```
    /* Write buffer to card reader then clear buffer. */
    write(card_handle, buffer, 17);
    memset(buffer, 0, 20);

    /* Read from card reader until data exists. The 'while'  */
    /* is how a read is normally done if you are waiting for */
    /* a card to be swiped.                                  */
    while (0 == read(card_handle, buffer, 20));

    /* Display the contents of the card reader buffer. */
    ptr = 0;
    for (i=1; i<4; i++) {
       printf("\fTRACK %d: ",i);
       switch (buffer[ptr+1]) {
          case 0: write(STDOUT, &buffer[ptr+2], buffer[ptr]-
2);
                  break;
          case 1: write(STDOUT, "NO DATA", 7);      break;
          case 2: write(STDOUT, "NO START", 8);     break;
          case 3: write(STDOUT, "NO END", 6);       break;
          case 4: write(STDOUT, "BAD BCC", 7);      break;
          case 5: write(STDOUT, "PARITY ERR", 10); break;
       }
       ptr=ptr+buffer[ptr];
       SVC_WAIT(2000);
    }

    /* Close the card reader. */
    close(card_handle);
}
```

Bar Code Reader

All OMNI 300 Series terminals support ANSI Standard Code 39 bar code reading, returning the ASCII equivalent (including extended ASCII) of the scanned 3-of-9 bar code symbol.

❖ *The bar code reader cannot operate concurrently with other devices or terminal operations. When actively reading a bar code symbol, all other devices are disabled during the read.*

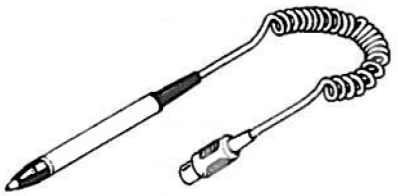


Figure 8-4. Bar Code Wand

Bar Code Reader Functions

open()

This function prepares the bar code reader for subsequent processing.

```
#include <io.h>

hBAR = open("/dev/bar", 0);
int hBar;
```

`open()` prepares the firmware to accept and store bar code reads; the terminal ignores all bar code scans before the `open()` call.

read()

This operation transfers bar code data from the internal buffer to the application buffer.

```
#include <io.h>

bytes_read = read (hBar, buffer, size);
int bytes_read, hBar, size;
char *buffer;
```

Each `read()` transfers bar code data into the application buffer and returns the number of bytes actually read: 0 if no data is available, or -1 if there is an error.

It is best to set `size` to the maximum value. If the bar code data exceeds the buffer size, it will be truncated and all truncated data is discarded. To recover the data, set `size` to the maximum value, and repeat the bar code scan.

The data returned represents the ASCII equivalent of the scanned symbol, regardless if the symbol was scanned in a forward or reverse direction.

write()

This operation transfers data from the application's buffer to the internal bar code buffer.

```
#include <io.h>

bytes_written = write (hBar, buffer, size);
int bytes_written, hBar, size;
char *buffer;
```

The data will be returned on the next `read()` operation.

This operation is useful for test program development and program debugging by simulating bar code reads with known data.

ioctl()

Use this function to retrieve bar code attributes and raw sample information.

```
#include <io.h>

result = ioctl(hBar, GetCtrl | type, buffer);
int result, hBar;
unsigned int type;
void *buffer;
```

Valid `type` values are:

`0` = Get information about the device driver; no meaningful result is returned. The information will be returned in `buffer` (passed as `&buffer`), which must be a structure of the form:

```
struct {char sample_size;
        char read_type;
        int buffer_max;
        long sample_rate;
        } buffer;
```

where:

`sample_size` is 1.

`read_type` is the decoding algorithm; 1 will be returned to represent Code 39.

`buffer_max` is the maximum number of characters in the array.

`sample_rate` is the number of samples per second, which will be 15980.

`1` = Retrieve any existing samples. This operation must be performed after the bar code scan and prior to a `read()` call.

`buffer` receives the sample data as an array of bytes. The number of bytes read is returned in `result` (0 if no scan has been made).

A `read()` operation must be performed to empty the internal bar code buffer, and another scan done, before the next `ioctl()` is issued. Otherwise, the same data will be returned.

The bar code `ioctl()` function is rarely used by applications. However, ioctl() is useful during bar code testing or when writng a custom bar code decoder (i.e., UPC, 2-of-5, etc.).

close()

Releases the handle associated with the bar code reader and disables the device from reading subsequent bar code passes.

```
#include <io.h>

status = close(hBar);
int status, hBar;
```

Bar Code Example

```
/* BARCODE.C                                              */
 /* Demonstrates the use of barcode functions. */

#include <config.h>
int bar_handle,bytes_read;
char buffer[20];

main () {

   /* Open barcode reader. */
   bar_handle = open("/dev/bar", 0);

   /* Write to the barcode buffer. */
   write(bar_handle, "0123456789ABCDEF", 16);

   /* Read from barcode reader until data exists.  */
   /* This loop is how a read is normally done if  */
   /* you are waiting for a barcode to be scanned. */
   do
      bytes_read = read(bar_handle, buffer, 20);
   while (bytes_read == 0);

   /* Display what was read from the barcode buffer. */
   write(STDOUT, buffer, bytes_read);

   /* Close the barcode reader. */
   close(bar_handle);
}
```

Real Time Clock

All OMNI 300 Series terminals support a real time clock device which maintains the current date and time, and provides a source of periodic interrupts for system timing services. Timer ticks occur at the rate of 64 ticks per second.

❖ *The number of ticks per second varies between OMNI terminals (i.e., between 300 and 400 Series models). For portability, VeriFone recommends that the constant* TICKS_PER_SECOND *(defined in both* <io.h> *and* <config.h>*) be used where this data is needed.>OMNI 400 series terminal*

Real Time Clock Functions

open()

Explicitly opens the clock/calendar device, returning its associated device handle.

```
#include <io.h>

hClock = open ("/dev/clock", 0);
int hClock
```

OMNI 300 Series terminals—but *not* OMNI 400 Series terminals—regard the real time clock as always open; the logical device name DEV_CLOCK, defined in file <io.h> and <config.h>, may be used by other I/O functions in place of the device handle hClock. This usage will make the application less portable to 400 Series platforms and is not recommended.

❖ *The library function* SVC_CLOCK() *may be used to access the clock without opening it.*

read()

Places the system date, time, and day of the week in an application buffer as a 15-byte ASCII character array (not a NULL-terminated string).

The `size` parameter determines how many bytes are actually read, which should match the value of `bytes_read`. The 15-byte ASCII character array is returned in buffer in the format `yyyymmddhhmmssd`,

where:

`yyyy`=year, `mm`=month, `dd`=day,
`hh`=hour, `mm`=minutes, `ss`=seconds, and
`d`=day number (0=Sunday ...6=Saturday)

```
#include <io.h>

bytes_read = read (hClock, buffer, size);
int  bytes_read, hClock, size;
long *buffer;
```

write()

Sets the system's date and time.

```
#include <io.h>

bytes_written = write (hClock, buffer, size);
int  bytes_written, hClock, size;
long *buffer;
```

The contents of `buffer` are in the same format as `read()`, except only 14 bytes are actually used; the day of the week (15th byte) is determined by the system. The `size` parameter must be set to 14; the year must be in the range 1990–2089.

Since no validity checks are performed on the values written to the clock, ensure your application always passes valid date and time values.

- *The system tick counter (SVC_TICKS) is stored in a long integer and is reset to zero upon power cycles or system restarts.*

ioctl()

The ioctl() function is used for simple timing applications.

The value of the current system tick count is obtained using ioctl() with the GetCtrl constant.

```
#include <io.h>

result = ioctl (hClock,GetCtrl|type,&buffer);
int  result, hClock;
unsigned int type;
char *buffer;
```

Valid type values are:

0 = Compares the current system tick value against the value in buffer.

1 = Copies the current system tick value into the caller's buffer.

This function can be used to determine whether or not a specified period of time has elapsed. This is accomplished by reading the current tick count value, adding to it the desired delay time in sixty-fourths of a second, and repeatedly calling ioctl() with type set to 0. This compares the contents of buffer with the current system tick count. When the current timer tick count becomes equal to (or exceeds) the value in buffer, the return value is set to 0. If the current timer tick count is less than the value in the buffer, 0X0001 is returned.

If type is set to 1, the system tick count is returned in buffer as a long integer.

close()

Releases the resources associated with the clock handle.

```
#include <io.h>

status = close (hClock);
int status, hClock;
```

Clock Example

```
 /* CLOCK.C                                                    */
 /* This program demonstrates the clock functions. */

#include <stdio.h>
#include <io.h>
#include <txosvc.h>

int clock_handle, i;
long ticks, newticks;
char current[15];

main() {

   /* Open the clock. */
   clock_handle = open("/dev/clock", 0);

   /* Get the current time. */
   read(clock_handle, current, 15);

   /* Enable the clock display and wait 5 seconds. */
   ioctl(clock_handle, SetCtrl | 1, 0);
   SVC_WAIT(5000);

   /* Change the clock to be just before the year 2000. */
   write(clock_handle, "19991231235950", 14);
```

```
    SVC_WAIT(15000);

    /* Disable the clock and print a message. */
    ioctl(clock_handle, SetCtrl | 0, 0);
    printf("\fWELCOME TO 2000!");
    SVC_WAIT(5000);

    /* Restore the terminal's original date and time. */
    /* Note that the time will be off by 20 seconds.  */
    write(clock_handle, current, 14);

    /* Time a loop by saving the current ticks, execute */
    /* the loop, and get the new tick value.            */
    ioctl(clock_handle, GetCtrl | 1, (char *)&ticks);
    for (i=0; i<10000; i++);
    ioctl(clock_handle, GetCtrl | 1, (char *)&newticks);
    printf("\fTICKS = %ld",newticks-ticks);

    /* Close the clock device. */
    close(clock_handle);
}
```

Clock Timing Example

```
#include <io.h>

void wait_sec(n,clock_h)
      int n;       /* Number of seconds to wait */
      int clock_h; /* Valid opened clock handle */
   (
      long timer;

/* Get the current system tick count */

         ioctl(clock_h,GetCtrl|1,&timer);

/* Increment timer to be n seconds in the future */
/* TICKS_PER_SEC is defined as 64 in io.h        */

         timer=timer+n*60*TICKS_PER_SEC;

/* Wait for the timer to expire before returning */

         while(ioctl(clock_h,GetCtrl|0,&timer));

      )
```

C H A P T E R

9

Communication Devices

This chapter describes the communication devices available on OMNI 300 Series terminals:

- Serial Async Port (COM1)
- Sync/Async Serial Port (COM3)
- PIN Pad/Bar Code Port
- Internal Modem (COM4)
- LAN Port (COM4)

Table 9-1. OMNI 300 Series Terminals Communication Devices

OMNI Model	Serial Async Port*	Serial Sync/Async Port	PIN Pad/ Bar Code Port	Internal Modem	LAN Port
380 Dial	DTR	—	Yes	Yes	—
385 LAN	No DTR	Async only	Yes	—	Yes
385 Dial	No DTR	Async only	Yes	Yes	—
390 Dial	DTR	—	Yes	Yes	—
395 LAN	No DTR	Yes	Yes	—	Yes
395 Dial	No DTR	Yes	Yes	Yes	—
460 Dial	DTR	—	Yes	Yes	—

** Note that the Serial Async Port differs between models. Some models support the DTR signal required for full Hayes-compatible support of external modems while others do not, providing a subset of that standard.*

For your convenience, an icon appears next to each subsection in this chapter to indicate which terminals utilize the particular communication device being discussed.

Also see *Appendix C. Model Specifications.*

❖ *New OMNI 300 Series terminals may be released with port configurations not included in this manual. Check your terminal's Reference Manual for port specifications, using the information in this chapter as a guideline for programming the device. Also check the current* `<io.h>` *and* `<config.h>` *file.*

Accessing Devices

Communications devices are accessed in the same manner as files, using the same basic set of function calls: `open()`, `read()`, `write()`, `ioctl()`, and `close()`. This basic set is augmented for certain devices, such as the modem and LAN ports. The basic function calls generally return a result of -1 when an error condition occurs, setting `errno` to a specific error code. These error codes, along with descriptive comments, are defined in `<errno.h>`. Device-specific information can be found in each subsection of this chapter.

The communication device names implemented in the OMNI 300 Series terminals, defined in `<io.h>`, are:

```
#define DEV_COM1    "/dev/com1"   /* Serial async/printer port (COM1) */
#define DEV_PINPAD  "/dev/pinpad" /* PIN pad device */
#define DEV_BAR     "/dev/bar"    /* Bar code device */
#define DEV_COM3    "/dev/com3"   /* Serial sync/async port (COM3) */
#define DEV_MODEM   "/dev/com4"   /* Internal modem device (COM4) */
#define DEV_LAN     "/dev/lan"    /* VFI peer-to-peer LAN (COM4) */
```

All communication devices must be explicitly opened before they can be used. Either the `DEV_...` form (which is case sensitive) or the `"/dev/..."` form (which is not case

sensitive) of the device name may be used in the `open()` as follows:

```
lan_handle = open (DEV_LAN, 0);
```

- or the equivalent -

```
lan_handle = open ("/dev/lan", 0);
```

Thus `/DEV/COM1` and `"/dev/com1"` both work. Some devices must also be configured before they can be accessed. This is the case, for instance, with the LAN and modem. Details on any relevant configuration process are provided along with the individual discussions of each device in the following sections.

❖ *See Section 8, OMNI Peer-to-Peer LAN for information on using OMNI 3X5 LAN operations.*

Device Overview

Serial Async Port (COM1)

380 DIAL
385 LAN
385 DIAL
390 DIAL
395 LAN
395 DIAL
460 DIAL

The 8-pin DIN connector supports a printer or other RS-232 device. The port handles data rates up to 19,200 baud, in either packet or character mode. This port may connect to an external modem and perform Hayes-compatible operations described in this chapter under *Modem Interface*. Note that some 300 Series models support DTR signals on this port (and are thus fully Hayes-compatible) and some do not. Refer to Table 9-1 for your specific terminal.

Serial Sync/Async Port (COM3)

385 LAN
395 LAN
395 DIAL

This 8-pin DIN connector (COM3) is available only on LAN-type terminal models and the OMNI 395 dial-type terminal. It provides RS-232 serial support for external devices, such as synchronous (SDLC) or asynchronous modems. It operates at up to 19,200 baud in either packet or character mode and accepts Hayes-compatible operations

described in this section under Modem Interface. The port does not support DTR signals, but has two clock inputs to support synchronous operation in a "slave" mode.

380 DIAL
385 LAN
385 DIAL
390 DIAL
395 LAN
395 DIAL
460 DIAL

PIN Pad/Bar Code Port

The 6-pin DIN connector is an RS-232 serial port that does not have any handshake lines. It operates at 9600 baud in either character or packet mode. Although designed to connect to either a PIN pad-type entry device or bar code external device, it can be used as a general-purpose communication port. The OMNI 460 terminal uses this port for its internal printer.

380 DIAL
385 DIAL
390 DIAL
395 DIAL
460 DIAL

Internal Modem (COM4)

The internal modem is a full-function, Hayes-compatible modem supporting character, packet, VISA 1st Generation, SDLC, and VeriFone's ZONTALK 2000 communications.

385 LAN
395 LAN

LAN Port (COM4)

This is a LAN communication port supporting a proprietary VeriFone peer-to-peer LAN protocol (CSMA/CA) at transmission speeds up to 19,200 baud.

Modes of Operation

Communications ports, with the exception of the LAN port on corresponding models, support both character mode and packet mode operation. In addition, communications protocols such as VISA 1st Generation and SDLC, discussed later in this section, are supported by these ports, either through the internal modem (in corresponding models) or the serial communications ports. The LAN

port only supports a VeriFone-proprietary peer-to-peer LAN protocol and is discussed in *Chapter 10*.

Character Mode

In character mode, all inputs and outputs are treated as individual bytes; the data is not handled as a packet. From a system viewpoint, character mode data is simply a stream of input bytes and a stream of output bytes; no data validation or additional processing is performed. All "intelligence" must reside in the application.

At the application level, data may be written or read up to 1016 characters at a time. The maximum number of characters can be increased via the *B entry in the CONFIG.SYS file (254 times the number of buffers allocated with *B). The limit for a single read or write is 4064 bytes.

When writing data in character mode, it is best to write several characters (up to 254) at a time in order to use the system communications buffers most efficiently. The *B entry in the CONFIG.SYS file will likely need to be increased when using large communication packets (over 254 bytes).

Packet Mode

This communication mode sends and receives data in predefined packets. A packet is defined by a start character (usually STX), an end character (usually ETX), and a

```
buffer.trailer.packet_parms.count = 0;
   /* no checksum */
buffer.trailer.packet_parms.count = 1;
   /* 1-byte LRC  */
buffer.trailer.packet_parms.count = 2;
   /* 2-byte CRC  */
```

packet check field.

In packet mode, start and stop characters (see <ascii.h>) must be defined along with a checksum characteristic:

The communication device drivers do not verify the packet check characters, but merely use the parameter to determine how many characters following the ETX are to be included in the record to be passed to the caller on the

1 byte	*up to 4060 bytes*	*1 byte*	*2 bytes*
STX	packet data	ETX	packet check

next device read operation. If checking is desired, the application must provide it; possibly using the SVC_CRC_CALC() routine.

A typical packet looks like:

In packet mode, the input stream is assembled into buffers. If the next character in the stream is a STX, then the STX, all characters which follow up to and including the ETX, and the defined number of check bytes are assembled into a record and passed to the application on the next valid read() operation. Each read() should normally allow for the largest possible packet, otherwise, characters may be truncated. Packets cannot include more than 4060 data bytes.

If the next character in the input stream is not an STX, the character is delivered to the application as a 1-byte packet. Typically, such packets will contain control characters like ACK or NAK.

Output in packet mode consists of writing a prepared buffer delivered by the caller's write() operation. The application must build the packet, including STX, data, ETX, and check characters before sending the packet to the device.

The operating system neither generates nor checks the LRC/CRC values.

Initializing Device Settings

Use the structure `Opn_Blk` (defined in `<device.h>`) to specify the baud rate, data format, and other parameters for TXO communication devices. Settings are initialized, or reset, using an `ioctl()` call with the `type` parameter set to `SetCtrl|0`. The structure is passed to the device as `&buffer`. This call must be made prior to any device read or write. The current structure can be obtained using an `ioctl()` call with the type parameter set to `GetCtrl|1`.

❖ *Not all* `Opn_Blk` *settings described in this section may be relevant to a particular device. For example, not all devices support all listed protocols or synchronous communication.*

The `<device.h>` definition for this structure is shown below:

```
struct Opn_Blk
  { RAW rate;          /* baud rate     */
    RAW format;        /* data format   */
    RAW protocol;      /* protocol code */
    RAW parameter;
    union
     {
       struct                  /* if packet mode protocol: */
         { RAW stx_char;       /* start char for packets   */
           RAW etx_char;       /* end char for packets     */
           RAW count;          /* no. of bytes in checksum */
         } packet_parms;
       struct
         { unsigned int other_parm[8];
         } other_parms;
     } trailer;
  };
```

The following example defines port initialization as 1200 baud, async, 7E2, packet mode with STX and ETX, and 1-byte LRC check:

```
mdm_Opn_Blk.rate = Rt_1200;
mdm_Opn_Blk.format = Fmt_A7E1 | Fmt_2stp;
mdm_Opn_Blk.protocol = P_pakt_mode;
mdm_Opn_Blk.parameter = 0;
mdm_Opn_Blk.trailer.packet_parms.stx_char = STX;
mdm_Opn_Blk.trailer.packet_parms.etx_char = ETX;
mdm_Opn_Blk.trailer.packet_parms.count    = 1;
```

Setting Baud Rate

Baud rates are set by the rate field as follows:

Constant	Baud Rate
Rt_300	300
Rt_600	600
Rt_1200	1200
Rt_2400	2400
Rt_4800	4800
Rt_9600	9600
Rt_19200	19,200

Setting Data Bits and Parity

The number of data bits and parity are determined by the format field as follows:

Constant	Data Bits	Parity
Fmt_A7E1	7	even
Fmt_A7N1	7	none
Fmt_A7O1	7	odd
Fmt_A8E1	8	even
Fmt_A8N1	8	none
Fmt_A8O1	8	odd
For synchronous format:		
Fmt_SDLC	8	no

All constants listed above assume the use of one stop bit. If two stop bits are required, the constant Fmt_2stp may be added to any of the constants listed above, e.g., Fmt_A7E1|Fmt_2stp.

Additional values may be included to modify any of the basic data formats:

Constant	Operation
Fmt_2stp	Two stop bits
Fmt_auto	Auto-enable flow control
Fmt_DTR	DTR assertion
Fmt_RTS	RTS assertion

Setting Protocol (Mode of Operation)

The protocol field specifies the communications protocol (character mode, packet mode, etc.) as follows:

P_char_mode	Character mode
P_pakt_mode	Packet mode
P_visa1gen	Visa First Generation protocol
P_sdlc_sec	Hypercom Fast Connect protocol
P_sdlc_norm	Hypercom Normal Connect protocol

Visa First Generation Initialization

In OMNI 300 Series terminals, support for the VISA First Generation protocol has been enhanced beyond the VISA specification.

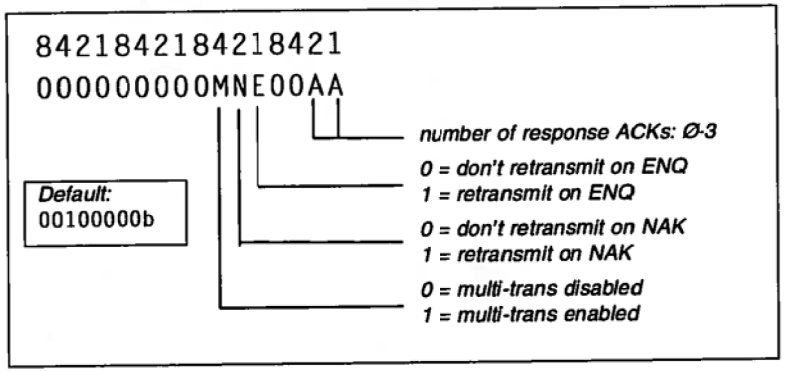
Enhancements to the protocol allow you to customize the following options (with Opn_Blk defaults listed on the right):

How many response ACKs (0 to 3)?—Default: 0
Retransmit on ENQ?—Default: No
Retransmit on NAK?—Default: Yes
Enable multi-transaction mode—Default: No

To modify the default values, use the Opn_Blk parameter:

```
buffer.trailer.other_parms.other_parm[0]
```

Format the Opn_Blk parameter as follows, noting that only five bits in the low-order byte are used:



Default modes are defined as 00100000b (0x20), indicating that no ACKs are sent, do not retransmit in response to ENQs, NAKs are not transmitted, and the multi-transaction mode is not enabled.

To change the defaults, the Opn_Blk other_parm[0] must be modified. For example, to enable three ACKs, retransmit on ENQs, retransmit on NAKs, and enable multi-transaction mode, add the following setting to the Opn_Blk prior to initializing the modem:

```
buffer.trailer.other_parms.other_parm[0] = 0x73;
```

Standard VISA mode varies slightly from the default mode in that one response ACK should be sent. To change the mode to be in line with the standard, other_parm[0] should be set to 0x21:

```
buffer.trailer.other_parms.other_parm[0] = 0x21;
```

SDLC Initialization

When using SDLC, the following additional setting is required in the Opn_Blk structure:

```
buffer.parameter = 0x30;
```

This parameter defines the SDLC secondary station address.

To connect to any SDLC host other than a Hypercom NAC, use the following in the Opn_Blk structure:

```
modem_blk.protocol = P_sdlc_norm;
```

To connect to a Hypercom NAC, use the following in the Opn_Blk structure:

```
modem_blk.protocol = P_sdlc_sec;
```

The following example shows a typical Opn_blk setup for access to a Hypercom NAC:

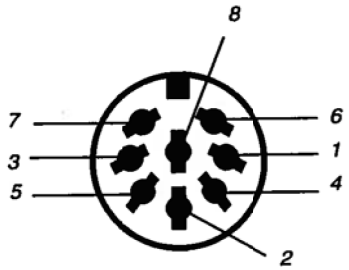
```
struct Opn_Blk modem_blk; /*Modem Control Block */

memset (modem_blk, '\0', sizeof(modem_blk));
/* Configure modem for sync communications */
modem_blk.rate = Rt_1200;
                /* 1200 baud */
modem_blk.format = Fmt_SDLC;
                /* SDLC format */
modem_blk.protocol = P_sdlc_sec;
                /* for Hypercom access */
modem_blk.parameter = 0x03;
                /* Poll ID - All NACs are 0x30 */
```

380 DIAL
385 LAN
385 DIAL
390 DIAL
395 LAN
395 DIAL
460 DIAL

Serial Port

All OMNI 300 Series terminals implement RS-232 communication support via the COM1 (printer) port, with slight model-specific variations. Some models also implement a COM3 serial port. Table 9-1 identifies the ports available per model.



Serial Port Pinouts			
1	GND	2	DCD
3	RTS	4	CTS
5	RXD	6	TXD

Three types of serial communications are supported: COM1 with DTR signaling, COM1 without DTR signaling, and COM3. All three types may be used in either synchronous or asynchronous communications with external modems or other communications devices, and all three use the same DIN-8 connector for the physical interface. See Appendix E for port specifications.

Serial Communications Function Calls

open()

Prepares a serial port for use by a device. The port remains inactive until an `ioctl()` call is made to set the baud rate and protocol parameters.

```
hCOM = open ("/dev/com1", 0);
int  hCOM;
```

The device name in the example above is `com1`, however, `com3` may be substituted.

read()

Transfers the contents of the internal device buffer to the application's buffer.

```
#include <io.h>

bytes_read = read (hCOM, buffer, size);
int  bytes_read, hCOM, size;
char *buffer;
```

Each `read()` transfers `size` bytes from the RS-232 port buffer to the application buffer and returns the number of bytes actually read, 0 if no data is available, or -1 if an error occurs.

Unread buffered data is handled differently, depending upon the mode. In packet mode, up to `size` bytes are put into `buffer`; any portion of the packet larger than `size` bytes is truncated and lost. In character mode, up to `size` bytes are transferred into the buffer; if more data is available, it remains in the port's buffer for the next `read()` operation.

write()

Transfers the contents of the application's buffer to the internal device buffer.

```
#include <io.h>

bytes_written = write (hCOM, buffer, size);
int bytes_written, hCOM, size;
char *buffer;
```

Each `write()` transfers `size` bytes from `buffer` to the port's buffer. If the operation is successful, `bytes_written` equals `size`. If the serial device driver has no available buffer space, or an error occurs, -1 is returned, with `errno` set accordingly.

Once in the device buffer, the data is transferred to the transmitter each time the transmit buffer goes empty.

close()

Releases the resources associated with the communications port handle. The prototype is:

```
#include <io.h>

result = close (hCOM);
int result, hCOM;
```

ioctl()

Several device operations may be performed using `ioctl()`, including:

- Port initialization
- Resetting error conditions
- Resetting BRK, RTS and DTR* control signals
- Retrieving device status

** Note: Only COM1 of certain models supports the DTR signal. See Table 9-1.*

Function prototype is:

```
#include <io.h>

result  =  ioctl (hCOM, type, buffer);
int  result,h COM;
unsigned int type;
void *buffer;
```

This function normally returns 0 if the operation succeeds, or -1 if an error occurs. One exception is the `GetCtrl | 0` command, which returns a 1 if output is pending for the device.

ioctl() operations are specified via the type parameter, which consists of a set of commands described briefly in the table below, and in detail on the following pages.

RS-232 ioctl() Commands	
Command	**Operation**
SetCtrl | 0	Port initialization
SetCtrl | 1	Reset port error conditions
SetCtrl | 2	Set/Reset BRK, DTR* or RTS signals
GetCtrl | 0	Check output pending, read status bytes
GetCtrl | 1	Gets Opn_Blk structure

** Note: Only COM1 of certain models supports the DTR signal. See Table 9-1 on page 9-1.*

SetCtrl | 0 — Port Initialization

This command initializes the serial port.

The type parameter is set to SetCtrl | 0x0000, and buffer contains an Opn_Blk structure (defined in <device.h>). This structure controls port baud rate, data format, protocol and other parameters for subsequent RS-232 operations. See Initializing Device Settings for a description of Opn_Blk structure values.

SetCtrl | 1 — Reset Error Conditions

Parity, framing, and overrun errors may be reset by setting the ioctl() type parameter to SetCtrl | 0x0001. A buffer address must be provided to satisfy the compiler, although the buffer parameter is not actually used.

SetCtrl | 2 — Control Signal Set/Reset

This type parameter sets or resets the port's line break level (BRK), RTS and DTR* control signals. The buffer must contain a single byte with its bits set accordingly:

Byte	BRK	RTS	DTR
00	-	-	-
01	-	-	+
02	-	+	-
03	-	+	+
04	+	-	-
05	+	-	+
06	+	+	-
07	+	+	+

** Note: Only COM1 of certain models supports the DTR signal. See Table 9-1.*

❖ *Asserting BRK does not imply any form of system supplied timeout for stopping the condition. The programmer must provide this mechanism.*

GetCtrl | 0 — Obtain Port Status

When type = GetCtrl | 0, ioctl() returns 1 if there is pending output, or 0 if no output is pending.

In addition, this command copies current port status information to a four-byte buffer as follows:

Byte 1: Number of input messages pending
Byte 2: Number of failed output messages pending
Byte 3: Number of output slots available
Byte 4: Current signal information:

0x80	Set if break/abort detected
0x40	Reserved, always zero
0x20	Set if CTS detected*
0x10	Set if ring indicator present
0x08	Set if DCD present*
0x04	Set if framing error detected

Frame, overrun and parity errors are latched until reset by an "intellegent" protocol, or by calling `ioctl(hCOM, SetCtrl | 1, 0)`.

0x02	Set if overrun error detected
0x01	Set if parity error detected

** Note: Available on COM1 only. For models without DTR support, you must set* `Fmt_auto` *to be able to read CTS and DCD.*

Parity errors are latched until reset by calling `ioctl()` with `SetCtrl | 1`, as described earlier in this section.

The returned value in `result` will indicate whether or not any output is currently queued or being transmitted. The example below demonstrates how to close the port while ensuring that all buffered data is sent:

```
while (ioctl(hCOM, GetCtrl+0, &buffer) != 0)
     printf ("\fOUTPUT PENDING");
close (hCOM);
```

`GetCtrl | 1` — Obtain the Current Opn_Blk

Setting `type` to `GetCtrl | 0x0001` copies the current `Opn_Blk` structure into the caller's buffer.

Serial Communications Example

```
/*************************************************************/
/* RS232CHR.C                                                */
/* This program demonstrates the use of the RS-232 device    */
/* communications between two OMNI 380 terminals. This       */
/* program operates in character mode and communicates to    */
/* RS232PKT.C which functions in packet mode.                */
/*                                                           */
/* Download the programs to separate 380s and start RS232CHR */
/* first, then RS232PKT.  RS232PKT will send:                */
/*    <STX>HELLO<ETX><lrc>                                   */
/* RS232CHR will respond with:                               */
/*    <STX>WORLD<ETX><lrc>                                   */
/*************************************************************/

#include <stdio.h>
#include <config.h>
#include <device.h>
#include <io.h>
#include <ascii.h>
#include <strings.h>
#include <txostd.h>
#include <txosvc.h>

int rs232_handle,bytes_read,i;
char buffer[20];
struct Opn_Blk rs232_Opn_Blk;

main () {

   /* Open the RS-232 port (com1) */
   rs232_handle = open("/dev/com1", 0);

   /* Define the protocol as:
         19200 baud, async, 7 data bits, even parity,
         2 stop bits, and character mode.
   */
```

```
    rs232_Opn_Blk.rate = Rt_19200;
    rs232_Opn_Blk.format = Fmt_A7E1 | Fmt_2stp;
    rs232_Opn_Blk.protocol = P_char_mode;
    rs232_Opn_Blk.parameter = 0;
    ioctl(rs232_handle, SetCtrl | 0, &rs232_Opn_Blk);

    /* Read the first 8 bytes received. */
    /* Expecting <STX>HELLO<ETX><lrc>   */
    for (i=0; i<8; i++)
       while (read(rs232_handle, &buffer[i], 1) == 0);

    /* Display the "HELLO" */
    clrscr();
    write(STDOUT,&buffer[1],5);

    /* Create a buffer with STX, "WORLD", ETX and LRC. */
    buffer[0]=STX;
    strcpy(&buffer[1],"WORLD");
    buffer[6]=ETX;
    buffer[7]=SVC_CRC_CALC(0, &buffer[1], 6);

    /* Output the buffer. */
    write(rs232_handle, buffer, 8);

    /* Wait for all the output to clear out before */
    /* we close the port or data may be lost.      */
    while (ioctl(rs232_handle, GetCtrl | 0, buffer));
    close(rs232_handle);

    /* Just for fun, display the buffer received from */
    /* the GetCtrl | 0 above.                         */
    SVC_WAIT(1500); printf("\f%d INPUT MSGS PND",buffer[0]);
    SVC_WAIT(1500); printf("\f%d FAILED OUTPUTS",buffer[1]);
    SVC_WAIT(1500); printf("\f%d SLOTS AVAIL",   buffer[2]);
    SVC_WAIT(1500); printf("\fSIGNAL INFO: %x",  buffer[3]);
}
```

```
/************************************************/
RS232PKT.C
This program demonstrates the use of the RS-232
device by communicating with another OMNI terminal.
This program operates in packet mode and
communicates to RS232CHR.C which functions in
character mode.

Download the programs to separate terminals and start
RS232CHR first, then RS232PKT.
RS232PKT will send:
   <STX>HELLO<ETX><lrc>
RS232CHR will respond with:
   <STX>WORLD<ETX><lrc>
/************************************************/

#include <config.h>
#include <device.h>
#include <io.h>
#include <ascii.h>

int rs232_handle,bytes_read;
char buffer[20];
struct Opn_Blk rs232_Opn_Blk,ob_copy;

main () {
   /* Open the RS-232 port (com1) */
   rs232_handle = open("/dev/com1", 0);

/* Define the protocol as:
     19200 baud, async, 7E2, packet mode with STX
     and ETX, and 1-byte LRC check. */
   rs232_Opn_Blk.rate = Rt_19200;
   rs232_Opn_Blk.format = Fmt_A7E1 | Fmt_2stp;
   rs232_Opn_Blk.protocol = P_pakt_mode;
   rs232_Opn_Blk.parameter = 0;
   rs232_Opn_Blk.trailer.packet_parms.stx_char = STX;
   rs232_Opn_Blk.trailer.packet_parms.etx_char = ETX;
   rs232_Opn_Blk.trailer.packet_parms.count    = 1;
   ioctl(rs232_handle, SetCtrl | 0, &rs232_Opn_Blk);

   /* Create a buffer with STX, "HELLO", ETX & LRC. */
   buffer[0]=STX;
   strcpy(&buffer[1],"HELLO");
   buffer[6]=ETX;
   buffer[7]=SVC_CRC_CALC(0, &buffer[1], 6);
```

```
    /* If LRC is valid, display the data. */
    clrscr();
    if (SVC_CRC_CALC(0, &buffer[1], bytes_read-2) ==
        buffer[bytes_read-1])
       write(STDOUT, &buffer[1], bytes_read-3);
    else
       write(STDOUT, "LRC ERROR", 9);
    SVC_WAIT(1500);

    /* Get a copy of the current open block.  */
    /* Just here to show how it is done.      */
    ioctl(rs232_handle, GetCtrl | 1, &ob_copy);

    /* Close the RS-232 device. */
    close(rs232_handle);
}
```

380 DIAL
385 DIAL
385 LAN
390 DIAL
395 LAN
395 DIAL
460 DIAL

Sync/Async Serial Port (COM3)

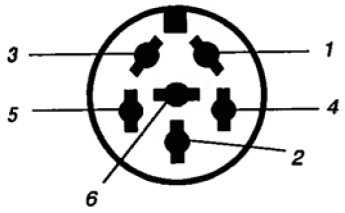
OMNI 3X5 LAN terminals and the OMNI 395 dial-type terminal include a second serial (RS-232) port for synchronous or asynchromous communications to external devices, such as a synchronous (SDLC) modem, asynchronous modem, or other RS-232 device. Characteristics of this port are:

- Up to 19,200 baud rate
- Support for either packet or character mode
- Accepts Hayes-compatible commands
- Does not support DTR signals
- Two clock inputs to support synchronous operation in a "slave" mode.

All serial port library functions are available for COM3 operation. Note that the COM3 port does not support a DTR signal.

PIN Pad Port

All OMNI 300 Series terminals are designed to work with VeriFone PINpad 101™ and PINpad 201™. Any other PIN pad functioning in a similar manner may also be connected to the PIN pad port.



PIN Pad Port Pinouts			
1	+ 5 volts 4.7 ohm	2	Bar code receive
3	PIN pad receive	4	PIN pad transmit
5	Ground	6	+9 volts unregulated

Although the PIN pad port does not provide control signals (such as RTS, CTS, etc.), it may be used for general purpose serial communications or as a printer port. The OMNI 460™ terminal utilizes this port for its internal printer. See Appendix C for more information on the OMNI 460 terminal.

PIN Pad Port Function Calls

open()

Prepares the PIN pad port for use. The port remains inactive until a call to `ioctl()` is made to set the baud rate and protocol parameters.

```
#include <io.h>

bytes_read = read (hPIN, buffer, size);
int bytes_read, hPIN, size;
char *buffer;
```

read()

Transfers the contents of the internal device buffer to the application's buffer.

```
#include <io.h>

bytes_read = read (hPIN, buffer, size);
int bytes_read, hPIN, size;
char *buffer;
```

Each `read()` transfers `size` bytes from the PIN pad buffer into the application buffer and returns the number of bytes actually read: 0 if no data is available, or -1 if an error occurs.

Unread buffered data is handled differently, depending upon the mode. In packet mode, up to `size` bytes are put into `buffer`; any portion of the packet larger than `size` bytes is truncated and lost. In character mode, up to `size` bytes are written into the buffer; if more data is available, it remains in the port's buffer for the next `read()` operation.

write()

Transfers the contents of the application's buffer to the internal PIN pad buffer.

```
#include <io.h>

bytes_written = write (hPIN, buffer, size);
int bytes_written, hPIN, size;
char *buffer;
```

Each `write()` transfers `size` bytes from `buffer` to the PIN pad buffer. If the operation is successful, `bytes_written` equals `size`. If the PIN pad device driver has no available buffer space, or an error occurs, -1 is returned.

Once in the PIN pad buffer, the data is transferred to the transmitter each time the transmit buffer goes empty.

close()

Releases the internal resources associated with the internal PIN pad device handle:

```
#include <io.h>

result = close (hPIN);
int result, hPIN;
```

ioctl()

Device operations that may be performed using `ioctl()` include:

- Device initialization
- Resetting error conditions
- Retrieving device status

To initialize the PIN pad port, the following prototype is used:

```
#include <io.h>

result = ioctl (hPIN, SetCtrl | type, buffer);
int result, hPIN;
unsigned int type;
void *buffer;
```

This function normally returns 0 if the operation succeeds, or -1 if an error occurs. One exception is the `GetCtrl | 0` command, which returns a 1 if output is pending for the device.