FILE TRANSFER BASICS:

This is an edited and condensed excerpt from The Modem Reference, written by Michael A. Banks and recommended by the Associated Press, The Smithsonian Magazine, Jerry Pournelle in Byte, et al.

(From Chapter 9)
File Transfers

When you first connect with an online system, you usually communicate with that system one line at a time. This kind of communication is fine when you're participating in an interactive online activity, such as browsing a system, using realtime conference, reading E-mail, etc. But it's impractical when you need to send or receive large blocks of text or non-text data, programs, or other files for use offline or at a later time. For these types of exchanges, you need to make use of file transfer.

The idea of sending a file over telephone lines may be a bit intimidating at first. What, you may ask yourself, if something goes wrong? What do I need to know? Does file transfer involve complex procedures? In partial answer to those questions, consider this: modem file transfers are today almost as common as long-distance voice telephone calls. Tens of thousands of executable programs, database and spreadsheet files, and documents of all types flow over telephone lines 24 hours a day, seven days a week. The vast majority of these transfers go through without a hitch, and are conducted by people who possess less technical knowledge than you have gained by reading the earlier chapters in this book.

This is not to say that file transfer is strictly a "plug-and-go" proposition. Knowledge of certain terms and concepts is necessary. And, as with the other elements of telecomputing discussed in this book, it helps to know something of what goes on behind the scenes.

Which brings us to the purpose of this chapter, which is to show you what file transfer is all about. There are certain methods common to all kinds of file transfers, and I'll explain those in detail here. I'll also show you how transfers are conducted, and how you can successfully send and receive files using your microcomputer. All of the basic information you need is right here, along with technical details and hands-on examples.

Read as much--or as little--of the technical explanations as you find necessary to achieve a working understanding of file-transfer techniques. But don't skip over any topic entirely; you should be familiar with each of the various file-transfer methods. Take the time to look over the examples, too. These will prepare you for transferring files on your own.

Simply defined, file transfer is the process of sending a file--text, binary data, or program--from one computer to another. The computers involved may range from mainframe and mini-computers operated by businesses, governments, or online services, to home and business microcomputers.

Regardless of the type of file or the size of the computers

involved, the basic steps involved are fairly simple: The sending computer's hardware and software read the contents of a file from a mass storage device (usually a hard or floppy disk). As it reads the file, the computer sends what it reads to its modem, which converts the data to analog form and transmits it to the receiving system via telephone line.

The receiving modem converts the data back to digital form and sends it to its computer. The receiving computer reads the data and stores it in a file. (Note that what is actually sent is a copy of a file--the file on the receiving end remains intact.) ((NOTE: The hardcopy of THE MODEM REFERENCE contains several figures, showing various "hands-on" examples of file transfer procedures and processes.))

Depending upon the transfer method in use, the sending and receiving systems may perform error checks and exchange information about the transfer along the way.

From the hardware viewpoint, this transmission takes place in pretty much the same way as the transmission of keyboard input, although serial ports and modems may handle flow control and error checking differently.

On the software end, both the sending and the receiving system have to be given commands to initiate transfer. Online systems often use special software file-transfer protocols to handle data flow and error checking during transfer.

Uploads and Downloads:

In case you missed it earlier, a file transfer can take place in either of two directions--to your computer, or from it. The terms used to describe file transfers refer to the direction in which data is transmitted, and are relative. If you are sending data to another computer, you are uploading; if you are receiving data, you are downloading.

File Transfer Methods:

ASCII Transfer and Error-Checking Protocols. There are two basic categories of file transfer methods: ASCII transfer, and what are usually called error-checking protocol transfers. ASCII transfer is the transfer of straight text files. Error-checking protocols transfer files in groups of bytes, and use sophisticated error-checking routines to verify the integrity of each group sent.

Batch File Transfers. Only one file can be transferred at a time via an asynchronous dial-up system of the type focused on by this book; if you want to transfer more than one file, you usually have to issue new commands for each file. However, some file transfer protocols, like Kermit (q.v.), let you specify a group (or batch) of files to be transmitted, after which the program takes over and handles the commands and transfer operations for each file. This kind of operation is known as batch processing, or batch file transfer. It's especially convenient when you have to upload or download a large number of files, but cannot be at your computer to direct and supervise the transfers.

File Transfer Channels:

There are two possible channels for file transfer. The first is direct (or sender-to-receiver) file transfer, in which the individuals who wish to transfer a file or files link their computers directly. The second channel is what I call "third party" transfer, in which an online system serves as an intermediary and storage place for files. This third party is usually a BBS or an online service. We'll take a closer look at

using these channels in a later segment of this chapter.

ASCII FILE TRANSFER:

For most computer users, the simplest type of file transfer is an ASCII transfer. Limited in a practical sense to files that contain only 7-bit ASCII characters, it is commonly used to transfer small- and medium-sized "straight" text files. ("7-bit ASCII characters" here refers to the letters, numbers, punctuation symbols, and spaces found on most keyboards.)

ASCII transfer can sometimes be used with transfer binary files, if they are output as text in hexadecimal form, or, as is the case with some BASIC programs, in straight ASCII text. This is, however, accommodated by few systems, and very tricky to get right. (Binary files of many types can be converted to 7-bit ASCII for transmission, too. They must be reconverted--offline--before use at the receiving end, however.)

8-bit ASCII transfer is possible, too--provided both systems are set up for 8-bit communication. But the applications for 8-bit transfer are limited. Many communications programs offer the option of "stripping" (removing) the eighth bit from bytes in 8-bit files, thus enabling you to transfer 8-bit text files--like those produced by WordStar--in 7-bit (and human-readable) form.

How ASCII Transfer Works:

When a file is transmitted via ASCII protocol, it is read from a computer's disk or memory, and sent character-by-character, just as it would be sent if you were typing it at your keyboard. The rate of transmission is much faster than you can type, though (unless you are a very fast typist and comparing yourself to 300 bps transmission!). When the file is received, it is handled just like any other text input--read into the computer's buffer, then written to disk or stored in RAM.

Data Buffering and Flow Control. Almost all computer systems use data buffering, which means that incoming or outgoing data is temporarily stored in RAM in what is called a buffer. DAta buffering speeds up file transfer because it eliminates the number of times a transmission must be paused while the receiving or sending computer accesses its disk.

At the sending end of a file transfer, enough data is read from disk to fill a RAM buffer, then sent from RAM. When the send buffer is empty, the disk is read again. Because the buffer holds enough data for several seconds of transmission, there is less disk access (i.e., the sending computer doesn't have to stop to read data from its disk each time it sends a line).

The reverse occurs at the receiving end, with data being written to disk only when the receive buffer is filled. The flow of incoming data is paused when this occurs, and is usually controlled by XON/XOFF (^Q/^S) protocol. Additional flow control may consist of turnaround characters or time delays, or hardware (modem or serial port) control. (Flow control, turnaround characters, and time delays are discussed in detail in Chapter 5.)

The overall process is pretty much the same as exchanging commands and textual data with a remote system in realtime. (As you've seen, the same systems of buffering and flow control are used.) Aside from the speed, the main difference between realtime data entry and text-file transfer is that, when a file is transferred, text is generated and stored at each end by the respective computers, without human involvement.

Advantages of ASCII File Transfer:

The main advantage of using ASCII file transfer is that it's easy. All you have to do is tell the remote system that you want to send or receive a file, switch to your communications software's command mode, and type something like SEND <file name> to transmit a file, or RECEIVE or CAPTURE <filename> to receive a file.

ASCII transfer is simple to implement in a program, too. All that's required are a capture buffer to temporarily store incoming data, a series of commands to read and write to a disk file, and a simple flow-control technique. With the rare exception of disk read- and write-commands, virtually any communications program or online system has everything it needs to support ASCII file transfer built in.

Because it is so easy to use and implement, almost all online systems provide ASCII file transfer (something that is not true with all error-checking protocols). And, even if an online system doesn't provide an ASCII download command, turning on your communications program's capture-to-disk while a text file is displayed by the online system is the same as performing an ASCII file download.

Disadvantages of ASCII File Transfer:

Where large files or "noisy" telephone connections are involved, data may be lost or garbled during an ASCII transfer. This is because parity checking (the only type of error-checking protocol used with ASCII transfers) is sometimes not used (as is the case when communications parameters are 8N1). Even when it is used, parity-checking rarely provides for re-transmission of garbled data (see Chapter 3). (Error-checking protocols do re-transmit garbled data.) The chance of losing or fouling up data increases with communications speed, too.

Another major drawback of using ASCII file transfer in the modern world of telecomputing is its inability to handle binary data files, programs, and most 8-bit files, for reasons discussed in the paragraphs immediately following.

ERROR-CHECKING (BINARY) FILE TRANSFER PROTOCOLS: WHAT THEY ARE, AND HOW AND WHY THEY'RE USED

In the early days of telecomputing, file transfers consisted of simple 7-bit ASCII text files--messages, program source code, reports, and the like. As telecomputing evolved, there emerged a need to transfer other kinds of files. A reliable method of error checking was also needed. Error-checking file transfer protocols (sometimes called "binary protocols," or simply "protocols") were developed to meet these needs.

As noted earlier, error-checking protocols operate by sending data in discrete groups, whose integrity is tested at the receiving end to insure error-free transmission. Error-checking protocols can be used to transfer any type of file, from binary data files and machine-language programs to 7-bit text files.

Transferring Binary Data and Programs with Error-Checking Protocols:

Programs, "binary" data files, and certain other types of files cannot be transferred via conventional ASCII transfer methods because such files may contain any or all of the 128 "standard" characters from the 7-bit ASCII group (including control-characters), as well as special 8-bit characters. 7-bit ASCII transfer methods can deal only with 7-bit alphanumeric characters; control-characters may be ignored or perceived by software or hardware as commands, and 8-bit characters are truncated.

When you attempt to transfer anything other than 7-bit

alphanumeric characters via modem, all sorts of problems can pop up. Here are a few examples:

1. The receiving system may interpret some characters (like as ^S) as a flow-control or other command character and "lock up."
2. The receiving system may receive an "end of file" marker (typically a ^Z), and stop accepting data, resulting in a partial transfer.
3. 8-bit characters will be truncated (i.e., the final bit in each byte ignored), which will result in the wrong characters being "received."
4. The receiving computer or either of the modems may, depending on their internal makeup and configuration, interpret 7-bit control-characters or 8-bit characters in other, unpredictable ways, resulting in lost data, system lockup, disconnection, etc.

All of these problems are avoided with error-checking protocol transfer, because control-characters are transmitted in such a way that they are not perceived as such by the receiving system.

Too, protocol transfers usually take place using 8 data bits, so 8-bit characters (when sent individually) are handled without truncation. If the transfer takes place at 7 bits, as is the case when the Kermit protocol is used, 8-bit characters are translated into "passable" 7-bit characters for transmission.

On top of all this, error-checking protocols provide the bonus of reliable error checking. When you're transferring a large file of any type, error-checking is a very welcome bonus, because the odds are high that some sort of telephone-line noise or other garbage will be introduced into the file. When binary data or program files are involved, error-checking becomes a necessity, because it's almost impossible to find garbled data in such files after download.

Transferring ASCII Text Files with Error-Checking Protocols:

Error-checking protocols can be used to transfer 7-bit text files as easily as other types of files. In fact, error-checking protocol transfer is to be preferred when you transfer large text files, for two reasons. First, as I've pointed out before, the simple parity checks performed during an ASCII file transfer are all but worthless. By comparison, error-checking protocols can transfer files with a reliability of more than 99%. Second, depending on the method used, error-checking protocol transfer is often faster than ASCII transfer.

Add to these advantages the byte-by-byte reports and other extras provided by some communications programs during file transfer, and it's easy to see that error-checking protocols are by far the better way to transfer files. I've found error-checking protocols to be so useful that I routinely use the Xmodem or Kermit error-checking protocol for even the smallest text-file transfers. (I reserve ASCII transfer for use with systems that don't accommodate error-checking protocols, and for inserting small disk files into the middle of messages I'm composing online.)

HOW ERROR-CHECKING PROTOCOL TRANSFER WORKS:

When a file is transferred using an error-checking protocol, data is transmitted in groups of characters called blocks (sometimes referred to as "packets" or "frames"), rather than one byte or one line at a time. The blocks are usually of a fixed size, such as 128 bytes or 1024 bytes.

A simplified description of the process goes like this: The sending computer transmits no data until it has read enough from

a file to make up a block, which it then transmits. When the block is received at the other end, the receiving computer unpacks the block and adds it to the file it is writing.

During a protocol transfer, the computer systems involved usually exchange information about the transmission, relying on a mutually-recognized set of control signals to signal data receipt or error, mark the end of a transmission, and perform other chores related to file transfer and error-checking. These control signals sometimes vary from one protocol to another, and consist of device and communications control characters from the ASCII character set designated by the American National Standards Institute (ANSI). Table 9.1 Provides a list of these codes and their applications in device and communication control.

Error-Checking:

Accurate error-checking is accomplished by any of several methods during protocol transfer. Information about the number and type of bytes or bits in each block may be included with the block in what is called a block "header," or transmitted before or after the block. Additional bits may be added to each block for the receiving system to use in determining (by calculations based on the sum of the binary ones in a block) whether a block has been properly received. Or, the sending and receiving systems may exchange information about the content of blocks after a block or group of blocks is transferred.

Suffice to say, the integrity of each block is accurately checked. If the block is "good," the receiving system sends an acknowledgment signal (usually an ACK) to tell the sending system to transmit the next block. If a block is "bad" (i.e., the information about the block doesn't agree with the block contents as received), the receiving system asks the sending system to re-send the block by sending a negative acknowledgment signal (usually a NAK).

Retries:

As indicated above, "bad" data blocks are automatically re-transmitted by protocol transfer systems. However, a transfer is terminated if a block is re-transmitted a set number of times (usually 9) without success. This provides extra insurance against bad data getting through, and also eliminates the possibility of computers being tied up for hours while a bad block is transmitted over and over again. The number of retries can usually be set within a communications program, and some online systems will allow you to do the same in an online profile or configuration area.

Timeouts:

Protocol transfer systems will wait for a specified period of time to receive a block of data or, when transmitting, for an acknowledgment that data was properly received. If this time limit is exceeded, the protocol transfer is terminated. This is called a "timeout." As with the number of retries, protocol transfer timeout can usually be set within a communications program and on some online systems.

Buffers:

Most error-checking protocols take advantage of data buffering, which means that (as is the case with ASCII file transfers) data is temporarily stored in RAM in what is called a buffer. Data buffering speeds up file transfer because it eliminates the number of times a transmission must be paused while the receiving or sending computer accesses its disk.

Data buffers usually hold several blocks' worth of data,

which means the sending computer doesn't have to stop to read data from its disk each time it sends a block. For the same reason, the receiving system doesn't have to pause to write each time it receives a block. Data is written to disk only when the receive buffer is filled.

Bit "Overhead" and Speed:

You may have wondered whether the addition of extra bits for error-checking purposes adds enough "overhead" to a file to slow down a transfer. It might, if not compensated for. Most error-checking protocols, however, remove the start-, stop-, and parity-bits from each byte before including the byte in a packet. The net result is that the total number of bits transferred is less than it would be if the bytes were sent via ASCII transfer.

Some protocols compress data before transmitting it, too. (The data is, of course, decompressed before being stored at the receiving end.)

Reports:

As you know, error-checking protocols exchange information on the status of a transfer, but these are not intended to be viewed by the computer user. However, most implementations of error-checking protocol in communications programs provide some sort of ongoing report on the status of protocol file transfers. Information such as the number of blocks and/or bytes transferred, the percentage of the transmission completed, and more is available at any point during the transmission.

Status reports from two different programs--COM-AND and MIRROR II--are shown in Figures 9.2 and 9.3.

Online systems often report on the final status of error-checking protocol transfers with a quick summary that scrolls onto the screen at the conclusion of each transfer. As shown below in figures 9.4 and 9.5, these reports vary in detail from one system to another.

This is a general description of the way error-checking protocols work; there are some more esoteric approaches, but this is the most common. The descriptions of specific error-checking protocols that follow contain additional details on particular protocols.

COMMON ERROR-CHECKING PROTOCOLS AND HOW THEY WORK:

Discussing every public domain, commercial/proprietary, and machine-specific error-checking protocol in existence is beyond the scope of this book. However, we'll take a look at the most popular protocols in public use here, along with machine- and system-specific protocols, and proprietary protocols.

Xmodem and Variations:

Xmodem file transfer protocol (sometimes called MODEM7 or Xmodem/Checksum) is an error-checking protocol you'll encounter very frequently. Created in 1978 by Ward Christensen and placed immediately into the public domain, Xmodem has become a de facto standard. It can be found on almost all BBSs and online systems that offer error-checking protocols. And, if a communications program offers any error-checking protocols at all--even one--Xmodem will be among them (systems and programs that use strictly proprietary protocols excepted, of course).

How Xmodem Works. Xmodem transfers files in 128-byte blocks. It adds an extra bit--called a checksum--to each block, which the receiving system uses to calculate whether or not the block was accurately transmitted. (A complex algorithm, based on

the contents of the block, is used for this calculation.) If the checksums don't agree, the receiving computer requests the sending computer to re-transmit the packet by sending a NAK. Otherwise, the receiver sends an ACK, and the sending system transmits the next block. This process is repeated for each block until the entire file is transferred, or until the transfer is aborted by the user, by too many retries, or by a timeout.

Although it is a superb error-checking protocol, Xmodem has a couple of drawbacks. It cannot be used to communicate at 7 data bits, as it transmits files in 8-bit format only. (If you try to use Xmodem with a 7-bit system, not all the bits transmitted will be received, or the system may lock up.) If a hardware error-checking protocol or flow control is in effect, some Xmodem characters can be perceived as control characters, with unpredictable effects.

You may encounter some versions of Xmodem (or MODEM7) that offer a batch file processing. Such implementations are generally restrictive and difficult to use, however.

Overall, Xmodem is a superior approach to error-checking protocol design. It is relatively easy to use, and offers 96% reliability--extremely high where file transfers are concerned, and far better reliability than can be achieved with ASCII transfers.

Xmodem CRC. CRC (Cyclic Redundancy Check) is an Xmodem option that modifies how Xmodem checks for errors. CRC adds a second checksum bit to each block to enhance error checking. The fact that an extra bit must be transmitted with each block makes for a noticeable difference in transmission times only for extremely large files, and the gain in efficiency is worth it. Using the CRC option with Xmodem increases reliability to an amazing 99.6%!

Xmodem programs on online systems that use CRC usually have the capability to detect whether or not another system is using CRC, and then use CRC or not, as appropriate. GEnie is one example of a system that can detect and adjust to the presence or absence of the CRC option.

(Don't count on every system being able to sense CRC, though; if you plan to use CRC during file transfers with a particular online system, look for a terminal settings option on the system where you can specify Xmodem/CRC as a default, or always select Xmodem/CRC specifically at a download menu. If there is no such selection available, the system can probably adjust for CRC automatically. If, on the other hand, nothing happens when you try to send or receive a file using Xmodem/CRC, you'll probably have to fall back on "straight" Xmodem.)

WXmodem. WXmodem stands for "Windowed Xmodem." Like Xmodem, WXmodem transmits 128-byte blocks; unlike Xmodem, it does not wait between blocks for an ACK or NAK. It monitors for those signals, but it "assumes" each block has been transmitted successfully, and immediately sends the next block.

In transmitting blocks in this nonstop manner, the sending computer is always one to four blocks ahead of the receiving computer's ACKs or NAKs. (The receiving computer's buffer must be large enough to accommodate this slightly faster influx of data, or WXmodem will not work.) The difference between the block being sent and the most currently received ACK or NAK is called the window--hence WXmodem. Under ideal situations, WXmodem keeps track of this difference, and thus knows which block is referred to and must be re-transmitted if a NAK is received.

Kermit:

Kermit (yes, it's named after Kermit the Frog) is equally as popular as Xmodem. Created at Columbia University in 1981, it was designed to be more flexible and convenient to use than Xmodem. It's not yet implemented on as many online systems and programs as Xmodem, but this is changing as computer users discover the program's usefulness.

How Kermit Works. Kermit is similar to Xmodem in that it transfers files in blocks--or packets, as they are referred to in Kermit. It also resembles Xmodem in its use of a checksum technique for error checking (a checksum bit--based on packet contents--is included with each packet).

Special Features. Kermit differs from Xmodem in several ways, not the least of which is the fact that it can transfer files using 7 data bits. Where necessary, Kermit converts 8-bit characters in a file to 7-bit characters, by stripping the 8th bit and sending it as a separate bit. Kermit also converts control characters into other ASCII characters that can be "safely" transmitted.

Another interesting feature of Kermit is that its packet sizes can be changed to accommodate fixed packet sizes on a remote system, or varying transmission conditions.

Finally, Kermit programs can re-synchronize their transmissions if interrupted by line noise--something else that isn't true of Xmodem.

"Wild Card" Transfers. If these advantages aren't enough, Kermit also allows "wild card" file transfers, which means you can use an asterisk in place of a file name or extension to transfer all files of a type. (Examples: Typing MIKE.* would transfer all files named "MIKE," no matter what their extensions. Typing *.TXT would transfer all files with the extension TXT.)

File Compression. Kermit saves time by using a clever transmission technique in which repeating characters in certain kinds of files are sent only once; this can result in a significant time savings.

Naturally, both the sending and receiving computers must use the same Kermit protocol, as quite a bit of decoding is required at the receiving end.

The Kermit Server Mode. Most versions of Kermit also offer what is called a "server" mode, which is a mode in which your software will take over and issue all commands necessary for sending and receiving files.

Ymodem:

Ymodem operates in a manner very similar to that of Xmodem, the major difference between the two being the fact that Ymodem transmits data in 1024-byte (1K) blocks, rather than in 128-byte blocks. Its major application is transmitting very large files.

You may encounter a UNIX-based version of Ymodem called YAM, which is actually the protocol from which Ymodem is derived.

Ymodem Advantages. Ymodem's large block size increases transfer speed significantly when few or no errors are encountered. Too, some implementations of Ymodem (called Batch-YAM) offer batch processing.

Ymodem Disadvantages. At first glance, Ymodem may seem imminently superior to Xmodem, due to its larger block size. It is--sometimes. The checksum error-correcting scheme used by Ymodem is similar to that of Xmodem/CRC in that it uses two

checksum bits at the end of each block. However, it differs in such wise that, if there is a lot of line noise, the re-transmission of blocks can slow down transfer significantly.

Another drawback of Ymodem is the fact that it adds to (pads) a block with extra nul characters to make sure it is exactly 1024 bytes in size. This means extra transmission time, too.

Zmodem:

Zmodem operates much faster than most other protocols, as it does not wait for ACKs when it sends data blocks. It only recognizes and acts on NAKs from the remote system. Zmodem blocks are 512 bytes in size. The protocol is relatively fast, even though it uses no buffering, which means a pause for disk access each time it sends or receives a block. Efficiency rates of 99% (239 cps @ 2400 bps) are routinely achieved with many online services' implementations of Zmodem.

Machine-specific Protocols:

There are a number of machine-specific protocol transfer programs, among them Cat-Fur for the Apple II series of computers, Punter for the Commodore 64/128, and Telink for the IBM PC.

These protocols aren't generally used by online services, but can be found on BBSs and a few online services that cater to specific computers.

System-specific Protocols:

Certain online services offer special file-transfer protocols designed to make optimum use of their software. CompuServe, for example, offers B Protocol, which is quite popular among users. It is supplied with all CompuServe VIDTEX software (see Chapter 5), as well as with many PD, shareware, and commercial programs. If your favorite communications program doesn't accommodate B Protocol, you may be able to find a "standalone" B Protocol (usable with almost any communications programs) on a BBS or other online system.

Proprietary Protocols:

Various modem and software manufacturers have developed what are called proprietary error-checking protocols. These are protocols that use special file transfer and error-checking techniques developed by the manufacturer and not released to the public.

As with other error-checking protocols, successful use of a proprietary protocol requires that the same protocol be used at each end of a file transfer. Because some such protocols are available only with a specific modem or software package, the product in question must be used at both ends. This obviously means, in turn, that the manufacturer or publisher sells more modem units or copies of software.

This may sound overly mercenary, but it's not all bad. Proprietary protocols often offer advantages that make restriction to a particular modem and/or software package worthwhile. Some of those advantages are discussed in the following paragraphs.

Software Protocols. Proprietary error-checking protocols implemented in software vary in design and features, but generally use a block- or packet-based system of file transfer.

Sometimes a proprietary protocol is the only protocol

provided with a communications program. Because such a program focuses on one job--transferring files via a specific protocol--it may include features such as file compression and adaptive parameter block size, in addition to what its designers feel is the best file transfer technique available. (In many cases, this may be close to the truth.) If you use this kind of program, however, you can transfer files only with programs that use the same protocol.

Ideally, a program that comes with a proprietary protocol should provide some of the more common protocols, such as Xmodem and Kermit, as well.

Some proprietary software protocols are available with more than one program. The Hayes protocol, for instance, is offered by many programs other than Hayes' designated software, Smartcom. The same is true of Crosstalk's protocol, which is implemented in Mirror II, among others.

No matter if a proprietary protocol is available with only one program or with several, using one may result in improved transfer efficiency and ease of use. This is because the implementations of proprietary protocols do not vary, and use the same set of commands and signals. Too, you can usually use the default communications parameters provided with the program, which greatly streamlines operation.

Hardware Protocols. Several error-checking protocols are implemented in hardware by modem manufacturers, among them MNP and ARQ. When these protocols are used, data is collected into packets (or "frames") before transmission. Virtually all such hardware protocols use an approach similar to the checksum/CRC method to check for errors.

Hardware protocols are particularly effective in eliminating the effects of telephone line noise, and may offer data compression and other enhancements.

On the negative side, using a hardware protocol can cause minor delays during realtime operations. When the protocol is "on," the modem may not send typed-in characters until enough have been entered to fill a packet. Or, it may wait until a certain number of milliseconds have passed before sending characters--to make certain that no more are immediately forthcoming. Such delays are sometimes perceptible, sometimes not, but it is best to disable hardware protocols during direct, realtime communication with another system. (Save the error-checking protocol for use when you're transferring files, or if a connection is bad.)

Hardware error-checking protocols sometimes interfere with software error-checking protocol transfer, too. This kind of problem can be overcome, but you'll probably have to consult with the modem's manufacturer.

Modem manufacturers are particularly competitive in this area, and proprietary protocols are jealously guarded. However, a number use licensed file-transfer protocols like MNP, so it is possible to use some hardware protocols with modems from different manufacturers.

STANDARDS AND SELECTIONS: WHAT TO LOOK FOR

The communications software package you select should offer some means of ASCII file transfer. And, whether or not you intend to do much uploading or downloading right now, make sure you get a program that has at least one binary file transfer protocol. Sooner or later, you'll find a reason to transfer a file via modem.

Which Protocol?

Generally speaking, it's a good idea to have on hand as many protocols as possible. But Xmodem and Kermit are by far the most popular error-checking protocols in existence. You'll find either or both available on any online service or BBS that offers protocol file transfer--Xmodem more often than Kermit. So, you should at least have Xmodem capability. (Fortunately, almost all communications software packages include Xmodem.)

After Xmodem, I recommend Kermit--because of its efficiency and because it will eventually be on as many systems as Xmodem.

If you intend to deal with a BBS or online service that offers a system-specific protocol, it's a good idea to have that protocol, too--even if the system in question offers Xmodem, Kermit, or other protocols. As I implied earlier, a system's own protocol tends to be the fastest and most efficient.

The other general error-checking protocols discussed in this chapter will be of interest to you if you like to experiment, and you can probably find a good reason to use each of them. However, I suggest that you do your training with Xmodem and/or Kermit, if for no other reason than the fact that a lot of information is available on using those protocols.

As for machine-specific protocols, remember that you will find them in use on a limited number of systems. Proprietary protocols are, of course, recommended only under special circumstances.

SAMPLE FILE TRANSFERS

I've already presented a number examples of file transfer in this chapter, but we'll wrap things up with a few more examples of specific transfer methods.

Incidentally, the menus, prompts, and messages from BBSs and online services in the illustrations are exactly as you would see them if you dialed up these systems. The communications software windows and commands probably differ from what your computer and software will display, but the steps depicted are pretty much the same no matter what your system.

ASCII Download:

An ASCII file transfer can be as simple as opening your software's capture buffer and commanding the remote system to display the desired text file. Some systems offer a more formal ASCII download, though, providing prompts and sending specific end-of-file characters. DELPHI is one such system.

When you select ASCII Download at a DELPHI download menu, the system sends the file, followed by a delay, Control-Z, and Control-G (bell). These signal some communications programs to close the capture file. A portion of the ASCII download process is shown below, in Figure 9.24.

ASCII Uploads:

As with an ASCII download, an ASCII upload can be fairly simple: let the remote system know you are going to send text, supply a file name, and then use your software's TRANSMIT or SEND command to send the text. As shown in Figure 9.25, you usually have to signal the receiving system via a control-character, carriage return, or special command character when you've finished uploading.

Timed and Prompted Uploads. As I mentioned a while back, the buffers in certain areas on some online systems cannot take

ASCII input at high speed. Such a system may require that your system wait a specified time between each line it sends, or that a line be sent only when a specified character, called a turnaround character, is sent. (This is rarely found outside of message entry areas nowadays; most database and personal file area buffers are designed to handle text at full speed.)

On systems that use a time delay, the upload will look just like a "straight" upload, except that the flow of text will be slowed down greatly.

When turnaround character prompting is used, you must set your system's turnaround character to match that used by the remote system (it's usually ?, >, or :). Once the turnaround character is set, your software will wait for the turnaround character (usually ?, >, or :) before it sends each line. (Some communications programs require you to specify that the turnaround-character prompting be used, while others "turn on" this feature whenever a turnaround character is set.) Figure 9.26 shows a portion of a prompted ASCII upload.

Xmodem Download:

The Xmodem download procedure is fairly straightforward on most systems. Xmodem protocol is either selected at a menu, or exists as a system default (set by you or by the system). In the example that follows--Figures 9.27 through 9.30--you'll see the steps involved. (Note that GEnie has the Xmodem protocol as a system default. You can use CRC, as I am in this example, and GEnie will adopt to it automatically. This is true of most BBSs, as well.)

Note that GEnie provides a rather complete report on the outcome of the download. Had I aborted the download, the system would have responded with: ((sample screen here))

Xmodem Upload:

Next, we'll take a look at an Xmodem upload on DELPHI. As with an ASCII upload, the system asks me for a filename when I type the Xmodem upload command (XUPLOAD). After I've entered the filename, I'm asked whether the file is text or binary. That question answered (use "binary" if you're in doubt), I'm prompted to send the file, as shown in Figure 9.32.

I'm using Xmodem/CRC in this example, but I didn't have to specify the CRC option because it is stored in my DELPHI system profile as my file-transfer protocol-of-choice.

When the download is complete, DELPHI provides me with a rather brief status report (Figure 9.33).

The procedures for other protocol downloads and uploads are not unlike those shown for Xmodem. The basic steps are the same: Let the remote system know you are going to download or upload, provide a filename and protocol choice if necessary, and initiate the transfer process on your end when prompted to do so. You may have to set packet sizes with some protocols, or provide commands to specify file groups in a batch transfer; consult your communications program's documentation for more information.

File Transfer Tips:

A file transfer can go wrong in a number of ways, but the most common sources of trouble during file transfer include telephone line noise, bad files, and unmatched protocols. You can't anticipate every problem, but there are certain precautions and procedures you can observe to minimize the possibility of trouble during file transfer.

Setting Up and Signing On:

You should, of course, make sure your equipment is properly connected and that you have set the proper communications parameters for the system you are dialing up before you attempt to sign on to a system.

You'll want to give special attention to telephone line conditions when you first sign on. Watch for evidence of line noise (in the form of "garbage" characters appearing on your screen). If it appears that you have a noisy line, sign off and redial to get a better connection. If the problem persists, wait an hour or two before attempting the file transfer.

If the weather in your area is bad, postpone the file transfer until things quiet down. Telephone line noise frequently accompanies bad weather--especially when lightning is present (and you shouldn't be using your computer during a lightning storm in any event).

Observing Protocol:

Make sure you select the proper protocol when you tell the remote system you want to transfer a file. Sometimes one letter's difference--entering an X rather than a K, for example--can result in your system trying to communicate with the remote system under the wrong protocol.

If you can't get a protocol to work with a particular system, experiment a bit with different settings. Because implementations of protocols differ slightly from system to system, you may occasionally find that "modified" protocols don't work. For example, two bulletin board systems that I call regularly implement Xmodem/CRC in a manner that is slightly different from the way my communications program handles it. Even though I select Xmodem/CRC when I want to upload a file, and initiate an Xmodem/CRC upload at my end, my software refuses to communicate with the BBSs software. So, I have to "fall back" to straight Xmodem to transfer files on these systems.

If a system offers a specific protocol (like B Protocol on CompuServe), use it. You'll find it faster and more reliable than other protocols every time.

Aborting a Transfer:

If you feel that a transfer is not going right, wait a few seconds before aborting it. Either your system or the remote system may correct the problem by resending bad data, or abort the transfer themselves.

If you do abort the transfer, you may have to issue an additional command to the remote system to let it know you want to abort the transfer. This is usually done by sending several ^Cs or ^Xs (the remote system will usually tell you how to abort before you initiate the transfer). In some cases, your software may send the proper abort signal to the remote system when you tell it to abort the transfer at your end.

If for some reason the remote system doesn't respond to the abort commands or to any other commands, refer to the "In Case of Fire..." section at the end of Chapter 7.

A NOTE ON TRANSFERRING PROGRAMS BETWEEN NON-COMPATIBLE COMPUTERS

I've already established that text files can be transferred between computers of varying types, but what about programs? The answer is no and yes. No, you cannot transfer IBM programs from, say, an IBM PC to a Commodore 64 and expect them to run on Commodore. This is because the content of the IBM files would be

absolutely meaningless to the Commodore. You can, however, transfer a Commodore program or data file from an IBM PC to a Commodore 64--I've done it. And how did I get a Commodore program on an IBM disk? Well, I downloaded the program from GEnie, where it was stored on a Honeywell mainframe computer. I then had a friend with a Commodore 64 dial up my computer direct, and I uploaded the program to him, It worked perfectly on his machine.

This may seem a bit odd until you consider that what I did was no different from what happens when a program is uploaded to and stored on an online service mainframe or mini-computer, where disk formats are definitely alien. As long as the content of a program file remains intact, it can be stored in any disk format for later transfer to a computer that can use it; modem transfer is the key. So, if you're online and see a file you'd like to get for a friend with a "foreign" computer, don't hesitate to download it for later transfer to his or her computer.

If you found this excerpt useful, you may want to pick up a copy of the book from which it was excerpted, THE MODEM REFERENCE, recommended by Jerry Pournelle in Byte, The New York times, The Smithsonian Magazine, various computer magazines, etc. (Excerpts from this book accompany this file.) THE MODEM REFERENCE published by Brady Books/Simon & Schuster, and is available at your local B. Dalton's, WaldenSoftware, Waldenbooks, or other bookstore, either in stock or by order. Or, phone 800-624-0023 to order direct.

In addition to explaining the technical aspects of modem operation, communications software, data links, and other elements of computer communications, the book provides detailed, illustrated "tours" of major online services such as UNISON, CompuServe, DELPHI, BIX, Dow Jones News/Retrieval, MCI Mail, Prodigy, and others. It contains information on using packet switching networks and BBSs, as well as dial-up numbers for various networks and BBSs, and the illustrations alluded to in this excerpt.

You'll also find hands-on guides to buying, setting up, using, and troubleshooting computer communications hardware and software. (And the book "supports" all major microcomputer brands.)

Want the lowdown on getting more out of your word processor? Read the only book on word processing written by writers, for writers: WORD PROCESSING SECRETS FOR WRITERS, by Michael A. Banks & Ansen Dibel (Writer's Digest Books). WORD PROCESSING SECRETS FOR WRITERS is available at your local B. Dalton's, Waldenbooks, or other bookstore, either in stock or by order. Or, phone 800-543-4644 (800-551-0884 in Ohio) to order direct.

Do you use DeskMate 3? Are you getting the most out of the program? To find out, get a copy of GETTING THE MOST OUT OF DESKMATE 3, by Michael A. Banks, published by Brady Books/Simon & Schuster, and available in your local Tandy/Radio Shack or Waldenbooks store now. Or, phone 800-624-0023 to order direct.

Other books by Michael A. Banks
UNDERSTANDING FAX & E-MAIL (Howard W. Sams & Co.)
THE ODYSSEUS SOLUTION (w/Dean Lambe; SF novel; Baen Books)
JOE MAUSER: MERCENARY FROM TOMORROW (w/Mack Reynolds; SF novel; Baen Books)
SWEET DREAMS, SWEET PRICES (w/Mack Reynolds; SF novel; Baen Books)
COUNTDOWN: THE COMPLETE GUIDE TO MODEL ROCKETRY (TAB Books)
THE ROCKET BOOK (w/Robert Cannon; Prentice Hall Press)
SECOND STAGE: ADVANCED MODEL ROCKETRY (Kalmbach Books)

For more information, contact:

Michael A. Banks
P.O. Box 312

Milford, OH 45150