Working with the IFS in RPG IV

Scott Klement

Working with the IFS in RPG IV
by Scott Klement

This eBook is intended to help an experienced RPG IV programmer learn how to read, write and manipulate documents within the Integrated File System on an IBM iSeries/400 server.

It is assumed that the reader of this tutorial is already familiar with the RPG IV language, including prototypes sub-procedures and service programs.

Trademarks

Table of Contents

Chapter 1. Introduction to the IFS

The purpose of this book is to teach you how to work with stream files in the Integrated File System from an ILE RPG/400 (RPG IV) program.

1.1. What is the Integrated File System?

Traditionally, we've worked with a file system on OS/400 that was made up of libraries. Within each library are objects that are assigned a specific "object type" such as a file, a program or a command. Each object type has a strictly defined layout. Files, for example, contain members, which then contain records, which contain fields. Each of these pieces is given a strict definition of what it is, how it works, and how it can be used.

By contrast, other operating systems, such as UNIX, MS-DOS and Windows use file systems where each object is simply a collection of bytes. Applications can be written to write and read these bytes as data files, but they can also view them as programs, pictures, sounds, video files, or anything else that a programmer can dream up. In other words, their contents are not strictly defined by the operating system.

At some point in it's history it was decided that OS/400 should be extended to work with these "stream files."

Some problems needed to be solved in order to do this, however, because although these file systems are all similar, they are not exactly the same. MS-DOS filenames can be 8 characters long with a 3-character "extension", and cannot contain spaces in the filename. Windows extends the MS-DOS capability by adding the ability to have a much longer file name, plus they now allow spaces. And UNIX allows spaces and long filenames, and even makes the distinction between upper & lower case letters. In other words, in Windows "MyFile.txt" and "myfile.txt" would refer to the same file, but in UNIX they refer to two different files.

In order to make OS/400 capable of working with files and folders that adhere to all of these different rules, the Integrated File System (IFS) was born. In the IFS, many different file systems can be accessed using a common interface. Special directory names are used to denote which file system you're referring to. You can even define your own file system that uses your own user-defined rules if you wish! (But, we won't be covering that in this book.)

1.2. What is a stream file?

With the exception of the original "library" file system, most of the file systems in the IFS store their objects in the "loosely-defined" file structure that's common in Windows and UNIX environments. This type of object is known as a "stream file" because the data in it is thought of as "one continuous stream of bytes." In other words, there's a start of the file, and an end to the file, but nothing else is defined. A program can use this big, long string of bytes for any purpose that it likes.

1.3. What different file systems can I work with in RPG?

There are many file systems defined to work in the IFS. Here we will just list a few of them, so that you get the idea:

File system	Description	Works Like
/QSYS.LIB	The traditional "Library file system"	OS/400

File system	Description	Works Like
/QDLS	The "Document Library Services" (OfficeVision) file system	MS-DOS
/QOpenSys	The "Open Systems" file system	UNIX
/ ("root")	The "root" file system	Windows

Most of the file systems not mentioned here are for accessing data that is not stored on your AS/400's hard drive, but is accessed on optical media, or on another computer somewhere over your LAN. Rest assured that all of these file systems can be accessed from an RPG program using the same Integrated File System interfaces that we use in this book.

1.4. IFS information in the Information Center

IBM provides a series of Application Program Interfaces (APIs) which we will use to access the IFS. These APIs are designed to be compatible with those used in UNIX environments.

Unfortunately, since most UNIX programming is done in the C programming language, almost all of the documentation assumes that you are a C programmer.

Throughout this book, we will be looking at the C documentation, and translating it into RPG for our purposes. We will make our RPG implementation as much like the C implementation as possible, so that it will be relatively easy to use the IBM manuals to find the information you're looking for in the future.

Don't worry if you're not a C programmer! In this eBook, we will explain each API in RPG terms, plus sample programs will be provided that you can use as a guide.

For general information about the Integrated File System, such as the concepts behind it, follow these steps:

1. Open up your Information Center CD, or point your web browser at:
   http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
2. If asked, choose the area of the world you live in, the language that you speak, and the release of OS/400 that you need.
3. Using the navigator bar on the left-hand side of the screen, click "Database and File Systems", then "File Systems and Management", then "Integrated File Systems Concepts."

> **Tip:** If your web browser has trouble with the navigation bar, (and many do) you can get to the same place by clicking on the "Site Map" instead.

For reference information on the APIs themselves, follow these:

1. Open up your Information Center CD, or point your web browser at:
   http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm
2. If asked, choose the area of the world you live in, the language that you speak, and the release of OS/400 that you need.

3. Using the navigator bar on the left-hand side of the screen, click "Programming", then "CL and APIs", then "OS/400 APIs", then "APIs by category".

   Tip: If your web browser has trouble with the navigation bar, (and many do) you can get to the same place by clicking on the "Site Map" instead.

4. From the list of categories, click on "UNIX-Type", and then click on the "Integrated File System APIs" topic.

1.5. An IFS "Hello World" Application

I don't know about you, but I'm falling asleep reading this book. I'm not a historian, I'm a programmer! Lets go! Let's write some code, already!

Type the following code into a source member called CH1HELLO, and we'll try to make it work:

```
 ** HELLO WORLD in the IFS Example:
 **  (From Chapter 1)
 **
 **  To compile:
 **     CRTBNDRPG CH1HELLO SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 **

H DFTACTGRP(*NO) ACTGRP(*NEW)


 ** API call to open a stream file
 **
D open            PR            10I 0 extproc('open')
D   path                          *   value options(*string)
D   oflag                       10I 0 value
D   mode                        10U 0 value options(*nopass)
D   codepage                    10U 0 value options(*nopass)

D O_WRONLY        C                   2
D O_CREAT         C                   8
D O_TRUNC         C                   64

D RW              C                   6
D R               C                   4
D OWNER           C                   64
D GROUP           C                   8

 ** API call to write data to a stream file
 **
D write           PR            10I 0 extproc('write')
D   fildes                      10I 0 value
D   buf                           *   value
D   nbyte                       10U 0 value
```

```
 ** API call to close a stream file
 **
D close           PR            10I 0 extproc('close')
D   fildes                      10I 0 value


D fd              S             10I 0
D data            S             12A
D msg             S             52A

C* Create an empty file called 'helloworld':
c                   eval      fd = open('/helloworld':
c                                       O_CREAT+O_TRUNC+O_WRONLY:
c                                       RW*OWNER + RW*GROUP + R )
c                   if        fd < 0
c                   eval      Msg = 'open failed!'
c                   dsply                   msg
c                   eval      *inlr = *on
c                   return
c                   endif

C* Write 'Hello World' to the file:
c                   eval      data = 'Hello World!'
c                   callp     write(fd: %addr(data): %size(data))

C* Close the file:
c                   callp     close(fd)

c                   eval      *inlr = *on
```

You typed all of that already? Wow, that was quick, good job! The instructions at the top tell you how to compile it. But, of course, you'll need to substitute the source library that you used for the "XXX" that I put in the comments.

For example, if your source library is called "TASTYSRC", you would type: **`CRTBNDRPG CH1HELLO SRCFILE(TASTYSRC/QRPGLESRC) DBGVIEW(*LIST)`**

> **Note:** I don't recommend that name, it makes you want to stop and eat your computer.

Once you've succeeded in compiling the program, you can run it just by typing: **`CALL CH1HELLO`**

If a problem occurs when you run this program, you'll get a DSPLY message on the screen telling you that something failed. If nothing happens on your screen, rejoice! That's what we wanted to happen! In the next topic, we'll talk about what that program did, and verify that things worked.

1.6. Looking at our example from OS/400

In our hello world example, we defined a bunch of prototypes in our D-specs that told the system what APIs we wanted to call, and how the parameters were to be passed to them. If you don't understand how prototypes work, this might be a good time to whip out a book on prototypes and brush up.

When we get to the C-specs, you'll see that we are calling the open() API, like this:

```
c                   eval      fd = open('/helloworld': ❶
c                                       O_CREAT+O_TRUNC+O_WRONLY:❷
c                                       (6*64)+(6*8)+(4))❸
c                   if        fd < 0 ❹
c                   eval      Msg = 'open failed!'
c                   dsply                   msg
c                   eval      *inlr = *on
c                   return
c                   endif
```

❶ Here we tell the API that the file we want to open is in the root directory ("/") and is called "helloworld", and that the result of the call should be stored in the variable called "fd" (which is shorthand for "file descriptor")

❶ These flags tell the API that we want to create the file if it doesn't already exist (O_CREAT), truncate the file to zero bytes if it does exist (O_TRUNC) and that we're opening the only for writing to it (O_WRONLY) and not for reading.

"Truncating" the file can be thought of as "clearing" the file. All it does is delete any data that currently exists in the file.

❸ This confusing looking equation is the file's "mode", also called the "permissions" that we're giving to it. This tells the operating system who is allowed to read this file, who is allowed to write to it, and who is allowed to execute/search it (if it is a program or directory, respectively)

Because this "mode" is confusing, and hard-to-read, I use named constants to make it clearer. If you look at the previous example, you'll see that I used named constants to make it easier to read. I'll use some more standardized (though, arguably harder to read) constants in a later chapter.

We'll talk more about file modes in the upcoming chapter. For now, just know that we're giving read & write permission to ourselves, read & write permissions to anyone in our "group", and read-only permissions to everyone else.

❹ If an error occurred, the system will return a -1 to our "fd" variable. Therefore, we check for an error, and if one occurred, we'll report it to the user, and end the program.

We then proceed to write the string 'Hello World!' to the file, close it, and end the program.

To see if it worked, return to your trusty OS/400 command line and type:**`WRKLNK '/*'`**

OS/400 will bring up the "Work with Object Links" screen which shows you a list of all of the objects in the root directory of the IFS. (Depending on what's stored here on your system, there could be many pages of information.) Find the file called 'helloworld' in the list of object links. You'll note that it has a 'Type' of 'STMF', which means that it is a stream file.

Place a **`5`** next to it, and OS/400 will show you the contents of the stream file. You should now see that it contains the words "Hello World!" just as we expected. Press the **`F3 key`** to get back to the "Work With Object Links" screen.

Unless you have some nifty use for the helloworld file you created, let's delete it now, by putting a **`4`** next to it to delete it.

Yes, it's really that simple! Aside from the prototypes and maybe the file mode, this program is really short and easy to follow. Using this same technique, you can write almost anything to a stream file! We'll cover this more in depth, and show you how to read a stream file in the next chapter.

Chapter 2. The Basics of Stream Files

2.1. Opening Files with the open() API

2.1.1. Creating a header member

More than half of the code in our "Hello World" program was prototypes and other definitions. If we were C programmers, we wouldn't have to worry about this part of the coding, because IBM ships "header members" with OS/400 and ILE C/400 that C programmers can simply "include" at the top of their programs.

Since IBM doesn't do that for us as RPG programmers, what we'll do is create our own header member. We'll call it "IFSIO_H" which stands for "Integrated File System I/O Header Member," in that member, we'll put our prototypes, constants, etc. After we've done that, we'll no longer need to type those prototypes into every program. One /copy and we're done!

The first prototype that we'll add will be for the open() API.

2.1.2. Prototyping the open() API

The UNIX-type APIs manual shows us a prototype (written in C) for the open() API. It looks like this:

```
int❶ open❷(const char *path❸, int oflag❹, ...❺);
```

❶ The "int" here signifies what type of value the procedure returns. The "int" data type in C is identical to a "10I 0" variable in RPG IV.

❷ The word "open" here signifies the name of the sub-procedure that's being called. Like all things in C, procedure names are case-sensitive. This is important to us because, in RPG they are not case-sensitive! In fact, if we don't do anything special, the RPG compiler will convert the procedure name to all uppercase before binding. Therefore, when we make our RPG prototype, we'll use the EXTPROC() keyword to refer to open as an all-lowercase procedure name.

❸ This is where we specify the name of a path that we want to access. The "char" means character, and the "*" means pointer. So, what this procedure needs from us is a pointer that points to a character variable. You see, in C, character strings are implemented by specifying a starting point in memory, and then reading forward in memory until a x'00' (called a "null") is encountered.

Fortunately, RPG's options(*string) keyword, and the %str() built-in-function, make it easy for us to convert to and from C's string format.

❹ This is an integer ("int") which defines the flags that are used by the open() API. As I mentioned, the C "int" data type is equivalent to the "10I 0" data type in RPG.

❺ These three periods signify that any number of optional parameters may follow. This is difficult for us, since RPG prototypes need to know in advance what the data types of the parameters are.

Fortunately, if you read IBM's documentation, you find out that there are only two optional parameters that follow, and they are both unsigned integers. One is for specifying the mode, which is used when O_CREAT is passed as a flag, the other is for specifying a code page, which we will talk more about in chapter 5.

Now that we know how the C prototype works, we can write our own RPG prototype. Here's what I came up with:

```
D open            PR            10I 0 extproc('open')
D   path                          *   value options(*string)
D   oflag                       10I 0 value
D   mode                        10U 0 value options(*nopass)
D   codepage                    10U 0 value options(*nopass)
```

Please add that prototype to your IFSIO_H member now, unless of course, you've decided to just install mine, in which case you already have it.

The first thing that you might notice is that all of the parameters are passed by value. That's because C is expecting to receive a pointer, an integer, an unsigned integer and another unsigned integer. If we passed these parameters by reference, it would actually receive 4 memory addresses, rather than receiving the actual data.

As a general rule, when a C program expects to receive something by value, it will simply list the data type followed by the variable. If it wants to receive the address of the variable, it will ask for a pointer to that variable by putting a "*" in front of it. For example "int oflag" is a request for an integer, passed by value, whereas "int *oflag" would be the same integer passed by reference.

Wait a minute! Wouldn't that mean that the "char *path" should be passed by reference, instead of by value?! Yes, that's true. In fact, we could've coded path as:

```
D   path                        1024A
```

However, if we did that, we'd have to assign a length to "path", and the C version allows path to be of any length. The trick is, passing a pointer by value is the same thing as passing a variable by reference. In either case, what actually gets passed from procedure to procedure is an address in memory. But, if we use a pointer by value, and we use "options(*string)", the RPG compiler will automatically allow any length string, and will automatically convert it to C's format by adding the terminating "null" character at the end. This saves us some work.

Finally, you'll notice that the mode and codepage parameters are set up with "options(*nopass)". This means that they're optional, and we only need to pass them when we need them.

2.1.3. The path parameter

path is pretty self-explanatory. It is the path to the file in the IFS that we want to open. In the IFS, the "/" character is used to separate the different components of the path. The first component may be a "/" to signify the root directory. Thereafter, each section of the path refers to a directory name until we reach the last component, which specifies the filename.

For example, consider the following path name:

```
/ifsebook/chapter2/examples/myfile.txt
```

The leading "/" means to start at the root of the IFS. If it was not specified, the path name would actually start at whatever our current working directory was, and continue from there. But since it has a "/" we're telling it that the path to the file actually starts at the root of the system.

The word "ifsebook" refers to a directory. The word "chapter2" refers to a sub-directory that's inside the "ifsebook" directory. The word "examples" refers to another sub-directory, this one is inside the "chapter2" directory, and finally, "myfile.txt" refers to the object that's in the "examples" directory.

Let's try another, somewhat more familiar, example:

```
/QSYS.LIB/qgpl.lib/qrpglesrc.file/proof.mbr
```

This tells us to go back to the "/" root directory, then look at the QSYS.LIB directory (which is also known as the QSYS library) and that within that directory is a directory called the qgpl.lib directory (which is also known as the QGPL library) and within that, there's a file called QRPGLESRC which contains a member called "PROOF".

2.1.4. The oflag parameter

oflag is where we specify the options we want to use when opening the file. What we're actually passing here is a string of 32 bits, each of which specifies a different option. The rightmost bit specifies "Read only", then moving one bit to the left, that bit specifies "Write only", and the next bit specifies "reading and writing" and the next bit specifies "create the file if it doesn't exist," etc.

In order to make our lives easier, rather than actually specifying the bits manually, we define a series of flags, that when added together, will turn on the bits that we desire.

For example, we use the number 8 to signify "create if the file doesn't exist" and the number 2 to signify "write only". This makes more sense if you convert those numbers to binary. The decimal number 8 is 1000 in binary. The decimal number 2 is 10 in binary. So you see, when we specify the number 8, we actually specify that we want the 4th bit (counting from the right) to be turned on. When we specify 2, we are specifying that the 2nd bit be turned on. If we add those two numbers together, 8+2=10. If you convert the decimal 10 to binary you get 1010 (bits 4 and 2 are both on). Because each of these numbers turns on a single bit, we refer to them as "flags", and they supply us with a convenient way to which options we want to pass to the open() API.

So that we don't need to mess with the bit values later in this book, let's add those flags to our IFSIO_H member now. This is what we need to add:

```
D*********************************************************************
D*  Flags for use in open()
D*
D* More than one can be used -- add them together.
D*********************************************************************
D*                                          Reading Only
D O_RDONLY        C                   1
D*                                          Writing Only
D O_WRONLY        C                   2
D*                                          Reading & Writing
D O_RDWR          C                   4
D*                                          Create File if not exist
D O_CREAT         C                   8
D*                                          Exclusively create
```

```
D O_EXCL          C                   16
D*                                            Truncate File to 0 bytes
D O_TRUNC         C                   64
D*                                            Append to File
D O_APPEND        C                   256
D*                                            Convert text by code-page
D O_CODEPAGE      C                   8388608
D*                                            Open in text-mode
D O_TEXTDATA      C                   16777216
```

2.1.5. The mode parameter

mode is used to specify the access rights that this file will give to users who want to work with it. Like the "oflag" parameter, this parameter is treated as a series of bits. The rightmost 9 bits are the ones that we're concerned with, and they're laid out like this:

```
user:       owner    group    other
access:     R W X    R W X    R W X
bit:        9 8 7    6 5 4    3 2 1
```

These bits specify Read, Write and Execute access to 3 types of users. The first is the file's owner, the second is users with the same group profile as the file's owner, and the third is all other users.

For example, if I wanted to specify that the owner of the file can read and write to the file, that people in his group can only read the file, and that everyone else has no access at all, I'd specify the following bits: 110 100 000. If you look at those bits as the binary number 110100000, and convert it to decimal, you'd get 416. So, to assign those permissions to the file, you'd call open() with a 3rd parameter of 416.

Just as we did for the "oflags" parameter, we'll also specify bit-flags for the mode, which we can add together to make our programs easier to read. Please add these to your IFSIO_H member:

```
D*****************************************************************
D*      Mode Flags.
D*         basically, the mode parm of open(), creat(), chmod(),etc
D*         uses 9 least significant bits to determine the
D*         file's mode. (peoples access rights to the file)
D*
D*           user:       owner    group    other
D*           access:     R W X    R W X    R W X
D*           bit:        8 7 6    5 4 3    2 1 0
D*
D* (This is accomplished by adding the flags below to get the mode)
D*****************************************************************
D*                                        owner authority
D S_IRUSR         C                   256
D S_IWUSR         C                   128
D S_IXUSR         C                   64
D S_IRWXU         C                   448
D*                                        group authority
```

```
D S_IRGRP         C                   32
D S_IWGRP         C                   16
D S_IXGRP         C                   8
D S_IRWXG         C                   56
D*                                         other people
D S_IROTH         C                   4
D S_IWOTH         C                   2
D S_IXOTH         C                   1
D S_IRWXO         C                   7
```

Now, instead of specifying "416", we can simply add together S_IRUSR+S_IWUSR+S_IRGRP, which specifies "read access for user", "write access for user" and "read access for group", respectively.

2.1.6. The codepage parameter

If you specify the O_CODEPAGE flag in the oflag parameter, you must use this parameter to specify which code page the file will be assigned.

We will talk about that more in Chapter 5, in our discussion of text files.

2.1.7. The return value of the open() API

The *return value* of the open() API is a "file descriptor". It is an integer that we will pass to all of the other IFS APIs that we call so that they know which file we are referring to. If something goes wrong, and the system is not able to open the file that we requested, it will return a value of -1 instead of a file descriptor. So, whenever we call open() we will check for this, and treat -1 as an error.

2.1.8. Code snippet showing the use of the open() API

Here's a code snippet that uses the open() API:

```
c                   eval      path = '/QIBM/UserData/OS400/DirSrv' +
c                                  '/slapd.conf'
c                   eval      flags = O_WRONLY + O_CREAT + O_TRUNC

c                   eval      mode =  S_IRUSR + S_IWUSR
c                                   + S_IRGRP

c                   eval      fd = open(%trimr(path): flags: mode)
c                   if        fd < 0
c                   goto      bomb_out
c                   endif
```

2.2. Closing a file with the close() API

If your head is spinning after reading about the open() API, you'll be glad to know that the close() API is comparatively simple.

The close() API is used to close a file that we opened when we called the open() API. Here is the C language prototype for close(), as it is listed in IBM's UNIX-type APIs manual:

```
int close(int fildes);
```

Simple, yes? The "int" is the return value. The "close" is the name of the procedure. The "int fildes" is the only parameter to the procedure, and it's just an integer, which is "10I 0" in RPG.

So, the RPG prototype will look like this:

```
D close           PR            10I 0 extproc('close')
D   fildes                      10I 0 value
```

See? It returns a "10I 0", because the C prototype returned an "int". It accepts one parameter, which is also a "10I 0" because the C prototype's only parameter was an "int". The parameter is passed by value because we don't want to pass the address, and we use "extproc()" to make sure the compiler doesn't try to call "CLOSE" instead of "close".

Great.

There's one small problem. This same close() API is used by both socket connections (TCP/IP communications API) and also by the IFS APIs that we're using. That's a problem because if you ever tried to use both sockets and IFS in the same program, the definitions would conflict, and the program wouldn't compile.

So, we're going to use a little "compiler directive" magic to make sure that the two close() prototypes never conflict, by making the prototype look like this:

```
D/if not defined(CLOSE_PROTOTYPE)
D close           PR            10I 0 extproc('close')
D   fildes                      10I 0 value
D/define CLOSE_PROTOTYPE
D/endif
```

And then, when the day comes that we make a header member for sockets, we'll have to remember to also put that same /define logic in the sockets header member. Do you see how it works? Pretty cool eh?

Here's an example of calling the close() API:

```
c                   callp     close(fd)
```

2.3. Writing streams with the write() API

The write() API is used to write bytes to a stream file.

The reason that I say "bytes" as opposed to saying "letters" or "words" is that a byte containing any value can be written. We are not limited to just alphanumeric strings of text. We can write the contents of a packed decimal variable, for example, or an entire data structure.

All you have to tell write() is an area of memory (and again, it doesn't care what's in that area) and length. It copies the bytes from memory to disk.

Here's what the C language prototype of write() looks like, as printed in the UNIX-type APIs manual:

```
int❶ write(int fildes❷, const void *buf❸, size_t nbyte❹);
```

Now that's a sexy looking API!

❶ The return value is an "int", which is a 32-bit signed integer, identical to the RPG "10I 0" data type. The write() API returns the number of bytes written to disk. If something goes wrong, this number will be smaller than the amount we told it to write, so we can use this to detect errors, as well.

❷ The first parameter is also an integer, and this one represents the file descriptor, which is the value that we got from the open() API.

❸ Something new! What the heck could "const void *buf" mean? Well, my friend, it's quite simple. The "const" is just like the RPG "const" keyword. It simply means that the API cannot and will not change the contents of the parameter. But does "void *" mean that it's a pointer to a void? No, not really. It means that this pointer can point to anything. It doesn't have to be a character, a number, or a structure. It can point to any byte value in memory.

❹ And finally, we have "size_t nbyte". It's pretty clear that "nbyte" is short for "number of bytes to write". But what is that funny "size_t" thing?

It's a "user-defined" type. It's similar in some ways to the "like" keyword in RPG. The idea is that on different platforms, the way that a byte size is stored may be different. To write code that's reusable, they have a header member called "sys/types.h" which describes the actual data types of things like "size_t", and then when you change to a different platform and use different header files, the program compiles and works on that platform as well.

On the AS/400, size_t is defined to be a 32-bit unsigned integer. In other words, it's the same as the RPG "10U 0" data type.

Now that we've covered that, here's the RPG version of the prototype. You'll want to add this to the IFSIO_H header member:

```
D write           PR            10I 0 extproc('write')
D   fildes                      10I 0 value
D   buf                           *   value
D   nbyte                       10U 0 value
```

See? Not so bad. Just a couple of integers and a pointer. no sweat.

Here's a code snippet showing a program that calls the write() API:

```
c                   eval      wrdata = 'THE QUICK BROWN FOX JUMP'
c                   if        write(fd: %addr(wrdata): %size(wrdata))
c                             < %size(wrdata)
c                   goto
c                   endif
```

2.4. Reading a stream file with the read() API

The read() API is the exact opposite of the write() API. It reads bytes of data from a stream file, and stores them into the area of memory that you point it to.

Here's the C language prototype of read():

```
int read(int fildes, void *buf, size_t nbyte);
```

This prototype is so much like the write() API that I won't even describe the process of converting it from C to RPG. From reading about write(), you should already understand. So, without any more long-winded ramblings, here's the RPG prototype:

```
D read            PR            10I 0 extproc('read')
D   fildes                      10I 0 value
D   buf                           *   value
D   nbyte                       10U 0 value
```

Make sure you put that in your copy of the IFSIO_H member.

In many ways, read() is very similar to write(). Like it's counterpart, it can be used to work with any byte values. It does not care if the data is numeric or character or part of a graphic or sound format. And, like write() it takes 3 parameters, and returns an integer. There are, however, some crucial differences:

- Obviously, it reads instead of writes. Otherwise it would be silly to call it "read!"
- Note that the "buf" argument is no longer marked as "const". That means that the API definitely can change the contents of the variable that buf points to!

  In fact, that's where read will store the information that it loads from the stream file.
- The "nbyte" parameter tells read the size of the variable that the "buf" parameter is pointing to. It's true that read() will try to read that many bytes from the disk, but if you're at the end of the stream file, read() may read fewer bytes than you've specified in the "nbyte" argument. Don't treat that as an error!

You call the read() API like this:

```
c                   eval      len = read(fd: ptr2buf: %size(buf))
c                   if        len < 1
c                   goto      no_more_to_read
c                   endif
```

2.5. Example of writing and reading data to a stream file

The last few topics have probably given you lots of new things to think about. It's time to play with them!

First of all, let's create a directory to write all of our stream files to. This will keep us from cluttering up the root directory of the IFS with our tests. Let's call this new directory "ifstest." We can create it by typing the following command at our OS/400 command-line:

```
CRTDIR DIR('/ifstest')
```

If you get an error that you do not have sufficient authority to create it, or something like that, you may need to speak with your system administrator. Tell him that you need a sandbox to play in!

Here's a program which both writes and reads data from a stream file. It also demonstrates one of the properties of a stream file -- the data is stored as a continuous stream of bytes, not in records.

Take a look at it, guess what you think it does, then try it out. It's pretty cool!

```
     * CH2WRRD: Example of writing & reading data to a stream file
     *  (From Chap 2)
     *
     * To compile:
     *   CRTBNDRPG CH2WRRD SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
     *
    H DFTACTGRP(*NO) ACTGRP(*NEW)

    D/copy IFSEBOOK/QRPGLESRC,IFSIO_H

    D fd              S             10I 0
    D wrdata          S             24A
    D rddata          S             48A
    D flags           S             10U 0
    D mode            S             10U 0
    D Msg             S             50A
    D Len             S             10I 0

    C*****************************************************************
    C* Example of writing data to a stream file
    C*****************************************************************
    c                   eval      flags = O_WRONLY + O_CREAT + O_TRUNC

    c                   eval      mode =  S_IRUSR + S_IWUSR
    c                                   + S_IRGRP
    c                                   + S_IROTH

    c                   eval      fd = open('/ifstest/ch2_test.dat':
    c                                       flags: mode)
    c                   if        fd < 0
    c                   eval      Msg = 'open(): failed for writing'
    c                   dsply                   Msg
    c                   eval      *inlr = *on
    c                   return
    c                   endif

    C* Write some data
    c                   eval      wrdata = 'THE QUICK BROWN FOX JUMP'
    c                   callp     write(fd: %addr(wrdata): %size(wrdata))

    C* Write some more data
```

```
c                   eval      wrdata = 'ED OVER THE LAZY GIRAFFE'
c                   callp     write(fd: %addr(wrdata): %size(wrdata))

C* close the file
c                   callp     close(fd)

C*****************************************************************
C* Example of reading data from a stream file
C*****************************************************************
c                   eval      flags = O_RDONLY

c                   eval      fd = open('/ifstest/ch2_test.dat':
c                                       flags)
c                   if        fd < 0
c                   eval      Msg = 'open(): failed for reading'
c                   dsply                   Msg
c                   eval      *inlr = *on
c                   return
c                   endif

c                   eval      len = read(fd: %addr(rddata):
c                                        %size(rddata))
c                   eval      Msg = 'Length read = ' +
c                                   %trim(%editc(len:'M'))
c     Msg           dsply
c                   dsply                   rddata

c                   callp     close(fd)

c                   eval      *inlr = *on
c                   return
```

As before, there are instructions on compiling the program near the top of the source. Give it a try. I'll be right back, I'm going to go get a cold beverage.

Ahhhh... lime soda.

2.6. Error handling

The world is an imperfect place. Things go wrong. Sometimes a file can't be opened, or sometimes a tyrannical system administrator won't let us access something. It can be rough.

One of the problems with the example programs that we've written so far is that, although they detect when something went wrong, they couldn't tell us what the problem was. They know something happened, but they don't know what.

2.6.1. Retrieving the error number.

Like most of the UNIX-type APIs, our IFS functions return their error information using the C language "errno" variable. The idea is that there is a global variable called "errno" which a C program can check after something has gone wrong. The result is an integer that corresponds to a specific error message.

On the AS/400, the "errno" variable is actually returned by a sub-procedure that, for C programmers, gets called behind-the-scenes. So, for us to check errno, all we have to do is call that sub-procedure, and get the return value.

The sub-procedure that returns error information is called "__errno" and is part of the ILE C runtime library which is installed on every AS/400. The C language prototype for "__errno" looks like this:

```
int *__errno(void);
```

What that means is that the procedure is called __errno, and it returns a ("int *") pointer to an integer. The "void" signifies that there are no parameters.

In RPG, you can't start a sub-procedure name with the underscore character, so we'll add another symbol to the front of the prototype to make it work. The result looks like this:

```
D @__errno        PR              *   ExtProc('__errno')
```

Now, you'll note that although we're looking for an integer, this procedure actually returns a pointer. Yech! So what we'll do is create a simple sub-procedure that gets an integer from the area of memory that the pointer points at. That's a very simple sub-procedure, and it looks like this:

```
P errno           B
D errno           PI            10I 0
D p_errno         S               *
D retval          S             10I 0 based(p_errno)
c                   eval      p_errno = @__errno
c                   return    retval
P                 E
```

2.6.2. What does the error number mean?

So, now we know that errno can be called, and it will give us an integer that tells us which error has occurred. But, what does the number mean? For example, if we got back the number 3401, how would we know what went wrong?

In C, there's a source member which programmers use that contains constants for each error number. For example, it will define the constant EACCES to be the number 3401. The C program can compare errno to EACCES, and if they match, it knows that the user does not have enough access (or "authority") to carry out the function.

In fact, if you look at the text in the IBM Information Center that explains (for example) the write() API, you'll see that under "Error Conditions" it says "If write() is not successful, errno usually indicates one of the following errors . . ." and then goes on to list errors like [EACCES] and [ENOSPC]. These error conditions are nothing more than the named constants that I mentioned above.

Since the "errno" stuff can be used by other APIs besides the ones that this book covers, we will place these in their own header member. That way, you can include them into your future programs without also including the code that's IFS-specific.

I've called my /copy member "ERRNO_H". If at all possible you should consider using the one that I provide with this book. In it, I've put RPG named constants corresponding to all the values of errno that I know about. Since it would be tedious for you to find all of these values and type them in, you may as well just use mine!

2.6.3. Getting a human-readable error message

In addition to the named constants for each error number, it's useful to have a "human-readable" error message that corresponds to each error number. For example, when you want to print a message on the screen explaining what went wrong, you'd probably rather say "No such path or directory" rather than "Error 3025 has occurred."

The ILE C/400 runtime library contains a procedure called "strerror()" for this purpose. When you call strerror() with an error number as a parameter, it returns a pointer to a variable length, null-terminated, error message. Here's the C and RPG prototype for strerror():

```
 char *strerror(int errnum);

D strerror        PR              *   ExtProc('strerror')
D    errnum                    10I 0 value
```

In addition to strerror(), you can also view each error number as a message in an OS/400 message file. The QCPFMSG message file contains all of the C error numbers prefixed by a "CPE". For example, the error ENOENT is 3025. If you type **`DSPMSGD CPE3025 MSGF(QCPFMSG)`** at the command prompt, it will show you a message that says "No such path or directory." Likewise, if you looked up CPE3401, you'd see the human-readable message "Permission denied."

2.6.4. Utilities for communicating errors

As you may already know, OS/400 programs usually return error information from one program to another by sending a "program message". When an error occurs which causes a program to fail, the program usually sends back a program message of type "escape" to it's caller.

For the sake of making error handling easier, we will create two simple sub-procedures that we can use to send back "escape messages", and add these to our ERRNO_H file, so all of our programs can use them.

The first utility is called "die". It will send back any user supplied error message under a message number of CPF9897. This is useful for supplying simple text error messages in our example programs. Here's the code:

```
P die             B
D die             PI             1N
D    msg                      256A   const

D QMHSNDPM        PR                    ExtPgm('QMHSNDPM')
D   MessageID                   7A   Const
D   QualMsgF                   20A   Const
D   MsgData                    256A   Const
```

```
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                  256A

D dsEC            DS
D  dsECBytesP              1     4I 0 inz(%size(dsEC))
D  dsECBytesA              5     8I 0 inz(0)
D  dsECMsgID               9    15
D  dsECReserv             16    16
D  dsECMsgDta             17   256

D MsgLen          S              10I 0
D TheKey          S               4A

c     ' '           checkr    msg           MsgLen
c                   if        MsgLen<1
c                   return    *off
c                   endif

c                   callp     QMHSNDPM('CPF9897': 'QCPFMSG   *LIBL':
c                               Msg: MsgLen: '*ESCAPE':
c                               '*': 3: TheKey: dsEC)

c                   return    *off
P                 E
```

The other utility function is called "EscErrno". We will pass an error number as an argument to this function, and it will send back the appropriate CPExxxx error message as an escape message to the calling program.

EscErrno is useful when we want our programs to crash and report errors that the calling program can monitor for individually. For example, a calling program could be checking for CPE3025, and handle it separately than CPE3401.

Here is the code for EscErrno:

```
P EscErrno        B
D EscErrno        PI             1N
D   errnum                      10i 0 value

D QMHSNDPM        PR                  ExtPgm('QMHSNDPM')
D   MessageID                    7A   Const
D   QualMsgF                    20A   Const
D   MsgData                      1A   Const
D   MsgDtaLen                   10I 0 Const
D   MsgType                     10A   Const
D   CallStkEnt                  10A   Const
D   CallStkCnt                  10I 0 Const
D   MessageKey                   4A
D   ErrorCode                  256A

D dsEC            DS
```

```
D  dsECBytesP             1      4I 0 inz(%size(dsEC))
D  dsECBytesA             5      8I 0 inz(0)
D  dsECMsgID              9     15
D  dsECReserv            16     16
D  dsECMsgDta            17    256

D TheKey          S              4A
D MsgID           S              7A

c                   move      errnum        MsgID
c                   movel     'CPE'         MsgID

c                   callp     QMHSNDPM(MsgID: 'QCPFMSG   *LIBL':
c                               ' ': 0: '*ESCAPE':
c                               '*': 3: TheKey: dsEC)

c                   return    *off
P                 E
```

What I've done in my ERRNO_H is put the codes for the all of the procedures (errno, die, and escerrno) at the bottom, and enclosed them in these:

```
 /if defined(ERRNO_LOAD_PROCEDURE)
.... procedure code goes here ....
 /endif
```

This allows us to include all of the error handling code in our programs by copying the header member twice, once without the "errno_load_procedure" symbol defined, which goes in our D-specs, and once with the "errno_load_procedure" symbol defined, which goes where our sub-procedures go.

2.7. Our last example with error handling added

Here's an example of the error handling code that we discussed in the last section. All I did here is take the sample code that we wrote in section 2.5 above, and add error checking to it. When something goes wrong, the program calls die() to signal an abnormal end.

```
 * CH2ERRNO: Example of writing & reading data to a stream file
 *   with error handling.  (This is the same as CH2WRRD except
 *   for the error handling)
 *  (From Chap 2)
 *
 * To compile:
 *   CRTBNDRPG CH2ERRNO SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSeBOOK/QRPGLESRC,IFSIO_H
D/copy IFSeBOOK/QRPGLESRC,ERRNO_H
```

```
D fd              S             10I 0
D wrdata          S             24A
D rddata          S             48A
D flags           S             10U 0
D mode            S             10U 0
D ErrMsg          S            250A
D Msg             S             50A
D Len             S             10I 0

C*****************************************************************
C* Example of writing data to a stream file
C*****************************************************************
c                   eval      flags = O_WRONLY + O_CREAT + O_TRUNC

c                   eval      mode =  S_IRUSR + S_IWUSR
c                                   + S_IRGRP
c                                   + S_IROTH

c                   eval      fd = open('/ifstest/ch2_test.dat':
c                                       flags: mode)
c                   if        fd < 0
c                   eval      ErrMsg = %str(strerror(errno))
c                   callp     die('open() for output: ' + ErrMsg)
c                   endif

C* Write some data
c                   eval      wrdata = 'THE QUICK BROWN FOX JUMP'
c                   if        write(fd: %addr(wrdata): %size(wrdata))<1
c                   eval      ErrMsg = %str(strerror(errno))
c                   callp     close(fd)
c                   callp     die('open(): ' + ErrMsg)
c                   endif

C* Write some more data
c                   eval      wrdata = 'ED OVER THE LAZY GIRAFFE'
c                   if        write(fd: %addr(wrdata): %size(wrdata))<1
c                   eval      ErrMsg = %str(strerror(errno))
c                   callp     close(fd)
c                   callp     die('open(): ' + ErrMsg)
c                   endif

C* close the file
c                   callp     close(fd)

C*****************************************************************
C* Example of reading data from a stream file
C*****************************************************************
c                   eval      flags = O_RDONLY

c                   eval      fd = open('/ifstest/ch2_test.dat':
c                                       flags)
c                   if        fd < 0
```

```
c                   eval      ErrMsg = %str(strerror(errno))
c                   callp     die('open() for input: ' + ErrMsg)
c                   endif

c                   eval      len = read(fd: %addr(rddata):
c                                        %size(rddata))
c                   if        len < 1
c                   eval      ErrMsg = %str(strerror(errno))
c                   callp     close(fd)
c                   callp     die('read(): ' + ErrMsg)
c                   endif

c                   eval      Msg = 'Length read = ' +
c                                   %trim(%editc(len:'M'))
c     Msg           dsply
c                   dsply                   rddata

c                   callp     close(fd)

c                   eval      *inlr = *on
c                   return

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

The only reason that I used die() instead of EscErrno() to handle errors in this program is that with die() I can easily add text explaining where the error occurred. In the next section, I'll give an example of EscErrno().

2.8. Example of writing raw data to a stream file

As I've mentioned in previous sections, stream files can be used for any byte values, not just for words and other text. As a proof of concept, I thought it might be fun to generate a very small MS-DOS program as a stream file.

Programs under OS/400 are stored in *PGM objects. You can't directly open and manipulate a *PGM object on the AS/400. You have to write source code, and let the compiler compile it.

However, on the PC, all objects are stored as stream files. It doesn't matter if it's a data file, a program, an audio file, etc. Every object is stored as a stream file!

To prove that, here's an RPG program that actually generates a PC program. The stream file that it outputs can be downloaded to your PC, and run. (Provided, of course, that the PC is able to run MS-DOS programs).

```
 * CH2RAWDTA: Example of writing non-text to a stream file
 *  (From Chap 2)
 *
 * To compile:
 *   CRTBNDRPG CH2RAWDTA SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
```

```
D fd              S             10I 0
D wrdata          S             79A
D err             S             10I 0

c                   eval      wrdata = x'B409BA0C01CD21B8004CCD21' +
c                                      x'416C6C206F626A6563747320' +
c                                      x'6F6E207468652050432172' +
c                                      x'652073746F72656420696E20' +
c                                      x'2273747265616D2066696C65' +
c                                      x'73222C206576656E2070726F' +
c                                      x'6772616D732124'

c                   eval      fd = open('/ifstest/littlepgm.com':
c                                       O_WRONLY+O_CREAT+O_TRUNC:
c                                       S_IRUSR + S_IWUSR + S_IXUSR
c                                     + S_IRGRP + S_IXGRP
c                                     + S_IROTH + S_IXOTH)
c                   if        fd < 0
c                   callp     EscErrno(errno)
c                   endif

c                   if        write(fd: %addr(wrdata): %size(wrdata))
c                                  < %size(wrdata)
c                   eval      err = errno
c                   callp     close(fd)
c                   callp     EscErrno(err)
c                   endif

c                   callp     close(fd)

c                   eval      *inlr = *on
c                   return

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

This program outputs a stream file called `littlepgm.com` into our `/ifstest` directory. To run this program you'll need to transfer it to your PC. One way to do this, if your system is set up for it, would be to map a network drive to your AS/400. Another good choice would be to FTP the file to your PC. If you decide to use FTP, make sure that you use binary mode.

Once you've got it on your PC, you should run the program from an MS-DOS prompt.

Chapter 3. Other simple, but helpful IFS commands

3.1. Checking existence and permissions to files

In the last chapter, we covered the basics of how stream files work. It's all down hill from here!

One question that I've been asked many times is: "I use the CHKOBJ command in my CL programs. How can I do the same thing with a file in the IFS?"

The answer is the access() API. Access() can be used to check two things: whether the file exists, and whether it's accessible for reading, writing or execution.

The C-language prototype for the access() API looks like this:

```
int access(const char *path, int amode);
```

The prototype is quite simple, and I think by now you're already getting the hang of it, so without further ado, here's the RPG version:

```
❶     D access          PR            10I 0 ExtProc('access')
❷     D   Path                          *   Value Options(*string)
❸     D   amode                       10I 0 Value
```

Please add this to the IFSIO_H copy member, if you're typing it in yourself.

❶ The access API returns an integer which can either be 0 if the file is accessible, or -1 if for some reason it is not.

Like most UNIX-type APIs, we can check errno after calling access to find out why the file wasn't accessible.

❷ This is the path name of the IFS object that we want to check the accessibility of.

❸ This is the access that we want to check. This is another one of those "bit flag" fields, similar to the ones we've been using in the open() API.

The *amode* parameter uses the rightmost 3 bits of the parameter to determine which access we want to check. If the right-most bit is on, access() checks for execute authority. The next bit to the left checks for write access, and the 3rd bit from the right checks for read access.

If none of the bits in the *amode* parameter are set, the API will only check to see if the object exists.

Just like we did for the other bit-flags that we've used, we will define named constants to both to make our code easier to follow, and also to match the constants that are already defined for the C programmers.

```
**************************************************************************
* Access mode flags for access()
*
*   F_OK = File Exists
*   R_OK = Read Access
*   W_OK = Write Access
*   X_OK = Execute or Search
**************************************************************************
```

```
D F_OK            C                   0
D R_OK            C                   4
D W_OK            C                   2
D X_OK            C                   1
```

Here's a sample of calling access() in an RPG program:

```
c                   if        access(%trimr(myfile): F_OK) < 0
c                   eval      err = errno
c                   if        err = ENOENT
c                   callp     die('Errrm... can"t find that file!')
c                   else
c                   callp     die(%str(strerror(err)))
c                   endif
c                   endif

c                   if        access(%trimr(myfile): R_OK) < 0
c                   eval      err = errno
c                   if        err = EACCES
c                   callp     die('It"s there, but YOU can"t read ' +
c                                 'it!  Nyaahh! Nyaahh!')
c                   else
c                   callp     die(%str(strerror(err)))
c                   endif
c                   endif
```

3.2. Example of checking for an object in the IFS

Here's an example that's also a useful utility. We will create a command called "CHKIFSOBJ" which works like the OS/400 CHKOBJ command, except that it operates on an IFS object.

CHKOBJ allows you to check if the file exists, and also allows you to optionally check if you have authority to use it. So, our IFS version will do the same. To do that, we need two parameters, the path name of the object to check, and the authority to check for.

Here's the command definition to do that:

```
CMD         PROMPT('Check for IFS Object')
PARM        KWD(OBJ) TYPE(*CHAR) LEN(640) MIN(1) +
              PROMPT('Object')
PARM        KWD(AUT) TYPE(*CHAR) LEN(10) RSTD(*YES) +
              DFT(*NONE) VALUES(*NONE *R *RW *RX *RWX +
              *W *WX *X) PROMPT('Authority')
```

Take a look at the "AUT" parameter. It allows you to specify "*NONE" if you just want to see if an object exists, or to check for read, write, execute or any combination of them. It's just like access(), really!

Now, here's the RPG code that our command will run:

```
* CH3CHKOBJ: Example of checking for an object in the IFS
```

```
 *  (From Chap 3)
 *
 * To compile:
 *   CRTBNDRPG CH3CHKOBJ SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *   CRTCMD CMD(CHKIFSOBJ) PGM(CH3CHKOBJ) SRCFILE(xxx/QCMDSRC)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D Path            S            640A
D Authority       S             10A
D AMode           S             10I 0

 ** Warning:  call this program from the command.  If you call
 **       it directly, because "Path" is larger than 32 bytes.
 **       See http://faq.midrange.com/data/cache/70.html
 **

C     *entry        plist
c                   parm                    Path
c                   parm                    Authority

C* First, just check if the file exists:
c                   if        Access(%trimr(Path): F_OK) < 0
c                   callp     EscErrno(errno)
c                   endif

C* Next, check if the current user has authority:
c                   if        Authority <> '*NONE'

c                   eval      amode = 0

c                   if        %scan('R':Authority) > 0
c                   eval      amode = amode + R_OK
c                   endif
c                   if        %scan('W':Authority) > 0
c                   eval      amode = amode + W_OK
c                   endif
c                   if        %scan('X':Authority) > 0
c                   eval      amode = amode + X_OK
c                   endif

c                   if        access(%trimr(Path): amode) < 0
c                   callp     EscErrno(errno)
c                   endif

c                   endif

c                   eval      *inlr = *on

 /DEFINE ERRNO_LOAD_PROCEDURE
```

```
/COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

3.3. Changing permissions on an existing IFS Object

The API that allows us to change the permissions of an IFS object is called "chmod", which stands for "change mode". Here's the C-language prototype for chmod(), along with my RPG equivalent:

```
int chmod(const char *path, mode_t mode)

D chmod           PR            10I 0 ExtProc('chmod')
D   path                          *   Value options(*string)
D   mode                        10U 0 Value
```

The "mode" parameter works exactly the same as the "mode" parameter on the open() API. You use the same bit-flags to assign permissions to the "owner", the "group" and "others".

The difference between chmod() and open() is that chmod() does not open or create a new file, but instead changes the access permissions on an existing IFS object.

For example, let's say that you had already run the example program from chapter 2 called "CH2RAWDTA", and you know that it wrote an object to disk called "littlepgm.com". But now, you decided that you didn't want Bill from Accounting to be able to download your program! Since it's already there, you'd want to remove "read" permissions from the object.

To do that, you'd do something like this:

```
c                   if        chmod('/ifstest/littlepgm.com':
c                                 S_IRUSR + S_IWUSR + S_IXUSR) < 0
c                   callp     EscErrno(errno)
c                   endif
```

We assigned Read, Write and Execute authority to the file's owner, but we gave no authority to "the group" or to "others", so, Bill won't be able to read it.

3.4. Example of changing an IFS objects permissions

To demonstrate the use of the chmod() API, we'll create a simple command that you can use to change the permissions on an IFS object.

Our command will need to know the path name of the IFS object, and the permissions to be assigned for the Owner, the Group and for everyone else. Our command source will look like this:

```
CMD         PROMPT('Change File Mode')
PARM        KWD(OBJ) TYPE(*CHAR) LEN(640) MIN(1) +
              PROMPT('Object')
PARM        KWD(USER) TYPE(*CHAR) LEN(10) RSTD(*YES) +
              DFT(*NONE) VALUES(*NONE *R *RW *RX *RWX +
              *W *WX *X) PROMPT('Owner permissions')
```

```
         PARM       KWD(GROUP) TYPE(*CHAR) LEN(10) RSTD(*YES) +
                      DFT(*NONE) VALUES(*NONE *R *RW *RX *RWX +
                      *W *WX *X) PROMPT('Group Permissions')
         PARM       KWD(OTHER) TYPE(*CHAR) LEN(10) RSTD(*YES) +
                      DFT(*NONE) VALUES(*NONE *R *RW *RX *RWX +
                      *W *WX *X) PROMPT('Others Permissions')
```

And the program that processes the command will look like this. Compile it, run it, laugh, cry, let it become a part of your being.

```
 * CH3PERM: Example changing an IFS object's permissions
 *  (From Chap 3)
 *
 * To compile:
 *   CRTBNDRPG CH3PERM SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D Path            S            640A
D UserPerm        S             10A
D GroupPerm       S             10A
D OtherPerm       S             10A
D Mode            S             10I 0

 ** Warning:  call this program from the command.  If you call
 **       it directly, because "Path" is larger than 32 bytes.
 **       See http://faq.midrange.com/data/cache/70.html
 **

C     *entry        plist
c                   parm                    Path
c                   parm                    UserPerm
c                   parm                    GroupPerm
c                   parm                    OtherPerm

c                   eval      Mode = 0

C* Calculate desired user permissions:
c                   if        %scan('R': UserPerm) > 0
c                   eval      Mode = Mode + S_IRUSR
c                   endif
c                   if        %scan('W': UserPerm) > 0
c                   eval      Mode = Mode + S_IWUSR
c                   endif
c                   if        %scan('X': UserPerm) > 0
c                   eval      Mode = Mode + S_IXUSR
c                   endif

C* Calculate desired group permissions:
c                   if        %scan('R': GroupPerm) > 0
```

```
c                   eval      Mode = Mode + S_IRGRP
c                   endif
c                   if        %scan('W': GroupPerm) > 0
c                   eval      Mode = Mode + S_IWGRP
c                   endif
c                   if        %scan('X': GroupPerm) > 0
c                   eval      Mode = Mode + S_IXGRP
c                   endif

C* Calculate desired permissions for everyone else:
c                   if        %scan('R': OtherPerm) > 0
c                   eval      Mode = Mode + S_IROTH
c                   endif
c                   if        %scan('W': OtherPerm) > 0
c                   eval      Mode = Mode + S_IWOTH
c                   endif
c                   if        %scan('X': OtherPerm) > 0
c                   eval      Mode = Mode + S_IXOTH
c                   endif

C* Change the file's access mode:
c                   if        chmod(%trimr(path): Mode) < 0
c                   callp     die(%str(strerror(errno)))
c                   endif

c                   eval      *inlr = *on

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

3.5. Retrieving Stream File Stats

Sometimes its useful to be able to look up information about a stream file. Information such as the file's size, access permissions, and the time it was last modified are all available using the stat() API.

Here is the C-language prototype for the stat() API:

```
int❶ stat(const char *path❷, struct stat *buf❸)
```

❶ The return value is an integer. Possible values are 0 which indicate success, or -1 if an error occurred.

❷ This is the path of the IFS object that you wish to get information about. This argument is a C-style string, so we use the options(*string) keyword in RPG so that the system will convert it to C's format for us.

❸ Here it wants a pointer to a stat data structure. This will be easy code in the prototype: we just define it as a pointer. However, we will also need to create a data structure in the same format as the stat data structure in C, which I'll explain below.

Here is the corresponding RPG prototype:

```
D stat            PR            10I 0 ExtProc('stat')
D   path                          *   value options(*string)
D   buf                           *   value
```

In C, when you define a data structure, you first define the layout of the data structure. In this layout you list the subfields and their data types. Then, you declare variables that use this layout.

So, in that C-language prototype above, it tells us that *buf* is a pointer to any variable which uses the "stat layout".

Here is the definition for "struct stat" (the "stat layout") from the C language header member:

```
struct stat {
 mode_t          st_mode;       /* File mode                       */
 ino_t           st_ino;        /* File serial number              */
 nlink_t         st_nlink;      /* Number of links                 */
 uid_t           st_uid;        /* User ID of the owner of file    */
 gid_t           st_gid;        /* Group ID of the group of file   */
 off_t           st_size;       /* For regular files, the file
                                 * size in bytes                   */
 time_t          st_atime;      /* Time of last access             */
 time_t          st_mtime;      /* Time of last data modification  */
 time_t          st_ctime;      /* Time of last file status change */
 dev_t           st_dev;        /* ID of device containing file    */
 size_t          st_blksize;    /* Size of a block of the file     */
 unsigned long   st_allocsize;  /* Allocation size of the file     */
 qp0l_objtype_t st_objtype;     /* AS/400 object type              */
 unsigned short st_codepage;    /* Object data codepage            */
 char            st_reserved1[62]; /* Reserved                     */
 unsigned int    st_ino_gen_id  /* file serial number generation id */
};
```

The first line simply tells us that this is a structure definition called "stat".

The remaining lines, except for the last one, are a simple list of subfields, and their data types. For example, the last subfield in this data structure is called "st_ino_gen_id" and it is an unsigned integer.

To duplicate this in RPG, what we'll do is create a normal RPG data structure. But then, we'll base that structure on a pointer. Then, when we want to use the structure in our programs, we'll simply use the LIKE() keyword to declare character strings of the same size, and move the pointer so that we can reference the subfields. (Don't worry, if that's not clear to you, yet. I'll give you an example shortly that will help you understand.)

Also, you may have noticed that both the structure definition and the API are called "stat". That's not a problem in C since structure definitions use a separate name space from procedure calls. However, it is a problem in RPG. So, we'll call our data structure "statds" instead of "stat". That way, the name won't conflict with the name of the API.

Here is the RPG definition of the stat data structure:

```
D p_statds        S               *
D statds          DS                  BASED(p_statds)
D  st_mode                       10U 0
D  st_ino                        10U 0
D  st_nlink                       5U 0
D  st_pad                         2A
```

```
D  st_uid                       10U 0
D  st_gid                       10U 0
D  st_size                      10I 0
D  st_atime                     10I 0
D  st_mtime                     10I 0
D  st_ctime                     10I 0
D  st_dev                       10U 0
D  st_blksize                   10U 0
D  st_allocsize                 10U 0
D  st_objtype                   12A
D  st_codepage                   5U 0
D  st_reserved1                 62A
D  st_ino_gen_id                10U 0
```

Now, when we call stat() in RPG, we'll do something like this:

```
D MyStat          S                   like(statds)
D MySize          S             10I 0

 * get stat info into "MyStat":
c                   if        stat('/path/to/file.txt':
c                                   %addr(mystat)) < 0
c                   callp     EscErrno(errno)
c                   endif

 * move structure to overlay the "mystat" info:
c                   eval      p_statds = %addr(mystat)

 * read the file's size into MySize:
c                   eval      MySize = st_size
```

3.6. Adding a *SAME option to the permission changer

In our last sample project, we created a program that assigned new access permissions to an IFS object. Now, let's update that example to allow "*SAME" to be specified.

This would, for example, allow you to change the permissions for the owner, without affecting the permissions for the group or anyone else.

To do that, what we'll do is retrieve the file's mode using stat(). Then, we'll preserve the bits from the original mode where *SAME is specified.

Here's the updated command source:

```
CMD        PROMPT('Change File Mode')
PARM       KWD(OBJ) TYPE(*CHAR) LEN(640) MIN(1) +
             PROMPT('Object')
PARM       KWD(USER) TYPE(*CHAR) LEN(10) RSTD(*YES) +
             DFT(*SAME) VALUES(*NONE *R *RW *RX *RWX +
             *W *WX *X *SAME) PROMPT('Owner permissions')
```

```
         PARM       KWD(GROUP) TYPE(*CHAR) LEN(10) RSTD(*YES) +
                      DFT(*SAME) VALUES(*NONE *R *RW *RX *RWX +
                      *W *WX *X *SAME) PROMPT('Group Permissions')
         PARM       KWD(OTHER) TYPE(*CHAR) LEN(10) RSTD(*YES) +
                      DFT(*SAME) VALUES(*NONE *R *RW *RX *RWX +
                      *W *WX *X *SAME) PROMPT('Others Permissions')
```

And here's the new RPG source:

```
 * CH3PERM2: Example of changing permissions of an IFS object w/*SAME
 *  (From Chap 3)
 *
 * To compile:
 *   CRTBNDRPG CH3PERM2 SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D Path            S            640A
D UserPerm        S             10A
D GroupPerm       S             10A
D OtherPerm       S             10A
D MyStat          S                   like(statds)

D                 DS
D Mode                          10I 0
D CharMode3                      1A   overlay(Mode:3)
D CharMode4                      1A   overlay(Mode:4)

 ** Warning:  call this program from the command.  If you call
 **       it directly, because "Path" is larger than 32 bytes.
 **       See http://faq.midrange.com/data/cache/70.html
 **

C     *entry        plist
c                   parm                    Path
c                   parm                    UserPerm
c                   parm                    GroupPerm
c                   parm                    OtherPerm

C* Retrieve current file mode:
c                   if        stat(%trimr(path): %addr(mystat)) < 0
c                   callp     die(%str(strerror(errno)))
c                   endif

c                   eval      p_statds = %addr(mystat)
c                   eval      Mode = st_mode

C* Calculate desired user permissions:
c                   if        UserPerm <> '*SAME'
```

```
c                   bitoff    x'FF'         CharMode3
c                   bitoff    x'C0'         CharMode4

c                   if        %scan('R': UserPerm) > 0
c                   eval      Mode = Mode + S_IRUSR
c                   endif
c                   if        %scan('W': UserPerm) > 0
c                   eval      Mode = Mode + S_IWUSR
c                   endif
c                   if        %scan('X': UserPerm) > 0
c                   eval      Mode = Mode + S_IXUSR
c                   endif

c                   endif

C* Calculate desired group permissions:
c                   if        GroupPerm <> '*SAME'

c                   bitoff    x'38'         CharMode4

c                   if        %scan('R': GroupPerm) > 0
c                   eval      Mode = Mode + S_IRGRP
c                   endif
c                   if        %scan('W': GroupPerm) > 0
c                   eval      Mode = Mode + S_IWGRP
c                   endif
c                   if        %scan('X': GroupPerm) > 0
c                   eval      Mode = Mode + S_IXGRP
c                   endif

c                   endif

C* Calculate desired permissions for everyone else:
c                   if        OtherPerm <> '*SAME'

c                   bitoff    x'07'         CharMode4

c                   if        %scan('R': OtherPerm) > 0
c                   eval      Mode = Mode + S_IROTH
c                   endif
c                   if        %scan('W': OtherPerm) > 0
c                   eval      Mode = Mode + S_IWOTH
c                   endif
c                   if        %scan('X': OtherPerm) > 0
c                   eval      Mode = Mode + S_IXOTH
c                   endif

c                   endif

C* Change the file's access mode:
c                   if        chmod(%trimr(path): Mode) < 0
c                   callp     die(%str(strerror(errno)))
c                   endif
```

```
c                   eval      *inlr = *on

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Note: We're using RPG's BITOFF operation to turn the bits from the original off before we set them. This ensures that they end up with the correct values, even if they were previously set to something.

3.7. Deleting IFS objects

You may have noticed that the IBM commands for working with the IFS frequently use the term "link." For example, the WRKLNK command ("Work with Links") is used to browse the IFS. The RMVLNK command ("Remove Link") is used to delete stream files.

You may be wondering "What's a link?"

To understand this, you need to make the distinction between the data that's stored in the file, and the file's name which shows up in the directory.

The data itself is called the "file". The name which you find in a directory is just a way of referring to that data. In essence, it's a link to the file.

In fact, it's possible to have more than one link to the same data. When that happens, the same file data may appear in more than one directory and/or under more than one different file name. Each one is considered a separate link, even though it's the same data.

When you remove a link to a stream file, the system will first remove the file name from the directory, and then it will check if this was the last link to the file. If the deleted link was the last link, the file's data will also be removed.

The API to delete a link is called "unlink()". And it's C-language prototype looks like this:

```
int unlink(const char *path)
```

Can't be much simpler than that, can it? It accepts only one parameter, and it's a null-terminated character string. It returns an integer, which will be a 0 if the API was successful or a -1 if it failed.

Here's the corresponding RPG prototype:

```
D unlink          PR            10I 0 ExtProc('unlink')
D   path                          *   Value options(*string)
```

So, if you wanted to delete the stream file called "littlepgm.com", you'd write code that looks like this:

```
c                   if        unlink('/ifstest/littlepgm.com') < 0
c                   callp     EscErrno(errno)
c                   endif
```

3.8. Renaming IFS objects

It's easy to change the name of an IFS object. The system gives us a rename() API which can be called to do that.

The rename() API operates on the file's link, rather than on the file's data. This means that doing a rename() is much more efficient than copying the file, and deleting the old copy! All it has to change is the link, not the data.

Here is both the C and RPG prototypes for the rename() API. Once again, I won't bore you with the details of translating from C to RPG, since it is very straightforward on this API.

```
 int rename(const char *old, const char *new)

D rename          PR            10I 0 ExtProc('Qp0lRenameKeep')
D   old                           *   Value options(*string)
D   new                           *   Value options(*string)
```

One detail that should be noted is that the external procedure name isn't "rename" as you might expect, it's "Qp0lRenameKeep". The reason for this is that there are actually two different rename APIs. The difference between them is what happens when the "new name" is already the name of another IFS object.

If you're calling Qp0lRenameKeep, and the "new name" already exists, the system will return an error, protecting you from accidentally deleting the existing file.

If, instead, you call Qp0lRenameUnlink, the system will unlink the existing filename, and then proceed with the renaming. I never use this option because I feel it's safer, and more intuitive, to use Qp0lRenameKeep. I can always call unlink() beforehand if I really want to unlink an existing file.

Calling rename() from an RPG program is easy. Just do something like this:

```
c                   if        rename('/ifstest/oldsmellyfile.dat':
c                                    '/ifstest/newshinyfile.dat') < 0
c                   callp     EscErrno(errno)
c                   endif
```

3.9. Example of renaming and deleting IFS objects

To demonstrate the use of the unlink() and rename() APIs, here's a sample program that can be used to either rename or delete a stream file.

What it does is ask for two parameters. The first is the current pathname of an IFS object. The second parameter is either the new pathname, or the special value *DELETE.

If *DELETE is specified, our program will bring up a window asking for confirmation before actually calling the unlink() API.

Here's the command source:

```
            CMD         PROMPT('Rename or Delete an IFS object')
            PARM        KWD(OLD) TYPE(*CHAR) LEN(640) +
                          PROMPT('Original (OLD) Object Name')
            PARM        KWD(NEW) TYPE(*CHAR) LEN(640) +
```

```
                    CHOICE('Character or *DELETE') +
                    PROMPT('New Object Name or *DELETE')
```

Here's the DDS for the Window that it pops up:

```
A                                      DSPSIZ(24 80 *DS3)
A          R DUMMYREC
A                                      ASSUME
A                                  1   2' '
A          R RENDELS1
A                                      WINDOW(9 30 6 20)
A                                  2   1'File:'
A            SCFILE        14   O  2   7
A                                  3   1'Size:'
A            SCSIZE        10Y 0O  3   7EDTCDE(L)
A                                  5   1'Really? Delete it?'
A            SCREALLY       1   I  5 20
```

Here's the RPG code that makes it all work:

```
 * CH3RENDEL: Example of deleting/renaming objects in the IFS
 *  (From Chap 3)
 *
 * To compile:
 *   CRTBNDRPG CH3RENDEL SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

FCH3RENDELSCF   E             WORKSTN

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D upper           C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
D lower           C                   'abcdefghijklmnopqrstuvwxyz'

D Path            S            640A
D NewPath         S            640A
D LowerNewPath    S            640A
D MyStat          S                   like(statds)
D Len             S             10I 0
D Pos             S             10I 0

 ** Warning:  call this program from the command.  If you call
 **       it directly, because "Path" is larger than 32 bytes.
 **       See http://faq.midrange.com/data/cache/70.html
 **

C     *entry        plist
c                   parm                    Path
c                   parm                    NewPath

c     upper:lower   xlate     NewPath       LowerNewPath
```

```
c                   if        LowerNewPath = '*delete'
c                   exsr      KillIt
c                   else
c                   exsr      NewIdentity
c                   endif

c                   eval      *inlr = *on


C*************************************************************
C* Kill off the file (Delete it from the IFS)
C*************************************************************
CSR   KillIt        begsr
C*-------------------------
C* Retrieve current file stats:
c                   if        stat(%trimr(path): %addr(mystat)) < 0
c                   callp     die(%str(strerror(errno)))
c                   endif

C* Get file size from stats
c                   eval      p_statds = %addr(mystat)
c                   eval      scSize = st_size

C* Strip directory names from front of pathname:
c                   eval      Len = %len(%trimr(path))
c                   eval      Pos = Len
c                   dow       Pos > 0
c                   if        %subst(path:Pos:1) = '/'
c                   leave
c                   endif
c                   eval      Pos = Pos -1
c                   enddo
c                   if        Pos<Len and %subst(path:Pos:1) = '/'
c                   eval      scFile = %subst(path:Pos+1)
c                   else
c                   eval      scFile = path
c                   endif

C* Ask user if he/she REALLY wants to delete it?
c                   exfmt     RENDELS1

C* Then ignore his choice and delete it anyway.
C* (just kidding)
c                   if        scReally = 'Y'
c                   if        unlink(%trimr(path)) < 0
c                   callp     die(%str(strerror(errno)))
c                   endif
c                   endif
C*-------------------------
CSR                 endsr
```

```
C*************************************************************
C* Give the file a new identity.  A new purpose in life!
C* (okay, rename it...)
C*************************************************************
CSR   NewIdentity   begsr
C*-------------------------
c                   if        rename(%trimr(Path): %trimr(NewPath))<0
c                   callp     die(%str(strerror(errno)))
c                   endif
C*-------------------------
CSR                 endsr


 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Chapter 4. Accessing stream files randomly

In chapter 2, we wrote data to the file at the start, and then read it from the start of the file. We always read the file sequentially. In this chapter, I'll show you how to read the file randomly.

4.1. Positioning to a given point in the file

In a standard physical file (one without key fields) you can position by record number using the SETLL and SETGT operations. This makes it possible to access a file "randomly" (or, in other words, you don't have to read through the file sequentially, you can "jump around".)

For a stream file, to read the file randomly, you use the "lseek()" API. However, since stream files are not organized into records by themselves, the lseek() API doesn't seek to a record number. Instead, you give it an "offset" which indicates a number of bytes, rather than records, to jump to.

Here is the C-language prototype for the lseek() API, followed by the corresponding RPG prototype:

```
off_t lseek(int fildes, off_t offset, int whence)

❶D lseek           PR            10I 0 ExtProc('lseek')
❷D   fildes                      10I 0 value
❸D   offset                      10I 0 value
❹D   whence                      10I 0 value
```

❶ The return value will be the new offset from the beginning of the file if lseek() was successful, otherwise it will be -1, and errno will be set.

❷ This is just the file descriptor of the stream file that you've opened.

❸ Here you put the offset. The offset is the number of bytes to move forward in the file from the point that's specified in the next parameter. If you wish to move backward instead of forward, you can specify a negative number. You can also specify zero if you want to move to exactly the point given in the next parameter.

❹ The "whence" parameter specifies where you want to move to, or start counting your offset from. It can be one of three named constants:

- **`SEEK_SET`** means that the offset will be from the beginning of the file. An offset of 0 would be the very first byte in the file.
- **`SEEK_CUR`** means that the offset will be from the current position in the file. For example, if you wanted to re-read the last 5 bytes, you could code **`SEEK_CUR`** with an offset of -5
- **`SEEK_END`** means that the offset will be from the end of the file.

Now that I've told you about the named constants, it'd probably be a good idea to add them to our /copy member, eh?

```
D SEEK_SET        C                   CONST(0)
D SEEK_CUR        C                   CONST(1)
D SEEK_END        C                   CONST(2)
```

Here's a sample of jumping to the end of the file in RPG:

```
c                   if        lseek(fd: 0: SEEK_END) < 0
c                   callp     EscErrno(errno)
c                   endif
```

How about jumping to the 157th byte from the start of the file?

```
c                   if        lseek(fd: 157: SEEK_SET) < 0
c                   callp     EscErrno(errno)
c                   endif
```

Or, if we're already at byte 157, we could easily seek forward to byte 167, like this:

```
c                   if        lseek(fd: 10: SEEK_CUR) < 0
c                   callp     EscErrno(errno)
c                   endif
```

4.2. Example of using lseek() to jump around the file

Okay, here's a real example of using lseek() to jump around. Look over the code, and see if you can tell what it's going to do, then compile and run it.

```
 * CH4RANDOM: Example of random access to an IFS object
 *  (From Chap 3)
 *
 * To compile:
 *   CRTBNDRPG CH3RANDOM SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D fd              S             10I 0
D err             S             10I 0
D wrdata          S             48A
D rddata          S             22A
D ShowMe          S             48A   varying

c                   eval      fd = open('/ifstest/ch2_test.dat':
c                                    O_WRONLY+O_CREAT+O_TRUNC:
c                                    S_IRUSR+S_IWUSR+S_IRGRP)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

C* Write some data
```

```
c                   eval      wrdata = 'THE QUICK BROWN FOX JUMP' +
c                                      'ED OVER THE LAZY GIRAFFE'
c                   if        write(fd: %addr(wrdata): %size(wrdata))<1
c                   eval      err = errno
c                   callp     close(fd)
c                   callp     die('write(): ' + %str(strerror(errno)))
c                   endif

c                   callp     close(fd)

c                   eval      fd = open('/ifstest/ch2_test.dat':
c                                    O_RDONLY)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   eval      %len(ShowMe) = 0

C* Read the first 16 bytes
c                   callp     read(fd: %addr(rddata): 16)
c                   eval      ShowMe = ShowMe + %subst(rddata:1:16)

C* Jump to byte 41 of the file
C* and read 7 bytes
c                   if        lseek(fd: 41: SEEK_SET) < 0
c                   callp     die('lseek(): ' + %str(strerror(errno)))
c                   endif
c                   callp     read(fd: %addr(rddata): 7)
c                   eval      ShowMe = ShowMe + %subst(rddata:1:7)

C* Jump to byte 19 of the file
C* and read 22 bytes
c                   if        lseek(fd: 19: SEEK_SET) < 0
c                   callp     die('lseek(): ' + %str(strerror(errno)))
c                   endif
c                   callp     read(fd: %addr(rddata): 22)
c                   eval      ShowMe = ShowMe + %subst(rddata:1:22)

C* Jump to byte 16 of the file
C* and read 3 bytes
c                   if        lseek(fd: 16: SEEK_SET) < 0
c                   callp     die('lseek(): ' + %str(strerror(errno)))
c                   endif
c                   callp     read(fd: %addr(rddata): 3)
c                   eval      ShowMe = ShowMe + %subst(rddata:1:3)

c                   callp     close(fd)

C* Show what we read
c                   dsply                   ShowMe

c                   eval      *inlr = *on
```

```
/DEFINE ERRNO_LOAD_PROCEDURE
/COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

4.3. Organizing a stream file into records

Stream files are not record-based, but rather are just a collection of bytes. However, when you're working with records in your favorite physical file, what are they? Nothing more than a fixed-length bunch of bytes, right?

Theoretically, if we wrote fixed-length chunks of data to a stream file, and called those chunks "records", then we could use lseek() to jump to the start of each record, and read it just like a non-keyed physical file!

But, why would you ever do that? After all, the DB2/400 physical files are much more efficient, aren't they? Well, yes. But, let's say our file was going to be read directly by a PC program... aha! The PC program probably doesn't understand how to access the physical file, but it sure knows how to read a stream file!

Calculating the offset where a record in a stream file starts should be pretty easy. If we know, for example, that a record is 68 bytes long, and we want to jump to the 10th record in a file, all we have to do is multiply, right? Well, not exactly. If the 10th record is at position 680, then that would mean that the first record would be at 1 x 68, or position 68. But, actually, the first record should be at offset 0, since that's the start of the file.

So, the formula for finding the offset for a start of a record is always: Offset = (RecordNo - 1) x RecordLength

4.4. Calculating number of records in a file

Figuring out how many records are in a stream file that has been organized into records is also quite simple. We just take the size of the file and divide it by the record length.

To simplify this slightly, I'm going to introduce another new API call. The fstat() API.

Here are the C and RPG prototypes:

```
int fstat(int fildes, struct stat *buf)
```

```
D fstat           PR            10I 0 ExtProc('fstat')
D   fildes                      10I 0 value
D   buf                           *   value
```

I'm not going to explain the details of fstat(), since it's exactly like stat(), which we covered earlier.

The only difference between fstat() and the stat() API that we discussed in chapter 3, is that the fstat() API operates on a file descriptor, whereas the stat() API operates on a path name.

In other words, you use fstat() on a file that you've already opened, and you use stat() on a file that you don't need to open.

Once we've called fstat(), we can use the st_size subfield of the statds data structure to find the size of the stream file, and then divide it by our record length to find out how many records we have, like this:

```
c                   if        fstat(fd: %addr(mystat)) < 0
c                   callp     die('fstat(): ' + %str(strerror(err)))
```

```
c                   endif

c                   eval      p_statds = %addr(mystat)
c                   eval      numrec = st_size / record_len
```

4.5. Example of reading/writing/updating records in a stream file

This example will create a stream file that contains some demonstration records. It will then show you how to look up the records, update some of the information in them, and even search through the file to find them.

Here's the code. Once again, read the code over and see if you can figure out what it does. Then, run the program and see if you're right!

```
 * CH4FIXED: Example of fixed-length records in an IFS file
 *  (From Chap 4)
 *
 * To compile:
 *   CRTBNDRPG CH4FIXED SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D dsRecord        DS
D   PartNo                      10S 0
D   Quantity                    10I 0
D   UnitOfMeas                  3A
D   Price                       7P 2
D   Description                 50A

D fd              S             10I 0
D err             S             10I 0
D MyStat          S                   like(statds)
D recno           S             10I 0
D NumRec          S             10I 0
D SaveRec         S             10I 0
D Pos             S             10I 0

c                   exsr      MakeFile

c                   eval      fd = open('/ifstest/ch4_records': O_RDWR)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   exsr      UpdateEx
c                   exsr      SearchEx
```

```
c                   callp     close(fd)

c                   eval      *inlr = *on


C*****************************************************************
C* This creates a file in the IFS containing fixed-length
C* records.
C*****************************************************************
CSR   MakeFile      begsr
C*------------------------
c                   eval      fd = open('/ifstest/ch4_records':
c                                       O_WRONLY+O_CREAT+O_TRUNC:
c                                       S_IRUSR+S_IWUSR+S_IRGRP)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   eval      PartNo = 5001
c                   eval      Quantity = 14
c                   eval      UnitOfMeas = 'BOX'
c                   eval      Price = 7.95
c                   eval      Description = 'BLUE WIDGETS'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

c                   eval      PartNo = 5002
c                   eval      Quantity = 6
c                   eval      UnitOfMeas = 'BOX'
c                   eval      Price = 3.95
c                   eval      Description = 'RAINBOW SUSPENDERS'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

c                   eval      PartNo = 5003
c                   eval      Quantity = 19
c                   eval      UnitOfMeas = 'SEA'
c                   eval      Price = 29.95
c                   eval      Description = 'RED BICYCLE SEATS'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

c                   eval      PartNo = 5004
c                   eval      Quantity = 8
c                   eval      UnitOfMeas = 'ITM'
c                   eval      Price = 93512.80
c                   eval      Description = 'REALLY EXPENSIVE ITEMS'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

c                   eval      PartNo = 5005
c                   eval      Quantity = 414
c                   eval      UnitOfMeas = 'BAT'
c                   eval      Price = 11.41
c                   eval      Description = 'BATS IN THE BELFRY'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))
```

```
c                   eval      PartNo = 5006
c                   eval      Quantity = 125
c                   eval      UnitOfMeas = 'BOX'
c                   eval      Price = 1.23
c                   eval      Description = 'KATES OLD SHOES'
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

c                   callp     close(fd)
C*------------------------
CSR                 endsr


C*************************************************************
C* This demonstrates updating our fixed-length record file:
C*************************************************************
CSR   UpdateEx      begsr
C*------------------------
C* Someone bought a box of suspenders, and we want to change
C* the quantity:

C* The suspenders are in record number 2, so get them now:
c                   eval      recno = 2
c                   eval      pos = %size(dsRecord) * (recno-1)
c                   callp     lseek(fd: pos: SEEK_SET)
c                   callp     read(fd: %addr(dsRecord):%size(dsRecord))

c                   eval      Quantity = Quantity - 1
c                   callp     lseek(fd: pos: SEEK_SET)
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))

C* Kate's shoes need to move faster, put them on sale!
c                   eval      recno = 6
c                   eval      pos = %size(dsRecord) * (recno-1)
c                   callp     lseek(fd: pos: SEEK_SET)
c                   callp     read(fd: %addr(dsRecord):%size(dsRecord))

c                   eval      Price = 0.25
c                   callp     lseek(fd: pos: SEEK_SET)
c                   callp     write(fd:%addr(dsRecord):%size(dsRecord))
C*------------------------
CSR                 endsr


C*************************************************************
C* This demonstrates searching for a record in the file:
C*************************************************************
CSR   SearchEx      begsr
C*------------------------
C* GASP! I can't remember the record number for Bats!
C* The part number was 5005, let's search for it
c                   if        fstat(fd: %addr(mystat)) < 0
c                   eval      err = errno
```

```
c                   callp     close(fd)
c                   callp     die('fstat(): ' + %str(strerror(err)))
c                   endif

c                   eval      p_statds = %addr(mystat)
c                   eval      numrec = st_size / %size(dsRecord)
c                   eval      SaveRec = -1

c                   for       recno = 1 to numrec
c                   eval      pos = %size(dsRecord) * (recno-1)
c                   callp     lseek(fd: pos: SEEK_SET)
c                   callp     read(fd: %addr(PartNo): %size(PartNo))
c                   if        PartNo = 5005
c                   eval      SaveRec = recno
c                   leave
c                   endif
c                   endfor

c                   if        SaveRec = -1
c                   callp     close(fd)
c                   callp     die('Part no 5005 not found!')
c                   endif
C*------------------------
CSR                 endsr


 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Chapter 5. Text files

5.1. How do text files work?

As I mentioned at the start of this eBook, a stream file is simply a long string of bytes. How that string of bytes is used is up to the software that reads and writes the file.

One of the most common ways of organizing the data in a stream file is called a "text file." Text files are made up of human-readable text organized into variable-length records called "lines." Each line is intended to be printed on a single row of a display or a page of paper.

Text files are important to use because they are so widely used. This list is just a sample of the many places text files are used:

- Nearly all printers accept text files as the their primary input format.
- All source code for PC programming languages, including C, C++, Visual Basic, Java, Perl and many others, is stored in a text file.
- Text files are the basis for more advanced file formats. HTML, XML, CSV and many other formats are text files with additional formatting added.
- All Internet e-mail is transmitted in text format. Even when you send pictures, movies or sound, they are converted to something that follows the rules of a text file before sending, and converted back after they’re received.
- Nearly every major operating system comes with tools that work with text files. Windows comes with the "Notepad" program that edits text files. MS-DOS came with the "edit" and "edlin" programs. Unix comes with many tools for working with text, vi, ex, ed, grep, sed, awk, and more! Our familiar OS/400 even comes with commands such as "EDTF" and "DSPF" which are used to edit and display text files.

In order to make it easy to store variable-length lines, a special character is written to signify the end of a line. This character is called, appropriately, "end-of-line character" or more commonly the "new line character."

Unfortunately, not everyone agrees on what character should be used as the new line character. On Unix systems, they use the "LF" (line feed) character. On Macintosh systems, the "CR" (carriage return) character is used to indicate a new line, and for Microsoft Windows, they use the two characters in combination, CR followed by LF.

In our examples, we will use the Windows convention of "CRLF" since it is the most widely used. Changing your code to use just CR or just LF is easy enough to do if you need to work with one of the other formats.

5.2. Writing text data to a stream file

As mentioned previously, some human-readable text, followed by the CR and LF characters, will be viewed as a line of text. Consider the following code snippet:

```
D CRLF            C                   const(x'0d25')
D text            S             14a

c                   eval      text = 'Hello there.' + CRLF
```

If I whip out my handy-dandy EBCDIC chart, I see that x'0D' is the EBCDIC code for carriage return, and x'25' is the EBCDIC code for line feed. Therefore, the variable called "text" above contains one line of text.

Consider this, more complicated, example:

```
D CRLF            C                   const(x'0d25')
D text            S            500

c                   eval      text = 'Hello there.' + CRLF + CRLF +
c                                'Nice day for a cruise.' + CRLF +
c                                'Maybe I"ll buy a yacht!'
```

Think about this: How many lines are stored in the variable called "text", above? Be careful before you answer, it's a trick question.

The answer is three and a half. Why? The first CRLF ends the first line, so the first line will read "Hello there!" in the text file. The next line ends when we encounter the next CRLF, which happens right away! That means the second line is blank. The third line says "Nice day for a cruise." and ends with another CRLF. The rest of the text, however, does not end with a CRLF. It's just part of a line.

All we have to do to put that text into a text file is call write(). This code would do the job:

```
c                   callp     write(fd: %addr(text):
c                                   %len(%trimr(text)) )
```

One small problem: If we let it be written like this, our text file would be invalid. Why? Because, remember, we left off the CRLF at the end of the last line.

To make sure that we don't make that mistake again, why not create some helper routines? What we'll do is create a service program, called IFSTEXTR4 and put our text file routines into it. Then, we'll use it for all the text files we write.

The routine to write a line will be simple. All it needs to do is accept parameters (just like write() does) for the descriptor, the text, and the text length. Then, we'll just stick a CRLF into the file after that.

Here's our IFSTEXTR4 service program, with the writeline() procedure:

```
 ** Service program to assist in creating TEXT files in the IFS
 **

H NOMAIN OPTION(*NOSHOWCPY: *SRCSTMT)

D/copy ifsebook/qrpglesrc,ifsio_h
D/copy ifsebook/qrpglesrc,ifstext_h

 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  The concept here is very simple:
 *     1) Write the data passed to us into the stream file.
 *     2) Add the end of line characters.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P writeline       B                   export
D writeline       PI            10I 0
D   fd                          10I 0 value
```

```
D   text                          *   value
D   len                         10I 0 value

D rc1             S             10I 0
D rc2             S             10I 0
D eol             S              2A

C* write the text provided
c                   if        len > 0
c                   eval      rc1 = write(fd: text: len)
c                   if        rc1 < 1
c                   return    rc1
c                   endif
c                   endif

C* then add the end-of-line chars
c                   eval      eol = x'0d25'
c                   eval      rc2 = write(fd: %addr(eol): 2)
c                   if        rc2 < 1
c                   return    rc2
c                   endif

c                   return    rc1 + rc2
P                 E
```

And then, we'll also need a prototype. That's what the /copy line for the "IFSTEXT_H" member is for. We'll put our prototypes in that member, so they can be easily accessed from other programs.

Here's the IFSTEXT_H member, so far:

```
 /if defined(IFSTEXT_H)
 /eof
 /endif

 /define IFSTEXT_H

D writeline       PR            10I 0
D   fd                          10I 0 value
D   text                          *   value
D   len                         10I 0 value
```

Don't compile it, yet! We're also going to add a routine for reading text files.

5.3. Reading text data from a stream file

We know what a line of text looks like now. We also know how to write lines to disk. What we don't know, yet, is how to read them.

The big difference between reading lines and writing lines is that when you write the data, you already know how long the line needs to be. But when you read, you don't. You won't know how long it is until you've found the CRLF sequence in the text!

Now, we could solve that by reading one byte from disk at a time, until one of them turned out to be the new line sequence, but that would not run efficiently, because disk hardware is designed to read from disk in larger chunks. So, what we'll do is read a whole buffer of data, and then parse that data looking for our new line sequence.

We'll save any characters in the buffer that occur after the new line, so that we can use them as the start of our next line of text.

The RPG code that I came up with looks like this:

```
P readline        B                   export
D readline        PI            10I 0
D   fd                          10I 0 value
D   text                          *   value
D   maxlen                      10I 0 value

D rdbuf           S           1024A   static
D rdpos           S             10I 0 static
D rdlen           S             10I 0 static

D p_retstr        S               *
D RetStr          S           32766A   based(p_retstr)
D len             S             10I 0

c                   eval      len = 0
c                   eval      p_retstr = text
c                   eval      %subst(RetStr:1:MaxLen) = *blanks

c                   dow       1 = 1

C* Load the buffer
c                   if        rdpos>=rdlen
c                   eval      rdpos = 0
c                   eval      rdlen=read(fd:%addr(rdbuf):%size(rdbuf))

c                   if        rdlen < 1
c                   return    -1
c                   endif
c                   endif

C* Is this the end of the line?
c                   eval      rdpos = rdpos + 1
c                   if        %subst(rdbuf:rdpos:1) = x'25'
c                   return    len
c                   endif

C* Otherwise, add it to the text string.
c                   if        %subst(rdbuf:rdpos:1) <> x'0d'
c                               and len<>maxlen
c                   eval      len = len + 1
c                   eval      %subst(retstr:len:1) =
```

```
c                                %subst(rdbuf:rdpos:1)
c                   endif

c                   enddo

c                   return    len
P                 E
```

Add that routine to the IFSTEXTR4 service program, and then add this code to the prototypes in the IFSTEXT_H member:

```
D readline        PR            10I 0
D   fd                          10I 0 value
D   text                          *   value
D   maxlen                      10I 0 value
```

One more thing: Because we're writing a service program, and we want our code to be as useful as possible, we're going to write binder source that tells the CRTSRVPGM how other programs can bind to us.

If you haven't done this before, don't worry about it. It's really, really simple. It's just a list of what gets exported. Here's the entire code of our binding source:

```
STRPGMEXP
  EXPORT SYMBOL(WRITELINE)
  EXPORT SYMBOL(READLINE)
ENDPGMEXP
```

See? Very simple. Put that code into a member called "IFSTEXTR4" in a source file called QSRVSRC. Now let's compile it:

```
CRTRPGMOD IFSTEXTR4 SRCFILE(IFSEBOOK/QRPGLESRC) DBGVIEW(*LIST)
CRTSRVPGM IFSTEXTR4 EXPORT(*SRCFILE) SRCFILE(IFSEBOOK/QSRVSRC) TEXT('IFS Text file service program')
```

Finally, to make it easy to compile the programs that use this service program, we'll create a binding directory. This only involves running two commands:

```
CRTBNDDIR IFSEBOOK/IFSTEXT TEXT('IFS Text binding directory')
ADDBNDDIRE BNDDIR(IFSEBOOK/IFSTEXT) OBJ((IFSTEXTR4))
```

5.4. Example of writing and reading text files

Here's an example that uses our new service program. What it does is it creates a text file. Then it brings up the OS/400 text file editor to let you make changes to it. Finally, it reads back the text file, and displays the first 52 bytes of each line using the DSPLY op-code.

```
* CH5TEXT: Example of creating/reading a text file in the IFS
```

```
 *  (From Chap 5)
 *
 * To compile:
 *   CRTBNDRPG CH5TEXT SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   len                         15P 5 const

D fd              S              10I 0
D line            S             100A
D len             S              10I 0
D msg             S              52A

c                   exsr      MakeFile
c                   exsr      EditFile
c                   exsr      ShowFile
c                   eval      *inlr = *on


C*************************************************************
C* Write some text to a text file
C*************************************************************
CSR   MakeFile      begsr
C*------------------------
c                   eval      fd = open('/ifstest/ch5_file.txt':
c                                   O_TRUNC+O_CREAT+O_WRONLY:
c                                   S_IWUSR+S_IRUSR+S_IRGRP+S_IROTH)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   eval      line = 'Dear Cousin,'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'I love the way you make' +
c                               ' cheese fondue.'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
```

```
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'Thank you for being so cheesy!'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'Sincerely,'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '     Richard M. Nixon'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   callp     close(fd)
C*------------------------
CSR                 endsr


C*************************************************************
C*  Call the OS/400 text editor, and let the user change the
C*  text around.
C*************************************************************
CSR   EditFile      begsr
C*------------------------
c                   callp     cmd('EDTF STMF("/ifstest/' +
c                                       'ch5_file.txt")': 200)
C*------------------------
CSR                 endsr


C*************************************************************
C*  Read file, line by line, and dsply what fits
C*  (DSPLY has a lousy 52-byte max... blech)
C*************************************************************
CSR   ShowFile      begsr
C*------------------------
c                   eval      fd = open('/ifstest/ch5_file.txt':
c                                 O_RDONLY)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   dow       readline(fd: %addr(line): %size(line))>=0
c                   eval      Msg = line
c     Msg           dsply
c                   enddo

c                   callp     close(fd)
```

```
c                   eval      Msg = 'Press ENTER to continue'
c                   dsply                   Msg
C*------------------------
CSR                 endsr


 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Try it out by compiling it and running it. Add some text and/or remove some text from the text file, and notice that the program will read it correctly.

5.5. Using code pages with text files

So, now we've created some text files, but there's one small problem. They're all in EBCDIC! This means that although we can read them using OS/400, they're utterly useless on the PC.

Now, we could call an API like QDCXLATE or iconv() to convert the text to ASCII before we write it to disk, but then our OS/400 programs wouldn't be able to work with it!

This problem can be easily solved, because the IFS allows us to assign a code page to our stream file. There are many code pages used to support all of the various characters used all around the world.

For our examples, we will use code page 37, which represents EBCDIC in the United States where I live, and code page 819 which represents ASCII here. If you're interested in using other code pages, please see the National Language Support manual which is published by IBM, and included in your Softcopy Library or Information Center.

The O_CODEPAGE flag on the open() API is used to assign a code page to a text file when the file is created.

So, when we want to assign a code page, we do this:

```
c                   callp     unlink('/ifstest/somefile.txt')

c                   eval      fd = open('/ifstest/somefile.txt':
c                                    O_CREAT+O_WRONLY+O_CODEPAGE:
c                                    mode: 819)
```

Note that a new code page is assigned only if a new file is created. Therefore, if we really want to make sure that it's going to be assigned, we delete the file first.

The O_TEXTDATA flag on the open() API is used to tell the API that we're working with text data, so we want the system to translate the data for us. Note that it only does the translation if the file already exists, and is assigned a different code page than the one that's associated with our job.

To open the file so that the translation takes place, we'd open it like this:

```
c                   eval      fd = open('/ifstest/somefile.txt':
c                                         O_RDWR+O_TEXTDATA)
```

Now when we read or write any data to that file, it will automatically be translated to or from the code page that was assigned to it.

5.6. Example of writing & creating an ASCII text file

Here we will take the same example that we created earlier in the chapter, and add code page support to it. The data that we write will actually be translated to ASCII for us.

Note also that the EDTF command still works, even though it's now in ASCII. The EDTF command uses the code page that we assigned, and understands that it needs to translate it as well!

You can also take the stream file, transfer it to your PC, and open it up with Notepad, Wordpad or even Word. Because the files are in ASCII, they can be used almost anywhere!

```
 * CH5ASCII: Example of text file in ASCII mode
 *  (From Chap 5)
 *
 * To compile:
 *   CRTBNDRPG CH5ASCII SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   len                         15P 5 const

D fd              S              10I 0
D line            S             100A
D len             S              10I 0
D msg             S              52A
D err             S              10I 0

c                   exsr      MakeFile
c                   exsr      EditFile
c                   exsr      ShowFile
c                   eval      *inlr = *on


C*************************************************************
C* Write some text to a text file
C*************************************************************
CSR   MakeFile      begsr
C*------------------------
C* Make sure we don't have an old file that might be in the way
C* (ENOENT means it didnt exist to begin with)
c                   if        unlink('/ifstest/ch5_file.txt') < 0
c                   eval      err = errno
```

```
c                   if        err <> ENOENT
c                   callp     die('unlink(): ' + %str(strerror(err)))
c                   endif
c                   endif

C* Create a new file, and assign it a code page of 819:
c                   eval      fd = open('/ifstest/ch5_file.txt':
c                                     O_CREAT+O_WRONLY+O_CODEPAGE:
c                                     S_IWUSR+S_IRUSR+S_IRGRP+S_IROTH:
c                                     819)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif
c                   callp     close(fd)

C* Now re-open the file in text mode.  Since it was assigned a
C* code page of 819, and we're opening it in text mode, OS/400
C* will automatically translate to/from ASCII for us.
c                   eval      fd = open('/ifstest/ch5_file.txt':
c                                       O_WRONLY+O_TEXTDATA)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   eval      line = 'Dear Cousin,'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'I love the way you make' +
c                               ' cheese fondue.'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'Thank you for being so cheesy!'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = ' '
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'Sincerely,'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)
```

```
c                   eval      line = '     Richard M. Nixon'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   callp     close(fd)
C*-------------------------
CSR                 endsr


C*************************************************************
C*  Call the OS/400 text editor, and let the user change the
C*  text around.
C*************************************************************
CSR   EditFile      begsr
C*-------------------------
c                   callp     cmd('EDTF STMF("/ifstest/' +
c                                        'ch5_file.txt")': 200)
C*-------------------------
CSR                 endsr


C*************************************************************
C*  Read file, line by line, and dsply what fits
C*  (DSPLY has a lousy 52-byte max... blech)
C*************************************************************
CSR   ShowFile      begsr
C*-------------------------
c                   eval      fd = open('/ifstest/ch5_file.txt':
c                                   O_RDONLY+O_TEXTDATA)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

c                   dow       readline(fd: %addr(line): %size(line))>=0
c                   eval      Msg = line
c     Msg           dsply
c                   enddo

c                   callp     close(fd)

c                   eval      Msg = 'Press ENTER to continue'
c                   dsply                 Msg
C*-------------------------
CSR                 endsr


 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

5.7. Example of a report in ASCII

Now, let's do something a little more practical as an example. Have you ever had someone ask if a report can be sent to their PC, instead of printed? Here's an example of doing that with a stream file.

Since the files that a report would typically be printed off of are not a standard part of OS/400, I decided to make a report by listing the objects in a given library. The library will be a parameter.

It might be a good exercise for you to create your own report. Use mine as a sample, but take a report that's commonly used in your company, and convert it to write a text file as well as the printer output.

Anyway, here's the code:

```
 * CH5LIBLIST: Example of a report in ASCII text format
 *  (From Chap 5)
 *
 * To compile:
 *   CRTBNDRPG CH5LIBLIST SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

FQADSPOBJ  IF   E           K DISK    USROPN

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   len                         15P 5 const

D FmtDate         PR             10A
D   mmddyy                        6A   const

D fd              S              10I 0
D line            S             100A
D len             S              10I 0
D LineNo          S              10I 0

c     *entry        plist
c                   parm                    MyLib           10

C*****************************************************
C* Create a file containing the objects we wish to report
C*****************************************************
C                   callp     cmd('DSPOBJD OBJ('+%trim(MyLib)+'/*ALL)'+
C                               ' OBJTYPE(*ALL) OUTPUT(*OUTFILE) ' +
C                               ' OUTFILE(QTEMP/QADSPOBJ) ' +
C                               ' OUTMBR(*FIRST *REPLACE)': 200)

C*****************************************************
C* Open the list of objects:
C*****************************************************
c                   callp     cmd('OVRDBF FILE(QADSPOBJ) TOFILE(' +
```

```
c                                   'QTEMP/QADSPOBJ)': 200)
c                   open      QADSPOBJ

C*****************************************************
C* Open a stream file to write report to:
C*****************************************************
c                   eval      fd = open('/ifstest/object_report.txt':
c                                   O_CREAT+O_TRUNC+O_CODEPAGE+O_WRONLY:
c                                   S_IRWXU+S_IRWXG+S_IROTH: 819)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif
c                   callp     close(fd)

c                   eval      fd = open('/ifstest/object_report.txt':
c                                         O_TEXTDATA+O_WRONLY)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

C*****************************************************
C* Create the report
C*****************************************************
c                   exsr      Heading
c                   read      QADSPOBJ

c                   dow       not %eof(QADSPOBJ)
c                   exsr      WriteObj
c                   read      QADSPOBJ
c                   enddo


C*****************************************************
c* Clean up and exit
C*****************************************************
c                   callp     close(fd)
c                   close     QADSPOBJ
c                   callp     cmd('DLTOVR FILE(QADSPOBJ)': 50)
c                   callp     cmd('DLTF QTEMP/QADSPOBJ': 50)

c                   eval      *inlr = *on


C*===============================================================
C* Write a heading on the report
C*===============================================================
CSR   Heading       begsr
C*------------------------
C* x'0C' = Form Feed
C                   eval      line = x'0c'
c                   eval      len = 1
c                   callp     writeline(fd: %addr(line): len)
```

```
C                   eval      line = 'Listing of objects in ' +
c                               %trim(MyLib) + ' library'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = *blanks
c                   eval      len = 0
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = 'Object Name'
c                   eval      %subst(line: 15) = 'Object Type'
c                   eval      %subst(line: 30) = 'Object Size'
c                   eval      %subst(line: 45) = 'Last Modified'
c                   eval      %subst(line: 60) = 'Last Used'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '----------'
c                   eval      %subst(line: 15) = '----------'
c                   eval      %subst(line: 30) = '----------'
c                   eval      %subst(line: 45) = '-------------'
c                   eval      %subst(line: 60) = '---------'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      LineNo = 5
C*------------------------
csr                 endsr


C*================================================================
C* Add an object to the report
C*================================================================
CSR   WriteObj      begsr
C*------------------------
c                   if        LineNo > 60
c                   exsr      Heading
c                   endif

c                   eval      Line = odObNm
c                   eval      %subst(line: 15) = odobtp
c                   eval      %subst(line: 30) = %editc(odobsz:'L')
c                   eval      %subst(line: 45) = FmtDate(odldat)
c                   eval      %subst(line: 60) = FmtDate(odudat)
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      LineNo = LineNo + 1
C*------------------------
csr                 endsr


 *++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
 * Format a date into human-readable YYYY-MM-DD format:
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P FmtDate         B
D FmtDate         PI            10A
D   mmddyy                       6A   const

D Temp6           S               6  0
D TempDate        S                D
D Temp10          S              10A

C* If date isn't a valid number, return *blanks
c                   testn                   mmddyy               99
c                   if        *in99 = *off
c                   return    *blanks
c                   endif

C* If date isn't a valid MMDDYY date, return *blanks
c                   move      mmddyy        Temp6
c     *mdy          test(de)                Temp6
c                   if        %error
c                   return    *blanks
c                   endif

C* Convert date to ISO format, and return it.
c     *mdy          move      Temp6         TempDate
c     *iso          move      TempDate      Temp10
c                   return    Temp10

P                 E

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Chapter 6. Additional Text Formats

6.1. Comma Separated Values (CSV) Format

One type of text file that's commonly used is called "Comma Separated Values" (CSV) format. In CSV, you create a file, where each "record" in the file is a line of text, each line is broken up into "fields", and each field is separated by a comma.

CSV is useful for taking data from an OS/400 physical file and importing it into another program. Perhaps the most popular program is Microsoft Excel, but there are many others that will accept CSV as an input.

The OS/400 command called CPYTOIMPF can be used to create a stream file in CSV format without us having to write any code to do it. So, why are we doing this? One reason is that it helps you understand a little more about stream files. Another reason is that sometimes a little detail comes up that CPYTOIMPF does not support, and when that happens it's nice to have an alternative. Finally, there are times when your CSV file needs to contain data that requires program logic to generate.

6.2. Example of creating a CSV file

In chapter 5, we created a report in plain ASCII format that showed the objects in a library. Now, we will find the same data, and put it into a CSV-formatted stream file.

```
 * CH6CSV: Example of a report in ASCII CSV format
 *  (From Chap 6)
 *
 * To compile:
 *   CRTBNDRPG CH6CSV SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

FQADSPOBJ  IF   E           K DISK    USROPN

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   len                         15P 5 const

D FmtDate         PR             10A
D   mmddyy                        6A   const

D fd              S              10I 0
D line            S             100A
D len             S              10I 0
D LineNo          S              10I 0
```

```
c     *entry        plist
c                   parm                    MyLib           10

C*****************************************************
C* Create a file containing the objects we wish to report
C*****************************************************
C                   callp     cmd('DSPOBJD OBJ('+%trim(MyLib)+'/*ALL)'+
C                               ' OBJTYPE(*ALL) OUTPUT(*OUTFILE) ' +
C                               ' OUTFILE(QTEMP/QADSPOBJ) ' +
C                               ' OUTMBR(*FIRST *REPLACE)': 200)

C*****************************************************
C* Open the list of objects:
C*****************************************************
c                   callp     cmd('OVRDBF FILE(QADSPOBJ) TOFILE(' +
c                                 'QTEMP/QADSPOBJ)': 200)
c                   open      QADSPOBJ

C*****************************************************
C* Open a stream file to CSV data to:
C*****************************************************
c                   eval      fd = open('/ifstest/object_report.csv':
c                               O_CREAT+O_TRUNC+O_CODEPAGE+O_WRONLY:
c                               S_IRWXU+S_IRWXG+S_IROTH: 819)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif
c                   callp     close(fd)

c                   eval      fd = open('/ifstest/object_report.csv':
c                                         O_TEXTDATA+O_WRONLY)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

C*****************************************************
C* Create the report
C*****************************************************
c                   read      QADSPOBJ

c                   dow       not %eof(QADSPOBJ)
c                   exsr      WriteObj
c                   read      QADSPOBJ
c                   enddo


C*****************************************************
c* Clean up and exit
C*****************************************************
c                   callp     close(fd)
c                   close     QADSPOBJ
c                   callp     cmd('DLTOVR FILE(QADSPOBJ)': 50)
c                   callp     cmd('DLTF QTEMP/QADSPOBJ': 50)
```

```
c                   eval      *inlr = *on



C*=============================================================
C* Add an object to the report
C*=============================================================
CSR   WriteObj      begsr
C*-------------------------
c                   eval      Line = '"' + %trim(odobnm) + '",' +
c                               '"' + %trim(odobtp) + '",' +
c                               %trim(%editc(odobsz:'L')) + ',' +
c                               '"' + %trim(FmtDate(odldat)) + '",' +
c                               '"' + %trim(FmtDate(odudat)) + '"'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)
C*-------------------------
csr                 endsr


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * Format a date into human-readable YYYY-MM-DD format:
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P FmtDate         B
D FmtDate         PI            10A
D   mmddyy                       6A   const

D Temp6           S              6  0
D TempDate        S               D
D Temp10          S             10A

C* If date isn't a valid number, return *blanks
c                   testn                   mmddyy               99
c                   if        *in99 = *off
c                   return    *blanks
c                   endif

C* If date isn't a valid MMDDYY date, return *blanks
c                   move      mmddyy        Temp6
c     *mdy          test(de)                Temp6
c                   if        %error
c                   return    *blanks
c                   endif

C* Convert date to ISO format, and return it.
c     *mdy          move      Temp6         TempDate
c     *iso          move      TempDate      Temp10
c                   return    Temp10

P                 E

 /DEFINE ERRNO_LOAD_PROCEDURE
```

```
/COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

See if you can figure out what that code is doing, then compile it and run it. You can open up the results using EDTF or Windows Notepad, and see what the output looks like.

Then, open up the same file using Microsoft Excel. Note that each field appears in a separate column in the spreadsheet. Now, you can use Excel's different capabilities to sort the file, or total up the columns, or whatever you'd like to do.

Next, you might try using this technique with one of your company's reports. Have you had a user ask if he could import the results of a report into Excel? This could be used as a way to do that!

6.3. HTML (web page) format

We already saw in Chapter 5 that text files could be opened in Microsoft Word. They don't have any fancy characteristics, however. No bold, no fonts, no underlines... Just plain boring text.

HTML is an easy way to add some of these extra bells and whistles. HTML stands Hypertext Markup Language and is the language that web pages are written in. Since HTML documents can be read by Microsoft Word, as well as web browsers, it is a useful format to be able to write data into.

Unfortunately, fully explaining the features of HTML would be far beyond the scope of this document. However, I will give you a quick idea of how it works.

HTML consists of "mark up tags". The concept is that you have this plain text document, and then you insert these tags to identify how parts of the document should be displayed.

Most HTML capabilities have a starting and an ending tag. For example, to mark something as "bold", you use the "b" tag. The starting tag looks like this: and the ending tag is the same, except that it starts with a slash like this: . Everything placed in between these tags will appear as bold text.

```
This is normal.  <b>This is bold.</b>
```

6.4. Example of creating an HTML file

This example will, once again, generate a report of the objects in library. However, this time we've used HTML to format the report.

Here's the code:

```
 * CH5LIBLIST: Example of a report in HTML format
 *  (From Chap 6)
 *
 * To compile:
 *   CRTBNDRPG CH6HTML SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

FQADSPOBJ  IF   E           K DISK    USROPN
```

```
D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D Cmd             PR                  ExtPgm('QCMDEXC')
D   command                    200A   const
D   len                         15P 5 const

D FmtDate         PR             10A
D   mmddyy                        6A   const

D fd              S              10I 0
D line            S             100A
D len             S              10I 0

c     *entry        plist
c                   parm                    MyLib           10

C*****************************************************
C* Create a file containing the objects we wish to report
C*****************************************************
C                   callp     cmd('DSPOBJD OBJ('+%trim(MyLib)+'/*ALL)'+
C                               ' OBJTYPE(*ALL) OUTPUT(*OUTFILE) ' +
C                               ' OUTFILE(QTEMP/QADSPOBJ) ' +
C                               ' OUTMBR(*FIRST *REPLACE)': 200)

C*****************************************************
C* Open the list of objects:
C*****************************************************
c                   callp     cmd('OVRDBF FILE(QADSPOBJ) TOFILE(' +
c                                 'QTEMP/QADSPOBJ)': 200)
c                   open      QADSPOBJ

C*****************************************************
C* Open a stream file to write report to:
C*****************************************************
c                   eval      fd = open('/ifstest/object_report.html':
c                               O_CREAT+O_TRUNC+O_CODEPAGE+O_WRONLY:
c                               S_IRWXU+S_IRWXG+S_IROTH: 819)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif
c                   callp     close(fd)

c                   eval      fd = open('/ifstest/object_report.html':
c                                       O_TEXTDATA+O_WRONLY)
c                   if        fd < 0
c                   callp     die('open(): ' + %str(strerror(errno)))
c                   endif

C*****************************************************
C* Create the report
```

```
C*************************************************************
c                   exsr      Heading
c                   read      QADSPOBJ

c                   dow       not %eof(QADSPOBJ)
c                   exsr      WriteObj
c                   read      QADSPOBJ
c                   enddo

c                   exsr      Footer


C*************************************************************
c* Clean up and exit
C*************************************************************
c                   callp     close(fd)
c                   close     QADSPOBJ
c                   callp     cmd('DLTOVR FILE(QADSPOBJ)': 50)
c                   callp     cmd('DLTF QTEMP/QADSPOBJ': 50)

c                   eval      *inlr = *on


C*===========================================================
C* Write a heading on the report
C*===========================================================
CSR   Heading       begsr
C*------------------------
C                   eval      line = '<html><head><title>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

C                   eval      line = 'Listing of objects in ' +
c                               %trim(MyLib) + ' library'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '</title></head>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<body>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

C                   eval      line = '<h1><center>' +
c                             'Listing of objects in ' +
c                               %trim(MyLib) + ' library</center></h1>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<center>' +
c                                    '<table width=90% border=3>'
```

```
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<tr>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<th><em>Object Name</em></th>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<th><em>Object Type</em></th>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<th><em>Object Size</em></th>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<th><em>Last Modified</em></th>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<th><em>Last Used</em></th>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '</tr>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)
C*------------------------
csr                 endsr


C*===============================================================
C* Add an object to the report
C*===============================================================
CSR   WriteObj      begsr
C*------------------------
c                   eval      line = '<tr>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<td>'+%Trim(odObNm)+'</td>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<td>'+%Trim(odObTp)+'</td>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<td align=right>' +
c                                    %trim(%editc(odObSz:'L')) + '</td>'
```

```
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<td align=center>' +
c                                    %trim(FmtDate(odldat)) + '</td>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '<td align=center>' +
c                                    %trim(FmtDate(odudat)) + '</td>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '</tr>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)
C*------------------------
csr                 endsr


C*============================================================
C* Finish up the HTML page
C*============================================================
CSR   Footer        begsr
C*------------------------
c                   eval      line = '</table></center>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)

c                   eval      line = '</body></html>'
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(fd: %addr(line): len)
C*------------------------
csr                 endsr


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * Format a date into human-readable YYYY-MM-DD format:
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P FmtDate         B
D FmtDate         PI            10A
D   mmddyy                       6A   const

D Temp6           S               6  0
D TempDate        S                D
D Temp10          S              10A

C* If date isn't a valid number, return *blanks
c                   testn                   mmddyy               99
c                   if        *in99 = *off
c                   return    ' '
c                   endif
```

```
C* If date isn't a valid MMDDYY date, return *blanks
c                   move      mmddyy        Temp6
c     *mdy          test(de)                Temp6
c                   if        %error
c                   return    ' '
c                   endif

C* Convert date to ISO format, and return it.
c     *mdy          move      Temp6         TempDate
c     *iso          move      TempDate      Temp10
c                   return    Temp10

P                 E

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Go ahead and run it, compile it, then bring up the resulting file using EDTF or the Windows Notepad. Then, try bringing the same file up in your web browser or in Word. Notice the affect of the various tags?

Chapter 7. Working with directories

7.1. How directories work

Back when we first discussed stream files, and path names, we talked about directories. We know that directories are similar to libraries in their ability to act as "containers" for stream files. In other words, stream files are stored inside directories in the IFS.

In addition to storing stream files inside a directory, most file systems also allow you to store directories inside a directory. A directory inside a directory is sometimes referred to as a "sub-directory." From a programmers perspective, directories and sub-directories behave exactly the same way, so we will simply refer to them all as "directories."

Each directory has a "mode" (access permission bits) just as the stream files do. Most file systems allow you to run chmod() to change the mode, just as you would with a file.

The mode of a directory acts slightly differently, however. If you have "read" permission to a directory, it means that you're allowed to see the list of files that it contains. If you have "write" permission, it means that you're able to add files, delete files, rename files, etc in the directory. If you have "execute" permission to a directory, it means that you're allowed to "search" the directory. Some programs, such as the "find" utility in QShell, will skip searching directories that you do not have execute authority to.

Just in case you missed it, let me stress this again: If you have "write" permission to a directory, you can add, delete, or rename files in that directory. Even though you don't have access to the file itself!

For example, let's say you created a directory called "scott". In that directory, you put a file called "dont_let_bob_read.txt". Let's say that the directory's mode allows read, write and execute to the owner, the group and everyone else. Let's say that the file gives read and write access to the owner, but nothing to the group or the world. Okay, now Bob sees his name on the file, he tries to open it with DSPF, but he can't. He can't read the file. So, he tries to delete it... He can! The file is gone.

Why can Bob do that? Because, technically, when you are adding, removing, or changing the names of stream files in a directory, you aren't modifying the file data. You're adding, removing or changing entries in the directory itself.

There are two "special" entries inside each directory, as well. They are "." (a single dot) and ".." (two dots). These signify the "current directory" and the "parent directory" respectively.

7.2. Creating directories

The API for creating a new directory (or sub-directory) in the IFS is called "mkdir()." This stands, appropriately, for "make directory."

Here are the C language and RPG prototypes for the mkdir() API. They're straightforward, so I will not bore you with the conversion details,

```
int mkdir(const char *path, mode_t mode)
```

```
❶D mkdir           PR            10I 0 ExtProc('mkdir')
❷D   path                          *   Value options(*string)
❸D   mode                        10U 0 Value
```

❶ mkdir() returns -1 if it failed, or 0 if it was successful

❷ The "path" parameter is where you specify the directory to be created. The format of the path is just like that specified in the open() API.

❸ The "mode" parameter is the access permissions to assign to the directory when it's created. You specify the mode using the same bit-flags that you used for the mode on the open() API.

For example, consider the following code:

```
C                   if        mkdir('/ifstest/ShoeTongue':
C                                     S_IRUSR+S_IWUSR+S_IXUSR+
C                                     S_IRGRP+S_IWGRP+S_IXGRP+
C                                     S_IROTH+S_IXGRP) < 0
C                   callp     EscErrno(errno)
C                   endif
```

This code creates a new sub-directory called "ShoeTongue" inside the a directory called "/ifstest". It grants read, write and search permissions to the owner, read, write and search permissions to the primary group, and read and search permissions to everyone else.

In order for this command to exist, the "/ifstest" directory must already exist. If not, the command will fail with "path or directory not found!"

7.3. Removing directories

What can be created, can also be destroyed. The rmdir() "Remove Directory" API is used to delete a directory.

Here are the C and RPG prototypes for the rmdir() API.

```
 int rmdir(const char *path)

D rmdir           PR            10I 0 ExtProc('rmdir')
D   path                          *   value options(*string)
```

In order for a directory to be deleted, it must be empty. You cannot delete a directory that still has files in it.

Here's a sample of deleting the ShoeTongue directory:

```
c                   if        rmdir('/ifstest/ShoeTongue') < 0
c                   callp     EscErrno(errno)
c                   endif
```

7.4. Switching your current directory

Just as there is a "current library" in the traditional file system, there is also a "current directory" in the IFS. When you supply a path name to an API, and that path name does not start with a slash character, it tells the API that you want to use the "current directory."

You can change your current directory by calling the chdir() "Change Directory" API.

The prototype for chdir() looks like this:

```
 int chdir(const char *path)

D chdir           PR            10I 0 ExtProc('chdir')
D   path                          *   Value Options(*string)
```

Not much to it, is there? You just call chdir() with one argument, the path to change to. The chdir() API will return a -1 if it fails, or a 0 if it was successful.

In this example, we will create and delete the ShoeTongue directory, just as we did in the code examples for mkdir() and rmdir(). However, we will use a "relative" directory name this time. We will switch our current directory to /ifstest, and then we won't have to specify it on the mkdir() and rmdir() APIs.

```
c                   if        chdir('/ifstest') < 0
c                   callp     EscErrno(errno)
c                   endif

c                   if        mkdir('ShoeTongue':
c                               S_IRUSR+S_IWUSR+S_IXUSR+
c                               S_IRGRP+S_IWGRP+S_IXGRP+
c                               S_IROTH+S_IXOTH) < 0
c                   callp     EscErrno(errno)
c                   endif

c                   if        rmdir('ShoeTongue') < 0
c                   callp     EscErrno(errno)
c                   endif
```

7.5. Opening Directories

Now that we know how to create and delete directories, and change our current directory, we also need to know how to read the contents of a directory.

The process of reading directories in an RPG program will require you to use 3 different APIs. They are opendir(), which opens the directory, readdir() which reads the next entry from a directory and closedir() which closes the directory when you're finished.

The opendir() API is similar in some ways to the open() API. It accepts a parameter that tells the name of a directory to open, and it returns a handle that can be used to read through that directory.

Here is the C-language prototype for the opendir() API:

```
DIR *opendir(const char *dirname)
```

We know that the "DIR *" means that it returns a pointer to a "DIR". But what data type is a "DIR"? We could figure that out -- if you hunt through the C header members, you'd eventually find it. But, as it turns out, we don't need it! All we need to do with the return value from opendir() is pass it as a parameter to the readdir() and closedir() APIs. Therefore, we don't care what the pointer points to! We can just treat it as a pointer.

When this API fails, it will return a *NULL pointer, and we can then check errno to find out which error occurred.

Therefore, the prototype for the opendir() API in RPG looks like this:

```
D opendir         PR              *   EXTPROC('opendir')
D  dirname                        *   VALUE options(*string)
```

When we want to open a directory, we simply pass the directory name as a parameter. For example, if our goal was to read the contents of the root directory of the IFS, we might write code that looks like this:

```
D d               S                   *
c                   eval      d = opendir('/')
c                   if        d = *NULL
c                   callp     EscErrno(errno)
c                   endif
```

7.6. Reading Directories

Once the directory has been opened, we will want to read it's contents. This is accomplished using the readdir() "Read Directory" API.

Here is the C language prototype for the readdir() API:

```
struct dirent *readdir(DIR *dirp)
```

The return value of readdir() is a pointer that points to a data structure in the "dirent" format. Back in the discussion of the stat() API, I explained that in C, you first define a data structure's format, and then you define structures that use that format. In this case, the format is called "dirent".

We will need to know how to define our own dirent data structures, since that's where the useful information from the directory comes from. However, for creating the RPG prototype, we really only needed to know that the return value is a pointer. So, here's the RPG prototype:

```
D readdir         PR              *   EXTPROC('readdir')
D  dirp                           *   VALUE
```

To get useful information out of this API, as I mentioned above, we need to make our own dirent structure. The definition of this structure in C looks like this;

```
struct dirent {
```

```
  ❶char           d_reserved1[16];  /* Reserved                        */
  ❷unsigned int   d_fileno_gen_id;  /* File number generation ID  @A1C*/
  ❸ino_t          d_fileno;         /* The file number of the file     */
  ❹unsigned int   d_reclen;         /* Length of this directory entry
                                       in bytes                       */
  ❺int            d_reserved3;      /* Reserved                        */
  ❻char           d_reserved4[8];   /* Reserved                        */
  ❼qlg_nls_t      d_nlsinfo;        /* National Language Information
                                       about d_name                   */
  ❽unsigned int   d_namelen;        /* Length of the name, in bytes
                                       excluding NULL terminator      */
  ❾char           d_name[_QP0L_DIR_NAME]; /* Name...null terminated   */
};
```

❶ The first subfield in the dirent data structure is simple. It's a 16-byte long character field marked as "reserved".

❷ Next, we have the file number generation ID. It's marked as an unsigned integer, which is equivalent to the RPG "10U 0" data type. This field probably won't be useful to you as an RPG programmer, so I won't explain what it contains.

❸ The file number is defined as a "ino_t". If we hunt through the C header members, we will find that ino_t is defined in the file `QSYSINC/SYS,TYPES` as an unsigned integer. Good, that's an RPG "10U 0". Again, this field isn't likely to be useful to an RPG programmer, so I won't explain it.

❹ The length of the directory entry is stored here, and it's marked as another unsigned integer. Lots of "10U 0" definitions for our structure, eh?

❺ This field is marked as "reserved" and is an integer, which we know is defined as "10I 0" in RPG.

❻ Yet another "reserved" field. This one is an 8-byte character field.

❼ The national language support information is stored here. This information allows us to determine what CCSID we should be using to display the filename, as well as what language it is in, and what country it is from.

If we hunt for "qlg_nls_t" in the C header members, we'll find it defined in `QSYSINC/SYS,TYPES`. It is defined to be a data structure, containing four more subfields, an integer containing the CCSID, a 2-byte field containing the country-id, a 3-byte field containing the language ID, and a 3-byte reserved field.

❽ This subfield contains the length of the filename that will be given in the next subfield. Since it is listed as an unsigned integer, we know that we need to make the RPG version use a "10U 0" variable.

❾ Finally, what we've been waiting for! The name of the file that this directory entry refers to. The constant "_QP0L_DIR_NAME" defines the maximum length that the file name can be. If we hunt through the C header members, we will find that this constant is set to be the number 640.

So, here's our RPG version of the dirent structure. Note that we've based it on a pointer. This is important, since the readdir() API allocates the memory for this structure, consequently, we need to be able to point our structure at it's memory.

```
D p_dirent        s               *
D dirent          ds                  based(p_dirent)
D   d_reserved1                 16A
D   d_fileno_gen_id...
D                               10U 0
```

```
D   d_fileno                    10U 0
D   d_reclen                    10U 0
D   d_reserved3                 10I 0
D   d_reserved4                  8A
D   d_nlsinfo                   12A
D     nls_ccsid                 10I 0 OVERLAY(d_nlsinfo:1)
D     nls_cntry                  2A   OVERLAY(d_nlsinfo:5)
D     nls_lang                   3A   OVERLAY(d_nlsinfo:7)
D     nls_reserv                 3A   OVERLAY(d_nlsinfo:10)
D   d_namelen                   10U 0
D   d_name                     640A
```

Once we've added these to our IFSIO_H member, we'll be able to call the readdir() API like this:

```
c                   eval      p_dirent = readdir(d)
c                   dow       p_dirent <> *NULL
C* ...do whatever we like with the contents
C*    of the dirent structure, here...
c                   eval      p_dirent = readdir(d)
c                   enddo
```

7.7. Closing an open directory

When we're done reading the directory, we need to close it. To do that, we will call the Close Directory API, closedir().

Here's the C and RPG prototypes for closedir():

```
 int closedir(DIR *dirp)

D closedir        PR            10I 0 EXTPROC('closedir')
D  dirhandle                      *   VALUE
```

Calling this API is quite simple, just pass it the variable that you received when you called opendir().

For example:

```
 c                   if        closedir(d) < 0
 c                   callp     EscErrno(errno)
 c                   endif
```

7.8. Example of reading a directory

In this simple example, we will open up our /ifstest directory that we've been using for all of our sample code, and we'll read the contents of it. For each entry in that directory, we'll use the DSPLY op-code to display the first 52 bytes of the file name.

```
 * CH7READDIR: Example of reading a directory in the IFS
 *  (From Chap 7)
 *
 * To compile:
 *   CRTBNDRPG CH7READDIR SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H

D dir             s               *
D Msg             S             52A

c                   eval      dir = opendir('/ifstest')
c                   if        dir = *NULL
c                   callp     die('opendir(): '+%str(strerror(errno)))
c                   endif

c                   eval      p_dirent = readdir(dir)
c                   dow       p_dirent <> *NULL
c                   eval      Msg = %subst(d_name:1:d_namelen)
c     msg           dsply
c                   eval      p_dirent = readdir(dir)
c                   enddo

c                   callp     closedir(dir)

c                   eval      Msg = 'Press ENTER to end'
c                   dsply                   Msg

c                   eval      *inlr = *on

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

If you want to experiment with it, you can change the directory name to other directories, and run it again. Or, maybe have it always list your current directory, and use chdir() to change which directory it lists.

7.9. Example of making a DIR command for QSHELL

Here's a slightly more complex example of reading a directory. This program is designed to be run under the QShell utility that IBM provides.

QShell is intended to emulate a UNIX command prompt, and therefore it provides the UNIX "ls" command to view directories. But, what if you like the way directories look in the MS-DOS "dir" command? Well... we could write out own DIR command using the APIs, right? Let's try!

One of the interesting things about QSHELL, is that it provides us with 3 stream file descriptors, already opened, which we can use for input and output to the shell.

Descriptor #0 is referred to as "standard input", but we will not use it in our example.

Descriptor #1 is referred to as "standard output". It's the place to write our directory listing to. We should treat it like a text file, which is to say that we start a new line by sending a CRLF sequence.

Descriptor #2 is referred to as "standard error". Any error messages that we need to communicate to the user should be sent there.

There's a lot more that I could explain about QSHELL, but alas, it's a bit beyond the scope of this document to fully explain this environment...

We'll also need to send text to the screen listing whether or not a file is a directory, and also the file size, because these are done in MS-DOS. Fortunately, we can use the stat() API to retrieve these.

Finally, we'll need to report the date that's associated with each file in the directory. We can get that from the stat() API as well, but all the dates in stat() are listed in Universal Time Coordinated (UTC) and not in our time zone. The dates that appear, are actually listed as a number of seconds since the "epoch" of January 1st, 1970. So, to make all of this behave as we want it to, we've got our work cut out for us. Still, hopefully when you see the code, you'll understand what's going on!

Here's the code:

```
 * CH7QSHDIR: More complex example of reading a dir in the IFS
 *  We try to make our output look like the MS-DOS "DIR" command
 *  (From Chap 7)
 *
 * To compile:
 *   CRTBNDRPG CH7QSHDIR SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D STDIN           C                   CONST(0)
D STDOUT          C                   CONST(1)
D STDERR          C                   CONST(2)

D S_ISDIR         PR             1N
D   mode                        10U 0 value

D CEEUTCO         PR                   ExtProc('CEEUTCO')
D   hours                       10I 0
D   minutes                     10I 0
D   seconds                      8F

D line            s           1024A
D len             s             10I 0
```

```
D dir             S               *
D filename        S            1024A   varying
D fnwithdir       S            2048A   varying
D mystat          S                    like(statds)
D curr            s            1024A
D curdir          s            1024A   varying
D dot             S              10I 0
D notdot          S              10I 0
D epoch           S                Z   inz(z'1970-01-01-00.00.00.000000')
D ext             S               3A
D filedate        S               8A
D filetime        S               6A
D modtime         S                Z
D mydate          S                D
D mytime          S                T
D shortfn         S               8A
D size            S              13A
D worktime        S               8A
D hours_utc       s              10I 0
D mins_utc        s              10I 0
D secs_utc        s               8F
D utcoffset       s              10I 0

 * Here's an example of what an MS-DOS directory listing
 * looks like:
 *
 *   Directory of C:\WINDOWS
 *
 *  .              <DIR>        10-24-00 10:28a .
 *  ..             <DIR>        10-24-00 10:28a ..
 *  COMMAND        <DIR>        05-08-00 11:54a COMMAND
 *  VB       INI         1,245  10-04-01  9:12p VB.INI
 *  LOCALS~1       <DIR>        10-06-01  9:44p Local Settings
 *  BDDKL    DRV         1,986  11-21-01  8:43p bddkl.drv
 *  BPKPC    DRV         3,234  11-21-01  8:43p bpkpc.drv
 *  PILCIKK  DRV         1,122  11-21-01  8:43p pilcikk.drv
 *  NEWRES~1 RC          1,440  12-12-01  2:02a newrestest.rc

c                   eval      *inlr = *on

C*****************************************
C* get the number of seconds between local
C* time an Universal Time Coordinated (UTC)
C*****************************************
c                   callp(e)  CEEUTCO(hours_utc: mins_utc: secs_utc)
c                   if        %error
c                   eval      utcoffset = 0
c                   else
c                   eval      utcoffset = secs_utc
c                   endif

C*****************************************
C* Use the getcwd() API to find out the
```

```
C* name of the current directory:
C*****************************************
c                   if        getcwd(%addr(curr): %size(curr)) = *NULL
c                   eval      line = 'getcwd(): ' +
c                                    %str(strerror(errno))
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDERR: %addr(line): len)
c                   return
c                   endif

c                   eval      curdir = %str(%addr(curr))

C*****************************************
C* open the current directory:
C*****************************************
c                   eval      dir = opendir(curdir)
c                   if        dir = *NULL
c                   eval      line = 'opendir(): ' +
c                                    %str(strerror(errno))
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDERR: %addr(line): len)
c                   return
c                   endif

c                   eval      line = ''
c                   eval      len = 0
c                   callp     writeline(STDOUT: %addr(line): len)

c                   eval      line = ' Directory of ' + curdir
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDOUT: %addr(line): len)

c                   eval      line = ''
c                   eval      len = 0
c                   callp     writeline(STDOUT: %addr(line): len)

c                   eval      p_statds = %addr(mystat)

c                   eval      p_dirent = readdir(dir)
c                   dow       p_dirent <> *NULL
c                   eval      filename = %subst(d_name:1:d_namelen)
c                   eval      fnwithdir = curdir + '/' + filename
c                   if        stat(fnwithdir: %addr(mystat))=0
c                   exsr      PrintFile
c                   endif
c                   eval      p_dirent = readdir(dir)
c                   enddo

c                   callp     closedir(dir)


C*===========================================================
C* For each file in the directory, print a line of info:
```

```
C*============================================================
CSR   PrintFile     begsr
C*------------------------
C* Separate into extension & short filename:
c     '.'           check     filename      notdot
c                   if        notdot = 0
c                   eval      ext = *blanks
c                   eval      shortfn = filename
c                   else
c                   eval      dot = %scan('.': filename: notdot)
c                   if        dot > 0
c                   eval      ext = %subst(filename:dot+1)
c                   eval      shortfn = %subst(filename: 1: dot-1)
c                   else
c                   eval      ext = *blanks
c                   eval      shortfn = filename
c                   endif
c                   endif

C* Show size if this is not a directory:
c                   if        S_ISDIR(st_mode)
c                   eval      size = '<DIR>'
c                   else
c                   eval      size = %editc(st_size: 'K')
c                   endif

C* figure out date & time:
c     epoch         adddur    st_atime:*S   modtime
c                   adddur    utcoffset:*S  modtime
c                   move      modtime       mydate
c                   move      modtime       mytime
c     *MDY-         move      mydate        filedate
c     *USA          move      mytime        worktime
c                   eval      filetime=%subst(worktime:1:5) +
c                                       %subst(worktime:7:1)

C* and write it to QSH STDOUT:
c                   eval      line = shortfn + ' ' + ext + '  ' +
c                               size + '  ' + filedate + ' ' +
c                               filetime + ' ' + filename
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDOUT: %addr(line): len)
C*------------------------
CSR                 endsr


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This tests a file mode to see if a file is a directory.
 *
 * Here is the C code we're trying to duplicate:
 *      #define _S_IFDIR    0040000                                     */
 *      #define S_ISDIR(mode) (((mode) & 0370000) == _S_IFDIR)
 *
```

```
 * 1) ((mode) & 0370000) takes the file's mode and performs a
 *      bitwise AND with the octal constant 0370000.  In binary,
 *      that constant looks like: 00000000000000011111000000000000
 *      The effect of this code is to turn off all bits in the
 *      mode, except those marked with a '1' in the binary bitmask.
 *
 * 2) ((result of #1) == _S_IFDIR)  What this does is compare
 *      the result of step 1, above with the _S_IFDIR, which
 *      is defined to be the octal constant 0040000.  In decimal,
 *      that octal constant is 16384.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P S_ISDIR         B
D S_ISDIR         PI             1N
D   mode                        10U 0 value

D                 DS
D  dirmode                1      4U 0
D  byte1                  1      1A
D  byte2                  2      2A
D  byte3                  3      3A
D  byte4                  4      4A

C* Turn off bits in the mode, as in step (1) above.
c                   eval      dirmode = mode

c                   bitoff    x'FF'         byte1
c                   bitoff    x'FE'         byte2
c                   bitoff    x'0F'         byte3
c                   bitoff    x'FF'         byte4

C* Compare the result to 0040000, and return true or false.
c                   if        dirmode = 16384
c                   return    *On
c                   else
c                   return    *Off
c                   endif
P                 E

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

You'll need to run this from the QSHELL. If your system doesn't have QSHELL installed, you've got a choice. You can either try to modify my code so that it doesn't need QSHELL, and make it work, or you can install QSHELL. QSHELL should be on your OS/400 CD's, and you should be able to install it free-of-charge.

To run this, do the following:

1. If you haven't done so already, compile the program according to the instructions at the top of the code.
2. Start the QSHELL by typing: **`STRQSH`** at your OS/400 command prompt.
3. Run the command by typing: **`/QSYS.LIB/IFSEBOOK.LIB/CH7QSHDIR.PGM`** at the QSHELL command prompt.

4. If that's too much to type, create a symbolic link to the program by typing:

   cd /ifstest

   ln -s /QSYS.LIB/IFSEBOOK.LIB/CH7QSHDIR.PGM dir

   And then adding the /ifstest directory to your path by typing:

   PATH=${PATH}:/ifstest

5. Now you can just type **dir** to list the contents of the directory.

7.10. Example of Reading a directory recursively

One more program. This one shows how to read through directories "recursively."

As you know, a directory can contain sub-directories. Those sub-directories can contain more sub-directories. How can you write a program that will process them all, when you don't know how many levels deep it can go?

The answer is "recursion." Recursion is the ability for a sub-procedure to call itself. Each new call to the sub-procedure has it's own copies of the variables that it uses (unless they are global or static.) This is useful to us, since it means that we can write a sub-procedure that lists out the contents of a directory. When it finds a sub-directory in that directory, it can call itself with the sub-directory's name. The new copy of the sub-procedure will process that directory, again looking for more directories, etc.

It's a bit of a difficult concept to understand the first time you see it. It helps to think of it as a "call stack", like programs have. Each time the procedure calls itself, think of it adding a new entry to that stack. When the newly called procedure ends, it's entry on the stack is removed, and the copy that made the call continues where it left off.

Once again, we'll use QSHELL. But this time, instead of looking like an MS-DOS DIR command, we'll just print the filename to the screen. Hopefully, when you see what it's outputting, you'll understand how the recursion works.

```
 * CH7RECURSE: Just in case that last example wasn't complicated
 *   enough!
 *  (From Chap 7)
 *
 * To compile:
 *   CRTBNDRPG CH7RECURSE SRCFILE(xxx/QRPGLESRC) DBGVIEW(*LIST)
 *
H DFTACTGRP(*NO) ACTGRP(*NEW) BNDDIR('QC2LE') BNDDIR('IFSTEXT')

D/copy IFSEBOOK/QRPGLESRC,IFSIO_H
D/copy IFSEBOOK/QRPGLESRC,ERRNO_H
D/copy IFSEBOOK/QRPGLESRC,IFSTEXT_H

D show_dir        PR            10I 0
D   curdir                    1024A   varying const

D STDIN           C                   CONST(0)
D STDOUT          C                   CONST(1)
D STDERR          C                   CONST(2)

D S_ISDIR         PR             1N
D   mode                        10U 0 value
```

```
D curr            s           1024A
D curdir          s           1024A   varying
D line            S           1024A
D len             S             10I 0

D cmd             PR                  ExtPgm('QCMDEXC')
D  command                     200A   const
D  length                       15P 5 const

c                   eval      *inlr = *on

c                   callp     cmd('DLYJOB DLY(10)': 20)

C*****************************************
C* Use the getcwd() API to find out the
C* name of the current directory:
C*****************************************
c                   if        getcwd(%addr(curr): %size(curr)) = *NULL
c                   eval      line = 'getcwd(): ' +
c                                    %str(strerror(errno))
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDERR: %addr(line): len)
c                   return
c                   endif

c                   eval      curdir = %str(%addr(curr))

C*****************************************
C*  Call our show_dir proc to show all
C*  of the files (and subdirectories) in
C*  the current directory.
C*****************************************
c                   callp     show_dir(curdir)
c                   return


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  prints all of the files in the directory to STDOUT.
 *
 *  if a subdirectory is found, this procedure will call
 *  itself (recursively) to process that directory.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P show_dir        B
D show_dir        PI            10I 0
D   curdir                    1024A   varying const

D mystat          S                   like(statds)
D dir             S                *
D line            S           1024A
D len             S             10I 0
D filename        S           1024A   varying
D fnwithdir       S           1024A   varying
```

```
D err             S             10I 0

C*****************************************
C* open the current directory:
C*****************************************
c                   eval      dir = opendir(curdir)
c                   if        dir = *NULL
c                   eval      err = errno
c                   eval      line = 'opendir(): ' +
c                                    %str(strerror(err)) +
c                                ', errno=' + %trim(%editc(err:'L'))
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDERR: %addr(line): len)
c                   if        err = EACCES
c                   return    0
c                   else
c                   return    -1
c                   endif
c                   endif

c                   eval      p_dirent = readdir(dir)

c                   dow       p_dirent <> *NULL

c                   eval      filename = %subst(d_name:1:d_namelen)
c                   eval      fnwithdir = curdir + '/' + filename

c                   if        filename<>'.' and filename<>'..'

c                   if        stat(fnwithdir: %addr(mystat))<0
c                   eval      line = 'stat(): ' +
c                                    %str(strerror(errno))
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDERR: %addr(line): len)
c                   return    -1
c                   endif

c                   eval      line = fnwithdir
c                   eval      len = %len(%trimr(line))
c                   callp     writeline(STDOUT: %addr(line): len)

c                   eval      p_statds = %addr(mystat)
c                   if        S_ISDIR(st_mode)
c                   if        show_dir(fnwithdir) < 0
c                   return    -1
c                   endif
c                   endif

c                   endif

c                   eval      p_dirent = readdir(dir)
c                   enddo
```

```
c                   callp     closedir(dir)

c                   return    0
P                 E


 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *  This tests a file mode to see if a file is a directory.
 *
 * Here is the C code we're trying to duplicate:
 *      #define _S_IFDIR    0040000                                    */
 *      #define S_ISDIR(mode) (((mode) & 0370000) == _S_IFDIR)
 *
 * 1) ((mode) & 0370000) takes the file's mode and performs a
 *      bitwise AND with the octal constant 0370000.  In binary,
 *      that constant looks like: 00000000000000011111000000000000
 *      The effect of this code is to turn off all bits in the
 *      mode, except those marked with a '1' in the binary bitmask.
 *
 * 2) ((result of #1) == _S_IFDIR)  What this does is compare
 *      the result of step 1, above with the _S_IFDIR, which
 *      is defined to be the octal constant 0040000.  In decimal,
 *      that octal constant is 16384.
 *+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
P S_ISDIR         B
D S_ISDIR         PI             1N
D   mode                        10U 0 value

D                 DS
D  dirmode                1      4U 0
D  byte1                  1      1A
D  byte2                  2      2A
D  byte3                  3      3A
D  byte4                  4      4A

C* Turn off bits in the mode, as in step (1) above.
c                   eval      dirmode = mode

c                   bitoff    x'FF'         byte1
c                   bitoff    x'FE'         byte2
c                   bitoff    x'0F'         byte3
c                   bitoff    x'FF'         byte4

C* Compare the result to 0040000, and return true or false.
c                   if        dirmode = 16384
c                   return    *On
c                   else
c                   return    *Off
c                   endif
P                 E

 /DEFINE ERRNO_LOAD_PROCEDURE
 /COPY IFSEBOOK/QRPGLESRC,ERRNO_H
```

Once again, you'll need to compile the program, start up QSHELL, and run it from the QSHELL prompt. The command that you'll run will be:

```
/QSYS.LIB/IFSEBOOK.LIB/CH7RECURSE.PGM
```