NOBUF/NO-WAIT IO

No-wait IO can be a very powerful tool in dealing with multi-terminal applications. It is an option of the MPE File System that allows a program to issue an IO request to a file/device and continue processing without having to wait for the IO to complete. This paper will explore the File System mechanism used to provide this facility, no-wait IO applications techniques, and no-wait IO considerations for the application programmer.

Madeline A. Lombaerde
Hewlett Packard
General Systems Division

January, 1980

INTRODUCTION

There are two situations that arise in dealing with IO requests: one is that the requesting process needs to have the IO complete before it can continue execution. Perhaps the process needs to have input data in order to decide what to do next; perhaps it requires assurance that the output successfully reaches the destination device before attempting further processing. In any case, the process must be 'blocked', that is, prevented from executing until the IO completes. This is what is referred to as 'waited (Wait) IO'.

The second situation is where the IO need not complete in order for the requesting process to continue. Perhaps the input will be required at a future time, but it is not needed immediately. Perhaps the process does not need to wait until the output reaches its final destination. The process does not have to be blocked: this is referred to as 'no-wait IO'.

The MPE File System alternates between these modes depending on the requirements and specifications of the user made at the time of opening the file (via FOPEN) and accessing the file (via FREAD, FWRITE, FCONTROL, etc.). Typically, when a file is opened on the HP3000, the File System sets up a buffer area outside of the user process stack data segment. Transfers to and from the file are handled through this buffer area (which is known to MPE as an extra data segment). When sequential reads are done from the file, the File System tries to keep all buffers full in anticipation of future requests by the user process. The File System makes IO requests whenever it decides one or more buffers need filling. Since these IO requests are made prior to the actual user process request, there is no reason to make the user process wait. Thus, anticipatory reading (which is also called 'pre-reading') is handled by the File System as no-wait IO transfers.

Similarly, when output is sent to buffers from the process stack data segment, the buffers are only written (posted) to the file/device when the buffer is considered full. This posting is done as no-wait IO. [A buffer is exactly one physical record (block) long; it is 'full' when the last (or only) logical record in that block is written.]

However, the File System cannot always use no-wait IO for buffered file transfers. If the user process requests a particular record which has not yet been brought into a buffer, then the File System must request that the process be "blocked" (make it wait) until the IO has completed (the desired record is in the buffer).

These decisions made by the File System when handling buffered files are transparent to the user: as long as the File System handles the IO transfers via the buffer(s) for that file, whether the process will have to wait or not will depend on the contents of the buffer(s) when the user request is made. [However, with fixed and undefined length record files, it is possible for the user to use the FREADSEEK intrinsic to request that a certain record be pre-read (no-wait) into a buffer.]

The situation changes when the user elects to access a file in 'nobuf' mode. In this mode, transfers go directly to or from the process stack data segment: no File System buffers are involved. The default mode is to make the process wait until the nobuf transfer has completed. The no-wait IO option (specified at file open time) allows the application process to request that the File System not force it to wait: when it requests a transfer, the user process should be allowed to continue once the IO request is made by the File System. This option is particularly useful in handling many terminals from one process but requires more careful use than 'wait IO'. The remaining sections will present detailed information on using the no-wait IO option with 'nobuf' files.

There are three basic elements involved in a nobuf transfer: the process stack data segment, File System tables and control blocks, and the device itself (with any special device dependent buffer area handled by the IO system, e.g. terminal buffers). These elements come into play whether the file was opened nobuf with the wait IO or the no-wait IO option.

When the user process requests the nobuf transfer, the information passed to the File System is as follows:

1. file number: a number (originally returned by FOPEN) that is an index into a File System table known as the AFT ('available' or 'active' file table), which is located in the stack data segment. The AFT entry corresponding to this file number allows the File System to locate the control blocks for this file.

2. source/target address: an address pointing to some location in the user portion of the stack data segment. Data will be transferred either from or to this location.

3. length: number of words or bytes to be transferred.

The File System locates the associated control blocks, does some bounds and other error checking; from the File System procedure FNOBUF it then calls the IO system, requesting that the transfer be done. This call is made to a procedure called ATTACHIO, which can be thought of as the door to the IO system. Some of the information passed to it is the device number, the starting sector address (if the device is a disc), the transfer length, the stack target (or source) address, type of transfer (read or write), and whether or not to make the process wait until the IO completes. The target/source address is converted to an absolute memory address; thus, the process stack data segment must be and is frozen in memory until the IO completes so that the target/source location cannot change while the transfer is in progress.

The information given to the IO system is formatted as an IO request and queued in a table (IO Queue) to await service. An index (the 'IOQ index') identifies this particular request and is the mechanism by which the File System is informed of the status of the transfer.

If the File System has specified that the process should wait until the IO completes, the IO routine will cause the process to be suspended. This is exactly the case when the file has been opened nobuf with the wait IO option. When the IO completes, the process is reawakened and the information on the status of the

transfer is passed back to the File System and ultimately to the application program that requested the transfer.

However, if the file was opened nobuf with the no-wait IO option, the File System will request that the IO system NOT suspend the process once the IO request has been initiated. Thus, the IO system returns control to the File System once it has finished making the request; the File System returns to its caller. Notice that, at this point in time, nothing is known about the status of the transfer. The user process can continue execution while the IO is in progress. However, the process will not know whether the IO has completed and if it was successful until it explicitly calls the File System to find out this information. For example, requested input data may have already been transferred into the stack data segment, but until the process calls either one of two special File System procedures (IOWAIT, IODONTWAIT), it will not know if the transfer has successfully completed.

APPLICATION OF NO-WAIT IO

How can this no-wait IO mechanism be an advantage to an application programmer? The fact that the process does not suspend on each read/write means that one program can be used to handle concurrent access to many terminals where the order of IO completion is random. In ordinary circumstances, any process can open many terminals; however, once a read is issued against one of them, the process waits until the input completes. Because this usually depends on human interaction, the length of time varies widely. Thus, all the other terminals handled by this process cannot send input or receive output until it is their "turn". In no-wait IO mode, the process can issue a no-wait read to each terminal and then wait until one of them completes. It could also handle other processing tasks until it finds out that a request has completed.

Of course, the no-wait option can be used with devices other than terminals, except for serial disc. A disc or a tape file can be opened nobuf, no-wait IO; this may be very useful when also handling multiple terminals in no-wait IO mode. The application may wish to issue reads to several terminals, having requested selection from a "menu" of possible tasks. Before checking for completion of any terminal reads, it could issue no-wait reads or writes from disc or tape in anticipation of work to be requested at one (or more) of the terminals.

The no-wait option can allow the programmer to overlap operations to be handled by an application program. Handling several terminals from one process instead of from individual processes belonging to separate jobs and sessions will dramatically cut down on the number of extra data segments and other system resources used. However, there are several factors that require careful consideration when using the no-wait IO option. These are best brought out by examining the basic programming sequence that is used to apply the no-wait IO option in a multi-terminal application.

NO-WAIT IO PROGRAMMING SEQUENCE

The first step is to open the file(s) using the FOPEN intrinsic, either directly or indirectly. [For the purposes of this discussion, all the File System intrinsics will be referred to directly; however, it is assumed that the reader is aware that all subsystems ultimately use File System intrinsics; that high level syntax check for, say, the COBOL "OPEN" statement results in a call to a COBOL library routine that calls FOPEN. Thus, the application programmer can effect the File System specifications even when they are made indirectly; this is usually done most easily by the :FILE command.]

When a file is to be FOPEN'd nobuf with the no-wait IO option, the process calling FOPEN must be running in privileged mode. This can be accomplished in a variety of ways:

a. In SPL, a procedure can have the OPTION PRIVILEGED statement. An outer block can be designated to run in privileged mode via the statement $CONTROL PRIVILEGED. In both these cases, the module will be in privileged mode during its entire execution time (unless the GETUSERMODE intrinsic is called).

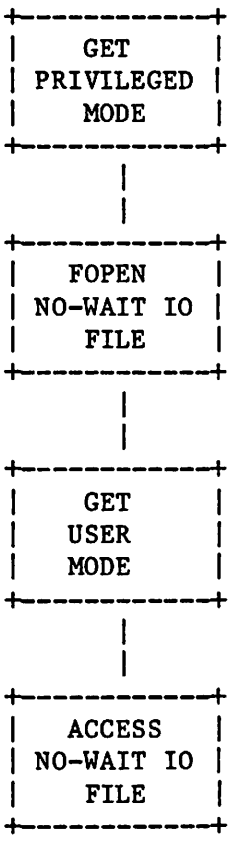
b. From SPL, FORTRAN, COBOL, BASIC, the GETPRIVMODE intrinsic can be called to place the current executing code into privilege mode. To return to user mode, call the GETUSERMODE intrinsic.

c. Regardless of how privileged mode is requested within the program, the program file MUST be given PM capability at PREP time. The program file's capability list is checked when attempting to grant privileged mode; if PM is not present, the request will not be granted. [Note that to :PREP a program file with CAP=PM the user (and thus the account) must have PM capability. To run a program file prep'd with CAP=PM, the program file must reside in a group (and thus an account) that has PM capability.]

FOPEN will check to see if the file is to be opened with the no-wait IO option. If this is the case, it checks to see if the caller was running in privileged mode; if not, an error will occur indicating illegal capability (FS error 2). If the caller is "privileged", FOPEN will set or disallow a number of associated options. Nobuf will be set; mutli-record mode, blocking*, and multi-access will be disallowed. [*Block factor is set to 1 for new files]. An error will result if the user tries to FOPEN with the no-wait IO option a file on a serial disc (serial disc error 11).

All the other work of FOPEN remains the same: it must get entry in the AFT for the file; it must build a physical access control block (PACB) for controlling access to the file; if the file resides on disc, the File System will need a file control block (FCB) for the file. If the file is new, space must be allocated (disc) or the device itself must be allocated. And so, FOPEN will complete all of its work just as it would if the file has been opened with the wait IO option.

Once FOPEN returns to the caller, there is no further requirement for the module to continue executing in privilege mode in order to access the file. Thus, the sequence of events used in the application program can be:



In order to access the no-wait IO file, the user will be using the normal transfer intrinsics such as FREAD, FWRITE, etc. and two other intrinsics specifically designed for use with no-wait IO files: IOWAIT, IODONTWAIT. To explain the basic sequence of events let's assume that what we want to accomplish is as

follows:

a. write a message out to the (no-wait) terminal

b. request input (no-wait) from the terminal.

The first step is to call FWRITE to send the message. Error checking after returning from FWRITE tells us whether or not the File Systen was able to request the transfer but it tells us nothing about the transfer itself since we did not wait for it to complete. There are several reasons why FWRITE might reject our call requesting a transfer, but the most common error programmers make with no-wait files is to request a transfer when a previous IO request is still pending. Thus, one of the first things done by FWRITE and the other transfer intrinsic is to find out if any no-wait IO is pending for this file. The File System does this by checking the fourth word of the four-word AFT entry for that file. If it is non-zero, it indicates that an IO request was queued but completion information has not yet been given to the application program. (File System Error 77, NO WAIT IO PENDING is set and an error condition is returned to the caller). Note that the IO may have completed: the File System has not yet been able to "clean up" the transaction. Since there is room in the AFT entry for only one such indicator, there can only be one no-wait IO request pending for any given file. All other requests will be rejected until the application program requests and receives completion information on this transfer. This is accomplished by calling IOWAIT or IODONTWAIT.

Let's assume that our FWRITE request was not rejected and that the IO request was set up and control returned to our program. At this point in time we have several options:

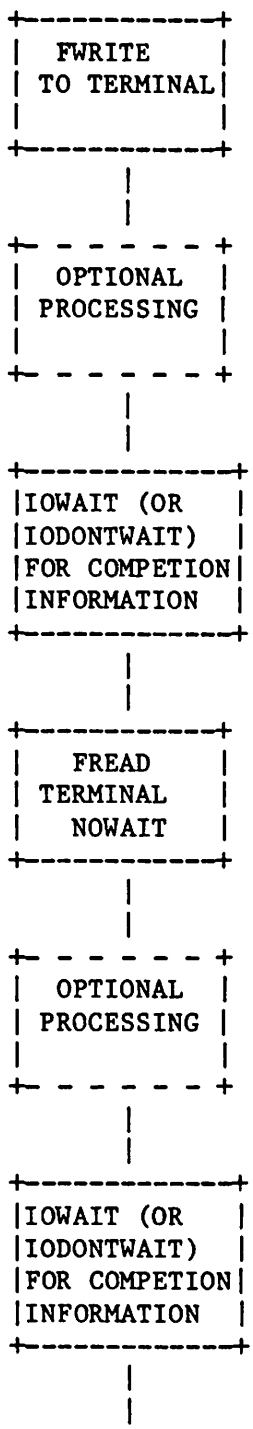
a. We can request completion information for that file transfer, waiting if it has not yet completed until it does complete (IOWAIT).

b. We can request completion information for that file transfer, returning to our program if it has not yet completed (IODONTWAIT). Realize, of course, this means we must again request completion information at a later time.

c. We can execute some other tasks, perhaps requesting transfer to/from other files (wait or no-wait). Once all these other tasks are complete, we will request completion information by either of the two methods above (IOWAIT or IODONTWAIT).

If there are several other terminals opened no-wait, we may want to issue no-wait FWRITE's to those terminals before requesting completion information on any of them. This would be option c. above. Most often, however, the program flow calls for a read following the write to a teminal. In this case the program would follow option a. by calling IOWAIT. IOWAIT will force the process to wait until the requested completion information is available. Thus, if our write transfer had not yet completed, our process would wait until it did.

Once we return from IOWAIT, we know the status of the transfer itself. If it was successful, we can go ahead and issue our FREAD to the terminals. Once again, the error information back from FREAD tells about the success of the request, not about the transfer itself. Note also, the only length FREAD can return is zero since the transfer has not yet taken place. After calling FREAD, we again have the option of doing some processing before checking to see if the read has completed. We may choose to issue no-wait reads to other terminals or process some other tasks. But just as when FWRITE was used, we must ultimately follow our call to FREAD with a IODONTWAIT to find out whether the read completed, if it did so successfully, and the length of the transfer. Thus, our sequence of events is:



[Note: Whenever IODONTWAIT is used, the process will regain control even if the IO has not yet completed. In this case, optional processing may be done, followed by subseqent calls to IOWAIT or IODONTWAIT until the desired IO completes.]

IO COMPLETION OPERATIONS

At this point, let's digress slightly to discuss how IOWAIT and IODONTWAIT actually work. In actual fact, IODONTWAIT is an entry point in the IOWAIT procedure. Their operation is identical except for the decision to force the process to wait if the IO has not yet completed: calling IOWAIT means the process will be suspended until completion; calling IODONTWAIT means it will not be suspended.

[Note: There are two reasons for including the following information in this paper. The first is to build some appreciation for the work the File System goes through on behalf of the user to handle no-wait IO. Secondly, it may be useful for an application programmer to realize that the search for IO completion is distributed over the range of files opened, based on the last file whose IO completion was processed by IOWAIT/IODONTWAIT.]

The caller has the option of requesting completion information for one specific file or for any no-wait file. This is done through the first parameter, the "filenum". If the parameter value supplied is not zero, then we are asking to know if IO has completed for that specific file. Of course, that number must correspond to an open file (that is, there must exist an AFT entry for it); if not, an error is returned (FS error 72 :Invalid file number). Also, the File System checks to see if IO was in fact pending for this file. If not, it sets File System error 79 (No No-Wait IO pending for Special File) and returns.

If an IO request was made but hasn't completed, the File System returns to the caller if IODONTWAIT was called. Both IOWAIT/IODONTWAIT are function procedures: a zero is the function value returned through IODONTWAIT when the IO for the specified file has not yet completed.

If IOWAIT was called, the process is suspended waiting for the IO to complete. When the IO completes (or if it had already completed), FNOBUF is called to process the transfer completion. The status is checked, error and condition codes returned as appropriate. When FNOBUF returns, IOWAIT/IODONTWAIT return the completion information to the caller, the function return value is set to the file number for the file whose IO completed.

There is additional work involved if the caller requests completion information for <u>any</u> file whose no-wait IO may have completed. The caller specifies this option by supplying a zero file number parameter or by not supplying that parameter at all. If this is the case, the File System procedure FINDWAITINGIO is called to determine if any no-wait IO has completed. It will

return to IOWAIT/IODONTWAIT a zero if no file's IO has completed or the file number of the file whose IO has completed. From that point, the File System processing proceeds as described above (specific file option). Let's consider what FINDWAITINGIO must do on behalf of the user to determine which, if any, file has completed.

Recall that in the stack data segment below DL (limit of user portion), there is a system area (called the PCBX, process control block extension) divided into three basic parts (PXGLOB, PXFIXED, PXFILE). The PXFILE area is used by the File System; it contains the AFT and some control blocks. One of the locations in this PXFILE area is set aside to contain the file number corresponding to the last file on which no-wait IO completion information was processed. One of the first things FINDWAITINGIO does is to pick up this file number and use it to determine where to begin looking in the AFT for IO completions. In essence, this file number tells the File System where it left off so it will begin with the entry for the next file in the AFT. Thus, it essentially adds one to the file number that it picked up out of the above-mentioned PXFILE location and proceeds to step through the AFT in a circular fashion until it winds up back where it "left off". Thus, for each file number, it

a. locates the AFT entry

b. checks to see if IO is pending for that file (fourth word is non-zero)

    1. if no IO is pending, go on to next file

    2. if IO is pending, check for completion

        a) if complete, can return information

        b) if not complete, put on top of stack the AFT entry number (file number) and the index corresponding to the pending IO request.

If the File System finishes this loop without finding any completed IO, there may be a list of stacked file numbers/IOQX's indicating there are pending IO's we can wait for. If not, an error is returned. Even if there are pending IO's, FINDWAITINGIO will return a zero if the option was "don't wait".

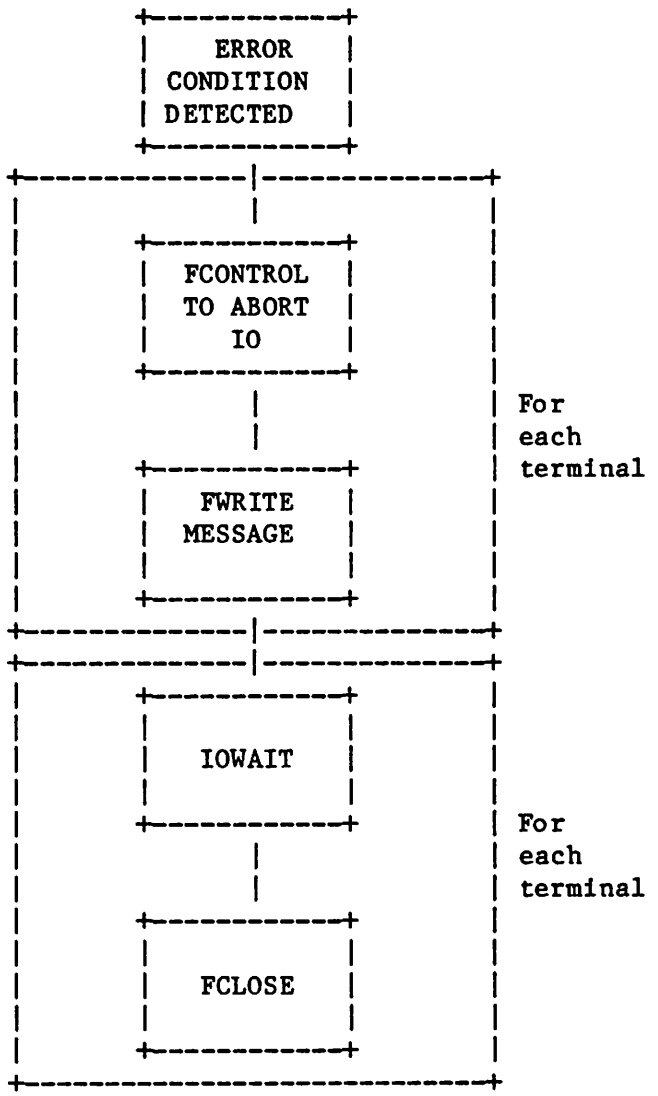
However, if the wait option was requested (IOWAIT), then the process will be suspended until some IO completes. Upon awakening, FINDWAITINGIO does not have to go through the AFT again to find which file's IO completed: it starts at the top of the stack and goes back down through its list of AFT (file) numbers/IOQ indexes checking each one to see if the IO completed.

As soon as it finds one that has completed, it returns that file number and exits back to the caller. If none in the list completed, then the process was awakened for some unknown reason. At any rate, FINDWAITINGIO will go back and recheck the AFT, rebuilding the stack list, and possibly waiting again. FINDWAITINGIO will return when it finally finds an IO completion for a no-wait file.

MISCELLANEOUS FILE OPERATIONS

Getting back to our discussion of file access, we have already covered the basic elements required to access the no-wait file. There are still a few points to mention with respect to file control and closing. With respect to the latter, the point to note is that a file with pending no-wait IO cannot be successfully FCLOSE'd by the user. IOWAIT, IODONTWAIT, or FCONTROL (see below) must be called in order to 'clean up' pending IO.

File control is done mainly through the FCONTROL intrinsic. Control codes 0 to 41 result in FCONTROL calling the IO system (ATTACHIO) with the request to "block" the process. This means that, even though the file is opened no-wait, the process will wait until the IO control function completes. Another point to note is that the special FCONTROL code (43) is provided specifically for aborting pending no-wait IO. This is particularly useful when the application program has to deal with error conditions. Suppose twenty-five terminals are opened no-wait by an application program and reads are pending at each. One of the reads completes and for some reason the program discovers, much to its dismay, that it must close down the entire application. If it wants to do this gracefully, it will probably want to write a message to the other terminals before closing them: this can't be done until the pending IO is cleaned up. Even if it doesn't want to send a message, FCLOSE will fail as long as IO is pending. FCONTROL (code 43) allows the program to abort pending IO by file number. The sequence of events might go something like this:



Some comments/cautions on the features of FCONTROL: if a read pending at a terminal is aborted while someone was entering data, no indication is returned to the program that input was in progress. This is probably not a concern if the IO was aborted because the application must be terminated; however, it is worth consideration when the IO is aborted for some other reason. Even if the IO has already completed, the application program will not know it (in addition to requesting the removal of the actual IO request, FCONTROL has to zero out the index kept in the file's AFT entry corresponding to the request).

It is probably worth reminding the reader that although we only mentioned FREAD and FWRITE, it is possible to use FREADDIR and FWRITEDIR with no-wait disc files.

NO-WAIT IO DESIGN CONSIDERATIONS

We will end our discussion of nobuf/no-wait IO by covering some considerations that the programmer should keep in mind when designing an application that accesses files with the no-wait option. [The points are made with no-wait terminal applications in mind; the reader is asked to apply them as needed to other types of no-wait file applications.]

1. One stack buffer per no-wait file will be required when any one or more IO requests may complete before the last one is completely processed by the application. In other words, if twenty-five terminals have no-wait reads pending, there should be a separate place in the stack for input from each of the terminals because there's no way to prevent one of the reads from completing once the IO is requested. If the program is still processing data in an input buffer, there should be no chance that this data could be overlayed by a subsequent IO completion. There should be twenty-five areas in the stack, each as big as the maximum amount to be read at one time. The address of each area is supplied to FREAD and is converted to a memory address when the IO request is set up. Maximum stack size will be a consideration if the buffer areas have to be quite large. If one program cannot handle all the terminals because of stack segment limitations, an alternative would be to have the program handle the maximum it can. A "driver" program could create and activate the program as many times as necessary until all terminals are handled. Thus perhaps one process can't handle all twenty-five, but two could (one handling 15, the other 10).

2. If an application has opened $STDIN as a no-wait file, the File System will not be able to check for logical end-of-file (:for $STDIN, :EOD for STDINX) unless the user passes the same input buffer address to IOWAIT (or IODONTWAIT) as it did to FREAD when it initially requested the input. The reason for this is that the File System must actually look in the buffer for the presence of a: (:EOD). If the buffer address isn't supplied, the File System can't look.

3. There is no reason why a program can't do multiple opens on a file. Therefore, an application may find it advantageous to open a terminal in no-wait mode for input and again in wait mode for output. This would allow the programmer to eliminate the the call to IOWAIT following each FWRITE. This is useful when the sequence of events calls for the process to always have to wait until a write to the terminal completes. The reads, on the other hand, are handled no-wait so that the application can handle other processing while terminal input is being entered.

4. It may be useful for an application to put a time limit on the no-wait read pending at a terminal. FCONTROL (option 4) allows the programmer to do this.

5. Extreme care must be taken to insure that any buffer area specified as the target (or source) of a no-wait transfer remains a valid buffer area until the transfer completes. Consider the following:

   Suppose an application program calls FREAD requesting that data be read no-wait into a buffer area that has been allocated in the dynamic portion of the stack. [The buffer is nothing more than an array local to a procedure (i.e. subroutine in FORTRAN, dynamic subprogram in COBOL).] Remember that the transfer is a nobuf transfer: it will go direct to the stack. FREAD passes this buffer area address along and it is converted to an absolute memory address (the stack data segment is frozen in memory). That absolute address is where the transferred data will be put - regardless of anything else that may have happened (other than an abort IO, of course).

   Now let's suppose that before the IO completed, the application (accidently or on purpose) exits from the procedure. What happens to local arrays? They are no longer valid and the stack space is reusable by calls to other procedures. It could very easily happen that the program calls some MPE (or subsystem) procedure which now uses as part of its local storage the very area designated to receive the input data. What if the procedure called is FOPEN or some other routine that deals with system table/directories, etc? When the input completes, the data will be transferred right on top of the procedure's local storage. It might proceed to update tables using totally erroneous data. It may indeed be quite some time before it (or some other part of MPE) discovers things are not right. However, sooner or later the result will almost certainly be a system failure of one sort or another. If you're lucky, only tables refreshed by INITIAL were corrupted. But it could be something like the system directory, in which case recovery will take much longer. The no-wait IO option requires PM because system managers must have a way to make sure that great care is taken when a no-wait IO application is developed.

   [It is not my intent to make no-wait IO sound so frightening that no one uses it. I do believe, however, that appreciation for its power must be matched by respect for its having potential for serious consequences if handled carelessly or without sufficient information.]

CONCLUSION

Nobuf/no-wait IO is a powerful tool, very useful in multiple-terminal applications. It is quite easy to use (although it must be used carefully) and can be a performance alternative to the typical one-process-per-terminal approach to handling terminals. It can be used with any language provided the IOWAIT/IODONTWAIT procedures can be used to obtain completion information. An example of a program handling no-wait IO is given in the MPE Intrinsics Manual.