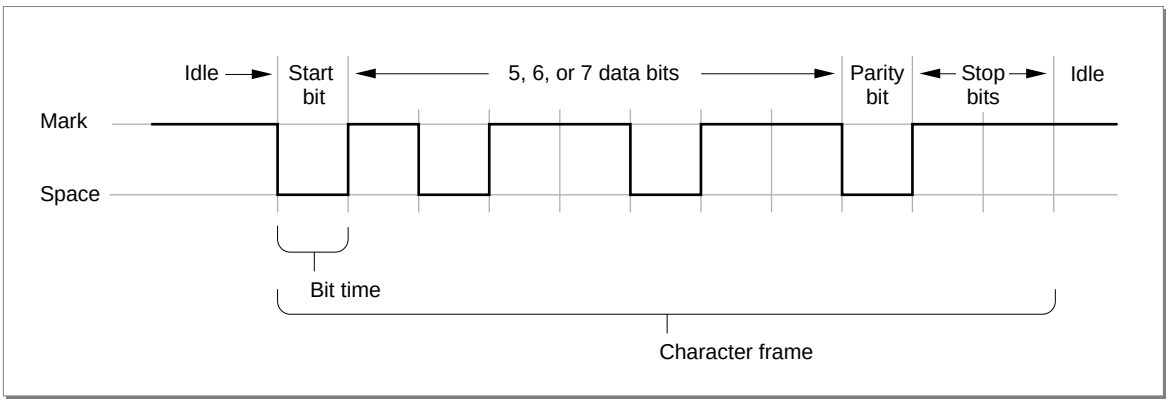
Asynchronous Serial Communication Protocol

This section provides an overview of the protocol that governs the lowest level of data transmission—how serialized bits are sent over a single electrical line. This standard rests on more than a century of evolution in teleprinter technology.

When a sender is connected to a receiver over an electrical connecting line, there is an initial state in which communication has not yet begun, called the idle or mark state. Because older electromechanical devices operate more reliably with current continually passing through them, the mark state employs a positive voltage level. Changing the state of the line by shifting the voltage to a negative value is called a space. Once this change has occurred, the receiver interprets a negative voltage level as a 0 bit, and a positive voltage level as a 1 bit. These transitions are shown in Figure 7-1.

The change from mark to space is known as the start bit, and this triggers the synchronization necessary for asynchronous serial transmission. The start bit delineates the beginning of the transmission unit defined as a character frame. The receiver then samples the voltage level at periodic intervals known as the bit time, to determine whether a 0-bit or a 1-bit is present on the line.

Figure 7-1 The format of serialized bits



The bit time is expressed in samples per second, known as ***baud*** (in honor of telecommunication pioneer Emile Baudot). This sampling rate must be agreed upon by

sender and receiver prior to start of transmission in order for a successful transfer to occur. Common values for the sampling rate are 1200 baud and 2400 baud. In the case where one sampling interval can signal a single bit, a baud rate of 1200 results in a transfer rate of 1200 bits per second (bps). Note that because modern protocols can express more than one bit value within the sampling interval, the baud rate and the data rate (bps) are not always identical.

Prior to transmission, the sender and receiver agree on a serial data format; that is, how many bits of data constitute a character frame, and what happens after those bits are sent. The Serial Driver supports frames of 5, 6, 7, or 8 bits in length. Character frames of 7 or 8 data bits are commonly used for transmitting ASCII characters.

After the data bits in the frame are sent, the sender can optionally transmit a parity bit for error-checking. There are various parity schemes, which the sender and receiver must agree upon prior to transmission. In odd parity, a bit is sent so that the entire frame always contains an odd number of 1 bits. Conversely, in even parity, the parity bit results in an even number of 1 bits. No parity means that no additional bit is sent. Other less-used parity schemes include mark parity, in which the extra bit is always 1, and space parity, in which its value is always 0. Using parity bits for error checking, regardless of the scheme, is now considered a rudimentary approach to error detection. Most communication systems employ more reliable techniques for error detection and correction.

To signify the end of the character frame, the sender places the line back to the mark state (positive voltage) for a minimum specified time interval. This interval has one of several possible values: 1 bit time, 2 bit times, or 1-1/2 bit times. This signal is known as the stop bit, and returns the transmission line back to idle status.

Electrical lines are always subject to environmental perturbations known as noise. This noise can cause errors in transmission, by altering voltage levels so that a bit is reversed (flipped), shortened (dropped), or lengthened (added). When this occurs, the ability of the receiver to distinguish a character frame may be affected, resulting in a framing error.

The break signal is a special signal that falls outside the character frame. The break signal occurs when the line is switched from mark (positive voltage) to space (negative voltage) and held there for longer than a character frame. The break signal resembles an ASCII NUL character (a string of 0-bits), but exists at a lower level than the ASCII encoding scheme (which governs the encoding of information *within* the character frame).

The RS-422 Serial Interface

The electrical characteristics of a serial communication connection are specified by various interfacing standards, one of which is the RS-422 standard used in all Macintosh computers. This standard is an enhancement of the RS-232 standard, with electrical characteristics modified to allow higher transmission rates over longer lines. Although the electrical voltage differences can be critical at times and should therefore not be ignored, most of the terminology and concepts remain the same across these two standards. For purposes of this discussion, it is convenient to treat these two standards as a single entity.

The specifications of the RS-422 and RS-232 interfacing standards are contained in documents available from the Electronic Industries Associations (EIA). The specifications cover several aspects of the connection between data terminal equipment and data communication equipment. These aspects include the electrical signal characteristics, the mechanical description of the interface circuits, and the functional description of the circuits.

The principal interface signals specified by the EIA are described in the following list. The term ***data terminal equipment*** (DTE) is used to describe the initiator or controller of the serial connection, typically the computer. The term ***data communication equipment*** (DCE) describes the device that is connected to the DTE, such as a modem or printer.

The RS-422/RS-232 signals are described below. For specific information about how these signals are used in Macintosh computers, see *Guide to the Macintosh Family Hardware*, second edition.

- *Data Terminal Ready* (DTR). The DTR signal indicates that the DTE (that is, your computer) is ready to communicate. Deasserting this signal causes the DCE to suspend transmission. The DTR signal is the most important control line for a modem, because when it is deasserted, most modem functions cease and the modem disconnects from the telephone line. In Macintosh computers, the DTR signal is connected to the CTS signal, discussed next.
- *Request to Send* (RTS) and *Clear to Send* (CTS). The RTS signal was originally intended to switch a half-duplex modem from transmit to receive mode. The computer would send an RTS signal to the modem and wait for the modem to respond by asserting CTS. Since most communications between microcomputers are full-duplex nowadays, RTS/CTS handshaking is not often used in its original form. Rather, in most full-duplex modems, the CTS signal is permanently asserted, and the RTS signal is not used. In Macintosh computers, the CTS signal is connected to the DTR signal.
- *Data Set Ready* (DSR). The DSR signal is not used by Macintosh computers and is usually permanently asserted on microcomputer modems. It was intended to signal the computer that the modem had made a proper connection to the telephone line and received an answer tone from the modem on the other end. Modern modems communicate this information by sending messages to the computer.
- *Transmitted Data* (TD). The TD signal carries the serial data stream from the DTE to the DCE. The EIA specifications dictate that the DTR, RTS, CTS, and DSR signals must be asserted before data can be transmitted, but this requirement is not strictly followed in the computer industry.
- *Received Data* (RD). The RD signal is the counterpart of the TD signal, and carries data from the DCE to the DTE. Although the EIA specifies that this signal be in the mark state when no carrier is present, this requirement is rarely adhered to.
- *Data Carrier Detect* (DCD). The DCD signal is not used by Macintosh computers. In systems that use the signal, it is asserted by the DCE when a carrier signal is received.
- *Ring Indicator* (RI). The RI signal is not used by Macintosh computers. In systems that use the signal, it is asserted by the DCE when the telephone line is ringing.

As you can see, implementations of the RS-422/RS-232 interface do not always correspond to the specifications set forth by the EIA. This is especially true when the DCE is not a modem.

About the Serial Driver

The Serial Driver is a part of the Macintosh Operating System that provides low-level support for asynchronous, interrupt-driven serial data transfers through the modem and printer ports.

The Serial Driver provides routines that allow you to

- initialize and terminate communication
- transmit and receive data
- examine and change communication settings

You access the Serial Driver routines using standard Device Manager functions such as open, close, read, write, control, and status. The Serial Driver also includes some convenience routines that you can call from Pascal or C.

The Serial Driver supports the following communication settings

- 5, 6, 7, or 8 data bits per character
- odd, even, or no parity
- 1, 1.5, or 2 stop bits
- 300 to 57600 baud transmission rates (depending on hardware capability)
- hardware or software flow control

The Serial Driver default settings are 9600 baud, 8 data bits per character, no parity, and 2 stop bits. Hardware handshaking is the default under System 7, although some earlier versions of the Serial Driver defaulted to software handshaking.

Additional control and status functions allow you to

- determine the version number of the Serial Driver
- change the input buffer from the default buffer to one that you specify
- obtain information about transmission errors such as overrun, framing, parity, and break signals.
- enable the automatic replacement of characters that have parity errors
- use an external timing signal for synchronous clocking

Macintosh Serial Architecture

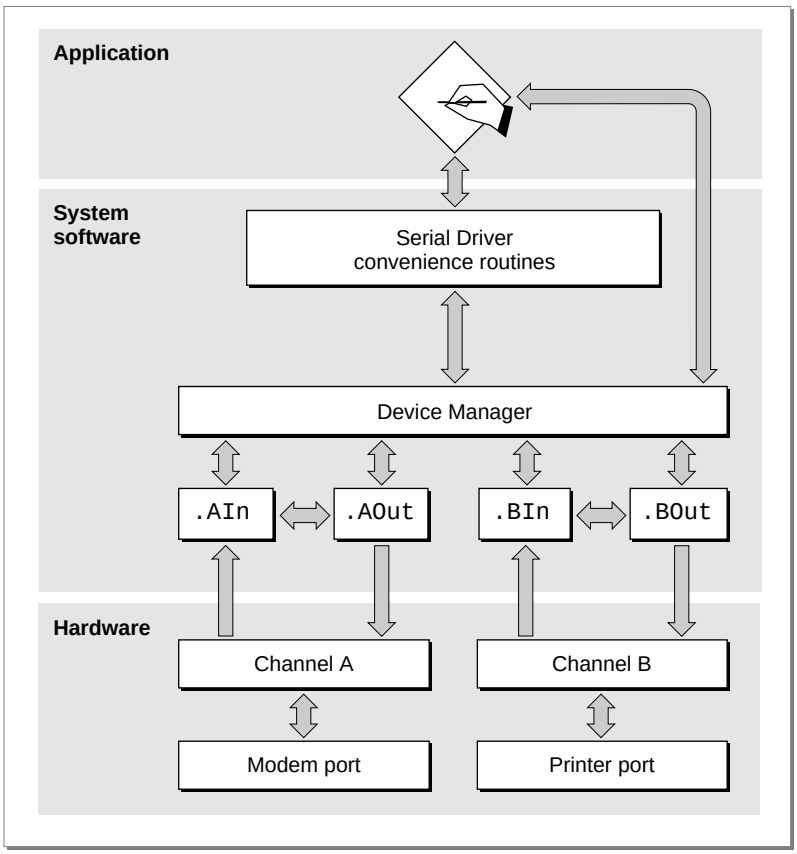
The Serial Driver consists of a set of four Macintosh device drivers and assorted convenience routines that interface to the Device Manager. Within the overall Macintosh software architecture, the location and boundaries of the Serial Driver are not sharply defined. This is because its role as mediator between applications and devices is supplemented by routines belonging to the Device Manager.

Although the hardware architecture of the serial ports varies, the Serial Driver provides a universal interface for applications. For example, some Macintosh computers use the

Zilog Z8530 Serial Communications Controller (SCC) microchip, while others use custom devices. By using the Serial Driver rather than relying on a particular hardware configuration, your application is compatible with all Macintosh computers.

Figure 7-2 shows the Serial Driver and its relation to the Macintosh serial architecture. Conceptually, there are three functional layers: the application layer, the system software layer, and the hardware layer. The Serial Driver, the Device Manager, and the four serial device drivers all exist within the system software layer. Although you normally access the Serial Driver through Device Manager routines, the Serial Driver interface includes a set of convenience routines such as `SerStatus` that provide a high-level interface to some functions.

Figure 7-2 The role of the Serial Driver



The four device drivers that control the serial ports differ from other Macintosh device drivers in that they share common internal routines and data structures, as illustrated by the horizontal interconnecting arrows in Figure 7-2. Each driver is associated with a communication channel, either Channel A or Channel B, and each channel is associated with a serial port. Channel A controls the modem port, and Channel B controls the printer port. Each channel has both an input driver and an output driver associated with

it. The drivers for the modem port are named .AIn and .AOut, and those for the printer port are named .BIn and .BOut.

Each input driver receives data from a serial port and transfers it to the application. Each output driver takes data from the application and sends it out its serial port. Although the input and output drivers for a port are closely related and share some of the same routines, each driver has its own device control entry data structure. This means that read and write operations can be processed simultaneously, which allows the Serial Driver to support full-duplex communication.

Because the input and output drivers are not completely distinct entities, some functions (for example, the SerReset function) only need to be invoked on the output driver—the desired operation occurs on the input side as well. Note, however, that you must always explicitly open and close both the input and output drivers.

Serial Communication Errors

Data received from the serial port passes through a hardware buffer and then into a software buffer managed by the input driver for the port. Characters are removed from the input driver's buffer each time an application calls the driver's read routine. Each input driver's buffer can initially hold up to 64 characters, but you can specify a larger buffer using the SerSetBuf function. You need to increase the input buffer size if the buffer fills up faster than your application can read from it, as indicated by overrun errors and lost data.

Because the serial hardware in some Macintosh computers relies on processor interrupts during I/O operations, overrun errors are possible if interrupts are disabled while data is being received at the serial port. To prevent such errors, the Disk Driver and other system software components are designed to store any data received by the modem port while they have interrupts disabled, and then pass this data to the port's input driver. Because the system software only monitors the modem port, the printer port is not recommended for two-way communication at data rates above 300 baud.

Note
AppleTalk is not subject to the same limitations because it is not interrupt-driven and does not use the Serial Driver. ◆

You can use the SerStatus function to detect the most common serial communication errors:

- Hardware overrun errors occur when the serial hardware input buffer overflows, usually because the input driver doesn't read it often enough.
- Software overrun errors occur when an input driver's buffer overflows, usually because the application doesn't issue read calls to the driver often enough.
- Parity errors occur when the serial hardware detects an incorrect parity bit.
- Framing errors occur when the serial hardware detects an error in the stop bits.
- Break errors occur when a break signal is received.

Overrun, parity, and framing errors are usually handled by requesting that the sender retransmit the affected data. Break errors are typically initiated by the user and handled as appropriate for the particular application. When an input driver receives a break signal, it terminates any pending read requests. You can terminate pending write requests by sending a `KillIO` request to the output driver.

Using the Serial Driver

The basic steps in using the Serial Driver are

1. Open the output device driver for the serial port, then open the input device driver. Always open both drivers, even if you only need one.
2. Optionally, allocate a buffer that is larger than the default 64-byte input buffer, and then use the `SerSetBuf` function to select the alternate buffer.
3. Set the handshaking mode.
4. Set the baud rate and data format.
5. Read or write the desired data.
6. When you are finished using the Serial Driver, terminate any pending I/O with the Device Manager `KillIO` function.
7. Restore the default input buffer.
8. Close the input and output drivers. Always close the input driver first.

The program shown in Listing 7-1 illustrates these steps. The following sections describe each step in more detail.

Listing 7-1 Using the Serial Driver

```
PROGRAM UsingTheSerialDriver;
{An example of the basic steps required to set up and use the Serial Driver.}
{ Note that all function calls demonstrated here are synchronous and thus   }
{ should not be called at interrupt time.                                   }
USES
   Serial;

VAR
   gOutputRefNum:    Integer;    {output driver reference number}
   gInputRefNum:     Integer;    {input driver reference number}
   gInputBufHandle:  Handle;     {handle to my input buffer}
   gOSErr:           OSErr;      {function results}
```

```
PROCEDURE MyOpenSerialDriver;
{Use the Device Manager OpenDriver function to open the drivers.}
BEGIN
   gOSErr := OpenDriver('.AOut', gOutputRefNum); {always open output first}
   IF gOSErr = noErr THEN
      gOSErr := OpenDriver('.AIn', gInputRefNum); {then open the input driver}
END;

PROCEDURE MyChangeInputBuffer;
{Replace the default input buffer with a larger buffer.}
CONST
   kInputBufSize = 1024; {size of my input buffer in bytes}
BEGIN
   gInputBufHandle := NewHandle(kInputBufSize); {allocate storage}
   HLock(gInputBufHandle); {lock it}
   SerSetBuf(gInputRefNum, gInputBufHandle^, kInputBufSize); {set the buffer}
END;

PROCEDURE MySetHandshakeOptions;
{Set flow control method and other options. Note that you only need to set}
{ the output driver; the settings are reflected on the input side.}
VAR
   mySerShkRec: SerShk; {serial handshake record}
BEGIN
   WITH mySerShkRec DO
      BEGIN
         fXOn := 0;     {turn off XON/XOFF output flow control}
         fCTS := 0;     {turn off CTS/DTR flow control}
         errs := 0;     {clear error mask}
         evts := 0;     {clear event mask}
         fInX := 0;     {turn off XON/XOFF input flow control}
         fDTR := 0;     {turn off DTR input flow control}
      END;
   {Use control call 14 instead of the SerHShake function}
   { because it allows control over DTR handshaking.}
   gOSErr := Control(gOutputRefNum, 14, @mySerShkRec); {csCode = 14}
END;

PROCEDURE MyConfigureThePort;
{Set baud rate and data format.  Note that you only need to set the}
{ output driver; the settings are reflected on the input side.}
CONST
   kConfigParam = baud2400+data8+noParity+stop10; {create bit field}
```

```
BEGIN
   gOSErr := SerReset(gOutputRefNum, kConfigParam); {configure the port}
END;

PROCEDURE MySendMessage;
{Use the Device Manager PBWrite function to send data to the output driver.}
VAR
   myMessage:        Str255;        {the data to send}
   myMsgLen:         LongInt;       {number of bytes to send}
   myParamBlock:     ParamBlockRec; {parameter block for the PBWrite function}
   myPBPtr:          ParmBlkPtr;    {pointer to the parameter block}
BEGIN
   myMessage := 'The Eagle has landed.';
   myMsgLen := Length(myMessage);   {get the size of the message string}
   WITH myParamBlock DO {fill in required fields of the parameter block}
      BEGIN
         ioRefNum := gOutputRefNum;    {write to the output driver}
         ioBuffer := @myMessage[1];    {pointer to the data}
         ioReqCount := myMsgLen;       {number of bytes to send}
         ioCompletion := NIL;          {no completion routine specified}
         ioVRefNum := 0;               {not used by the Serial Driver}
         ioPosMode := 0;               {not used by the Serial Driver}
      END;
   myPBPtr := @myParamBlock
   gOSErr := PBWrite(myPBPtr, FALSE);  {synchronous Device Manager request}
END;

PROCEDURE MyReceiveMessage;
{Use the Device Manager PBRead function to read data from the input driver.}
VAR
   myBuffer:         Str255;        {a buffer to receive the data}
   myReadCount:      LongInt;       {number of bytes to read}
   myParamBlock:     ParamBlockRec; {parameter block for the PBRead function}
   myPBPtr:          ParmBlkPtr;    {pointer to the parameter block}
BEGIN
   myBuffer := '';
   myReadCount := 0;
   gOSErr := SerGetBuf(gInputRefNum, myReadCount); {determine how many bytes}
                                                   { are in the input buffer}
   IF myReadCount > 0 THEN
      BEGIN
         WITH myParamBlock DO {fill in required fields of the parameter block}
            BEGIN
```

```
            ioRefNum := gInputRefNum;  {read from the input driver}
            ioBuffer := @myBuffer[1];  {pointer to my data buffer}
            ioReqCount := myReadCount; {number of bytes to read}
            ioCompletion := NIL;       {no completion routine specified}
            ioVRefNum := 0;            {not used by the Serial Driver}
            ioPosMode := 0;            {not used by the Serial Driver}
          END;
        myPBPtr := @myParamBlock;
        gOSErr := PBRead(myPBPtr, FALSE);{synchronous Device Manager request}
      END;
END;

PROCEDURE MyRestoreInputBuffer;
{Restore the default input buffer.}
BEGIN
   SerSetBuf(gInputRefNum, gInputBufHandle^, 0); {0 means restore default}
   HUnlock(gInputBufHandle); {release my old buffer}
END;

PROCEDURE MyCloseSerialDriver;
{Use the Device Manager KillIO function to terminate all current and pending}
{ operations, then close the drivers. Note that you only need to call KillIO}
{ on the output driver to terminate both input and output operations.}
BEGIN
   gOSErr := KillIO(gOutputRefNum); {terminate all pending I/O operations}
   IF gOSErr = noErr THEN
      gOSErr := CloseDriver(gInputRefNum); {close the input driver first}
   IF gOSErr = noErr THEN
      gOSErr := CloseDriver(gOutputRefNum); {then close the output driver}
END;

BEGIN {UsingTheSerialDriver}
   MyOpenSerialDriver;              {open the output and input drivers}
   MyChangeInputBuffer;             {replace the default input buffer}
   MySetHandshakeOptions;           {select flow control method}
   MyConfigureThePort;              {set baud rate and data format}
   MySendMessage;                   {send some bytes to the output driver}
   MyReceiveMessage;                {read some bytes from the input driver}
   MyRestoreInputBuffer;            {restore the default input buffer}
   MyCloseSerialDriver;             {terminate I/O and close the drivers}
END.
```

Opening the Serial Driver

Because the Serial Driver uses separate device drivers for the input and output functions, you need to open both drivers for two-way communication. On Macintosh computers with two serial ports, you access the modem port through the `.AIn` and `.AOut` drivers, and the printer port through the `.BIn` and `.BOut` drivers.

On computers with a single serial port, such as the Macintosh PowerBook Duo models, the serial port can be used for either modem or printer connections. There is only one serial channel on these models, which you access through the `.AIn` and `.AOut` drivers.

You open the serial port drivers using the Device Manager `OpenDriver` or `PBOpen` functions. You should always open the output driver first because the Serial Driver initializes its local variables for both the input and output drivers when you open the output driver. Opening the output driver also installs interrupt handlers and allocates and locks buffer storage for both input and output.

When the Serial Driver receives an open request it first verifies that the serial port is available and correctly configured. If the port is unavailable or not configured, the Serial Driver returns the result code `portInUse` or `portNotCf`. Any other errors, such as attempting to open the `.BIn` or `.BOut` driver on a Macintosh with only one serial port, return the `openErr` result code.

When a device driver is opened successfully, the Device Manager returns a driver reference number, which you use to identify the driver in subsequent I/O requests. Although the reference numbers of the serial input and output drivers have remained constant for some time, you should not assume these values are fixed. Because future versions of the Operating System may assign other reference numbers to these drivers, your application should always use the reference numbers returned by the Device Manager.

Because of hardware differences between the serial ports in some Macintosh models, you should use the printer port for output-only connections to devices such as printers, at a maximum data rate of 9600 baud. The printer port is not recommended for two-way communication at data rates above 300 baud.

Note
If AppleTalk is active you cannot open the printer port for serial communication unless AppleTalk is using an alternate connection, such as EtherTalk or TokenTalk. ◆

Specifying an Alternate Input Buffer

An optional but recommended practice is to increase the size of the input driver's buffer. The default buffer size, 64 bytes, is not always sufficient for sustained transfers at data rates above 300 baud. A larger buffer will help avoid buffer overruns and consequent loss of data. You can specify a buffer size of up to 32 KB, but 1 to 2 KB is usually sufficient.

You use the SerSetBuf function to specify an alternate input buffer, and also to reset the default buffer. To ensure compatibility and avoid heap fragmentation you must reset the default buffer before closing the input driver.

Setting the Handshaking Options

The recommended method of setting handshaking options is to send a control request to the output driver, with a csCode value of 14. This is equivalent to using the SerHShake function, but allows you to select DTR handshaking. To specify the desired options, you pass the following data structure to the driver:

```
TYPE SerShk =
   PACKED RECORD
      fXOn: Byte; {XON/XOFF output flow control enabled flag}
      fCTS: Byte; {CTS hardware handshake enabled flag}
      xOn:  Char; {XON character}
      xOff: Char; (XOFF character}
      errs: Byte; {error mask for input errs that cause abort}
      evts: Byte; {mask for status changes that cause events}
      fInX: Byte; {XON/XOFF input flow control flag}
      fDTR: Byte; {DTR input flow control (for csCode=14 only)}
   END;
```

The Data Terminal Ready (DTR) signal is normally asserted when the Serial Driver is opened and negated when it is closed. You can change this behavior using one of several control routines described in the section "Low-Level Routines," beginning on page 7-27.

The fields of the SerShk data structure are described in the section "Serial Driver Reference," beginning on page 7-18.

Setting the Baud Rate and Data Format

When you open the Serial Driver it configures the selected port with default settings of 9600 baud, 8 data bits, no parity bit, and 2 stop bits. You can change these settings using the SerReset function, described on page 7-19.

Reading and Writing to the Serial Ports

Once you have configured the serial port, you can read and write data using the Device Manager PBRead and PBWrite functions. These functions can be called either synchronously or asynchronously, as described in the chapter "Device Manager" in this book.

Synchronous I/O Requests

When you make a synchronous request to a device driver, the Device Manager places your request at the end of the driver's I/O queue and does not return control to your application until the request completes. To avoid hanging, your application needs to take steps to ensure that a request will complete properly before calling the Device Manager.

For example, because the `PBRead` function requires you to specify the number of bytes to be read, you need to determine how many bytes are in the input driver's buffer before you call `PBRead`. You can use the `SerGetBuf` function to determine how many characters are in the input buffer, as shown in Listing 7-1.

If you try to read more bytes than are available in the input buffer, the driver waits until it receives enough characters to satisfy your request. If the external serial device does not send the required number of bytes, there is no way for your application regain control of the processor or terminate the read request.

Similarly, the `PBWrite` function will not complete until the specified number of bytes have been transmitted to the external serial device. If the external device is holding off transfers through hardware or software handshaking, the Device Manager will never return control to your application. You can use the `SerStatus` function, described on page 7-25, to query the status of the output driver and determine if output is suspended by handshaking.

For more information about how synchronous I/O requests are processed, see the chapter "Device Manager" in this book.

Asynchronous I/O Requests

Asynchronous execution allows your application to continue to process user input or perform other tasks while waiting for serial I/O requests to complete. To take full advantage of asynchronous operation you should supply a completion routine for the Device Manager to call when an asynchronous request completes. You should also implement a timer function to notify your application if a request is not satisfied within a reasonable period.

See the chapter "Device Manager" in this book for information about how asynchronous I/O requests are processed.

Closing the Serial Driver

Before closing the Serial Driver you must restore the default input buffer using the `SerSetBuf` function. After restoring the default buffer, you can terminate any pending I/O using the Device Manager `KillIO` function. Finally, you should close the input and output drivers using the Device Manager `CloseDriver` or `PBClose` functions.

Synchronous Clocking

Although the Serial Driver does not support synchronous communication protocols, it does allow you to select an external timing signal for synchronous clocking between the sender and receiver. You connect the external timing signal to the handshake input (HSKi) signal on pin 2 of the serial port, and select external clocking by sending a control request to the output driver with a `csCode` value of 16 and bit 6 set in the `csParam` field. See the section "Low-Level Routines," beginning on page 7-27, for more information.

Serial Driver Reference

This section describes the programming interface to the Serial Driver. This interface consists of the Serial Driver routines and the Device Manager functions for accessing them. The Serial Driver defines two data structures, the serial handshake record and the serial status record, which are described along with the routines that use these structures (the `SerHShake` and `SerStatus` functions, respectively).

Serial Driver Routines

You can use the Serial Driver routines to

- reset and configure the serial port device drivers
- set the size of the serial input buffer
- set handshaking options
- set or clear a break signal
- determine the number of characters in the input buffer
- get status information for a serial port

This section describes the control and status routines unique to the Serial Driver, as well the convenience routines for accessing them. Other Serial Driver functions, such as reading and writing, are accessed through the Device Manager. For information about the Device Manager functions for opening, closing, and communicating with device drivers, see the chapter "Device Manager" in this book.

IMPORTANT

The Serial Driver convenience routines described in this section are always executed synchronously when called using the high-level interface. To execute these functions asynchronously you must use the equivalent low-level Device Manager control or status call (`PBControlAsync` or `PBStatusAsync`). The `csCode` value for each routine is listed in the assembly-language information section of the routine description. ▲

SerReset

You can use the `SerReset` function to reset the serial port drivers and configure the port for a specified transmission rate and character frame.

```
FUNCTION SerReset (refNum: Integer; serConfig: Integer): OSErr;
```

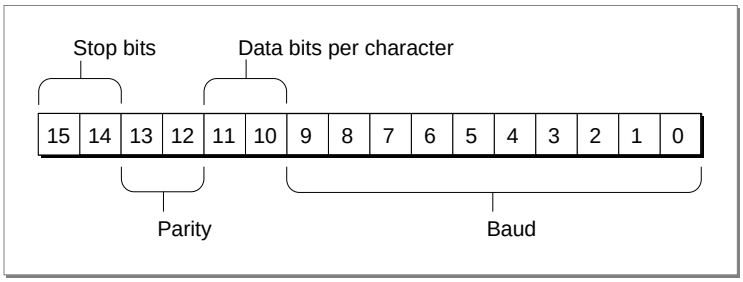
`refNum` The driver reference number of the serial output driver.

`serConfig` A 16-bit value that specifies the configuration information.

DESCRIPTION

The `SerReset` function resets the output and input device drivers for the serial port, and also configures the port according to the format of the `serConfig` parameter shown in Figure 7-3.

Figure 7-3 The `serConfig` parameter format



You can use the following constants to set the values of the bit fields in the `serConfig` parameter:

```
CONST
   baud300     = 380;      {300 baud}
   baud600     = 189;      {600 baud}
   baud1200    = 94;       {1200 baud}
   baud1800    = 62;       {1800 baud}
   baud2400    = 46;       {2400 baud}
   baud3600    = 30;       {3600 baud}
   baud4800    = 22;       {4800 baud}
   baud7200    = 14;       {9600 baud}
   baud9600    = 10;       {3600 baud}
   baud14400   = 6;        {14400 baud}
   baud19200   = 4;        {19200 baud}
```

```
baud28800   = 2;        {28800 baud}
baud38400   = 1;        {38400 baud}
baud57600   = 0;        {57600 baud}
stop10      = 16384;    {1 stop bit}
stop15      = -32768;   {1.5 stop bits}
stop20      = -16384;   {2 stop bits}
noParity    = 0;        {no parity}
oddParity   = 4096;     {odd parity}
evenParity  = 12288;    {even parity}
data5       = 0;        {5 data bits}
data6       = 2048;     {6 data bits}
data7       = 1024;     {7 data bits}
data8       = 3072;     {8 data bits}
```

For example, the default setting of 9600 baud, eight data bits, two stop bits, and no parity bit is equivalent to passing the following value in the `serConfig` parameter:

```
baud9600 + data8 + stop20 + noParity.
```

This value has a binary representation of `1100110000001010` and a hexadecimal representation of `$CC0A`.

ASSEMBLY-LANGUAGE INFORMATION

The `SerReset` function is equivalent to a Device Manager control request with a `csCode` value of 8. You pass the `serConfig` parameter in the `csParam` field (`csParam[0]` = `serConfig`).

RESULT CODES

`noErr`	0	No error

SerSetBuf

You can use the `SerSetBuf` function to increase the size of the serial input buffer, or to restore the driver's default buffer.

```
FUNCTION SerSetBuf (refNum: Integer; serBPtr: Ptr;
                    serBLen: Integer): OSErr
```

`refNum`	The driver reference number of the serial input driver.
`serBPtr`	A pointer to the new input buffer.
`serBLen`	The size of the new input buffer, or 0 to restore the default buffer.

DESCRIPTION

The `SerSetBuf` function replaces the input buffer for the specified input driver. The `serBPtr` parameter points to the buffer, and the `serBLen` parameter specifies the number of bytes in the buffer. The buffer must be locked while in use. Before closing the driver you must restore the default buffer by calling `SerSetBuf` with the `serBLen` parameter equal to 0.

ASSEMBLY-LANGUAGE INFORMATION

The `SerSetBuf` function is equivalent to a Device Manager control request with a `csCode` value of 9. You pass the `serBPtr` and `serBLen` parameters in the `csParam` field (`csParam[0]` = `serBPtr`; `csParam[4]` = `serBLen`).

RESULT CODE

`noErr`	0	No error

SerHShake

You can use the `SerHShake` function to set software handshaking options and other control information.

```
FUNCTION SerHShake (refNum: Integer; flags: SerShk): OSErr;
```

`refNum`	The driver reference number of the serial output driver.
`flags`	A pointer to a serial handshake record.

DESCRIPTION

The `SerHShake` function enables flow control, sets flow control characters, and specifies which conditions will cause input requests to be aborted.

Note that the `SerHShake` function has been superseded by a newer function that allows control over DTR handshaking. There is no high-level interface to the new function, you access it using a Device Manager control request with a `csCode` value of 14. This function uses the same `SerShk` data structure, but adds an additional field for DTR hardware flow control. See the section "Low-Level Routines," beginning on page 7-27, for a description of control routine 14.

The serial handshake record is defined by the `SerShk` data type:

```
TYPE SerShk =
PACKED RECORD
   fXOn:    Byte;    {XON/XOFF output flow control flag}
   fCTS:    Byte;    {CTS output flow control flag}
   xOn:     Char;    {XON character}
```

```
   xOff:    Char;    {XOFF character}
   errs:    Byte;    {mask for errors that will terminate input}
   evts:    Byte;    {mask for status changes that cause events}
   fInX:    Byte;    {XON/XOFF input flow control flag}
   fDTR:    Byte;    {DTR input flow control flag (csCode 14 only)}
END;
```

Field descriptions

`fXOn`	Set this byte to a non-zero value to enable XON/XOFF output flow control.
`fCTS`	Set this byte to a non-zero value to enable CTS output flow control.
`xOn`	If XON/OFF flow control is enabled, this field specifies the character to use for XON.
`xOff`	If XON/XOFF flow control is enabled, this field specifies the character to use for XOFF.
`errs`	Indicates which errors will cause input requests to be terminated, using the bit mask constants shown below.
`evts`	Indicates whether changes in the CTS signal or the break signal will cause the Serial Driver to post device driver events, using the bit mask constants shown below.
`fInX`	Set this byte to a non-zero value to enable XON/XOFF input flow control.
`fDTR`	Set this byte to a non-zero value to enable DTR input flow control. This field is only used by control function 14; it is ignored by the `SerHShake` function.

You can use the following constants as bit mask values for the `errs` field, to specify which errors will cause input requests to be aborted. Because these are bit mask values, you can sum them to specify more than one error condition.

```
CONST
   parityErr      = 16; {parity error}
   hwOverrunErr   = 32; {hardware overrun error}
   framingErr     = 64; {framing error}
```

You can use the following constants as bit mask values for the `evts` field, to specify which status changes will cause the Serial Driver to post device driver events. Because these are bit mask values, you can sum them to specify more than one event.

```
CONST
   ctsEvent    = 32;    {change in CTS signal}
   breakEvent  = 128;   {change in break signal}
```

▲ **WARNING**
Using device driver events is discouraged because interrupts are disabled during the event posting process, which may cause serial data to be lost or other events to be missed. Instead, you should use the `SerStatus` function to check the value of the `ctsHold` or `breakErr` flags in the serial status record. ▲

ASSEMBLY-LANGUAGE INFORMATION

The `SerHShake` function is equivalent to a Device Manager control request with a `csCode` value of 10. To specify DTR flow control, use a `csCode` value of 14 and set the `fDTR` flag to a non-zero value. You pass the `flags` parameter in the `csParam` field (`csParam[0]` = `flags`).

RESULT CODES

`noErr`	0	No error

SerSetBrk

You can use the `SerSetBrk` function to assert a break signal.

```
FUNCTION SerSetBrk (refNum: Integer): OSErr;
```

`refNum` The driver reference number of the serial output driver.

DESCRIPTION

The `SerSetBrk` function forces the output data line into the space state. To form a break signal, the line must be left in this state longer than a character frame.

ASSEMBLY-LANGUAGE INFORMATION

The `SerSetBrk` function is equivalent to a Device Manager control request with a `csCode` value of 12.

RESULT CODES

`noErr`	0	No error

SerClrBrk

You can use the `SerClrBrk` function to deassert the break signal.

```
FUNCTION SerClrBrk (refNum: Integer): OSErr;
```

`refNum` The driver reference number of the serial output driver.

DESCRIPTION

The `SerClrBrk` function restores the output driver to normal operation after asserting a break signal with the `SerSetBrk` function.

ASSEMBLY-LANGUAGE INFORMATION

The `SerClrBrk` function is equivalent to a Device Manager control request with a `csCode` value of 11.

RESULT CODES

`noErr`	0	No error

SerGetBuf

You can use the `SerGetBuf` function to determine the number of characters available in the driver's input buffer.

```
FUNCTION SerGetBuf (refNum: Integer; VAR count: LongInt): OSErr;
```

`refNum` The driver reference number of the serial input driver.

`count` On exit, the number of characters in the input buffer.

DESCRIPTION

The `SerGetBuf` function returns, in the `count` parameter, the number of characters present in the input buffer.

ASSEMBLY-LANGUAGE INFORMATION

The `SerGetBuf` function is equivalent to a Device Manager status request with a `csCode` value of 2. The `count` value is returned in `csParam` as a long word (`csParam[0]` = `count`).

RESULT CODES

`noErr`	0	No error

SerStatus

You can use the `SerStatus` function to obtain status information from the Serial Driver.

```
FUNCTION SerStatus (refNum: Integer; VAR serSta: SerStaRec):OSErr;
```

`refNum` The driver reference number of the serial input or output driver.

`serSta` A pointer to a serial status record.

DESCRIPTION

The `SerStatus` function returns status information for the specified input or output driver. This information includes error conditions, flow control status, and whether there are read or write operations pending. Because the serial status record is shared, the `SerStatus` function returns the same information whether you reference the input or output driver. The serial status record is defined by the `SerStaRec` data type:

```
TYPE SerStaRec =
PACKED RECORD
   cumErrs:    Byte;       {cumulative errors}
   xOffSent:   Byte;       {XOFF sent as input flow control}
   rdPend:     Byte;       {read pending flag}
   wrPend:     Byte;       {write pending flag}
   ctsHold:    Byte;       {CTS flow control hold flag}
   xOffHold:   Byte;       {XOFF flow control hold flag}
END;
```

Field descriptions

`cumErrs` A bit field that indicates what errors have occurred since the last time the `SerStatus` function was called. You can use the bit mask constants shown below to test for particular errors. Errors detected include software overrun, break asserted, parity error, hardware overrun, and framing error.

`xOffSent` A bit field that indicates if the driver has initiated input flow control by sending an XOFF character or negating the DTR signal. You can use the bit mask constants shown below to test for these conditions.

`rdPend` This field contains a non-zero value if the driver has a read operation pending.

`wrPend` This field contains a non-zero value if the driver has a write operation pending.

`ctsHold` This field contains a non-zero value if the driver has suspended output due to the CTS handshake signal.

`xOffHold` This field contains a non-zero value if the driver has suspended output due to receiving an XOFF character.

You can use the following constants as bit mask values for the `cumErrs` field, to detect which errors have occurred since the last time the `SerStatus` function was called. Because these are bit mask values, you can sum them to specify more than one error condition. The remaining bit values in the `cumErrs` field are reserved.

```
CONST
   swOverrunErr   = 1;     {software overrun error}
   breakErr       = 8;     {break signal asserted}
   parityErr      = 16;    {parity error}
   hwOverrunErr   = 32;    {hardware overrun error}
   framingErr     = 64;    {framing error}
```

You can use the following constants as bit mask values to test the `xOffSent` field for the specified conditions. The remaining bit values in the `xOffSent` field are reserved.

```
CONST
   dtrNegated     = 64;    {DTR signal was negated}
   xOffWasSent    = 128;   {XOFF character was sent}
```

IMPORTANT

Calling `SerStatus` resets `cumErrs` and other fields of the serial status record, so repeated calls to `SerStatus` may not return identical results. ▲

ASSEMBLY-LANGUAGE INFORMATION

The `SerStatus` function is equivalent to a Device Manager status request with a `csCode` value of 8; the serial status record is returned in the first 6 bytes of the `csParam` field (`csParam[0]` = `SerStaRec`).

You can execute the status request immediately, bypassing the I/O queue, by setting bit 9 of the trap word. You can set this bit by appending the word `IMMED` as the second argument to the trap macro. For example:

```
_Status, IMMED
```

This technique is recommended when you need to determine the current status of a port before issuing a subsequent I/O request.

RESULT CODES

`noErr`	0	No error

Low-Level Routines

This section describes the low-level Serial Driver routines that you can call using the Device Manager control and status functions. These calls should be made to the output device driver—they affect the input driver as well.

Serial Driver Version[status code 9]

`csCode` = 9 `csParam` = word

This status routine returns the version number of the Serial Driver in the `csParam` field. The version number is an integer value.

Set Baud Rate[control code 13]

`csCode` = 13 `csParam` = word

This control routine provides an additional method (besides the `SerReset` function) of setting the baud rate. You specify the baud rate value as an integer in the `csParam` field (for example, 9600). The Serial Driver attempts to set the serial port to the specified baud rate, or the closest baud rate supported by the hardware. The actual baud rate selected is returned in the `csParam` field.

Set Handshaking Options[control code 14]

`csCode` = 14 `csParam` = `SerShk` record

This control routine is identical to the `SerHShake` function (control code 10) with the additional specification of the `fDTR` flag in the eighth byte of the `SerShk` record. You enable DTR input flow control by setting this flag to a non-zero value. See the description of the `SerHShake` function on page 7-21 for information about the other fields of the `SerShk` record.

Set Miscellaneous Options[control code 16]

`csCode` = 16 `csParam` = byte

This control routine sets miscellaneous control options. Bits 0-5 are reserved and should be set to 0 for compatibility with future options. Bit 6 enables external clocking through the CTS handshake line (the HSKi signal on pin 2 of the serial port). Set bit 6 to 1 to allow an external device to drive the serial data clock. Set bit 6 to 0 to restore internal clocking. Bit 7 controls the state of the DTR signal when the driver is closed. When bit 7 is 0 (the default) the DTR signal is automatically negated when the driver closes. Set bit 7 to 1 if you want the DTR signal to be left unchanged when the driver is closed. This can be used to prevent a modem from hanging up or a printer from going offline when the driver closes.

Assert DTR[control code 17]

`csCode` = 17

This control routine asserts the DTR signal.

Negate DTR[control code 18]

`csCode` = 18

This control routine negates the DTR signal.

Simple Parity Error Replacement[control code 19]

`csCode` = 19 `csParam` = char

This control routine enables simple parity error replacement, in which incoming characters with parity errors are replaced by the ASCII character specified in `csParam` (for example, $FF). If a valid incoming character matches the replacement character, the most significant bit of the character is cleared. Therefore, if it is possible for your replacement character to appear in the data stream, you should use control code 20 instead. Set `csParam` to 0 to disable parity error replacement.

Extended Parity Error Replacement[control code 20]

`csCode` = 20 `csParam[0]` = char `csParam[1]` = char

This control routine enables extended parity error replacement. Incoming characters with parity errors are replaced by the ASCII character specified in `csParam[0]`. The difference between this routine and the simple version (control code 19) is that if a valid incoming character matches the parity replacement character, it is replaced by the alternate character specified in `csParam[1]`. Set `csParam[0]` to 0 to disable parity error replacement.

Note
The ASCII NUL character ($00) can be used as the alternate character but not as the parity replacement. ◆

Set XOFF State[control code 21]

`csCode` = 21

This control routine unconditionally sets the `xOffHold` flag, which is equivalent to receiving an XOFF character. If software handshaking is enabled, data transmission is halted until an XON character is received, or until you clear the XOFF state using control code 22.

Clear XOFF State[control code 22]

`csCode` = 22

This control routine unconditionally clears the `xOffHold` flag, which is equivalent to receiving an XON character. If software handshaking is enabled, data transmission is resumed.

Send XON Conditional[control code 23]

`csCode` = 23

This control routine sends an XON character for input flow control if the last input flow control character sent was XOFF.

Send XON Unconditional[control code 24]

`csCode` = 24

This control routine unconditionally sends an XON character for input flow control, regardless of the current state of input flow control.

Send XOFF Conditional[control code 25]

`csCode` = 25

This control routine sends an XOFF character for input flow control if the last input flow control character sent was XON.

Send XOFF Unconditional[control code 26]

`csCode` = 26

This control routine unconditionally sends an XOFF character for input flow control, regardless of the current state of input flow control.

Serial Hardware Reset[control code 27]

`csCode` = 27

This control routine resets the serial port hardware for a channel. Because this routine may leave the serial port in an unknown state, you must call the `SerReset` function before you use the port.

Summary of the Serial Driver

Pascal Summary

Constants

```
CONST
   {values for the transmission rate in the SerConfig parameter}
   baud300        = 380;         {300 baud}
   baud600        = 189;         {600 baud}
   baud1200       = 94;          {1200 baud}
   baud1800       = 62;          {1800 baud}
   baud2400       = 46;          {2400 baud}
   baud3600       = 30;          {3600 baud}
   baud4800       = 22;          {4800 baud}
   baud7200       = 14;          {7200 baud}
   baud9600       = 10;          {9600 baud}
   baud14400      = 6;           {14400 baud}
   baud19200      = 4;           {19200 baud}
   baud28800      = 2;           {28800 baud}
   baud38400      = 1;           {38400 baud}
   baud57600      = 0;           {57600 baud}

   {values for the number of stop bits in the SerConfig parameter}
   stop10         = 16384;       {1 stop bit}
   stop15         = -32768;      {1.5 stop bits}
   stop20         = -16384;      {2 stop bits}

   {values for the parity in the SerConfig parameter}
   noParity       = 0;           {no parity}
   oddParity      = 4096;        {odd parity}
   evenParity     = 12288;       {even parity}

   {values for the number of data bits in the SerConfig parameter}
   data5          = 0;           {5 data bits}
   data6          = 2048;        {6 data bits}
   data7          = 1024;        {7 data bits}
   data8          = 3072;        {8 data bits}
```

```
{bit mask values to test for indicated errors}
swOverrunErr   = 1;          {software overrun error}
breakErr       = 8;          {break occurred}
parityErr      = 16;         {parity error}
hwOverrunErr   = 32;         {hardware overrun error}
framingErr     = 64;         {framing error}

{bit mask values for the evts field in the SerShk record}
ctsEvent       = 32;         {CTS change}
breakEvent     = 128;        {break status change}

{bit mask value for the xOffHold field of the SerStaRec record}
dtrNegated     = 64;         {DTR signal was negated}
xOffWasSent    = 128;        {XOFF character was sent}
```

Data Types

```
TYPE
   SerShk =
   PACKED RECORD
      fXOn:    Byte;    {XON/XOFF output flow control flag}
      fCTS:    Byte;    {CTS output flow control flag}
      xOn:     Char;    {XON character}
      xOff:    Char;    {XOFF character}
      errs:    Byte;    {mask for errors that will terminate input}
      evts:    Byte;    {mask for status changes that cause events}
      fInX:    Byte;    {XON/XOFF input flow control flag}
      fDTR:    Byte;    {DTR input flow control flag (csCode 14 only)}
   END;

   SerStaRec =
   PACKED RECORD
      cumErrs:    Byte;    {cumulative errors}
      xOffSent:   Byte;    {XOFF sent as input flow control}
      rdPend:     Byte;    {read pending flag}
      wrPend:     Byte;    {write pending flag}
      ctsHold:    Byte;    {CTS flow control hold flag}
      xOffHold:   Byte;    {XOFF flow control hold flag}
   END;
```

Routines

```
FUNCTION SerReset           (refNum: Integer; serConfig: Integer): OSErr;
FUNCTION SerSetBuf          (refNum: Integer; serBPtr: Ptr;
                             serBLen: Integer): OSErr;
FUNCTION SerHShake          (refNum: Integer; flags: SerShk): OSErr;
FUNCTION SerSetBrk          (refNum: Integer): OSErr;
FUNCTION SerClrBrk          (refNum: Integer): OSErr;
FUNCTION SerGetBuf          (refNum: Integer; VAR count: LongInt): OSErr;
FUNCTION SerStatus          (refNum: Integer; VAR serSta: SerStaRec): OSErr;
```

C Summary

Constants

```
enum {
   /*values for the transmission rate in the SerConfig parameter*/
   baud300        = 380,         /*300 baud*/
   baud600        = 189,         /*600 baud*/
   baud1200       = 94,          /*1200 baud*/
   baud1800       = 62,          /*1800 baud*/
   baud2400       = 46,          /*2400 baud*/
   baud3600       = 30,          /*3600 baud*/
   baud4800       = 22,          /*4800 baud*/
   baud7200       = 14,          /*7200 baud*/
   baud9600       = 10,          /*9600 baud*/
   baud14400      = 6,           /*14400 baud*/
   baud19200      = 4,           /*19200 baud*/
   baud28800      = 2,           /*28800 baud*/
   baud38400      = 1,           /*38400 baud*/
   baud57600      = 0,           /*57600 baud*/

   /*values for the number of stop bits in the SerConfig parameter*/
   stop10         = 16384,       /*1 stop bit*/
   stop15         = -32768,      /*1.5 stop bits*/
   stop20         = -16384,      /*2 stop bits*/

   /*values for the parity in the SerConfig parameter*/
   noParity       = 0,           /*no parity*/
   oddParity      = 4096,        /*odd parity*/
   evenParity     = 12288,       /*even parity*/
```

```
   /*values for the number of data bits in the SerConfig parameter*/
   data5          = 0,           /*5 data bits*/
   data6          = 2048,        /*6 data bits*/
   data7          = 1024,        /*7 data bits*/
   data8          = 3072,        /*8 data bits*/

   /*bit mask values to test for indicated errors*/
   swOverrunErr   = 1,           /*software overrun error*/
   breakErr       = 8,           /*break occurred*/
   parityErr      = 16,          /*parity error*/
   hwOverrunErr   = 32,          /*hardware overrun error*/
   framingErr     = 64,          /*framing error*/

   /*bit mask values for the evts field in the SerShk record*/
   ctsEvent       = 32,          /*CTS change*/
   breakEvent     = 128,         /*break status change*/

   /*bit mask value for the xOffHold field of the SerStaRec record*/
   dtrNegated     = 64,          /*DTR signal was negated*/
   xOffWasSent    = 128          /*XOFF character was sent*/
};
```

Data Types

```
struct SerShk {
   char           fXOn;    /*XON/XOFF output flow control flag*/
   char           fCTS;    /*CTS output flow control flag*/
   unsigned char  xOn;     /*XON character*/
   unsigned char  xOff;    /*XOFF character*/
   char           errs;    /*mask for errors that will terminate input*/
   char           evts;    /*mask for status changes that cause events*/
   char           fInX;    /*XON/XOFF input flow control flag*/
   char           fDTR;    /*DTR input flow control flag (csCode 14 only)*/
};
typedef struct SerShk SerShk;

struct SerStaRec {
   char     cumErrs;       /*cumulative errors*/
   char     xOffSent;      /*XOFF sent as input flow control*/
   char     rdPend;        /*read pending flag*/
   char     wrPend;        /*write pending flag*/
   char     ctsHold;       /*CTS flow control hold flag*/
```

```
   char     xOffHold;      /*XOFF flow control hold flag*/
};
typedef struct SerStaRec SerStaRec;
```

Functions

```
pascal OSErr SerReset       (short refNum, short serConfig);
pascal OSErr SerSetBuf      (short refNum, Ptr serBPtr, short serBLen);
pascal OSErr SerHShake      (short refNum, const SerShk *flags);
pascal OSErr SerSetBrk      (short refNum);
pascal OSErr SerClrBrk      (short refNum);
pascal OSErr SerGetBuf      (short refNum, long *count);
pascal OSErr SerStatus      (short refNum, SerStaRec *serSta);
```

Assembly-Language Summary

Data Structures

Serial Handshake Record

0	`fXOn`	byte	XON/XOFF output flow control flag
1	`fCTS`	byte	CTS output flow control flag
2	`xOn`	byte	XOn character
3	`xOff`	byte	XOff character
4	`errs`	byte	mask for errors that will terminate input
5	`evts`	byte	mask for status changes that cause events
6	`fInX`	byte	XON/XOFF input flow control flag
7	`fDTR`	byte	DTR input flow control flag (`csCode` 14 only)

Serial Status Record

0	`cumErrs`	byte	cumulative errors
1	`xOffSent`	byte	XOFF sent as input flow control
2	`rdPend`	byte	read pending flag
3	`wrPend`	byte	write pending flag
4	`ctsHold`	byte	CTS flow control hold flag
5	`xOffHold`	byte	XOFF flow control hold flag

Device Manager Interface

Status Routines

Code	Parameters	Function
2	long	Return the number of bytes currently in the input data buffer (`SerGetBuf`).
8	6 bytes	Return status information (`SerStatus`).
9	word	Return driver version number.

Control Routines

Code	Parameters	Function
8	word	Set data rate and character frame (`SerReset`).
9	long, word	Specify either a new input buffer or the default buffer (`SerSetBuf).`
10	8 bytes	Set software handshaking and other control information (`SerHShake`).
11		Deassert the break signal (`SerClrBrk`).
12		Assert the break signal (`SerSetBrk`).
13	word	Set baud rate.
14	8 bytes	Equivalent to control code 10, plus DTR handshaking.
16	byte	Set miscellaneous control options.
17		Assert DTR.
18		Negate DTR.
19	byte	Simple parity error replacement.
20	2 bytes	Extended parity error replacement.
21		Set XOFF state.
22		Clear XOFF state.
23		Send XON for input flow control if XOFF was sent last.
24		Unconditionally send XON for input flow control.
25		Send XOFF for input flow control if XON was sent last.
26		Unconditionally send XOFF for input flow control.
27		Reset serial hardware channel.

Result Codes

`noErr`	0	No error
`openErr`	–23	Unable to open device driver
`portInUse`	–97	Port is in use
`portNotCf`	–98	Port is not configured