Async I/O About using asynchronous I/O operations

One of the advantages of using the low-level PBxxx functions is that nearly all of them can be run asynchronously; that is, you can start the I/O operation going and continue doing other work simultaneously.

Most PBxxx calls require two parameters; e.g.:

```
        OSErr PBFunction(pb, async );
```

In most applications, you will use FALSE as the *async* parameter. In this case, the File Manager automatically sets the ioCompletion field of the parameter block to 0 and the call is executed synchronously - control remains with the File Manager until the I/O is fully completed.

Although not needed by most programs, asynchronous I/O can speed up some operations. Unlike timeslicing (which simply apportions the CPU cycles among several tasks), asynchronous I/O time savings are real: normally, after initiating an I/O operation, the CPU sits idle until the disk controller issues an interrupt. By using that wasted time, you can improve performance. Some uses of asynchronous I/O include:

- When a user closes a file window: open the file, write the data and close the file asynchronously. Then immediately call the Window Manager and Quickdraw functions to update the screen, menus, etc. The screen gets updated in the time normally lost awaiting the disk controller interrupt.
- Data communications programs can be made more efficient if you read asynchronously while displaying the most recent character (or packet) that was read.
- Call **PBFlushVol** asynchronously to ensure volume integrity without being penalized by the I/O interval.
- Overlapping I/O between two devices (as when copying a file from one drive to another) can be significantly faster than reading the entire file before writing the duplicate.

To execute a File Manager function asynchronously, use TRUE for the *async* parameter. Before making the call, set the ioCompletion to the address of the function you want to in control as soon as the I/O is finished (such "completion routines" are described below).

Completion Polling

Optionally, you can set ioCompletion to 0. In that case, the File Manager immediately sets ioResult to 1, begins the I/O, and when the operation is finished, sets ioResult to an error code (0=no error). Thus, you can poll ioResult to find when the I/O is done. For instance:

```
IOParam    pb;

pb.ioRefNum = theFile;
pb.ioBuffer = myData;
pb.ioReqCount = myDataLen;
pb.ioPosMode = fsAtMark;
PBRead( &pb, TRUE );   /* asynchronous request */
```

```
while( pb.ioResult == 1 ) {   /* I/O still continues */
    /*. . . update the screen, print something, etc . . .*/

}
/*. . . I/O is completed . . .*/
if ( pb.ioResult != 0) { /* . . . some error occurred . . .  */ }
```

The File Manager places all parameter blocks into an I/O request queue, processes them in sequential order, and dequeues each request as soon as it is finished (see **GetFSQHdr**). The difference between synchronous and async I/O is that with the former, the system does the polling for you; it just waits around until the device driver indicates completion before returning control to the caller.

It is important to remember that each async request **must have its own parameter block** (with synchronous I/O, it is convenient to use and reuse the same physical area of memory on each call). Thus, you will probably need to allocate a series of parameter blocks, one for each operation that will be in the I/O queue at the same time.

Completion Routines

An I/O completion routine is a piece of code, normally written in assembly language. It is called immediately upon completion of an I/O request. Several important notes:

- **Don't use any functions that might move or purge memory.** Specifically, you can't use **DisposHandle** to deallocate the request's parameter block.
- Save and restore all **CPU registers** except A0, A1, and D0-D2.
- **Register A5 may be invalid**. A5 is critical to applications since it normally points to the base of the application global variable area. Use **SetA5** before accessing these globals and use **SetCurrentA5** before exiting the completion routine.
- Older versions of THINK Reference used to quote DTS (Apple **D**eveloper **T**echnical **S**upport) as saying

  "Interrupts may be disabled when you get control so **don't start anything that will take a long time** (you might lose data coming in on a serial line, for instance). Alas, this also means that you should NOT initiate another I/O request (a major drawback)."

  This is no longer the case. DTS now recommends that you DO use completion routines to "chain" I/O requests if possible. That is, if you need to make a number of I/O requests for a driver that supports async. I/O, the best way to do it is to have a completion routine that posts another async. request to the driver. The only kind of I/O request that should not be made from a completion routine is a synchronous one. Since some drivers do not support asynchronous I/O, the entire issue of async I/O might be moot, but you should investigate the driver you are calling, because the run time improvements can be dramatic.

  If you really want to do polling waiting for the flag to change, you must be able to process other events as well.

Given these restrictions, there appears to be precious little you can accomplish with a completion routine that you can't get done by simply polling ioResult.