Fast AmigaDOS I/O

by Martin Taillefer

Reading and writing data is crucial to most applications and is, in many cases, a major bottleneck. Using the Amiga's sophisticated file system architecture can help reduce, and sometimes eliminate, the time spent waiting for I/O to complete. This article presents six small routines that can greatly improve an application's I/O performance.

Typically, an application processes a file in the following manner:

Step 1: Open the file.
Step 2: Read some data (with the DOS library's Read() function).
Step 3: Process that data.
Step 4: Repeat steps 2 and 3 until the application is finished processing the file.
Step 5: Close file.

This sequence of steps is effective, but it does have a potential bottleneck. Whenever the application reads some data using the DOS Read() function, the Amiga has to put that task to sleep and ask the file system to fetch the data. The file system then starts up the disk hardware and reads the data. After the file system finishes reading the data, the operating system wakes up the application.

The problem is step 2. While the file system is busy reading data from the disk, the application is idle, waiting for the DOS I/O in Read() to complete. A more sophisticated application would initiate an *asynchronous* read, allowing the application to continue to do some other important chore while the file system is busy reading. If all goes well, the file system will be finished with the asynchronous read by the time the application is finished with its chore, so the application will not have to wait for any DOS I/O to complete before the application can access data.

Using the routines presented in this article, an application processes a file in the following manner:

Step 1: Open the file with OpenAsync(). This function opens the file and, if the file is opened for reading, OpenAsync() asks the file system to start reading data, asynchronously.
Step 2: Read some data with ReadAsync(). If the asynchronous read request that OpenAsync() sent has not completed, ReadAsync() will put the application to sleep until that request returns. Ideally, the read will have returned, so the application won't have to wait. ReadAsync() will also initiate a new asynchronous read so new file data is ready when the application needs it.
Step 3: Process the file data.
Step 4: Repeat steps 2 and 3 until the application processes all its file data.
Step 5: Close the file with CloseAsync().

Immediately after opening the file, OpenAsync() sends a request to the file system to get it reading data in the background. If all goes well, by the time the application gets around to reading the first byte of data, the file system has already copied the data into memory. That means the application doesn't need to wait and can immediately start processing the data. As soon as the application starts processing data from the file using ReadAsync(), ReadAsync() sends out a second request to the file system to fill up a second buffer. Once the application is done processing the first buffer, it starts processing the second one. When this happens, the file system starts filling up the first buffer again with new data. This process continues until the application has read all of its data. This technique is known as ‘‘double-buffered asynchronous I/O’’.

The set of functions presented below offer high-performance I/O using the technique described above. The interface is very similar to standard AmigaDOS files. These routines enable full asynchronous read/write of any file.

These functions are especially useful on an Amiga with a DMA (Direct Memory Access) hard drive. DMA makes it possible to transfer data to memory *at the same time* the CPU is busy executing a task's instructions. A DMA data transfer is truly parallel, so, under normal conditions, the CPU is operating at full speed, unaffected by the DMA transfer. This parallelism is what makes the set of accompanying routines so efficient. They exploit the fact that the Amiga can transfer an application's data while the application is busy processing other data.

Although these asynchronous routines make disk I/O much faster, they do have an important limitation. The routines do not support seeking into a file.