Files Chapter Seven

7.1 Chapter Overview

In this chapter you will learn about the *file* persistent data type. In most assembly languages, file I/O is a major headache. Not so in HLA with the HLA Standard Library. File I/O is no more difficult than writing data to the standard output device or reading data from the standard input device. In this chapter you will learn how to create and manipulate sequential and random-access files.

7.2 File Organization

A file is a collection of data that the system maintains in persistent storage. Persistent means that the storage is non-volatile – that is, the system maintains the data even after the program terminates; indeed, even if you shut off system power. For this reason, plus the fact that different programs can access the data in a file, applications typically use files to maintain data across executions of the application and to share data with other applications.

The operating system typically saves file data on a disk drive or some other form of secondary storage device. As you may recall from the chapter on the memory hierarchy (see "The Memory Hierarchy" on page 303), secondary storage (disk drives) is much slower than main memory. Therefore, you generally do not store data that a program commonly accesses in files during program execution unless that data is far too large to fit into main memory (e.g., a large database).

Under Linux and Windows, a standard file is simply a stream of bytes that the operating system does not interpret in any way. It is the responsibility of the application to interpret this information, much the same as it is your application's responsibility to interpret data in memory. The stream of bytes in a file could be a sequence of ASCII characters (e.g., a text file) or they could be pixel values that form a 24-bit color photograph.

Files generally take one of two different forms: *sequential files* or *random access files*. Sequential files are great for data you read or write all at once; random access files work best for data you read and write in pieces (or rewrite, as the case may be). For example, a typical text file (like an HLA source file) is usually a sequential file. Usually your text editor will read or write the entire file at once. Similarly, the HLA compiler will read the data from the file in a sequential fashion without skipping around in the file. A database file, on the other hand, requires random access since the application can read data from anywhere in the file in response to a query.

7.2.1 Files as Lists of Records

A good view of a file is as a list of records. That is, the file is broken down into a sequential string of records that share a common structure. A list is simply an open-ended single dimensional array of items, so we can view a file as an array of records. As such, we can index into the file and select record number zero, record number one, record number two, etc. Using common file access operations, it is quite possible to skip around to different records in a file. Under Windows and Linux, the principle difference between a sequential file and a random access file is the organization of the records and how easy it is to locate a specific record within the file. In this section we'll take a look at the issues that differentiate these two types of files.

The easiest file organization to understand is the random access file. A random access file is a list of records whose lengths are all identical (i.e., random access files require fixed length records). If the record length is *n* bytes, then the first record appears at byte offset zero in the file, the second record appears at byte offset *n* in the file, the third record appears at byte offset *n**2* in the file, etc. This organization is virtually identical to that of an array of records in main memory; you use the same computation to locate an "ele-

ment" of this list in the file as you would use to locate an element of an array in memory; the only difference is that a file doesn't have a "base address" in memory, you simply compute the zero-based offset of the record in the file. This calculation is quite simple, and using some file I/O functions you will learn about a little later, you can quickly locate and manipulate any record in a random access file.

Sequential files also consist of a list of records. However, these records do not all have to be the same length[1]. If a sequential file does not use fixed length records then we say that the file uses variable-length records. If a sequential file uses variable-length records, then the file must contain some kind of marker or other mechanism to separate the records in the file. Typical sequential files use one of two mechanisms: a length prefix or some special terminating value. These two schemes should sound quite familiar to those who have read the chapter on strings. Character strings use a similar scheme to determine the bounds of a string in memory.

A text file is the best example of a sequential file that uses variable-length records. Text files use a special marker at the end of each record to delineate the records. In a text file, a record corresponds to a single line of text. Under Windows, the line feed character marks the end of each record. Other operating systems may use a different sequence; e.g., Windows uses a carriage return/line feed sequence while the Mac OS uses a single carriage return. Since we're working with Windows here, we'll adopt the line feed end of line marker.

Accessing records in a file containing variable-length records is problematic. Unless you have an array of offsets to each record in a variable-length file, the only practical way to locate record *n* in a file is to read the first *n-1* records. This is why variable-length files are sequential-access – you have the read the file sequentially from the start in order to locate a specific record in the file. This will be much slower than accessing the file in a random access fashion. Generally, you would not use a variable-length record organization for files you need to access in a random fashion.

At first blush it would seem that fixed-length random access files offer all the advantages here. After all, you can access records in a file with fixed-length records much more rapidly than files using the variable-length record organization. However, there is a cost to this: your fixed-length records have to be large enough to hold the largest possible data object you want to store in a record. To store a sequence of lines in a text file, for example, your record sizes would have to be large enough to hold the longest possible input line. This could be quite large (for example, HLA allows lines up to 256 characters). Each record in the file will consume this many bytes even if the record uses substantially less data. For example, an empty line only requires one or a single byte (for the line feed character). If your record size is 256 bytes, then you're wasting 255 or 255 bytes for that blank line in your file. If the average line length is around 60 characters, then each line wastes an average of about 200 characters. This problem, known as *internal fragmentation*, can waste a tremendous amount of space on your disk, especially as your files get larger or you create lots of files. File organizations that use variable-length records generally don't suffer from this problem.

7.2.2 Binary vs. Text Files

Another important thing to realize about files is that they don't all contain human readable text. Object and executable files are good examples of files that contain binary information rather than text. A text file is a very special kind of variable-length sequential file that uses special end of line markers (line feeds) at the end of each record (line) in the file. Binary files are everything else.

Binary files are often more compact than text files and they are usually more efficient to access. Consider a text file that contains the following set of two-byte integer values:

```
1234
543
3645
32000
```

1. There is nothing preventing a sequential file from using fixed length records. However, they don't require fixed length records.


```
1
87
0
```

As a text file, this file consumes at least 27 bytes (assuming a single byte line feed at the end of each line). However, were we to store the data in a fixed-record length binary file, with two bytes per integer value, this file would only consume 14 bytes – half the space. Furthermore, since the file now uses fixed-length records (two bytes per record) we can efficiently access it in a random fashion. Finally, there is one additional, though hidden, efficiency aspect to the binary format: when a program reads and writes binary data it doesn't have to convert between the binary and string formats. This is an expensive process (with respect to computer time). If a human being isn't going to read this file with a separate program (like a text editor) then converting to and from text format on every I/O operation is a wasted effort.

Consider the following HLA record type:

```
type
    person:
        record
            name:string;
            age:int16;
            ssn:char[11];
            salary:real64;
        endrecord;
```

If we were to write this record as text to a text file, a typical record would take the following form (<nl> indicates the end of line marker, a line feed or line feed):

```
Hyde, Randall<nl>
45<nl>
555-55-5555<nl>
123456.78<nl>
```

Presumably, the next *person* record in the file would begin with the next line of text in the text file.

The binary version of this file (using a fixed length record, reserving 64 bytes for the *name* string) would look, schematically, like the following:

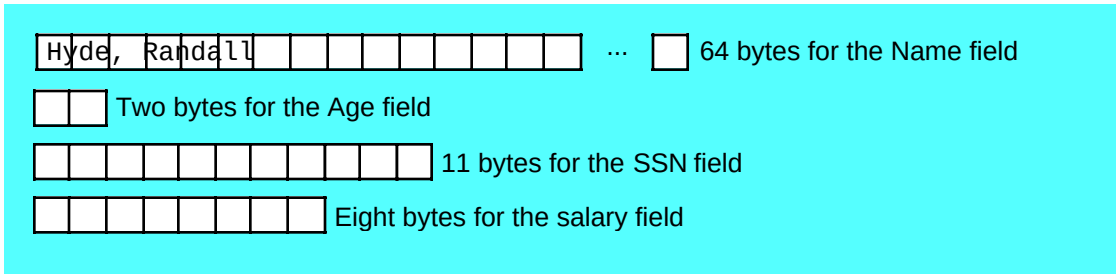

Figure 7.1 Fixed-lengthFormat for Person Record

Don't get the impression that binary files must use fixed length record sizes. We could create a variable-length version of this record by using a zero byte to terminate the string, as follows:

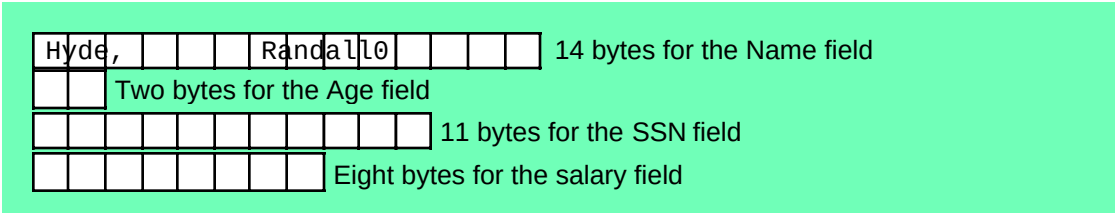

Figure 7.2 Variable-length Format for Person Record

In this particular record format the *age* field starts at offset 14 in the record (since the name field and the "end of field" marker [the zero byte] consume 14 bytes). If a different name were chosen, then the *age* field would begin at a different offset in the record. In order to locate the age, ssn, and salary fields of this record, the program would have to scan past the name and find the zero terminating byte. The remaining fields would follow at fixed offsets from the zero terminating byte. As you can see, it's a bit more work to process this variable-length record than the fixed-length record. Once again, this demonstrates the performance difference between random access (fixed-length) and sequential access (variable length, in this case) files.

Although binary files are often more compact and more efficient to access, they do have their drawbacks. In particular, only applications that are aware of the binary file's record format can easily access the file. If you're handed an arbitrary binary file and asked to decipher its contents, this could be very difficult. Text files, on the other hand, can be read by just about any text editor or filter program out there. Hence, your data files will be more interchangeable with other programs if you use text files. Furthermore, it is easier to debug the output of your programs if they produce text files since you can load a text file into the same editor you use to edit your source files.

7.3 Sequential Files

Sequential files are perfect for three types of persistent data: ASCII text files, "memory dumps", and stream data. Since you're probably familiar with ASCII text files, we'll skip their discussion. The other two methods of writing sequential files deserve more explanation.

A "memory dump" is a file that consists of data you transfer from data structures in memory directly to a file. Although the term "memory dump" suggests that you sequentially transfer data from consecutive memory locations to the file, this isn't necessarily the case. Memory access can, an often does, occur in a random access fashion. However, once the application constructs a record to write to the file, it writes that record in a sequential fashion (i.e., each record is written in order to the file). A "memory dump" is what most applications do when you request that they save the program's current data to a file or read data from a file into application memory. When writing, they gather all the important data from memory and write it to the file in a sequential fashion; when reading (loading) data from a file, they read the data from the file in a sequential fashion and store the data into appropriate memory-based data structures. Generally, when loading or saving file data in this manner, the program opens a file, reads/writes data from/to the file, and then it closes the file. Very little processing takes place during the data transfer and the application does not leave the file open for any length of time beyond what is necessary to read or write the file's data.

Stream data on input is like data coming from a keyboard. The program reads the data at various points in the application where it needs new input to continue. Similarly, stream data on output is like a write to the console device. The application writes data to the file at various points in the program after important computations have taken place and the program wishes to report the results of the calculation. Note that when reading data from a sequential file, once the program reads a particular piece of data, that data is no longer available in future reads (unless, of course, the program closes and reopens the file). When writing data to a

sequential file, once data is written, it becomes a permanent part of the output file. When processing this kind of data the program typically opens a file and then continues execution. As program execution continues, the application can read or write data in the file. At some point, typically towards the end of the application's execution, the program closes the file and commits the data to disk.

Although disk drives are generally thought of as random access devices, the truth is that they are only pseudo-random access; in fact, they perform much better when writing data sequentially on the disk surface. Therefore, sequential access files tend to provide the highest performance (for sequential data) since they match the highest performance access mode of the disk drive.

Working with sequential files in HLA is very easy. In fact, you already know most of the functions you need in order to read or write sequential files. All that's left to learn is how to open and close files and perform some simple tests (like "have we reached the end of a file when reading data from the file?").

The file I/O functions are nearly identical to the *stdin* and *stdout* functions. Indeed, *stdin* and *stdout* are really nothing more than special file I/O functions that read data from the standard input device (a file) or write data to the standard output device (which is also a file). You use the file I/O functions in a manner analogous to *stdin* and *stdout* except you use the *fileio* prefix rather than stdin or stdout. For example, to write a string to an output file, you could use the *fileio.puts* function almost the same way you use the *stdout.puts* routine. Similarly, if you wanted to read a string from a file, you would use *fileio.gets*. The only real difference between these function calls and their *stdin* and *stdout* counterparts is that you must supply an extra parameter to tell the function what file to use for the transfer. This is a double word value known as the *file handle*. You'll see how to initialize this file handle in a moment, but assuming you have a dword variable that holds a file handle value, you can use calls like the following to read and write data to sequential files:

```
fileio.get( inputHandle, i, j, k ); // Reads i, j, k, from file inputHandle.
fileio.put( outputHandle, "I = ", i, "J = ", j, " K = ", k, nl );
```

Although this example only demonstrates the use of *get* and *put*, be aware that almost all of the *stdin* and *stdout* functions are available as *fileio* functions, as well (in fact, most of the *stdin* and *stdout* functions simply call the appropriate *fileio* function to do the real work).

There is, of course, the issue of this file handle variable. You're probably wondering what a file handle is and how you tell the *fileio* routines to work with data in a specific file on your disk. Well, the definition of the file handle object is the easiest to explain – it's just a dword variable that the operating system initializes and uses to keep track of your file. To declare a file handle, you'd just create a dword variable, e.g.,

```
static
    myFileHandle:dword;
```

You should never explicitly manipulate the value of a file handle variable. The operating system will initialize this variable for you (via some calls you'll see in a moment) and the OS expects you to leave this value alone as long as you're working with the file the OS associates with that handle. If you're curious, both Linux and Windows store small integer values into the handle variable. Internally, the OS uses this value as an index into an array that contains pertinent information about open files. If you mess with the file handle's value, you will confuse the OS greatly the next time you attempt to access the file. Moral of the story – leave this value alone while the file is open.

Before you can read or write a file you must open that file and associate a filename with it. The HLA Standard Library provides a couple of functions that provide this service: *fileio.open* and *fileio.openNew*. The *fileio.open* function opens an existing file for reading, writing, or both. Generally, you open sequential files for reading or writing, but not both (though there are some special cases where you can open a sequential file for reading and writing). The syntax for the call to this function is

```
fileio.open( "filename", access );
```

The first parameter is a string value that specifies the filename of the file to open. This can be a string constant, a register that contains the address of a string value, or a string variable. The second parameter is a constant that specifies how you want to open the file. You may use any of the three predefined constants for the second parameter:

```
fileio.r
```



```
fileio.w
fileio.rw
```

fileio.r obviously specifies that you want to open an existing file in order to read the data from that file; likewise, *fileio.w* says that you want to open an existing file and overwrite the data in that file. The *fileio.rw* option lets you open a file for both reading and writing.

The *fileio.open* routine, if successful, returns a *file handle* in the EAX register. Generally, you will want to save the return value into a double word variable for use by the other HLA *fileio* routines (i.e., the *MyFileHandle* variable in the earlier example).

If the OS cannot open the file, *fileio.open* will raise an *ex.FileOpenFailure* exception. This usually means that it could not find the specified file on the disk.

The *fileio.open* routine requires that the file exist on the disk or it will raise an exception. If you want to create a new file, that might not already exist, the *fileio.openNew* function will do the job for you. This function uses the following syntax:

```
fileio.openNew( "filename" );
```

Note that this call has only a single parameter, a string specifying the filename. When you open a file with *fileio.openNew*, the file is always opened for writing. If a file by the specified filename already exists, then this function will delete the existing file and the new data will be written over the top of the old file (*so be careful*!).

Like *fileio.open*, *fileio.openNew* returns a file handle in the EAX register if it successfully opens the file. You should save this value in a file handle variable. This function raises the *ex.FileOpenFailure* exception if it cannot open the file.

Once you open a sequential file with *fileio.open* or *fileio.openNew* and you save the file handle value away, you can begin reading data from an input file (*fileio.r*) or writing data to an output file (*fileio.w*). To do this, you would use functions like *fileio.put* as noted above.

When the file I/O is complete, you must close the file to commit the file data to the disk. You should always close all files you open as soon as you are through with them so that the program doesn't consume excess system resources. The syntax for *fileio.close* is very simple, it takes a single parameter, the file handle value returned by *fileio.open* or *fileio.openNew*:

```
fileio.close( file_handle );
```

If there is an error closing the file, *fileio.close* will raise the *ex.FileCloseError* exception. Note that Linux and Windows automatically close all open files when an application terminates; however, it is very bad programming style to depend on this feature. If the system crashes (or the user turns off the power) before the application terminates, file data may be lost. So you should always close your files as soon as you are done accessing the data in that file.

The last function of interest to us right now is the *fileio.eof* function. This function returns true (1) or false (0) in the AL register depending on whether the current file pointer is at the end of the file. Generally you would use this function when reading data from an input file to determine if there is more data to read from the file. You would not normally call this function for output files; it always returns false[2]. Since the fileio routines will raise an exception if the disk is full, there is no need to waste time checking for end of file (EOF) when writing data to a file. The syntax for *fileio.eof* is

```
fileio.eof( file_handle );
```

The following program example demonstrates a complete program that opens and writes a simple text file:

```
program SimpleFileOutput;
```

2. Actually, it will return true under Windows if the disk is full.


```
#include( "stdlib.hhf" )

static
    outputHandle:dword;

begin SimpleFileOutput;

    fileio.openNew( "myfile.txt" );
    mov( eax, outputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.put( outputHandle, (type uns32 ebx ), nl );

    endfor;
    fileio.close( outputHandle );

end SimpleFileOutput;
```

Program 7.1 A Simple File Output Program

The following sample program reads the data that Program 7.1 produces and writes the data to the standard output device:

```
program SimpleFileInput;
#include( "stdlib.hhf" )

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput;

    fileio.open( "myfile.txt", fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.get( inputHandle, u );
        stdout.put( "ebx=", ebx, " u=", u, nl );

    endfor;
    fileio.close( inputHandle );

end SimpleFileInput;
```

Program 7.2 A Sample File Input Program

There are a couple of interesting functions that you can use when working with sequential files. They are the following:


```
fileio.rewind( fileHandle );
fileio.append( fileHandle );
```

The *fileio.rewind* function resets the "file pointer" (the cursor into the file where the next read or write will take place) back to the beginning of the file. This name is a carry-over from the days of files on tape drives when the system would rewind the tape on the tape drive to move the read/write head back to the beginning of the file.

If you've opened a file for reading, then *fileio.rewind* lets you begin reading the file from the start (i.e., make a second pass over the data). If you've opened the file for writing, then *fileio.rewind* will cause future writes to overwrite the data you've previously written; you won't normally use this function with files you've opened only for writing. If you've opened the file for reading and writing (using the *fileio.rw* option) then you can write the data after you've first opened the file and then rewind the file and read the data you've written. The following is a modification to Program 7.2 that reads the data file twice. This program also demonstrates the use of *fileio.eof* to test for the end of the file (rather than just counting the records).

```
program SimpleFileInput2;
#include( "stdlib.hhf" )

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput2;

    fileio.open( "myfile.txt", fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.get( inputHandle, u );
        stdout.put( "ebx=", ebx, " u=", u, nl );

    endfor;
    stdout.newln();

    // Rewind the file and reread the data from the beginning.
    // This time, use fileio.eof() to determine when we've
    // reached the end of the file.

    fileio.rewind( inputHandle );
    while( fileio.eof( inputHandle ) = false ) do

        // Read and display the next item from the file:

        fileio.get( inputHandle, u );
        stdout.put( "u=", u, nl );

        // Note: after we read the last numeric value, there is still
        // a newline sequence left in the file, if we don't read the
        // newline sequence after each number then EOF will be false
        // at the start of the loop and we'll get an EOF exception
        // when we try to read the next value.  Calling fileio.ReadLn
        // "eats" the newline after each number and solves this problem.

        fileio.readLn( inputHandle );


    endwhile;
```



```
    fileio.close( inputHandle );

end SimpleFileInput2;
```

Program 7.3 Another Sample File Input Program

The *fileio.append* function moves the file pointer to the end of the file. This function is really only useful for files you've opened for writing (or reading and writing). After executing *fileio.append*, all data you write to the file will be written after the data that already exists in the file (i.e., you use this call to append data to the end of a file you've opened). The following program demonstrates how to use this program to append data to the file created by Program 7.1:

```
program AppendDemo;
#include( "stdlib.hhf" )

static
    fileHandle:dword;
    u:uns32;

begin AppendDemo;

    fileio.open( "myfile.txt", fileio.rw );
    mov( eax, fileHandle );
    fileio.append( eax );

    for( mov( 10, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // display all the data from the file, including the
    // new data we just wrote to it:

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        fileio.readLn( fileHandle );

    endwhile;
    fileio.close( fileHandle );

end AppendDemo;
```

Program 7.4 Demonstration of the fileio.Append Routine

Another function, similar to *fileio.eof*, that will prove useful when reading data from a file is the *fileio.eoln* function. This function returns true if the next character(s) to be read from the file are the end of line sequence (carriage return, linefeed, or the sequence of these two characters under Windows, just a line feed under Linux). This function returns true or false in the EAX register if it detects an end of line sequence. The calling sequence for this function is

```
fileio.eoln( fileHandle );
```

If *fileio.eoln* detects an end of line sequence, it will read those characters from the file (so the next read from the file will not read the end of line characters). If *fileio.eoln* does not detect the end of line sequence, it does not modify the file pointer position. The following sample program demonstrates the use of *fileio.eoln* in the AppendDemo program, replacing the call to *fileio.readLn* (since *fileio.eoln* reads the end of line sequence, there is no need for the call to *fileio.readLn*):

```
program EolnDemo;
#include( "stdlib.hhf" )

static
    fileHandle:dword;
    u:uns32;

begin EolnDemo;

    fileio.open( "myfile.txt", fileio.rw );
    mov( eax, fileHandle );
    fileio.append( eax );

    for( mov( 10, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // display all the data from the file, including the
    // new data we just wrote to it:

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        if( !fileio.eoln( fileHandle )) then

            stdout.put( "Hmmm, expected the end of the line", nl );

        endif;

    endwhile;
    fileio.close( fileHandle );

end EolnDemo;
```


Program 7.5 fileio.eoln Demonstration Program.

7.4 Random Access Files

The problem with sequential files is that they are, well, sequential. They are great for dumping and retrieving large blocks of data all at once, but they are not suitable for applications that need to read, write, and rewrite the same data in a file multiple times. In those situations random access files provide the only reasonable alternative.

Windows and Linux don't differentiate sequential and random access files anymore than the CPU differentiates byte and character values in memory; it's up to your application to treat the files as sequential or random access. As such, you use many of the same functions to manipulate random access files as you use to manipulate sequential access files; you just use them differently is all.

You still open files with *fileio.open* and *fileio.openNew*. Random access files are generally opened for reading or reading and writing. You rarely open a random access file as write-only since a program typically needs to read data if it's jumping around in the file.

You still close the files with *fileio.close*.

You can read and write the files with *fileio.get* and *fileio.put*, although you would not normally use these functions for random access file I/O because each record you read or write has to be exactly the same length and these functions aren't particularly suited for fixed-length record I/O. Most of the time you will use one of the following functions to read and write fixed-length data:

```
fileio.write( fileHandle, buffer, count );
fileio.read( fileHandle, buffer, count );
```

The *fileHandle* parameter is the usual file handle value (a dword variable). The *count* parameter is an uns32 object that specifies how many bytes to read or write. The *buffer* parameter must be an array object with at least *count* bytes. This parameter supplies the address of the first byte in memory where the I/O transfer will take place. These functions return the number of bytes read or written in the EAX register. For *fileio.read*, if the return value in EAX does not equal *count's* value, then you've reached the end of the file. For *fileio.write*, if EAX does not equal *count* then the disk is full.

Here is a typical call to the *fileio.read* function that will read a record from a file:

```
fileio.read( myHandle, myRecord, @size( myRecord ) );
```

If the return value in EAX does not equal @size(myRecord) and it does not equal zero (indicating end of file) then there is something seriously wrong with the file since the file should contain an integral number of records.

Writing data to a file with *fileio.write* uses a similar syntax to *fileio.read*.

You can use *fileio.read* and *fileio.write* to read and write data from/to a sequential file, just as you can use routines like *fileio.get* and *fileio.put* to read/write data from/to a random access file. You'd typically use these routines to read and write data from/to a binary sequential file.

The functions we've discussed to this point don't let you randomly access records in a file. If you call *fileio.read* several times in a row, the program will read those records sequentially from the text file. To do true random access I/O we need the ability to jump around in the file. Fortunately, the HLA Standard Library's file module provides several functions you can use to accomplish this.

The *fileio.position* function returns the current offset into the file in the EAX register. If you call this function immediately before reading or writing a record to a file, then this function will tell you the exact

position of that record. You can use this value to quickly locate that record for a future access. The calling sequence for this function is

```
fileio.position( fileHandle ); // Returns current file position in EAX.
```

The *fileio.seek* function repositions the file pointer to the offset you specify as a parameter. The following is the calling sequence for this function:

```
fileio.seek( fileHandle, offset ); // Repositions file to specified offset.
```

The function call above will reposition the file pointer to the byte offset specified by the *offset* parameter. If you feed this function the value returned by *fileio.position*, then the next read or write operation will access the record written (or read) immediately after the *fileio.position* call.

You can pass any arbitrary offset value as a parameter to the *fileio.seek* routine; this value does not have to be one that the *fileio.position* function returns. For random access file I/O you would normally compute this offset file by specifying the index of the record you wish to access multiplied by the size of the record. For example, the following code computes the byte offset of record *index* in the file, repositions the file pointer to that record, and then reads the record:

```
intmul( @size( myRecord ), index, ebx );
fileio.seek( fileHandle, ebx );
fileio.read( fileHandle, (type byte myRecord), @size( myRecord ) );
```

You can use essentially this same code sequence to select a specific record in the file for writing.

Note that it is not an error to seek beyond the current end of file and then write data. If you do this, the OS will automatically fill in the intervening records with uninitialized data. Generally, this isn't a great way to create files, but it is perfectly legal. On the other hand, be aware that if you do this by accident, you may wind up with garbage in the file and no error to indicate that this has happened.

The *fileio* module provides another routine for repositioning the file pointer: *fileio.rSeek*. This function's calling sequence is very similar to *fileio.seek*, it is

```
fileio.rSeek( fileHandle, offset );
```

The difference between this function and the regular *fileio.seek* function is that this function repositions the file pointer offset bytes from the end of the file (rather than offset bytes from the start of the file). The "r" in "rSeek" stands for "reverse" seek.

Repositioning the file pointer, especially if you reposition it a fair distance from its current location, can be a time-consuming process. If you reposition the file pointer and then attempt to read a record from the file, the system may need to reposition a disk arm (a very slow process) and wait for the data to rotate underneath the disk read/write head. This is why random access I/O is much less efficient than sequential I/O.

The following program demonstrates random access I/O by writing and reading a file of records:

```
program RandomAccessDemo;
#include( "stdlib.hhf" )

type
    fileRec:
        record
            x:int16;
            y:int16;
            magnitude:uns8;
        endrecord;

const

    // Some arbitrary data we can use to initialize the file:
```



```
    fileData:=
        [
            fileRec:[ 2000, 1, 1 ],
            fileRec:[ 1000, 10, 2 ],
            fileRec:[ 750, 100, 3 ],
            fileRec:[ 500, 500, 4 ],
            fileRec:[ 100, 1000, 5 ],
            fileRec:[ 62, 2000, 6 ],
            fileRec:[ 32, 2500, 7 ],
            fileRec:[ 10, 3000, 8 ]
        ];


static
    fileHandle:         dword;
    RecordFromFile:     fileRec;
    InitialFileData:    fileRec[ 8 ] := fileData;



begin RandomAccessDemo;

    fileio.openNew( "fileRec.bin" );
    mov( eax, fileHandle );

    // Okay, write the initial data to the file in a sequential fashion:

    for( mov( 0, ebx ); ebx < 8; inc( ebx )) do

        intmul( @size( fileRec ), ebx, ecx );   // Compute index into fileData
        fileio.write
        (
            fileHandle,
            (type byte InitialFileData[ecx]),
            @size( fileRec )
        );

    endfor;

    // Okay, now let's demonstrate a random access of this file
    // by reading the records from the file backwards.

    stdout.put( "Reading the records, backwards:" nl );
    for( mov( 7, ebx ); (type int32 ebx) >= 0; dec( ebx )) do

        intmul( @size( fileRec ), ebx, ecx );   // Compute file offset
        fileio.seek( fileHandle, ecx );
        fileio.read
        (
            fileHandle,
            (type byte RecordFromFile),
            @size( fileRec )
        );
        if( eax = @size( fileRec )) then

            stdout.put
            (
                "Read record #",
                (type uns32 ebx),
                ", values:" nl
                "  x: ", RecordFromFile.x, nl
```



```
                    "  y: ", RecordFromFile.y, nl
                    "  magnitude: ", RecordFromFile.magnitude, nl nl
                );

            else

                stdout.put( "Error reading record number ", (type uns32 ebx), nl );

            endif;

        endfor;
        fileio.close( fileHandle );

    end RandomAccessDemo;
```

Program 7.6 Random Access File I/O Example

7.5 ISAM (Indexed Sequential Access Method) Files

ISAM is a trick that attempts to allow random access to variable-length records in a sequential file. This is a technique employed by IBM on their mainframe data bases in the 1960's and 1970's. Back then, disk space was very precious (remember why we wound up with the Y2K problem?) and IBM's engineers did everything they could to save space. At that time disks held about five megabytes, or so, were the size of washing machines, and cost tens of thousands of dollars. You can appreciate why they wanted to make every byte count. Today, data base designers have disk drives with hundreds of gigabytes per drive and RAID[3] devices with dozens of these drives installed. They don't bother trying to conserve space at all ("Heck, I don't know how big the person's name can get, so I'll allocate 256 bytes for it!"). Nevertheless, even with large disk arrays, saving space is often a wise idea. Not everyone has a terabyte (1,000 gigabytes) at their disposal and a user of your application may not appreciate your decision to waste their disk space. Therefore, techniques like ISAM that can reduce disk storage requirements are still important today.

ISAM is actually a very simple concept. Somewhere, the program saves the offset to the start of every record in a file. Since offsets are four bytes long, an array of dwords will work quite nicely[4]. Generally, as you construct the file you fill in the list (array) of offsets and keep track of the number of records in the file. For example, if you were creating a text file and you wanted to be able to quickly locate any line in the file, you would save the offset into the file of each line you wrote to the file. The following code fragment shows how you could do this:

```
static
    outputLine: string;
    ISAMarray: dword[ 128*1024 ]; // allow up to 128K records.
            .
            .
            .
    mov( 0, ecx );         // Keep record count here.
    forever

        << create a line of text in "outputLine" >>

        fileio.position( fileHandle );
```

3. Redundant array of inexpensive disks. RAID is a mechanism for combining lots of cheap disk drives together to form the equivalent of a really large disk drive.

4. This assumes, of course, that your files have a maximum size of four gigabytes.


```
        mov( eax, ISAMarray[ecx*4] );  // Save away current record offset.
        fileio.put( fileHandle, outputLine, nl ); // Write the record.
        inc( ecx );  // Advance to next element of ISAMarray.

        << determine if we're done and BREAK if we are >>

    endfor;

    << At this point, ECX contains the number of records and >>
    << ISAMarray[0]..ISAMarray[ecx-1] contain the offsets to >>
    << each of the records in the file.                      >>
```

After building the file using the code above, you can quickly jump to an arbitrary line of text by fetching the index for that line from the *ISAMarray* list. The following code demonstrates how you could read line *recordNumber* from the file:

```
mov( recordNumber, ebx );
fileio.seek( fileHandle, ISAMarray[ ebx*4 ] );
fileio.a_gets( fileHandle, inputString );
```

As long as you've precalculated the *ISAMarray* list, accessing an arbitrary line in this text file is a trivial matter.

Of course, back in the days when IBM programmers were trying to squeeze every byte from their databases as possible so they would fit on a five megabyte disk drive, they didn't have 512 kilobytes of RAM to hold 128K entries in the *ISAMarray* list. Although a half a megabyte is no big deal today, there are a couple of reasons why keeping the *ISAMarray* list in a memory-based array might not be such a good idea. First, databases are much larger these days. Some databases have hundreds of millions of entries. While setting aside a half a megabyte for an ISAM table might not be a bad thing, few people are willing to set aside a half a gigabyte for this purpose. Even if your database isn't amazingly big, there is another reason why you might not want to keep your *ISAMarray* in main memory – it's the same reason you don't keep the file in memory – memory is volatile and the data is lost whenever the application quits or the user removes power from the system. The solution is exactly the same as for the file data: you store the *ISAMarray* data in its own file. A program that builds the ISAM table while writing the file is a simple modification to the previous ISAM generation program. The trick is to open two files concurrently and write the ISAM data to one file while you're writing the text to the other file:

```
static
    fileHandle: dword;     // file handle for the text file.
    outputLine: string;    // file handle for the ISAM file.
    CurrentOffset: dword;  // Holds the current offset into the text file.
           .
           .
           .
    forever

        << create a line of text in "outputLine" >>

        // Get the offset of the next record in the text file
        // and write this offset (sequentially) to the ISAM file.

        fileio.position( fileHandle );
        mov( eax, CurrentOffset );
        fileio.write( isamHandle, (type byte CurrentOffset), 4 );

        // Okay, write the actual text data to the text file:

        fileio.put( fileHandle, outputLine, nl ); // Write the record.

        << determine if we're done and BREAK if we are >>
```



```
    endfor;
```

If necessary, you can count the number of records as before. You might write this value to the first record of the ISAM file (since you know the first record of the text file is always at offset zero, you can use the first element of the ISAM list to hold the count of ISAM/text file records).

Since the ISAM file is just a sequence of four-byte integers, each record in the file (i.e., an integer) has the same length. Therefore, we can easily access any value in the ISAM file using the random access file I/O mechanism. In order to read a particular line of text from the text file, the first task is to read the offset from the ISAM file and then use that offset to read the desired line from the text file. The code to accomplish this is as follows:

```
    // Assume we want to read the line specified by the "lineNumber" variable.

    if( lineNumber <> 0 ) then

        // If not record number zero, then fetch the offset to the desired
        // line from the ISAM file:

        intmul( 4, lineNumber, eax );   // Compute the index into the ISAM file.
        fileio.seek( isamHandle, eax );
        fileio.read( isamHandle, (type byte CurrentOffset), 4 ); // Read offset

    else

        mov( 0, eax );  // Special case for record zero because the file
                        // contains the record count in this position.

    endif;
    fileio.seek( fileHandle, CurrentOffset ); // Set text file position.
    fileio.a_gets( fileHandle, inputLine );   // Read the line of text.
```

This operation runs at about half the speed of having the ISAM array in memory (since it takes four file accesses rather than two to read the line of text from the file), but the data is non-volatile and is not limited by the amount of available RAM.

If you decide to use a memory-based array for your ISAM table, it's still a good idea to keep that data in a file somewhere so you don't have to recompute it (by reading the entire file) every time your application starts. If the data is present in a file, all you've got to do is read that file data into your *ISAMarray* list. Assuming you've stored the number of records in element number zero of the ISAM array, you could use the following code to read your ISAM data into the *ISAMarray* variable:

```
static
    isamSize: uns32;
    isamHandle: dword;
    fileHandle: dword;
    ISAMarray: dword[ 128*1024 ];
        .
        .
        .
    // Read the first record of the ISAM file into the isamSize variable:

    fileio.read( isamHandle, (type byte isamSize), 4 );

    // Now read the remaining data from the ISAM file into the ISAMarray
    // variable:

    if( isamSize >= 128*1024 ) then

        raise( ex.ValueOutOfRange );
```



```
    endif;
    intmul( 4, isamSize, ecx );  // #records * 4 is number of bytes to read.
    fileio.read( isamHandle, (type byte ISAMarray), ecx );

    // At this point, ISAMarray[0]..ISAMarray[isamSize-1] contain the indexes
    // into the text file for each line of text.
```

7.6 Truncating a File

If you open an existing file (using *fileio.open*) for output and write data to that file, it overwrites the existing data from the start of the file. However, if the new data you write to the file is shorter than the data originally appearing in the file, the excess data from the original file, beyond the end of the new data you've written, will still appear at the end of the new data. Sometimes this might be desirable, but most of the time you'll want to delete the old data after writing the new data.

One way to delete the old data is to use the *fileio.openNew* function to open the file. The *fileio.openNew* function automatically deletes any existing file so only the data you write to the file will be present in the file. However, there may be times when you may want to read the old data first, rewind the file, and then overwrite the data. In this situation, you'll need a function that will *truncate* the old data at the end of the file after you've written the new data. The *fileio.truncate* function accomplishes this task. This function uses the following calling syntax:

```
fileio.truncate( fileHandle );
```

Note that this function does not close the file. You still have to call *fileio.close* to commit the data to the disk.

The following sample program demonstrates the use of the *fileio.truncate* function:

```
program TruncateDemo;
#include( "stdlib.hhf" )

static
    fileHandle:dword;
    u:uns32;

begin TruncateDemo;

    fileio.openNew( "myfile.txt" );
    mov( eax, fileHandle );
    for( mov( 0, ecx ); ecx < 20; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;


    // Okay, let's rewind to the beginning of the file and
    // rewrite the first ten lines and then truncate the
    // file at that point.

    fileio.rewind( fileHandle );
    for( mov( 0, ecx ); ecx < 10; inc( ecx )) do

        fileio.put( fileHandle, (type uns32 ecx), nl );
```



```
    endfor;
    fileio.truncate( fileHandle );


    // Rewind and display the file contents to ensure that
    // the file truncation has worked.

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle )) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        fileio.readLn( fileHandle );

    endwhile;
    fileio.close( fileHandle );

end TruncateDemo;
```

Program 7.7 Using fileio.truncate to Eliminate Old Data From a File

7.7 File Utility Routines

The following subsections describe *fileio* functions that manipulate files or return meta-information about files (e.g., the file size and attributes).

```
program CopyDemo;
#include( "stdlib.hhf" )

begin CopyDemo;

    // Make a copy of myfile.txt to itself to demonstrate
    // a true "failsIfExists" parameter.

    if( !fileio.copy( "myfile.txt", "myfile.txt", true )) then

        stdout.put( "Did not copy 'myfile.txt' over itself" nl );

    else

        stdout.put( "Whoa!  The failsIfExists parameter didn't work." nl );

    endif;

    // Okay, make a copy of the file to a different file, to verify
    // that this works properly:

    if( fileio.copy( "myfile.txt", "copyOfMyFile.txt", false )) then

        stdout.put( "Successfully copied the file" nl );

    else
```



```
        stdout.put( "Failed to copy the file (maybe it doesn't exist?)" nl );

    endif;

end CopyDemo;

program FileMoveDemo;
#include( "stdlib.hhf" )

begin FileMoveDemo;

    // Rename the "myfile.txt" file to the name "renamed.txt".

    if( !fileio.move( "myfile.txt", "renamed.txt" )) then

        stdout.put
        (
            "Could not rename 'myfile.txt' (maybe it doesn't exist?)" nl
        );

    else

        stdout.put( "Successfully renamed the file" nl );

    endif;


end FileMoveDemo;
```

7.7.1 Computing the File Size

Another useful function to have is one that computes the size of an existing file on the disk. The *fileio.size* function provides this capability. The calling sequences for this function are

```
fileio.size( filenameString );
fileio.size( fileHandle );
```

The first form above expects you to pass the filename as a string parameter. The second form expects a handle to a file you've opened with *fileio.open* or *fileio.openNew*. These two calls return the size of the file in EAX. If an error occurs, these functions return -1 ($FFFF_FFFF) in EAX. Note that the files must be less than four gigabytes in length when using this function (if you need to check the size of larger files, you will have to call the appropriate OS function rather than these functions; however, since files larger than four gigabytes are rather rare, you probably won't have to worry about this problem).

One interesting use for this function is to determine the number of records in a fixed-length-record random access file. By getting the size of the file and dividing by the size of a record, you can determine the number of records in the file.

Another use for this function is to allow you to determine the size of a (smaller) file, allocate sufficient storage to hold the entire file in memory (by using *malloc*), and then read the entire file into memory using the *fileio.read* function. This is generally the fastest way to read data from a file into memory.

Program 7.10 demonstrates the use of the two forms of the *fileio.size* function by displaying the size of the "myfile.txt" file created by other sample programs in this chapter.

```
program FileSizeDemo;
#include( "stdlib.hhf" )
```



```
static
    handle:dword;

begin FileSizeDemo;

    // Display the size of the "FileSizeDemo.hla" file:

    fileio.size( "FileSizeDemo.hla" );
    if( eax <> -1 ) then

        stdout.put( "Size of file: ", (type uns32 eax), nl );

    else

        stdout.put( "Error calculating file size" nl );

    endif;

    // Same thing, using the file handle as a parameter:

    fileio.open( "FileSizeDemo.hla", fileio.r );
    mov( eax, handle );
    fileio.size( handle );
    if( eax <> -1 ) then

        stdout.put( "Size of file(2): ", (type uns32 eax), nl );

    else

        stdout.put( "Error calculating file size" nl );

    endif;
    fileio.close( handle );



end FileSizeDemo;
```

Program 7.8 Sample Program That Demonstrates the fileio.size Function

7.7.2 Deleting Files

Another useful file utility function is the fileio.delete function. As its name suggests, this function deletes a file that you specify as the function's parameter. The calling sequence for this function is

```
fileio.delete( filenameToDelete );
```

The single parameter is a string containing the pathname of the file you wish to delete. This function returns true/false in the EAX register to denote success/failure.

Program 7.11 provides an example of the use of the *fileio.delete* function.

```
program DeleteFileDemo;
```



```
#include( "stdlib.hhf" )

static
    handle:dword;

begin DeleteFileDemo;

    // Delete the "myfile.txt" file:

    fileio.delete( "xyz" );
    if( eax ) then

        stdout.put( "Deleted the file", nl );

    else

        stdout.put( "Error deleting the file" nl );

    endif;


end DeleteFileDemo;
```

Program 7.9 Example Usage of the fileio.delete Procedure

7.8 Directory Operations

In addition to manipulating files, you can also manipulate directories with some of the *fileio* functions. The HLA Standard Library includes several functions that let you create and use subdirectories. These functions are *fileio.cd* (change directory), *fileio.gwd* (get working directory), and *fileio.mkdir* (make directory). Their calling sequences are

```
fileio.cd( pathnameString );
fileio.gwd( stringToHoldPathname );
fileio.mkdir( newDirectoryName );
```

The *fileio.cd* and *fileio.mkdir* functions return success or failure (true or false, respectively) in the EAX register. For the *fileio.gwd* function, the string parameter is a destination string where the system will store the pathname to the current directory. You must allocate sufficient storage for the string prior to passing the string to this function (260 characters[5] is a good default amount if you're unsure how long the pathname could be). If the actual pathname is too long to fit in the destination string you supply as a parameter, the *fileio.gwd* function will raise the *ex.StringOverflow* exception.

The *fileio.cd* function sets the current working directory to the pathname you specify. After calling this function, the OS will assume that all future "unadorned" file references (those without any "\" or "/""/" characters in the pathname) will default to the directory you specify as the *fileio.cd* parameter. Proper use of this function can help make your program much more convenient to use by your program's users since they won't have to enter full pathnames for every file they manipulate.

The *fileio.gwd* function lets you query the system to determine the current working directory. After a call to *fileio.cd*, the string that *fileio.gwd* returns should be the same as *fileio.cd's* parameter. Typically, you would use this function to keep track of the default directory when your program first starts running. You

5. This is the default MAX_PATH value in Windows. This is probably sufficient for most Linux applications, too.

program will exhibit good manners by switching back to this default directory when your program terminates.

The *fileio.mkdir* function lets your program create a new subdirectory. If your program creates data files and stores them in a default directory somewhere, it's good etiquette to let the user specify the subdirectory where your program should put these files. If you do this, you should give your users the option to create a new directory (in case they want the data placed in a brand-new directory). You can use *fileio.mkdir* for this purpose.

7.9 Putting It All Together

This chapter began with a discussion of the basic file operations. That section was rather short because you've already learned most of what you need to know about file I/O when learning the *stdout* and *stdin* functions. So the introductory material concentrated on a file general file concepts (like the differences between sequential and random access files and the differences between binary and text files). After teaching you the few extra routines you need in order to open and close files, the remainder of this chapter simply concentrated on providing a few examples (like ISAM) of file access and a discussion of the *fileio* routines available in the HLA Standard Library.

While this chapter demonstrates the mechanics of file I/O, how you efficiently use files is well beyond the scope of this chapter. In future volumes you will see how to search for data in files, sort data in files, and even create databases. So keep on reading if you're interested in more information about file operations.

