Everything You Wanted to Know About Interfacing to the HP3000 PART II

John J. Tibbetts
Vice President, Research & Development
The DATALEX Company

INTRODUCTION

The title of the this talk is "Everything You Wanted to Know about Interfacing to the HP3000-Part II." In Part I, Ross Scroggs described in great detail characteristics of the internals of the asynchronous communications protocol, especially for the benefit of those who would wish to tie foreign devices onto an HP3000 through the asynchronous port. This talk — the second part — is intended to take that discussion into a specific direction and discuss how, specifically, to connect intelligent devices, in particular microcomputers, to the asynchronous communications protocol of the HP3000. Note immediately that we are restricting our discussion of microcomputer communication to the asynchronous communications protocol. The reason for this is simply that most microcomputers are easily configurable to communicate asynchronously. Few microcomputer hardware and software packages have been assembled so far which will use bisynchronous communications protocol. Consequently, the reality of the current state of microcomputers suggests that asynchronous communications protocol will be the standard way of hooking up your microcomputer to the HP3000.

This talk will have two major parts. In the first part, we will discuss what is at issue in terms of features and capabilities in a remote communications program. We will discuss in some detail what our standard approaches to handling such capabilities as terminal emulation, sending and receiving files, simultaneously printing to a local printer, and control of the communications protocol from either the local end — that is, the microcomputer end — or from the remote computer end. In this talk we will refer to the local side as being the microcomputer and the remote side as being the remote or the host computer.

In the second part of the talk we will describe how you can actually get such a program running on your own machine. The choices are twofold: either buy one or write one. By using the criteria we have established in part one of the talk, we will try to outline some of the considerations of doing either of these.

One final note before we begin is that the remote communications capabilities tend to be a very hardware- and software-specific part of microcomputing. Whereas one can usually take a CPM program written in BASIC, for instance, and run it on most or all CPM implementations, one cannot expect to do the same with remote software. Remote software usually has to talk to pieces of your microcomputer which the operating system tends not to know anything about. During the course of the talk we will periodically make reference to a specific capability that is required for the remote program and in the published paper we will annotate them as a capability bullet that you will need to either have supplied to you or you will have to implement on your microcomputer to get this particular capability to be implemented for your remote program.

TERMINAL EMULATION

The first and perhaps the easiest capability to implement on your microcomputer is the emulation of a simple terminal.

- Capability — handling the remote port. Any of the operations we will be discussing for remote communications program presume that your microcomputer has a separate usable asynchronous communications port. Your program should be able to perform the following operations on that port:
  - read a character
  - write a character
  - test to see if a character is ready to be read.

  This last capability is the one that is usually missing in the standard microcomputing operating environments. In particular, CPM implements the read and write character routines as the reader and punch devices, respectively, but do not have a standard driver entry for testing the status of the remote port. Usually, you have to specifically write this capability for your own hardware if it hasn't been provided to you by someone else.

The standard procedure for implementing a terminal emulator is to write a polling loop. In the polling loop a very tight program loop tests to see if a character has been entered, either at the remote port or at the keyboard. If the character has been entered on either one, the character is then read and written to its opposite port. Thus, a character entered at the keyboard of the microcomputer, when sensed, would be written to the remote port and vice versa. It is important to make the polling loop as quick as possible. No code should be included in that loop unless it is absolutely necessary. Especially if you are writing in a high level language and especially if that language is interpretive, such as BASIC, you may have speed problems when trying to emulate a terminal at higher baud rates, say over 2400 baud. When writing in assembly language this is usually less important and you will find that you can support virtually any standard baud rate.

When emulating a half duplex terminal, any character entered at the keyboard should be immediately written back to the screen, thus providing the local echo. If terminal emulation requirements stopped here, a terminal emulator would be a very easy piece of software to write. Unfortunately, there are usually a few special problems with the terminal emulator.

The first problem is handling the break key. Many timesharing systems do not require break keys, but as any member of the HP3000 audience knows, the break key is a very crucial part of terminal handling on the HP3000. Unfortunately, the break key is not a character in the way any other keystroke on the terminal is. When depressed, the break key actually changes the electrical state of the transmit pin.

- Capability — sending a break. Most microcomputer communication ports have a mechanism by which the output port can be put into a break state. It almost always requires assembly language programming to implement a break key function. The actual reasoning behind break key handling goes beyond the scope of this talk. Suffice it to say that the preferred technique of break key transmission is, when the break key of a terminal is sensed, to put the output port into the break state until either 200 milliseconds have elapsed or a character comes in on the remote port.

Thus, to properly handle the break key our polling loop now needs to be expanded to test to see if the value entered from the keyboard is the break signal. When the break signal is sensed, the polling loop should then, instead of sending that character, invoke the send break routine to send a break.

The second capability which makes the terminal emulator more difficult relates to simultaneous printing of the terminal interactions on a printer which is hooked up to the microcomputer. The whole issue of printers, and especially printers that might hold up communications flow, is dealt with in a subsequent section.

SENDING AND RECEIVING FILES

Probably the main, useful work we would like for a communications program is to send files from our microcomputer to our HP3000 and receive files from the HP3000 down to the microcomputer. At first glance this may seem to be a rather simple operation. To send the file we should simply read the file from the local storage medium on the microcomputer and write it out the remote port. To receive a file we should simply read from the remote port and write to the diskette. If it were only so simple . . . There are three issues which will significantly complicate the issue of sending and receiving files. They are:

1. The vast majority of computers need time for themselves. What I mean by this statement is that at various times in the life of a computer it needs time to handle data it has been sending or receiving. On an HP3000, if you should try to type characters into it before it has put a prompt character up, you know that you will lose those characters. On a microcomputer, if you try to enter characters into most microcomputers while it is reading and writing a diskette file, for example, those characters will be lost. These phenomena reflect the fact that most computers are not designed to be able to handle communications of their terminal or remote ports at any time they are activated. A newer line of more commercially oriented microcomputers are beginning now to feature interrupt systems that do have full functioning typeahead systems which greatly ameliorate these problems. However, these microcomputers are definitely in the minority. Thus, our communications program needs to somehow be able to allow each computer to have time for itself when it needs it.

2. Communications lines tend to be rather noisy, especially if we are using the telephone system to transmit our data. Since file integrity is usually important, we need to come up with some kind of error checking protocol which can detect errors in the transmission of the data being sent or received. Interestingly enough, most of the programs running now on microcomputers for sending and receiving data do not handle error detection. The reason is that so far most microcomputer users who are using communications programs are doing so to make use of timesharing networks such as The Source for sending and receiving programs. As more, real, data processing functions, which might relate to shared databases or distributed data entry, are being built, clear data transmission protocols will become very important.

3. Most systems have some kind of difficulty with binary transmissions. This may not be a problem in applications in which only textual data needs to be sent. As time goes on, one finds the need to send binary data, for instance, to distribute object code of programs through the communications program. Thus it becomes desirable to be able to send binary files.

This is a summary of the problems — now let's take a look at some of the possible solutions.

MESSAGE HANDLING

Message handling is the general title by which we refer to the problem of the traffic control of the data being passed back and forth between the micro and the HP3000. The message handling protocol determines when data can safely be sent or received so that we never go faster than either of the machines can accommodate. The very first thing that becomes apparent after some investigation and experimentation is that the send and receive case are quite different from one another with *this* pair of computers. This is unusual when compared to communications software usually existing between microcomputer and microcomputer. In that case, the communications message handling is usually symmetric; that is, whatever convention is used to control data flow on the send side is also used symmetrically in the other direction to control it on the receive side. We have to do extra work on the HP3000 since none of it asynchronous communications protocol was designed for access by an intelligent terminal, and it ends up having some asymmetric properties which we have to deal with.

First, let's consider the case of sending data from a microcomputer to an HP3000. The first fact one must always be aware of when trying to send data to an HP3000 through its asynchronous communication port is that it can only read data from a device when a read is up; that is, when a read has been issued from a program. If you try to type ahead on an HP3000, the data is lost. Fortunately, in the HP3000 communications software a character is always sent whenever a read is put up. That character is the Control-Q or the DCl character. Thus, any device trying to send data to the HP3000 can simply wait until it sees a DCl and then send its record of data terminated by a carriage return. This type of *data interlocking* is the preferred method of sending data to an HP3000. For instance, this is the mechanism that LINK-125 uses in its protocol. It simply invokes FCOPY, and when FCOPY puts up its first read, it hands a record of data to it, terminates it with a carriage return, and proceeds with file transmission in that fashion.

But this is not always good enough. Consider this case: a message has been transfered to the HP3000, a carriage return sent following it, the HP3000 has issued another read, has sent the DCl back along the phone line and suddenly there is a noise burst on the phone line. The DCl coming back to the microcomputer is lost. The microcomputer is waiting there to transmit its next record of data with the DCl and then deadlocks because it never sees the DCl. This type of deadlocking is characteristic of trying to make too much out of a simple interlocking protocol. What we really find as more desirable is to write a communications program on the HP3000 which talks to the program on the microcomputer. This will allow the microcomputer program and the HP3000 program to issue reads with timeouts which would require that after a certain amount of time we give up on a particular read because of lost protocol characters or a dropped line. In the particular case of the missing DCl — and this is only one of the pathological conditions that can arise — the microcomputer program can simply, after a certain amount of time, send a message up the line which says something like, "Hey, are you still there?" to which the HP3000 program will response, "Yes, I am still here and here is another DCl" or may not respond at all if the machine has crashed or the line has gone down. This concept of using programs on both sides is really what differentiates very simple dumping of files up and down the line from more sophisticated communications protocols. I feel that this approach is required for any serious use of communications, especially with any bulk of data transmission which we would like to move reliably back and forth. Using this "program-to-program" approach, we can also perform some other more sophisticated error checking which we will get into shortly.

As we leave the send case, note this important fact. Make sure that after a record has been sent to the HP3000 with its carriage return the very next piece of work the microcomputer does is to turn around and wait for the DCl before it does anything else. One might be tempted to put up the next read to the diskette to pull the next record off while waiting for the DCl. If your microcomputer has the appropriate communications typeahead software on its remote port, you might be able to get away with this. However, in general the mircrcomputer needs to wait for that interlock character to come back before it tries to do any other useful work. Otherwise, you will start missing DCls and your program will get hung up.

RECEIVING FILES

Just as we have done with the send case, let's examine the most trivial method of receiving files which would not rely on a program being run on the HP3000 side. The basic fact of life when receiving files is that the remote computer — the HP3000 — will be instructed to start sending down a file; perhaps we use FCOPY or the editor to start sending a file to us. The microcomputer is going to periodically need to write out the buffer it is accumulating to the disk drive. When it does this there will usually be a second in which it can't receive any data. The first approach is to just receive small files, in which case the microcomputer never writes out its data until it has collected the full file in memory. This of course limits the size of the file you can receive to the amount of available memory on the microcomputer, usually somewhere between 10,000 and 40,000 bytes. Obviously, this is an unsatisfactory method of receiving files unless your application is very limited. The next idea that comes to mind is making use of the X-on/X-off characteristics of the HP3000 to control this flow. As a human user, sometimes when a listing is coming out too quickly onto the CRT, we stop the flow by typing the

X-off key which is a Control-S and most of the time the HP3000 stops its transmission flow until you have done what you wanted and then you hit a Control-Q and the scrolling of the data output continues. Maybe we could have the microcomputer perform this function for us as a simple interlocking method.

The answer is that "Yes, we can," however, it is not the preferred method of receiving files. The reason for this is that, surprisingly, Control X-on/X-off protocol seems to have some holes in it on the HP3000 side. Someone told me that after some extensive testing they found that one out of five X-off characters seems to drop into a hole when sent to the HP3000. I have absolutely no way of verifying this other than to tell you it has happened a number of times to me. This doesn't make the use of this mechanism impossible, it simply complicates it somewhat.

Using this technique then, what you need to do is:

1. Build a large receive buffer.
2. Start receiving data until the buffer gets to 80% or 90% full.
3. Send an X-off character to the HP3000 but keep receiving the characters onto your microcomputer.
4. After some predetermined timeout time — perhaps a second of no characters coming in — assume it has finally absorbed the X-off character, and then you can proceed with your disk writes of the buffer.

But, if 4 or 5 characters have passed without stopping, send the X-off character again. Repeat these steps until the data transmission actually stops.

Just like the send file case, I recommend the use of a program on both sides. Using this technique we will simulate the kind of data interlocking protocol that the HP3000 uses. That is, every time the microcomputer is ready to issue a read to the remote HP3000 it will issue a character, perhaps for symmetry's sake a Control-Q, or any character of your choice. When that character is received by the program on the far side, that program will then send down the next record of data to the microcomputer followed by some standard termination character. After the message has been received, the microcomputer can set to work writing that message to the disk or doing whatever other housekeeping it would like to do. It then issues the next interlock character, and proceeds. This protocol also allows for the kind of timeout mechanisms that I described in the send case so that you can recover from lost transmission and especially lost protocol transmission. It will also easily accommodate the kind of error checking we will be talking about in the next section.

As always, there is a complication and a warning. Even in the case we have just described, we have not really built a symmetric communications protocol to the HP3000. The interlock character itself, which is going to be sent to the HP3000, has to be read by an HP3000 read. Of course, that HP3000 read will have a DCl coming right before it and any attempt to write the protocol character up the line before the HP3000 is able to read it results in a lost protocol character. Thus, some real world experimentation is usually needed in which some delay is required after the record has been received from the HP3000 so that it will have had time to finish writing the record and then put up the read which will read the next character interlock. It's for reasons like these, incidentally, that interfacing microcomputers to the HP3000 has not always been the simplest and the least frustrating of tasks.

The other item of note is that on reading characters into the microcomputer it is usually wise to strip out occurrences of the protocol characters that have accumulated in the asynchronous communication chip. These characters would be the line feed character which the HP3000 will usually tag onto the end of the carriage return unless you turn that off, and also the DCl character itself. Although these characters will be flying around during the transmision of the data, you don't want to include them into the data stream itself. They should be filtered out of the actual data flow.

ERROR HANDLING

Now that we have described the actual methods by which data can be sent and received, let's go on to the second defined problem in our data communications task — error handling.

The fact is that there are very few asynchronous communications protocols which go to this level, and I find this fact to be extremely regrettable. No serious large-scale interface of microcomputers to any kind of data processing network can be accomplished without real error checking. However, once we have built the proper send and receive frameworks with the right kinds of interlocking and assuming there is some intelligence and flexibility on both ends, it becomes rather easy to add the error handling phase. What are some of the usual techniques for adding error handling to the send and receive cases that we've described?

The standard mechanism, of course, is to add to each message sent or received some kind of check character or checksum which is used to check out the validity of the data. The simplest form of a checksum is an addition of the various character values of the message. For instance, if one record of data I am sending to the HP3000 is 40 characters long, the microcomputer can run through those 40 characters, add up the ASCII value of the 40 characters, and then produce a new character for the string and tag it onto the beginning or the end of that string. The HP3000 on the other end, when it has received the data, will go through the very same operation except that this time it will strip the character off and compare it with its own calculation of that string and see if they match.

There are some problems with the simple add-em-up checksum and there are many other sophisticated algorithms around — I can refer you to literally any book on communications for a description of CRC algorithms. The problem with CRC algorithm is that it's usually a fairly difficult algorithm to execute quickly enough on a microcomputer unless you are programming in an assembler language. The algorithm I have found to be very simple but very effective is an algorithm which adds and shifts the bits as the characters roll in. In this algorithm each character is added on to the checksum and then the checksum is multiplied by 2 which shifts all the bits to the higher order by one bit. Then it receives the next character and repeats the process. This final 16-bit quantity is then tagged onto the message.

You have to remember not to freely insert binary integers into the communications stream. Some adjustment of the value must be done when we are sending it to the HP3000 to make certain we are not sending a character it will have difficulty receiving.

Once a message has been sent to the other side with a checksum on it, the other side has the opportunity to examine that message and respond. The typical response is for the receiver to send back some predetermined character message which says either: the data was received successfully and you should proceed to the next block; or, alternatively, the data was not received correctly so retransmit the block just transmitted. I usually include a third state in this message traffic which indicates that something terrible has happened on one end or the other and to abort the entire transmission process altogether. You can include in this function the ability for the user to hit some kind of escape key and abort the communicatons traffic.

One other item I have found to increase reliability is to add sequence numbers on each of the messages sent or received. This would ensure that in some pathological case we don't actually get the blocks out of order; that is, in a case where an entire block has dropped out of the communications traffic. Although this is fairly rare, there are actually certain conditions which can cause something like that to happen. A sequence number which is checked on both sides for each block transmitted can protect against this possibility.

BINARY TRANSFERS

We have mentioned previously that it is generally unreliable to transmit 8-bit binary characters from a microcomputer up to an HP3000. What are the possible ways around this problem? The standard way is to simply convert the 8-bit binary traffic into hexadecimal strings, that is convert a binary character 255 into the ASCII string FF, etc. Of course, you would probably immediately see this means that there is a 50% reduction in communications efficiency. This technique is usually simple to perform and it is useful when the binary traffic is somewhat limited. A technique that I prefer is to translate seven 8-bit bytes into eight 7-bit bytes. This is quickly accomplished by gathering the 8th level bit of the 7 bytes input and building another byte and tagging it onto the back end of each 7-byte block. This effectively chops the 8th level off the communications stream at transmission time and is then reassembled on the far side. If this technique is used on the entire message, including checksums, sequence numbers or any kind of message identifier on the block, the whole communications interface becomes considerably simpler.

USING PRINTERS

It is often desirable in a communications protocol to log the data to the printer. For instance, on receiving a file to a microcomputer you may want to get a listing of it. Alternatively, you may wish, during terminal emulation, to get a copy of that session onto a hardcopy printer.

Like everything else mentioned in this talk, there are hidden catches. It seems simple enough to be able to put in a switch in the software — for instance, in the polling loop of the terminal emulator — that when a character has been sent or received, it should be sent to the printer port. However, many printers don't print at the communications speed. We will therefore distinguish between a fast printer and a slow printer. In this context, fast and slow do not have any absolute meaning to them. Fast means that the printer operates faster than the current communications context, and slow means that the printer operates slower than the current communications context. For example, in a 300-baud environment most printers (for example, an Epson matrix printer or a TI-810) will be fast printers. However, at 1200 baud most of the inexpensive matrix printers are slow printers, that is, they cannot keep up with the 1200-baud stream. Surprisingly, even printers such as the TI-810 which are rated at from 120 to 150 characters per second often cannot keep up with the 1200-baud flow of data. Therefore, the determination of whether a printer is a fast or slow printer can only be done by running a series of tests.

As you might now be able to suspect, there is very little difficulty with a true, fast printer in our communications program. Any characer we wish to print we simply output to the printer port. However, on a slow printer we have to do more resource balancing in that there is now another resource in the communications environment which needs time of its own. Adding a slow printer to a communications program can easily double the complexity of the communications environment.

At this point let me summarize a few of the major elements of printer integration:

• If the communications program has been implemented, as I have been suggesting, with a pair of programs on either end which have an interlocking mechanism, the simplest approach of integrating a slow printer is to print out the block of data during the time that the program is performing activities such as writing to the diskette or reading from the diskette. That is, after the message has been sent or received and before the interlock causes the pair of programs to proceed, the buffer of data sent or received can be put to the printer. An unfortuante side effect of this approach is that the printer is only printing between records. This does not take advantage of the fact that there may be sufficient time during the actual communications transmission to have the printer doing some useful work.

• Improvement on this scheme requires a new capability:

> Capability — Printer Ready — The printer ready capability says that our communications software can sense when the printer is available; that is, when a character can be written to the printer in such a way that the printer buffer will absorb the character instantly.

With a printer-ready capability in our software, we can build a more sophisticated operating environment in which we have, in effect, a small spooler being operated. That is, any data which has been successfully sent or received and is ready to print can be added to a print buffer. This buffer is metered out to the printer when the printer is ready. It is important that the remote communications facility always have top priority. The other mechanism that needs to be in effect in this type of environment is that as the printer buffer gets close to being full, a flag will go up which will hold the interlock the next time around until the print buffer has been totally cleared. Although this mechanism sounds somewhat obtuse, it actually provides a very effective method of integrating a slow printer into a communications environment.

• Integrating a slow printer into the terminal emulator mode can be accomplished by using the X-on/X-off character techniques I described in the receive section. That is, if printing is active, a mechanism will go into effect, during terminal emulation mode, such that the microcomputer dispatches X-on/X-off to control the characters coming from the remote computer into the microcomputer.

BIDIRECTIONAL CONTROL

The last major capability we will discuss in our communications program is the ability for the remote computer to assume control of the communications program. This can be very desirable in applications in which an operator may activate a communications program and get online with an HP3000 and perhaps start a UDC. At that point the UDC might take over all control of the microcomputer through the communications program such that it can request files to be sent and received. The following are a couple of points concerning bidirectional control:

• The basic concept in bidirectional control is that the remote computer can have some escape character which it can send to the microcomputer during terminal emulation that will cause the remote computer to assume command of the microcomputer. Commands can then be dispatched by the remote computer directly to the microcomputer. Be sure that all of the issues previously mentioned about interlocking and protocol are also supported by any direct interaction between the remote computer and the microcomputer.

• It is very useful to be able to have a capability whereby the remote computer can ask for directory listings directly from the microcomputer. This gives the remote computer a list of what files may need to be sent or received.

• You may wish to give the remote program the ability to actually terminate the communications session itself and to remove the user from the terminal emulator mechanism.

IMPLEMENTATIONS

The best thing you can do with communications software is to buy it rather than develop it. Unfortunately, this assumes that someone has developed the type of software running on the type of machine you desire. As we've indicated during the course of this talk, remote communications software tends to be more hardware dependent than almost any other software running on your computer. Not only is it hardware dependent, it is also operating system dependent. Thus, on a single machine — for instance, the Apple which can run the Apple DOS, the PASCAL operating system, and CPM (if the CPM card is added) — each of these three operating systems has a different file system and each has different requirements for its communications program. This means that no one program will solve all of your problems.

Let's consider some of the available implementations. Under CPM, there are a couple of programs fairly well known in the CPM community for doing file-to-file transfers. They are a program called CROSSTALK and a program called COMMX. Both of them are available through the major CPM software distributors. Both of the programs feature a non-protocol mode and a protocol mode. In the non-protocol mode you can easily make the software talk to your HP3000 by setting the DCl character to the interlock character. Unfortunately, on both of these programs the protocol mode which includes the checksumming algorithms is only usable when the program is talking to another CPM program of its own type. Clearly, these programs are written for CPM systems to talk to other CPM systems, not to another computer system. This means that if you do wish to turn one of these systems into a protocol checked operating environment, you need to do a little extra work on it. If you have an HP125 you can acquire LINK-125 which does a good, but not error-checked, link with the HP3000.

None of the programs I have seen feature bidirectional control which would allow, as I have described in the talk, the remote computer to assume the control of the microcomputer.

If you are running some variant of the UCSD PASCAL or UCSD p-System operating environment, then, with all due modesty, there is no better communications software available than that provided by our own company. It incorporates in a table-driven fashion, ready-to-run for the HP3000, all of the capabilities described in this talk, that is: full error-checked protocol, the ability to support fast and slow printers, full bidirectional control, and blank compression of the data. All of the software for communicating with an HP3000 has been worked out in great detail. Incidentally, we also support protocol-oriented communication for other p-Systems — that is, for p-System to p-System communication — as well as communication with the IBM 370 interactive operating system such as CMS, CSS, or TSO, and DEC-10, -11, and -20 support.

Something new is that the software distributors for the UCSD p-System now have a CPM file compatibility mode which, when available, will mean that we can also use our communications software to send and receive CPM files as well.

CONCLUSION

If there is any theme for a discussion of communications software, it is *"There is More Than Meets the Eye."* As I have repeatedly stressed, the very best way of solving your communication problems is finding someone else who has already solved them and acquire the software from them. This is my very strong recommendation when attempting to establish a remote communications network for your system.

I would also refer to the other talk I am giving at this meeting which encompasses distributed processing applications using microcomputers. It is entitled "Microcomputer-based Transaction Processing with Your HP3000" and it goes into some detail about the state of the art in microcomputer software for distributed processing.