Device Driver for the MS-DOS® Operating System Installation Guide (Hostess *550* 16-Port)

Scope

Use this document to install the device driver for the MS-DOS operating system (the device driver was developed and tested using levels 5.0 and 6.2).

Prerequisites

This document assumes you have already installed MS-DOS and you have a basic understanding of the operating system.

Audience

This document is primarily for the System Administrator or the person who installs software and hardware on the system. The secondary audience includes the system user.

Organization

This guide contains the following information to install and use the device driver:

Section 1. Overview - Contains an installation overview and lists device driver features.

Section 2. Installation - Explains how to install the device driver.

Section 3. Developing Applications - Provides information to develop applications for the device driver in BASIC, C, and Assembly languages.

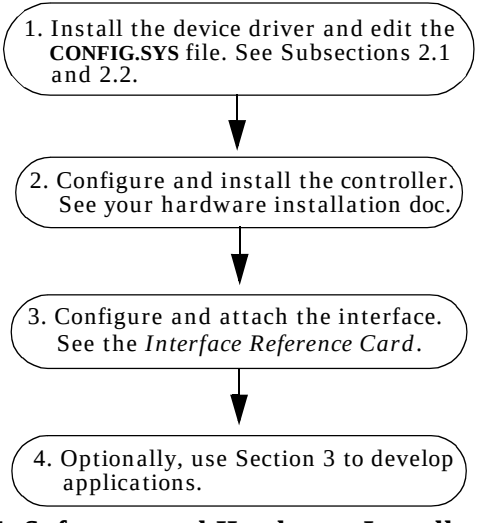
Section 4. Troubleshooting and Technical Support - Contains information that may help you resolve installation or operations problems. In addition, it lists information that you should gather before calling for technical support.

Section 1. Overview

1.1. Introduction

The Hostess *550* supports a 16550 UART.

Flowchart 1-1 shows an installation overview.



Flowchart 1-1. Software and Hardware Installation Overview

Note: *See* Section 4. Troubleshooting *if you have installation problems.*

1.2. Device Driver Features

The device driver has the following features:

- Fully interrupt driven
- The Hostess *550* supports FIFO mode

The device driver is a file I/O driver that supports DOS **IOCTL**. All communication parameters (including baud rate, parity, number of bits in character, stop bit, flow control, protocols, and so forth) can be changed by using an **IOCTL** with the proper function call (see Section 3. Developing Applications).

Once parameters are set, they stay set until they are changed by another **IOCTL** or the system is reset.

Section 2. Software Installation

This section contains the following information:

- Installing the device driver
- Setting up the **CONFIG.SYS** file
- Resolving errors associated with the **CONFIG.SYS** file

2.1. Device Driver Installation

Use the following steps to install the device driver:

1. Insert the device driver diskette.
2. Create a new directory at the **C:** prompt

   md \comtrol
3. Change to the Comtrol directory

   cd \comtrol
4. Copy the contents of the device driver to the Comtrol directory

   copy a: *.*

   where **a** is the drive the diskette is located in.
5. Edit the **CONFIG.SYS** file in your root directory or create a **CONFIG.SYS** file if one does not exist. See Subsection 2.2 to set up this file.
6. Reboot the system to initialize the **CONFIG.SYS** file.

Note: *The device driver installation is based on the MS-DOS operating system, level 6.0.*

After the device driver is installed, see your hardware installation documentation to configure and install the controller. Once the device driver and controller are both installed, a message appears on the screen during the startup process indicating that the device driver is active. If you get an error, see Subsection 2.3.

2.2. Setting Up the CONFIG.SYS File

The device driver is contained in three binary files:

- **COM.BIN**
- **HOSTESS.BIN**
- **PORT.BIN**

Software configuration involves the addition of several **DEVICE** statements to your **CONFIG.SYS** file in the root directory. These device statements specify the names of the three files contained in the device driver.

CONFIG.SYS contains three types of statements which configure the device driver:

device=*path*\com.bin
device=*path*\hostess.bin /required parameters (See Table 2-1)
device=*path*\port.bin /required parameters (See Table 2-2) **/optional parameters** (See Table 2-3)

Note: *For any number of controllers being installed, the* **device=com.bin** *statement is required only once in the* ***CONFIG.SYS*** *file. A* **device=hostess.bin** *statement is needed for each controller that is installed and a* **device=port.bin** *statement is required for each port that is installed.*

The following shows an example **CONFIG.SYS** file for a Hostess *550* 16-port controller. Although this example uses the same parameters for each port, you may choose different parameters for your own needs (see Tables 2-1 through 2-3).

device=\comtrol\com.bin
device=\comtrol\hostess.bin /3/2c0h
device=\comtrol\port.bin /port1/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port2/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port3/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port4/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port5/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port6/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port7/9600/n/8/1/1024/1024/0/0/ioctl

```
device=\comtrol\port.bin /port8/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port9/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port10/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port11/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port12/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port13/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port14/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port15/9600/n/8/1/1024/1024/0/0/ioctl
device=\comtrol\port.bin /port16/9600/n/8/1/1024/1024/0/0/ioctl
```

Example 2-1. Sample CONFIG.SYS File

The device statements in the **CONFIG.SYS** file should follow these rules:

- The required and optional parameters are separated by a slash.
- Spaces are not allowed between the slash and the parameter it delimits, however, any number of spaces may appear between a parameter and the next slash.
- The required parameters must be specified in the order listed in Tables 2-1 and 2-2.
- The optional parameters may be specified in any order.
- The **device=com.bin** statement provides the majority of the processing for the device driver. For any number of controllers being installed, this statement appears only once and must precede all other configuration statements in the file that pertain to the controller. No parameters are required.

- The **device=hostess.bin** statement provides the data structures and processing for an individual controller. This statement must be followed by the **device=port.bin** statements (16 statements) that pertain to the controller. Both of these statements are required for each controller installed.

  Refer to Section 2. Software Installation for information about **device=hostess.bin parameter1** and **parameter2** (interrupt request and base address).

Table 2-1. HOSTESS.BIN Required Parameters

Parameter	Description
parameter1 - COMM INTERRUPT REQUEST #	This parameter defines the I/O channel interrupt request line for the controller. This interrupt request line must not conflict with any other device in the system. Allowable values are 2, 3, 4, 5, 10, 11, or 12.(Refer to the hardware installation documentation for switch setting information.)
parameter2 - COMM BASE ADDRESS	This parameter defines the base I/O address for the group of I/O addresses that will be occupied by the controller. Each serial port occupies a contiguous block of eight I/O addresses: • A 16-port controller occupies 128 contiguous I/O addresses. These I/O addresses must not conflict with any other device in the system. (Refer to your hardware installation documentation for switch setting information.) Allowable values include one of the following: • Hexadecimal numbers ranging from 0 through 1FFFh • Decimal numbers ranging from 0 through 8191 Hexadecimal numbers are specified by adding an upper or lowercase “h” (280h for example).

Note: *Optional parameters do not exist for* ***HOSTESS.BIN****.*

The **PORT.BIN** file provides the data structures and processing for an individual serial port. Table 2-2 shows the required parameters for **PORT.BIN**.

Table 2-2. PORT.BIN Required Parameters

Parameter	Description
parameter1 - DEVICE NAME	This parameter specifies a 1 to 8 character device name for the serial port. This device name must not conflict with any existing file or device in the system.
parameter2 - BIT RATE	This parameter specifies the bit rate (or baud rate) in bits per second for the serial port. Allowable values include 50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200, 38400, or 56000.
parameter3 - PARITY	This parameter specifies the parity used by the serial port. Allowable values include one of the following: • **e** (specifying even parity) • **o** (specifying odd parity) • **n** (specifying no parity) These letters may be upper or lowercase.
parameter4 - DATA BITS	This parameter specifies the number of data bits used by the serial port. Allowable values include 5, 6, 7, or 8.
parameter5 - STOP BITS	This parameter specifies the number of stop bits used by the serial port. Allowable values are 1 or 2.

Table 2-2. PORT.BIN Required Parameters (Continued)

Parameter	Description
parameter6 - TRANSMIT BUFFER SIZE	This parameter specifies the transmit buffer size. Allowable values include 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, or 32768. If any other value is specified, it is rounded up to the next higher power of two.
parameter7 - RECEIVE BUFFER SIZE	This parameter specifies the receive buffer size. Allowable values are the same as those for TRANSMIT BUFFER SIZE.
parameter8 - TRANSMIT TIME-OUT VALUE	This parameter specifies the number of clock *ticks* that the device driver will wait before returning control to the application upon a transmit buffer full condition. The device driver will wait indefinitely if this value is FFFFh (65,535 decimal). Approximately 18.2 clock *ticks* occur per second on the IBM PC. Allowable values range from 0 through FFFFh (0 through 65535 decimal). When executing our sample programs, set this value to 0. The **IOCTL** option must be enabled for the serial port being used. The sample programs expect the **read** and **write** calls to return immediately, even if all of the data has not been moved. When the output of DOS commands are sent to a port on the controller (**copy *file* port1** or **type *file* > port1**), best results may be obtained by specifying **ffffh** for this value. DOS commands expect the **write** to return only when all of the data is moved. If the **write** returns before it is complete, an **abort**, **retry**, or **fail** message will appear. In addition, the **IOCTL** option should not be enabled.

Table 2-2. PORT.BIN Required Parameters (Continued)

Parameter	Description
parameter9 - RECEIVE TIME-OUT VALUE	This parameter specifies the number of clock *ticks* that the device driver will wait before returning control to the application upon a receive buffer empty condition. The device driver will wait indefinitely if this value is FFFFh (65,535 decimal). Approximately 18.2 clock *ticks* occur per second on the IBM PC. Allowable values range from 0 through FFFFh (0 through 65,535 decimal). When executing our sample programs, set this value to 0. The **IOCTL** option must be enabled for the serial port being used. The sample programs expect the **read** and **write** calls to return immediately, even if all of the data has not been moved. When the output of DOS commands are sent to a port on the controller (**copy *file* port1** or **type *file* > port1**), best results may be obtained by specifying **ffffh** for this value. DOS commands expect the **write** to return only when all of the data is moved. If the **write** returns before it is complete, an **abort**, **retry**, or **fail** message will appear. In addition, the **IOCTL** option should not be enabled.

Table 2-3 shows the optional parameters which may be specified for **PORT.BIN** if the function is required.

Table 2-3. PORT.BIN Optional Parameters

Parameter	Description
ioctl	This parameter is set to allow the device driver to recognize the DOS **IOCTL** function. **IOCTL** requests are used to perform input and output to a device through a separate channel from normal data input and output. **IOCTL** input and output are device dependent, but are usually used to exercise control of various features of the device. The device driver returns error and status information through the **IOCTL** input channel. The **IOCTL** output allows the application program to flush a transmit or a receive buffer, change a communication parameter (baud rate, parity, and so on), and change an optional parameter (**echo** option and so on). See Subsection 3.7.3.
echo	This parameter is set to allow the device driver to echo received characters back to the sender.
hifc (Hardware Input Flow Control)	This parameter enables hardware flow control of inbound data, through the DTR signal. If you use this option, the device driver deactivates the DTR signal when the receive buffer reaches 7/8 of capacity. When the receive buffer drops to 3/8 of capacity, the DTR signal is activated again.
hofc (Hardware Output Flow Control)	This parameter enables hardware flow control of outbound data, through the CTS signal. If you use this option, the device driver suspends transmission of data whenever it detects an inactive CTS signal. Transmission resumes when the CTS signal becomes active again.

Table 2-3. PORT.BIN Optional Parameters (Continued)

Parameter	Description
sifc (Software Input Flow Control)	This parameter enables software flow control of inbound data, through XON and XOFF (DC1 and DC3) characters. If you use this option, the driver transmits an XOFF character (ASCII DC3) when the receive buffer reaches 7/8 of capacity. When the receive buffer drops to 3/8 of capacity, the driver transmits an XON character (ASCII DC1).
sofc (Software Output Flow Control)	This parameter enables software flow control of outbound data, through XON and XOFF (DC1 and DC3) characters. If you use this option, the device driver suspends transmission of data whenever it receives an XOFF character (ASCII DC3). Transmission resumes when an XON (ASCII DC1) is received.
dtr (Data Terminal Ready)	This parameter causes the DTR signal to be asserted on the port.
rts (Request To Send)	This parameter causes the RTS signal to be asserted on the port.

2.3. Configuration Errors Associated with CONFIG.SYS

The following error message may appear:

'Bad or missing FILENAME'

This is an error that is issued by DOS when it is trying to load a device driver file that is specified by a **DEVICE=FILENAME** statement in **CONFIG.SYS**. The error is issued if DOS is unable to find the specified file. If this error occurs, correct the file name specified in the **DEVICE=FILENAME** statement in **CONFIG.SYS**.

Error messages may also be issued by the configuration software when the device driver files load. These error messages are issued when a required parameter is missing, incorrectly specified, or out of order. The error messages have the following form:

'configuration error: device : parameter name - message'

If error messages of this type appear, check the parameter specified by **parameter name** for the device specified by **device**.

Parameter errors that occur on **device=hostess.bin** statements will have a device name of HOSTESS*. If the **DEVICE NAME** parameter is missing from a **device=port.bin** statement, that port will be assigned the default device name PORT*x*.

If problems persist, see Section 4. Troubleshooting before calling technical support.

Once the device driver is installed, see your hardware installation documentation to install the controller.

Section 3. Developing Applications

3.1. Introduction

The device driver interfaces with any programming language which supports the following DOS **INT 21h** low level file I/O functions:

- **open** (prepares a port for I/O operation)
- **close** (terminates logical connection)
- **read** (retrieves data from a port)
- **write** (sends data to a port)
- **IOCTL** (accesses device specific functions and information)

The interface to the driver will vary depending on the language being used.

This section provides detailed information about interfacing to BASIC, C, and Assembly languages, including sample programs. Error handling and the use of the DOS **IOCTL** functions are also described.

Note: *See Subsection 3.7 for general DOS **IOCTL** information that is not language specific.*

3.1. Verifying Device Driver Installation

Verify the operation of the device driver and the validity of the parameters in the **CONFIG.SYS** file by using the loopback plug and the **TERM.EXE** sample application.

Note: *Before executing any of Comtrol's sample applications, set the Transmit Time-Out Value and the Receive Time-Out Value to 0 (see Table 2-2 for more information).*

1. Attach the loopback plug to the serial port you want to test.
2. Enter **term** to invoke the **TERM.EXE** terminal program, which was placed in the **\Comtrol** directory during the device driver installation. The **enter device name** message displays.
3. Enter the name of the port to test, as it was defined in the **CONFIG.SYS** file (**PORT.BIN**

parameter1).

4. Enter a character from the keyboard.

   If the character entered is displayed twice on the screen, the installation is correct.

   If you see only one character:

   - Check the switch settings on the controller (you may need to choose different switch settings), and the parameters specified in the **CONFIG.SYS** file.
   - Reboot the system and try again.

If problems persist, refer to Section 4. Troubleshooting for troubleshooting information.

3.2. Low Level File I/O Functions

The Comtrol device driver supports the following DOS **INT 21h** functions, which are commonly referred to as low level file I/O functions:

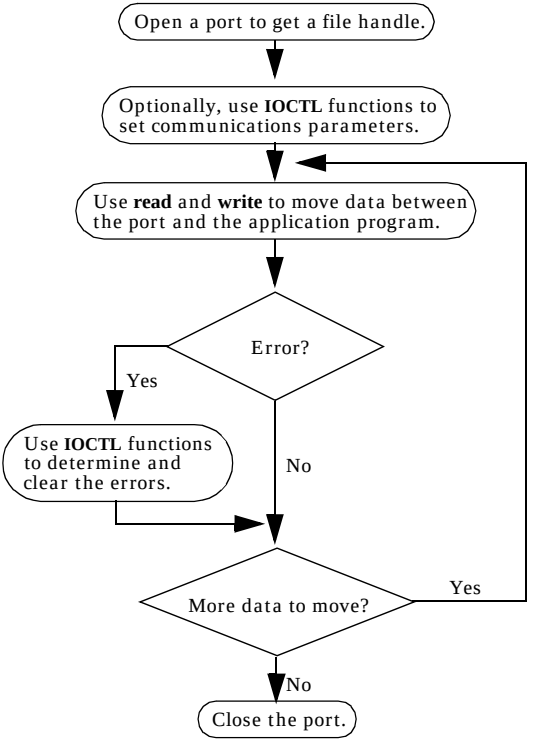
- **INT 21h** function **3Dh - open**
- **INT 21h** function **3Eh - close**
- **INT 21h** function **3Fh - read**
- **INT 21h** function **40h - write**
- **INT 21h** function **44h - IOCTL**
  - Subfunction **00h - get device information**
  - Subfunction **01h - set device information**
  - Subfunction **02h - device IOCTL read**
  - Subfunction **03h - device IOCTL write**

DOS **INT 21h** functions are accessed differently depending on the programming language being used. **open**, **read**, **write**, and **close** functions allow the passing of transmit and receive data between the port and your application program. **IOCTL** subfunctions allow your application to do the following:

- Configure communication parameters for a port
- Determine error types

- Clear errors
- Check buffer counts for the ports

The following flowchart shows the normal flow for low level file I/O functions:



Flowchart 3-1. Low Level File I/O Overview

3.3. BASIC Language

The following information for the BASIC language was developed and tested using IBM's BASIC interpreter, release 3.0 under DOS 3.10. The **TERM.BAS** program (which contains these instructions) was tested under Microsoft's Quick BASIC versions 3.0 and 4.0.

The statement and function descriptions that follow are abbreviated examples. For detailed information, see your BASIC programming reference materials.

3.3.1. open for BASIC (DOS INT 21h, Function 3Dh)

Prior to performing any I/O function, the program must make a logical connection between itself and the device. This is done through the BASIC **open** statement, which has the following format:

open *filespec* for *mode* as # *filenum*

Where:

filespec is the 1 to 8 character device name.

mode is OUTPUT or INPUT (APPEND should not be used).

filenum is an integer between 1 and the maximum number of files allowed.

The following shows an example **open** statement:

open "host3" for output as #1

This statement allows the program to begin executing output operations to the serial port with the HOST3 device name. HOST3 should be defined in the **DEVICE NAME** parameter of a **DEVICE=PORT.BIN** statement in the **CONFIG.SYS** file.

Once a device has been opened, I/O operations can execute in the direction defined by the mode parameter of the **open** statement.

Both input and output can be performed on the same device by using two **open** statements, one for input and one for output. Each of these statements must use a different **filenum**.

3.3.2. read for BASIC (DOS INT 21h, Function 3Fh)

Data received from a serial port is retrieved (read from a port) from the device driver through the BASIC **INPUT$** statement, which accesses the DOS **read** function. The following shows the format of this statement:

v$ = input$ (*n*, # *filenum*)

Where:

n is the number of characters to be read from the device.

filenum is the number used when the device was opened for input.

When this statement executes, the destination string variable (**v$**, in this case) contains the number of characters requested from the device driver receive buffer. If there are not enough characters in the receive buffer to satisfy the request, the device driver waits the number of clock ticks specified by the **TIME-OUT** parameter (in the **CONFIG.SYS** file). This allows more characters to be received. When the request is satisfied or the timer expires, control returns to the application program. The number of characters returned can be determined by checking the length of the string variable.

BASIC (or DOS) returns an **input past end** error (#62) to the application when the receive buffer is empty. If this or any other error occurs and an **ON ERROR** statement is not active, the program halts. You must set up an **ON ERROR** sequence to enable error trapping and to interrogate error variables. The **ON ERROR** statement transfers control to a subroutine that reads the variable **ERR** and provides error handling.

When an input error occurs, take the following actions:

1. If the value of **ERR** is 62, the device driver receive buffer is empty. The appropriate action depends on the application.
2. If the value of **ERR** is 57 (device I/O error), read the error flags through the BASIC **IOCTL$** function (see Subsection 3.7.2).
3. If the **ERR** variable is zero, read the **ERDEV** variable. If the value of **ERDEV** is 800C hexadecimal (device driver *general failure*), a more serious device driver error has occurred. The error flags should be read through the **IOCTL**, as described in the previous paragraph.

3.3.3. write for BASIC (DOS INT 21h, Function 40h)

Program data is sent to the device driver (written to a port) through the BASIC **PRINT #** statement, which accesses the DOS **write** function. The following shows the format of this statement:

print # ***filenum, list of expressions***

Where:

filenum is the number used when the device was opened for output.

list of expressions is a list of numeric or string expressions to be output to the device.

When this statement executes, the list of expressions is output to the device driver transmit buffer. If there is not enough room in the transmit buffer for all of the characters, an error condition is returned. If an error occurs and an **ON ERROR** statement is not active, the program halts.

Note: *Make sure you set up an* ***ON ERROR*** *sequence to interrogate the error variables and to take the appropriate action.*

When an output error occurs, take the following action:

1. If the value of **ERR** is non-zero, an error has occurred that is not directly related to the device driver. This error can most likely be corrected by altering the BASIC program.
2. If the **ERR** variable is zero, read the **ERDEV** variable. If the value of **ERDEV** is 800A hexadecimal (device driver "write fault"), there is not enough room in the transmit buffer for all of the characters. Continue trying to output characters to the transmit buffer until space is available, or ignore the error and go on to other processing.
3. The only other value that is returned in **ERDEV** on an output request is 800C hexadecimal (device driver *general failure*). If this value is returned, a more serious device driver error has occurred and the error flags should be read through the BASIC **IOCTL$** function (see Subsection 3.7.3).

3.3.4. IOCTL for BASIC (DOS INT 21h, Function 44h, Subfunctions 02h and 03h)

The **device IOCTL read** and **write** functions read or write a string of control information from or to the driver. These strings and their functions are described in Subsections 3.7.2 and 3.7.3.

Note: *BASIC does not provide support for subfunctions **00h** and **01h***

- **Reading IOCTL Information**

  IOCTL data is retrieved from the device driver through the BASIC **IOCTL$** statement, which accesses the DOS **device IOCTL read** function (**IOCTL** Subfunction **02h**).

  The following shows the format of the BASIC **IOCTL$** statement:

  v$ = ioctl$ (# filenum)

  Where:

  filenum is the number used when the device was opened.

  When this function executes, the destination string variable (**v$** in this case) contains a character string consisting of 11 bytes of device driver error and status information.

- **Writing IOCTL Information**

  IOCTL data is sent to the device driver through the **IOCTL** statement, which accesses the DOS **device IOCTL write** function (**IOCTL** Subfunction **03h**).

  The following shows the format of the **IOCTL** statement:

  ioctl # *filenum, string*

  Where:

  filenum is the number used when the device was opened.

  string is a string expression containing the **device IOCTL write** information.

3.3.5. close for BASIC (DOS INT 21h, Function 3Eh)

When finished with a port, the program severs the logical connection to the device through the BASIC **close** statement:

close # ***filenum***

Where:

filenum is the number used when the device was opened.

3.3.6. Sample Programs for BASIC

The following files contain sample programs for BASIC:

- **TERM.BAS**

  This file contains the source code for a simple terminal program that provides examples of all of the I/O statements and functions previously described.

- **IOCTL$.BAS**

  This file contains the source code for a program that displays the error and status information returned by the device driver.

- **IOCTL.BAS**

  This file contains the source code for a program that clears the transmit and receive buffers through the **IOCTL** output function.

3.4. C Language

The following information for the C language was developed and tested using Microsoft C. More detailed information can be found in your C compiler documentation.

The following header files are generally included to allow access to the DOS low level file I/O functions:

- **<types.h>**
- **<stat.h>**
- **<io.h>**
- **<fcntl.h>**

3.4.1. open for C (DOS INT 21h, Function 3Dh)

The C **open** function opens the port called **name** in the mode specified by **mode**. **open** returns a file handle for the port or, in the case of an error, a value of -1 is returned.

The following shows how to open a port. The return value from **open()** is a file handle. Subsequent operations on the port refer to the file handle obtained here.

handle = open("*HOS1*",O_RDWR|O_BINARY);

Where:

HOS1 is the port's device name specified in **CONFIG.SYS**.

3.4.2. close for C (DOS INT 21h, Functions 3Eh)

The C **close** function closes the port associated with **handle**. The **close** function returns zero if the port was successfully closed or, in the case of an error, a value of -1 is returned.

The following shows how to close a file:

close(handle);

3.4.3. read for C (DOS INT 21h, Function 3Fh)

The C **read** function attempts to read **count** bytes into **buf** from the port associated with **handle**. The **read** function returns:

- The number of bytes actually read
- -1 if an error occurred
- 0 if an attempt to read at end-of-file occurred

The following shows an example of a **read** function:

```
char  *buf;          /* character buffer to store read data */
int   retcnt;        /* number of characters actually read */
int   count;         /* number of characters requested to be read */

retcnt = read(handle,buf,count);
```

3.4.4. write for C (DOS INT 21h, Function 40h)

The C **write** function writes **count** bytes from **buff** into the file associated with **handle**. The **write** function returns:

- The number of bytes actually written
- -1 if an error occurred

The following shows an example of a **write** function:

```
char  *buf;          /* buffer containing write data */
int   sentcnt;       /* number of characters actually sent */
int   count;         /* number of characters requested to be sent */

sentcnt=write(handle,buf,count);
```

3.4.5. IOCTL for C (DOS INT 21h, Function 44h, Subfunctions 00h, 01h, 02h and 03h)

The C language does not provide standard functions that directly relate to the DOS **IOCTL** functions. These functions must be accessed through DOS **INT 21h**. For Microsoft C, this can be done by using the **intdos()** function.

Your compiler may vary from the following abbreviated examples. See your DOS programming documentation for a complete description of DOS **INT 21h** calls. See your compiler documentation for details concerning DOS **INT 21h** access. (This installation guide assumes that the reader is familiar with DOS **INT 21h** or has reference material pertaining to it.)

The following definitions may be required for Microsoft C to allow access to DOS **INT 21h**.

```
#include <dos.h>
int intdos(inregs,outregs);/* Microsoft C library function for INT 21h */
union REGS inregs;/* 8086 register unions for use with intdos() */
union REGS outregs;
```

The following is a **get device information** example, which uses DOS **INT 21h**, function **44**, subfunction **00h**:

```
inregs.h.ah = 0x44;          /* IOCTL */
inregs.h.al = 0;             /* subfunction 0, get device information */
inregs.x.bx = handle;        /* file handle from open() */
inregs.x.dx = 0;
intdos(&inregs,&outregs); /* information returned in outregs.x.dx */
```

Example 3-1. get device information for C

The following is a **set device information** example, which uses DOS **INT 21h**, function **44h**, subfunction **01h**:

```
inregs.h.ah = 0x44;          /* IOCTL */
inregs.h.al = 1;             /* subfunction 1, set device information */
inregs.x.bx = handle;        /* file handle from open() */
inregs.x.dx=INFO_WORD;    /* device information word equate */
intdos(&inregs,&outregs); /* DOS INT 21h */
```

Example 3-2. set device information for C

After the **intdos()** call, the port's information word will be set to the value of **INFO_WORD** (see Subsection 3.7.1).

The following shows how to perform a **device IOCTL read**:

```
inregs.h.ah = 0x44;          /* IOCTL */
inregs.h.al = 2;             /* subfunction 2, device IOCTL read */
inregs.x.bx = handle;        /* file handle */
inregs.x.cx = 11;            /* driver always returns 11 bytes */
inregs.x.dx = buf;           /* where to put string */intdos(&inregs,&outregs);/* DOS INT 21h */
```

Example 3-3. device IOCTL read for C

After the **intdos()** call, **buf** contains 11 bytes of status and error information (see Subsection 3.7.2 for a description of this information).

The following shows how to perform a **device IOCTL write**:

```
inregs.h.ah = 0x44;          /* IOCTL */
inregs.h.al = 3;             /* subfunction 3, device IOCTL write */
inregs.x.bx = handle;        /* file handle */
inregs.x.cx = command_size;  /* number of bytes in buffer */
                             /* number depends on command */
inregs.x.dx = buff;          /* pointer to data buffer */
intdos(&inregs,&outregs);    /* DOS INT 21h */
```

Example 3-4. device IOCTL write for C

After the **intdos()** call, the command string contained in **buf** executes (see Subsection 3.7.3 for the available commands).

3.4.6. Sample Programs for C

The **TERM1.C** file contains the source code for a simple terminal program that provides examples of some I/O operations previously described. This program contains the example of the replacement of the DOS critical error handler.

The executable code for this program is in the **TERM1.EXE** file. Link the **TERMERR.OBJ** file to **TERM1.OBJ**. **TERMLINK.BAT** is a batch file, which will create **TERM1.EXE**.

3.5. Assembly Language

The following information explains the interfacing requirements for Assembly language. These instructions were developed and tested using the Microsoft Macro Assembler, version 4.0 under the MS-DOS operating system, level 3.10.

The Assembly language interface to the device driver makes use of standard DOS function calls (through the **21h** software interrupt). The function descriptions that follow are abbreviated. For detailed information, see your DOS programming reference materials.

3.5.1. open for Assembly (DOS INT 21h, Function 3Dh)

Entry parameters include:

- AH = 3Dh
- DS:DX => device name (terminated by a null)
- AL = access code
  - 0 = read
  - 1 = write
  - 2 = read/write

Exit parameters include:

- If carry is clear, the operation was successful; AX = file handle
- If carry is set, the operation failed; AX = error code

The following example shows how to open a port:

```
...
call    get_device          ; get device name string and return it in DS:DX
mov     ah,3Dh              ; function 3Dh = open the file
mov     al,READ_WRITE       ; READ_WRITE is equal to 02h
int     21h                 ; invoke DOS function
                            ; handle returned in ax
...
```

Example 3-5. open for Assembly

3.5.2. close for Assembly (DOS INT 21h, Function 3Eh)

Entry parameters include:

- AH = 3Eh
- BX = file handle

Exit parameters include:

- If carry is clear, the operation was successful
- If carry is set, the operation failed; AX = error code

The following example shows how to close a file:

```
...
mov   bx,handle          ; file handle
mov   ah,3eh             ; function 3Eh = close the file
int   21h                ; invoke function
  ...
```

Example 3-6. close for Assembly

3.5.3. read for Assembly (DOS INT 21h, Function 3Fh)

Entry parameters include:

- AH = 3Fh
- BX = file handle
- DS:DX=> buffer (data destination)
- CX = number of bytes to be read

Exit parameters include:

- If carry is clear, the operation was successful; AX = number of bytes actually transferred (read)
- If carry is set, the operation failed; AX = error code

The following example shows how to read from a device:

```
...
mov   bx,handle              ; file handle
mov   ah,3Fh                 ; function 3Fh = read from file or device
mov   cx,1                   ; number of bytes to read
mov   dx,offset READ_BUF     ; buffer address (DS:DX)
int   21h                    ; invoke function
  ...
```

Example 3-7. read for Assembly

3.5.4. write for Assembly (DOS INT 21h, Function 40h)

Entry parameters include:

- AH = 40h
- BX = file handle
- DS:DX=> buffer (data source)
- CX = number of bytes to write

Exit parameters include:

- If carry is clear, the operation was successful; AX = number of bytes actually transferred (written)
- If carry is set, the operation failed; AX = error code

The following example shows how to write to a device:

```
...
mov   bx,handle          ; file handle
mov   ah,40h             ; function 40h = write to a file or device
mov   cx,num_bytes       ; number of bytes to write
mov   dx,offset write_buf ; address of the data to write(DS:DX)
int   21h                ; invoke function
  ...
```

Example 3-8. write for Assembly

3.5.5. IOCTL for Assembly (DOS INT 21h, Function 44h, Subfunctions 00h, 01h, 02h, and 03h)

Entry parameters include:

- AH = 44h
- BX = file handle
- AL = subfunction value (described below)

The exit parameter is function dependent (described below).

The I/O control function supports 12 subfunctions (DOS version 3.1), four of which are important when interfacing with the device driver:

- **AL = 0 (get device information)**
  - Function specific entry parameters
    none
  - Exit parameters

    If carry is clear, the operation was successful; DX = status information
    If carry set, operation failed; AX = error code

  The following is a **get device information** example (see Subsection 3.7.1 for more information):

```
...
mov   ah,44h          ; function 44h = IOCTL
mov   al,0            ; sub-function 00h = get device information
mov   bx,handle       ; file handle
int   21h             ; invoke function
  ...                 ; DX has device information word
```

Example 3-9. get device information for Assembly

- **AL = 1 (set device information)**
  - Function specific entry parameters
    DX = status information (DH must be zero)
  - Exit parameters

    If carry is clear, the operation was successful;
    If carry is set, the operation failed; AX = error code

The following is a **set device information** example (see Subsection 3.7.1 for more information):

```
...
mov   ah,44h          ; function 44h = IOCTL
mov   al,1            ; sub-function 01h = set device information
mov   bx,handle       ; file handle
mov   dx,INFO_WORD    ; equate for device information word
int   21h             ;
 ...
```

Example 3-10. set device information for Assembly

- **AL = 2 (device IOCTL read)**
  - Function specific entry parameters
    DS:DX=> destination buffer
    CX = number of bytes to read
  - Exit parameters

    If carry is clear, the operation was successful; AX = number of bytes actually transferred (read)

    If carry is set, the operation failed; AX = error code

The following shows how to perform a **device IOCTL read** (see Subsection 3.7.2 for more information):

```
...
mov   ah,44h              ; function 44h = IOCTL
mov   al,2                ; sub-function 02h = read from device
mov   bx,handle           ; file handle
mov   cx,11               ; CX = number of bytes to read
mov   dx,offset io_buf    ; buffer address (where to put string)
int   21h                 ; invoke function
  ...                     ; io_buf contains 11 bytes
```

Example 3-11. device IOCTL read for Assembly

- **AL = 3 (device IOCTL write)**
  - Function specific entry parameters
    DS:DX=> source buffer

    CX = number of bytes to write (size depends on the command)
  - Exit parameters
    If carry is clear, the operation was successful; AX = number of bytes actually transferred (written)

    If carry is set, the operation failed; AX = error code

The following shows how to perform a **device IOCTL write** (see Subsection 3.7.3):

```
...
mov   ah,44h              ; function 44h = IOCTL
mov   al,3                ; sub-function 03h = write to device
mov   bx,handle           ; file handle
mov   cx,command_size     ; CX = number of bytes to write
mov   dx,offset io_buf    ; buffer address (command string address)
int   21h                 ;
  ...
```

Example 3-12. device IOCTL write for Assembly

3.5.6. Sample Programs for Assembly

The **TERM.ASM** file contains the source code for a simple terminal program that provides examples of all of the I/O operations previously described. This program contains the example of the replacement of the DOS critical error handler. The executable code for this program is in the **TERM.EXE** file.

The **TERM.ASM** file contains **IOCTL** function calls that change the current parameter settings to the following:

- Baud rate - 19,200
- Stop bits - 1.5
- Transmit time-out - 1
- Receive time-out - 0
- DTR option - ON

3.6. Error Conditions

This subsection contains error processing information. It explains how the DOS device driver handles an error and what message numbers (for the error) are passed to the DOS critical error handler in the **REQUEST HEADER**.

If an unknown function (DOS sub-function for **INT 21h**) or unknown command (in **IOCTL** control output data) is chosen by the application, the **STATUS** word in the **REQUEST HEADER** (passed to DOS from the driver) is set to:

done + error + dderr3

This is interpreted as an unknown command to DOS.

Two classes of errors are returned by the device driver. These include the **write fault** error and the **general failure** error. When the device driver receives an output request, it adds characters to the transmit buffer. If the buffer is full, and the transmit time-out period (if set) expires, the device driver sets the **STATUS** word in the **REQUEST HEADER** to:

done + error + dderra

This is interpreted as a **write fault** error to DOS.

Prior to performing an input or output operation, the device driver checks the error flags. If any of the error flags are set, the device driver returns **general failure** to the application. This indicates to the application that a character error (overrun, parity, or framing error), a break interrupt, or a receive buffer overrun condition occurred.

The **STATUS** word in the **REQUEST HEADER** is set to:

done + error + dderrc

If the **IOCTL** function is not enabled (**IOCTL** option is not chosen in the **CONFIG.SYS** file), the application can not determine the type of error. The error flags are cleared by the device driver after one **general failure** indication is returned to the application. In this case, an **Abort**, **Retry**, or **Ignore** message appears on the screen and processing halts depending on the user's response.

For C or Assembly language, the DOS critical error handler needs to be replaced by the user's own error handler. BASIC does have its own error handler.

If the **IOCTL** function is enabled (**IOCTL** option ON), the application queries the error channel (through the **device IOCTL read** function) and determines the type of error that occurred. The device driver continues to return a **general failure** error until the error flags are cleared by a **device IOCTL read**.

Note: *The general failure message also displays on the return from the **IOCTL** control output data function. This occurs if the application requires changes in parameters, and the **IOCTL** option is not **ON**.*

DOS will generate a 24H interrupt based on the contents of the **STATUS** word in the **REQUEST HEADER**.

3.7. IOCTL Subfunctions

The **IOCTL** is used to retrieve or pass a user defined control string from the device driver. The following **IOCTL** subfunctions (**00h** through **03h**) are discussed in this subsection. The information is not language specific.

- **00h - get device information**
- **01h - set device information**
- **02h - device IOCTL read**
- **03h - device IOCTL write**

3.7.1. IOCTL get/set device information (Subfunctions 00h and 01h)

The format of the **get/set device information** is shown in Table 3-1 (see the programming reference materials for the MS-DOS operating system for detailed information).

Note: *Subfunctions **00h** and **01h** are not available with BASIC.*

Table 3-1. Device Information Word

Bit	Status Description	Notes
15	(reserved)	
14	**IOCTL**	Set to 1
13	(reserved)	
12	NETWORK	Always 0 for this driver
11	(reserved)	
10	(reserved)	
9	(reserved)	
8	(reserved)	
7	ISDEV	Always 1 for this driver
6	EOF	Always set to 1*****

***** *May be set to 0 by DOS, using an empty buffer **read**. If this occurs, you must set the EOF bit to a 1 to allow subsequent reads.*

Table 3-1. Device Information Word(Continued)

Bit	Status Description	Notes
5	BINARY	Set to 1 for binary mode Set to 0 for ASCII mode
4	(reserved)	
3	ISCLK	Always 0 for this driver
2	ISNUL	Always 0 for this driver
1	ISCOT	Always 0 for this driver
0	ISCIN	Set to 0 for a driver

***** *May be set to 0 by DOS, using an empty buffer* ***read****. If this occurs, you must set the EOF bit to a 1 to allow subsequent reads.*

Normally, only Bits 5, 6, or 14 need to be set for the device driver. To preserve the remaining bits, the information word should be read and the appropriate bits should be set or cleared and then written back out.

3.7.2. device IOCTL read (Subfunction 02h)

This subsection describes the 11 bytes associated with the **device IOCTL read** function. The **IOCTL** is used to pass error and status information from the driver to the application program. The following is a description of the 11 bytes associated with **device IOCTL read**:

- **Error Flags (bytes 1 and 2) -** See Table 3-2 for this information.
- **Overrun Error Count (byte 3)**

  This byte indicates the number of receive character overrun error conditions that have occurred since the last **IOCTL** function was used. This count is incremented each time an overrun error occurs, until a maximum value of 0FFh (255 decimal) is reached. The count is reset to zero when read by an **IOCTL** function.
- **Parity Error Count (byte 4)**

  This byte indicates the number of receive character parity error conditions that have occurred since the last **IOCTL** function was used. This count is incremented each time a parity error occurs, until a maximum value of 0FFh (255 decimal) is reached. The count is reset to zero when read by an **IOCTL** function.
- **Framing Error Count (byte 5)**

  This byte indicates the number of receive character framing error conditions that have occurred since the last **IOCTL** function was used. This count is incremented each time a framing error occurs, until a maximum value of 0FFh (255 decimal) is reached. The count is reset to zero when read by an **IOCTL** function.
- **Break Interrupt Count (byte 6)**

  This byte indicates the number of break interrupts, which occurred since the last **IOCTL** function was used. This count increments each time a break interrupt occurs, until a maximum value of 0FFh (255 decimal) is reached. The count resets to zero when read by an **IOCTL** function.

- **Receive Buffer Overrun Count (byte 7)**
  This byte indicates the number of receive buffer overflow error conditions (characters lost) since the last **IOCTL** function was used. This count is incremented each time a character is received, but there is no room in the receive buffer to store it. The count is not incremented beyond 0FFh (255 decimal) and is reset to zero when read by an **IOCTL** function.
- **Transmit Buffer Count (bytes 8 and 9)**
  This word (two bytes) indicates the number of characters in the transmit buffer.
- **Receive Buffer Count (bytes 10 and 11)**
  This word (two bytes) indicates the number of characters in the receive buffer.

Table 3-2 lists the various error flags associated with bytes 1 and 2 of the **device IOCTL read** function.

Table 3-2. device IOCTL read Error Flags (Bytes 1 and 2)

Bit	Hex	Error Description
0	0001h	undefined
1	0002h	overrun error occurred
2	0004h	parity error occurred
3	0008h	framing error occurred
4	0010h	break interrupt occurred
5	0020h	undefined
6	0040h	undefined

Table 3-2. device IOCTL read Error Flags (Bytes 1 and 2)(Continued)

Bit	Hex	Error Description
7	0080h	undefined
8	0100h	receive buffer overrun occurred
9	0200h	undefined
10	0400h	undefined
11	0800h	undefined
12	1000h	undefined
13	2000h	undefined
14	4000h	undefined
15	8000h	undefined

Note: *The error flags are cleared when an* ***IOCTL*** *function executes.*

3.7.3. device IOCTL write (Subfunction 03h)

This subsection describes the command string associated with the **device IOCTL write** function. Use the **IOCTL** to pass a user defined command string to the device driver. This data is not meant for the device to transmit, but for controlling the device.

Note: *To execute the **IOCTL** from an application, the user must set the **IOCTL** option ON for the port in the **CONFIG.SYS** file.*

The first byte of the command string is always set to the command code. Subsequent bytes, if any, are defined by the particular command.

The format of the command string is summarized as follows:

Byte 1 = **COMMAND CODE**:

1 (Flush Transmit Buffer)

2 (Flush Receive Buffer)

3 (Change Required Parameter)

4 (Change Optional Parameter)

Byte 2 = **PARAMETER CODE**

Byte 3 = **PARAMETER VALUE**

Byte 4 = **PARAMETER VALUE**

Note: *Bytes 2, 3, and 4 may not be used for all commands, since the number of bytes depends on the command being used.*

The following is a description of the string passed to the device driver for each of the four **COMMAND CODES**.

3.7.3.1. Flush Transmit Buffer String

COMMAND CODE (byte 1) = 1

Only **COMMAND CODE** is required for this command string. When **COMMAND CODE** is equal to 1, the transmit buffer count is reset to zero, discarding all characters currently in the buffer.

Use the following example command string to flush the transmit buffer of a port:

COMMAND CODE (byte 1) = 1 (Flush Transmit Buffer)
PARAMETER CODE (byte 2) = unused
PARAMETER VALUE (byte 3) = unused
PARAMETER VALUE (byte 4) = unused

Example 3-13. Flush Transmit Buffer

3.7.3.2. Flush Receive Buffer String

COMMAND CODE (byte 1) = 2

Only **COMMAND CODE** is required for this command string. When **COMMAND CODE** is equal to 2, the receive buffer count is reset to zero, discarding all characters currently in the buffer.

Use the following example command string to flush the receive buffer of a port:

COMMAND CODE (byte 1) = 2 (Flush Receive Buffer)
PARAMETER CODE (byte 2) = unused
PARAMETER VALUE (byte 3) = unused
PARAMETER VALUE (byte 4) = unused

Example 3-14. Flush Receive Buffer

3.7.3.3. Change Required Parameter String

COMMAND CODE (byte 1) = 3

PARAMETER CODE (byte 2) =

1 - change baud rate

2 - change parity

3 - change character size

4 - change number of stop bits

5 - change transmit time-out value

6 - change receive time-out value

PARAMETER VALUE (byte 3)= first byte of the parameter value in hex

PARAMETER VALUE (byte 4)= second byte of the parameter value in hex (if required)

The following is a list of **PARAMETER CODES** and their required **PARAMETER VALUES**:

1 (baud rate) - bytes 3 and 4 of the command string specify one of the following 16 bit values: 50, 75, 110, 134 (for 134.5), 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200, 38400, and 56000.

Note: *These values should be converted to a 16 bit hex value and placed in low byte, high byte form (byte 3 is low byte, byte 4 is high byte).*

2 (parity) - byte 3 of the command string specifies one of the following ASCII values:

- n no parity (ASCII 6Eh)
- o odd parity (ASCII 6Fh)
- e even parity (ASCII 65h)

3 (character size) - byte 3 of the command string specifies one of the following decimal values:

- 5 five bit character
- 6 six bit character
- 7 seven bit character
- 8 eight bit character

4 (number of stop bits) - byte 3 of the command string specifies one of the following decimal values:

- 1 one stop bit
- 2 two stop bits

5 (transmit time-out value) - bytes 3 and 4 of the command string specify a 16 bit value ranging from 0 to 65535 decimal (0 to FFFF hex)

6 (receive time-out value) - bytes 3 and 4 of the command string specify a 16 bit value ranging from 0 to 65535 decimal (0 to FFFF hex)

Use the following example command string to change a port's baud rate to 19200:

COMMAND CODE (byte 1) = 3 (Change Required Parameter)
PARAMETER CODE (byte 2) = 1 (change baud rate)
PARAMETER VALUE (byte 3) = 00h (low byte of 4B00h*)
PARAMETER VALUE (byte 4) = 4Bh (high byte of 4B00h*)

Example 3-15. Change Required Parameter

Note: **4B00h = 19200 decimal.*

3.7.3.4. Change Optional Parameter String

COMMAND CODE (byte 1) = 4

PARAMETER CODE (byte 2) = optional parameter bitmap

The optional parameter bitmap is one byte that indicates the optional parameters for a port that are active.

0 - on the proper position means optional parameter OFF

1 - on the proper position means optional parameter ON

bit position 0 **ECHO** option
bit position 1 **HIFC** option
bit position 2 **HOFC** option
bit position 3 **SIFC** option
bit position 4 **SOFC** option
bit position 5 **DTR** option
bit position 6 **RTS** option
bit position 7 **IOCTL** option

Note: *Bit 7 must always be ON for subsequent **IOCTL** calls.*

Use the following example command string to set the DTR on a port:

COMMAND CODE (byte 1) = 4 (Change Optional Parameter)
PARAMETER CODE (byte 2) = A0h* (bit mask)
PARAMETER VALUE (byte 3) = unused
PARAMETER VALUE (byte 4) = unused

Example 3-16. Change Optional Parameter

Note: **A0h is the bit mask needed to turn DTR (bit position 5) and **IOCTL** (bit position 7) on.*

If you need specific information about how to use an **IOCTL** subfunction with BASIC, C, or Assembly languages, refer to Table 3-1. For detailed information, see your programming reference materials.

If you experience technical problems, refer to Section 4. Troubleshooting before calling Comtrol Technical Support.

Section 4. Troubleshooting

4.1. Resolving Installation Problems

If installation fails or you are trying to resolve a problem, you should try the following before calling the Comtrol technical support line:

- Check the signals between your peripherals and the interface box to verify that they match.
- Check to make sure the cables are connected properly.
- Reseat the controller in the slot.
- Make sure that the expansion slot screw was replaced after inserting the controller.
- Verify that the switch settings are correct.

If you have not been able to get the controller operating:

1. Turn off your PC and insert the diagnostic diskette.
2. Boot the PC and follow the instructions provided by the diagnostic diskette.

Table 4-1 defines the 64-byte I/O address blocks from 0 through 3FFh and their known uses.

Table 4-1. System I/O Addresses – Up to 3FF

Address Block	Addresses Used	Description
000 – 03F		Reserved for Motherboard
040 – 07F		Reserved for Motherboard
080 – 0BF		Reserved for Motherboard
0C0 – 0FF		Reserved for Motherboard
100 – 13F		
140 – 17F		
180 – 1BF		
1C0 – 1FF	1F0 – 1F8	Fixed Disk
200 – 23F	218 – 21B	
240 – 27F	278 – 27F	LPT2, IDE controllers, and multifunction boards (game ports)
280 – 2BF		
2C0 – 2FF	2E8 – 2EF 2F8 – 2FF	COM4 COM2
300 – 33F	318 – 31B	

Table 4-1. System I/O Addresses – Up to 3FF(Continued)

Address Block	Addresses Used	Description
340 – 37F	378 – 37F	LPT1
380 – 3BF	3B0 – 3BF	Monochrome Display and LPT3
3C0 – 3FF	3D0 – 3DF 3E8 – 3EF 3F0 – 3F7 3F8 – 3FF	Graphics Monitor Adapter COM3 Floppy Disk Controller COM1

4.2. Placing a Support Call

Before you place a technical support call to Comtrol, please make sure that you have the following information.

Table 4-2. Support Call Information

Item	Your System Information
Controller type	16-port
Interface type	RS-232, RS-232/422, RS-422/485, RS-232/Current Loop
Base I/O address selection	
Interrupt (IRQ) number selection	
Operating system type and release	
Device driver release number	
PC make, model, and speed	
List of other devices in the PC and their addresses	

After you have gathered this information, contact Comtrol by email, FAX, or phone:

Comtrol Corporate Headquarters:

- Internet URL: www.comtrol.com
- email: support@comtrol.com
- FTP site: ftp.comtrol.com
- Phone: (612) 494-4100
- FAX: (612) 494-4199

Comtrol Europe:

- Internet URL: www.comtrol.co.uk
- email: support@comtrol.co.uk
- Phone: +44 (0) 1 869-323-220
- FAX: +44 (0) 1 869-323-211

Copyrights and Trademarks