RPG and the IFS

Presented by

Scott Klement

"There are 10 types of people in the world.
Those who understand binary, and those who don't."

What is the IFS?

The Integrated File System, or IFS, is an interface that IBM i programs can use to access various types of files, including:

- Stream (PC style) files can be stored on IBM i DASD with the same naming conventions as used in Windows, MS-DOS or Unix

- Stream files stored on a CD-ROM or optical drive

- Stream files stored on a Windows (or Samba) server

- Stream files stored on Unix/Linux servers

- Files stored on a different IBM i system

- "Normal" IBM i Libraries and Objects

2

What's integrated into the IFS?

To access a file system, start the path name with a different value...   (Some examples, later!)

`/QOPT`	Optical filesystem, CD-Rom
`/QFileSvr.400`	Other IBM i Systems and their directories
`/QNTC`	Windows file shares
`/QSYS.LIB`	IBM i traditional libraries and objects
`/QDLS`	OfficeVision and old Shared Folders
`/QOpenSys`	Unix-like case sensitive files and folders
`(your choice)`	Mounted NFS directories
`/`	Everything else falls under "root"

3

Tools to Explore the IFS Interactively

- The WRKLNK command from the IBM i command line in a 5250 session (similar to WRKOBJ!)

- QShell or PASE from a 5250 session

- iSeries Navigator

- Windows "Network Neighborhood" (but the various parts of the IFS must be "shared")

- NFS from Linux or another Unix-like system

- FTP, if you specify "`quote site NAMEFMT 1`"

However, today I'll show you how to access the IFS from your RPG programs!

4

Hello World

```
H DFTACTGRP(*NO)

 /copy ifsio_h

D fd              s              10I 0
D data            s              50A

 /free

    fd = open('/tmp/hello.txt': O_CREAT+O_TRUNC+O_WRONLY: M_RDWR);
    if (fd < 0);
        // check for error here.
    endif;

    data = 'Hello World!' + x'0d25';
    callp write(fd: %addr(data): %len(%trimr(data)));

    callp close(fd);

    *INLR = *ON;
 /end-free
```

The IFS isn't only for /FREE RPG

```
H DFTACTGRP(*NO)

 /copy ifsio_h

D name            s             100A    varying
D x               s              10I 0
D data            s              50A

c                 eval      name = '/tmp/hello.txt'
c                 eval      x = open( name
c                               : O_CREAT+O_TRUNC+O_WRONLY
c                               : M_RDWR )
c                 if        x < 0
 * check for error here
c                 endif

c                 eval      data = 'Hello World!' + x'0d25'
c                 callp     write(x : %addr(data)
c                               : %len(%trimr(data)) )
c                 callp     close(x)

c                 eval      *inlr = *on
```

6

Differences between RPG I/O and "UNIX-style" or "C-style" I/O

RPG "hard-wires" each file to the program

- You can use an OVRDBF, but the file will still "look" the same
- Each READ, WRITE, UPDATE, etc. is tied to a specific file
- If you want to process two files, you need separate F-specs, separate READ statements, etc.

That's not the way that C programs work!

- A single I/O operation can work on many files

Then, how does it know which file to operate on?

- The open API gives you a different number ("descriptor") for each file you open
- You supply this number to each subsequent API to tell it which file to access

7

One operation for many files

```
      D name              s              100A    varying
      d x                 s              10I 0
      d fd                s              10I 0 dim(10)

… later in program …

        for x = 1 to %elem(fd);

            name = '/tmp/test' + %char(x) + '.txt';
            fd(x) = open(name: O_CREAT+O_TRUNC+O_WRONLY: M_RDWR);

            if ( fd(x) < 0 );
                // check for error
            endif;

        endfor;
```

8

Without an F-spec, how do you…

Specify I=Input, O=Output, U=Update?
- You pass this in the "openflags" parameter to the API
- It can be O_RDONLY, O_WRONLY, or O_RDWR
- Many other things can be specified as you'll soon see!

What about the record length?
- The operating system does not organize IFS files into records
- Records are determined by program logic

What kind of program logic?
- Sometimes you use fixed-length records
- Other times, you might look for a special character sequence
- CR/LF (EBCDIC x'0d25') is a very common sequence
- You can use any sequence that you like

9

The open() API (slide 1 of 4)

The following is the RPG prototype for the IFS open() API. It can be found found in the IFSIO_H /COPY member.

```
D open            PR            10I 0 ExtProc('open')
D  path                          *   value options(*string)
D  openflags                    10I 0 value
D  mode                         10U 0 value options(*nopass)
D  ccsid                        10U 0 value options(*nopass)
D/if defined(*V5R2M0)
D  txtcreatid                   10U 0 value options(*nopass)
D/endif
```

Where:

✓Path = path name of the file in the IFS, such as:

   /home/scottk/notes/mytest.txt

✓Openflags = options for how the file is opened (read only, write only,

   create the file if it doesn't exist, etc.) – more about this later!

✓Mode = permissions ("authority") – more about this later!

✓Ccsid = For globalization and EBCDIC / ASCII translation. More later!

10

Open() API Considerations

The open API returns a file descriptor

- This is a number that distinguishes the file from other open files
- The number can be zero or higher
- If –1 is returned, an error prevented the file from opening

The file will remain open until you close it

- Even if the program ends with *INLR on, the file stays open
- RCLRSC <u>does not</u> close IFS files
- RCLACTGRP might work
- If all else fails, the file is closed when the job ends
- The job ends when you sign off, or when a batch job completes

11

Null-terminate and remove blanks

The *PATH* parameter

- There are no fixed-length strings in C. All strings are variable.
- Variable-length strings in C end with a x'00' character
- OPTIONS(*STRING) will automatically add that character
- You do not have to pass a pointer with OPTIONS(*STRING)

Pre-V3R7 examples will show code like the following:

```
D myPath           s              100A

c                  eval       myPath = %trim(file) + x'00'
c                  eval       x = open( %addr(myPath) : O_RDONLY )
```

- **Don't do it that way! Let the compiler do the work for you!**
- **Manually adding `x'00'` is unnecessary with options(*string)**
- **The use of `%addr()` and/or pointers is <u>not</u> needed!**
- **But <u>do</u> use `%trimr()` unless you use a `VARYING` field**

```
c                  eval       x = open( %trimr(file) : O_RDONLY )
  /free
       fd = open( %trimr(mydir) + '/' + %trimr(myfile) : O_RDONLY );
```

12

Some Path Examples

```
D path1            s              32768A
D path2            s               1024A    varying
D scFile           s                 60A
D wkFile           s                 60A    varying
 /free

        // Root file system, %trimr removes blanks
        path1 = '/tmp/test.txt';
        f1 = open(%trimr(path1): O_RDONLY);

        // QDLS file system, no blanks added!
        path2 = '/QDLS/MYFLR/MYFILE.DAT';
        f2 = open(path2: O_RDONLY);        // QDLS file system

        // Optical file system, no blanks added!
        f3 = open('/QOpt/job31760/addresses.csv': O_RDONLY);

        // A screen field will have blanks, but they can
        // be stripped when added to a VARYING field
        scFile = '/QNTC/SERVER4/policies/dress_code.txt';
        wkFile = %trimr(scFile);
        f4 = open(wkFile: O_RDONLY);
```

13

Open Flags (Each bit is an option)

The *OpenFlags* parameter
- This parameter represents a series of bits rather than a value
- Each bit in the binary number represents a different option

Create is (binary) 1000, or 8 in decimal.
Write only is (binary) 0010, or 2 in decimal.
If you specify them both, you specify (binary) 1010 – or 10 in decimal.
They can also be added in decimal. 8+2=10

14

More flags to use with the open() API

Bit	Description	Constant
29	Large File Support	O_LARGEFILE
28		
27	Inherit Mode of Dir	O_INHERITMODE
26		
25	Create as a text file	O_TEXT_CREAT
24	Read as a text file	O_TEXTDATA
23	Specify a codepage	O_CODEPAGE
22		
21		
20		
19	Don't Share	O_SHARE_NONE
18	Share for reading/writing	O_SHARE_RDWR
17	Share for write only	O_SHARE_WRONLY
16	Share for read only	O_SHARE_RDONLY
15		
14		
13		
12		
11		
10		
9		
8	Append any new data	O_APPEND
7		
6	Truncate the file to 0	O_TRUNC
5	Specify a CCSID	O_CCSID
4	Exclusively create	O_EXCL
3	Create the file if needed	O_CREAT
2	Open for read and write	O_RDWR
1	Open for writing only	O_WRONLY
0	Open for reading only	O_RDONLY

15

Open flags in IFSIO_H

Numbers in RPG are specified as decimal numbers. The easiest way to keep them straight is to code them as named constants in a separate member, and use `/copy` to bring them into your code. The following are some of the constants that I've defined in the `IFSIO_H` source member.

```
 *   00000000000000000000000000000001        Reading Only
D O_RDONLY         C                   1
 *   00000000000000000000000000000010        Writing Only
D O_WRONLY         C                   2
 *   00000000000000000000000000000100        Reading & Writing
D O_RDWR           C                   4
 *   00000000000000000000000000001000        Create File if needed
D O_CREAT          C                   8
 *   00000000000000000000000000010000        Exclusively create --
 *                                              open will fail if it
 *                                              already exists.
D O_EXCL           C                   16
 *   00000000000000000000000000100000        Assign a CCSID to new
 *                                            file.
D O_CCSID          C                   32
 *   00000000000000000000000001000000        Truncate file to 0 bytes
D O_TRUNC          C                   64
 *   00000000000000000000000100000000        Append to file
 *                                            (write data at end only)
D O_APPEND         C                   256
```

16

Why Adding Works...

When you include these definitions in your source, you can add them together to specify the options that you want:

```
flags = O_CREAT + O_TRUNC + O_WRONLY;
  74  =    8   +   64   +    2
```

Binary
```
O_WRONLY                 10
O_CREAT                1000
O_TRUNC          +  1000000
        total       1001010
```
(Which is decimal 74!)

Decimal
```
O_WRONLY                  2
O_CREAT                   8
O_TRUNC           +      64
         total           74
```
(Which is binary 1001010!)

- **The only way you can get 74 is by adding these 3 flags**
- **Specifying `VALUE` on the prototype makes it possible for you to use an expression**

```
// these flags will be totaled first, then the total
// value of 74 will be passed as a parm to the API

fd = open(name: O_CREAT+O_TRUNC+O_WRONLY: M_RDWR);
```

The MODE (Authority) Parameter

The *Mode* parameter

- The bits of this parm represent the file's authorities
- The *MODE* parameter is only used when creating a new file
- This parameter is only required if you specify the O_CREAT open flag
- An IFS file is owned by both a user and a group
- You specify separate authorities for the user, group and public

Owner			Group			Public		
256	128	64	32	16	8	4	2	1
R	W	X	R	W	X	R	W	X

18

Mode Flags in IFSIO_H

Since I only use a few different file permission modes, I like to specify named constants for the ones that I use.

```
D M_RDONLY          C                    const(292)
D M_RDWR            C                    const(438)
D M_RWX             C                    const(511)
```

M_RDWR is 438 because:

```
    Owner read & write   128 + 256 = 384
    Group read & write    32 +  16 =  48
   Public read & write     4 +   2 =   6
                           total   438
```

These are the "normal" flags if you want to specify each bit individually:

```
D S_IRUSR           C                    256
D S_IWUSR           C                    128
D S_IXUSR           C                    64

D S_IRGRP           C                    32
D S_IWGRP           C                    16
D S_IXGRP           C                    8

D S_IROTH           C                    4
D S_IWOTH           C                    2
D S_IXOTH           C                    1
```

19

CCSID and TXTCREATID parameters

The CCSID parameter
- This is the CCSID that will be assigned to a new file
- This parameter is only used with O_CCSID or O_CODEPAGE
- For new files, it is the CCSID the file is tagged with
- With O_TEXTDATA, it's also the CCSID of the data supplied
  (unless the V5R2 flag, O_TEXT_CREAT, is also given)
- In V4R5 and earlier, this is a code page, not a full CCSID

The TXTCREATID parameter
- Only available in V5R2 and later
- Specifies the CCSID of the data if the file is created
- Only used when O_TEXT_CREAT is specified
- O_TEXT_CREAT requires O_CCSID, O_CREAT and
  O_TEXTDATA to be specified

When used with the O_TEXTDATA and O_CCSID flags, these parms can be used to automatically translate ASCII to EBCDIC or vice-versa.

20

More about CCSIDs

Some CCSIDs that I use are:

37	EBCDIC for US, Canada, Netherlands, Portugal, Brazil, New Zealand, Australia
285	EBCDIC for United Kingdom
819	ISO 8859-1 ASCII
1252	ASCII used by Windows for Latin-1 alphabets
1208	UNICODE UTF-8 (Requires V5R1 and O_CCSID – not O_CODEPAGE)

This assigns CCSID 819 when the file is created:

```
fd = open('/tmp/ccsid_test.txt'
          : O_CREAT+O_CCSID+O_WRONLY
          : M_RDWR
          : 819 );
```

Examples of open() with CCSIDs

The following will create a file and translate any data written to ASCII:

```
file1 = open( '/hr/payroll/weekly_nacha.dat'
              : O_WRONLY + O_CREAT + O_TRUNC + O_CCSID
              : M_RDWR
              : 819 );
callp close(file1);

file1 = open( '/hr/payroll/weekly_nacha.dat'
              : O_WRONLY + O_TEXTDATA );
```

Note that the translation might not be right if the file already exists. CCSID is only assigned if the file is created. If you want to be sure, delete the file first:

```
unlink('/hr/payroll/weekly_nacha.dat');
```

In V5R2, the O_TEXT_CREAT flag lets you do the same thing in a single call to the open() API. When you specify a CCSID of 0, it means "the current job's CCSID."

```
file2 = open( '/hr/payroll/weekly_nacha.dat'
              : O_WRONLY + O_CREAT + O_TRUNC + O_CCSID
                + O_TEXTDATA + O_TEXT_CREAT
              : M_RDWR
              : 819
              : 0 );
```

The write() API

The `write()` API writes data from the computer's memory to the file.

The prototype for the `write()` API is in the `IFSIO_H` /copy member, and looks like this:

```
D write          PR        10I 0 ExtProc('write')
D  fildes                  10i 0 value
D  buf                      *    value
D  bytes                   10U 0 value
```

The parameters of the write() API are:
- `Fildes` = the descriptor that was returned by `open()`
- `buf` = pointer to memory where data to write resides
- `bytes` = number of bytes of memory to write

The `write()` API returns the length of the data written, or –1 if an error occurs.

```
len = write( file2: %addr(data): %len(data));

if (len < 1);
    // ack!  It failed!
endif;
```

Writing Different Data Types

Since the data to write is specified by a pointer, you are not limited to writing text.

```
D myPacked        s            9P 2

    callp write( fd: %addr(myPacked): %size(myPacked));
```

Beware! Translating packed or other data types to ASCII might cause them to be difficult to read. Non-text should never be written to a file when O_TEXTDATA is supplied.

If you're not comfortable with pointers, it's possible to prototype the `write()` API so that you don't need to use them:

```
D writeA          PR        10I 0 ExtProc('write')
D  fildes                   10i 0 value
D  buf                    65535A   const options(*varsize)
D  bytes                   10U 0 value
```

```
D  text                    100A

    callp writeA( fd: text: %len(%trimr(text)));
```

24

One More Write Example

- **This prototype works because passing a parameter by reference is the same thing, under the covers, as passing a pointer to that parameter by value.**

- **Options(*VARSIZE) makes it possible to pass strings of any length. Since the API determines the length from the 3rd parameter, any size variable is possible.**

- **The `const` keyword tells the compiler that the API won't change the value that's passed. This is helpful because it lets the compiler do some of the work. You can now pass an expression or VARYING field and let the compiler convert it to something the API will understand.**

The writeA() prototype works very nicely with VARYING fields:

```
D CRLF             c                   x'0d25'
D fd               s             10I 0
D html             s            500A   varying

    html =
    '<html>'                             + CRLF +
    ' <head>'                            + CRLF +
    '    <title>This is a test</title>'  + CRLF +
    ' </head>'                           + CRLF +
    ' <body>'                            + CRLF +
    '    <h1>This is a test HTML file</h1>' + CRLF +
    ' </body>'                           + CRLF +
    '</html>'                            + CRLF ;

    writeA( fd : html : %len(html) );
```

The read() API

The `read()` API reads data from a file in the IFS to memory. It's parameters, shown below, are very similar to that of the `write()` API.

```
D read              PR            10I 0 ExtProc('read')
D  fildes                         10i 0 value
D  buf                             *    value
D  bytes                          10U 0 value
```

The parameters of the `read()` API are:
- **`Fildes` = the descriptor that was returned by `open()`**
- **`buf` = pointer to memory that data will be stored in**
- **`bytes` = maximum amount of data the API can read on this call**
       (Make sure this value is never larger than your variable!)

The `read()` API returns the length of the data read. That length will be shorter than the `bytes` parameter if the end of the file was reached.

The API will return –1 if an error occurs.

26

Block Read Example

You can read a fixed-length record from an IFS file like this:

```
f = open('/home/klemscot/demo.data': O_RDONLY+O_TEXTDATA);
if ( f < 0 );
    // oh no! not again! ahhhhhhh!
endif;

len = read( f : %addr(data) : %size(data) );

if (len < %size(data));
   msg = 'Only ' + %char(len) + ' bytes were found!';
endif;

callp close(f);
```

This works well for a fixed-length chunk of data, but…

- **It won't work for a CR/LF style record!**
- **It always reads as much data as will fit (unless it reaches the end of the file) so how can you read until you get a CR/LF sequence?**

Reading Text Files

You can read one character at a time from a file in the IFS until you get a line feed character.

```
// because line is varying, the max length is two less than
// it's size in bytes.

LineMax = %size(line) - 2;

dow ( read( fd : %addr(char) : 1 ) = 1 );

      if ( char<>CR and char<>LF );
          line = line + char;
      endif;

      if ( char=LF or %len(line)=LineMax );
          except PrintLine;
          Line = '';
      endif;

enddo;
```

Although this method works, it's not very efficient. For some better alternatives, check out my web site, or the article "Text File Primer" from December 2004 issue of System iNEWS.

The close() API

When you're done with a file, you close it by calling the `close()` API.

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

Close API parameters:

 • `fildes` = file descriptor to close (returned by the `open()` API)

You should <u>always</u> close a file when you're done with it. It won't automatically be closed when the program ends. However, if the activation group is destroyed, or your job ends, the file will be closed.

Like most Unix-type APIs, `close()` will return a –1 if an error occurs. That's not very helpful, is it?

What can you do if the file won't close? Even though this returns a value, you can ignore that value by calling it with the `CALLP` opcode.

```
      callp close(fd);
```

Handling Errors

The APIs return –1 when an error occurs. Checking this value tells you that an error occurred, but it does not tell you what the error was!

The Unix-type APIs return error information by setting the value of a variable in the ILE C runtime library called "errno."

The `__errno()` function can be used to get a pointer to the error number.

```
D get_errno       pr            *   ExtProc('__errno')

D ptrToErrno      s             *
D errno           s            10I 0 based(ptrToErrno)

 /free

    fd = open('/DirThatDoesntExist/hello.txt'
             : O_CREAT+O_TRUNC+O_WRONLY: M_RDWR );

    if (fd < 0);
       ptrToErrno = get_errno();
       msg = 'open failed with error number ' + %char(errno);
       dsply msg '' wait;
       return;
    endif;
```

The error number will match a CPE message in the QCPFMSG message file.
For example, If errno = 3021, then type `DSPMSGD CPE3021` to get more info.

Nicer Error Messages

The `strerror()` function can be used to get a text message for an error number.

```
D strerror        pr                  *   ExtProc('strerror')
D   errno_val                    10I 0 value

         msg = %str(strerror(errno));
```

Note: In order to get `errno` or call the `strerror()` function, you must tell your program to bind to the ILE C runtime library.

On the H-spec:

```
H BNDDIR('QC2LE')
```

Additional binding directories can be listed on the same statement:

```
H BNDDIR('QC2LE': 'OTHERDIR': 'MYBNDDIR')
```

Error Constants

- Each `errno` value corresponds to a constant
- The following is a small sampling of constants that correspond to `errno`

```
     *  Permission denied.
D EACCES         C                3401
     *  Not a directory.
D ENOTDIR        C                3403
     *  No space available.
D ENOSPC         C                3404
     *  Improper link.
D EXDEV          C                3405
```

- These constants are listed in the API manuals instead of the actual numbers
- I've included my definitions of these constants in the ERRNO_H source member, available from my Web site. **http://www.scottklement.com/presentations/**
- The constants work nicely in programs for handling "expected errors"

```
/copy ERRNO_H

   ptrToErrno = get_errno();

   select;
   when   errno = EACCES;
      // handle "user doesn't have authority"
      //  by send a message to the security officer.

   other;
      // handle other errors here.
```

Reading Directories

- **Reading the contents of an IFS directory requires 3 APIs**
- **They are `OpenDir()`, `ReadDir()` and `CloseDir()`**
- **A fourth API called `rewinddir()` can be used to re-read the directory from the start**
- **The prototypes for these APIs are in the IFSIO_H member, and are listed below:**
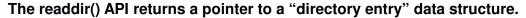
```
D opendir         PR              *    EXTPROC('opendir')
D  dirname                        *    VALUE options(*string)

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

D rewinddir       PR                   ExtProc('rewinddir')
D   dirp                          *    value

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

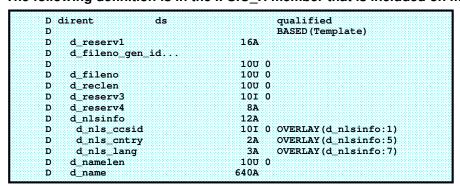**`Opendir()` opens up a directory, just as `open()` opens a file.**

- **`Dirname` = directory name to open. Notice `options(*string)`!**
- **`Opendir()` returns pointer to a "directory handle." It works like a descriptor. You pass it to the other APIs.**
- **`Opendir()` returns `*NULL` if an error occurs, and `errno` can be checked.**

The "`dirp`" parameter to the other APIs is the pointer that `opendir()` returned. 33

Directory Entries

The readdir() API returns a pointer to a "directory entry" data structure.

- **This data structure works nicely with `QUALIFIED` and `LIKEDS`**
- **The following definition is in the IFSIO_H member that is included on my site**

```
D dirent          ds                   qualified
D                                      BASED(Template)
D   d_reserv1                   16A
D   d_fileno_gen_id...
D                               10U 0
D   d_fileno                    10U 0
D   d_reclen                    10U 0
D   d_reserv3                   10I 0
D   d_reserv4                    8A
D   d_nlsinfo                   12A
D    d_nls_ccsid                10I 0 OVERLAY(d_nlsinfo:1)
D    d_nls_cntry                 2A   OVERLAY(d_nlsinfo:5)
D    d_nls_lang                  3A   OVERLAY(d_nlsinfo:7)
D   d_namelen                   10U 0
D   d_name                     640A
```

Usually there are two useful bits of informaton in a directory entry:

- **The `d_name` and `d_namelen` fields are useful for getting the file names**
- **The `d_nls_ccsid` field is useful for determining the CCSID of the file name.**

The `d_name` field is defined as 640 chars long, but most file names aren't that long. The unused part of it will not be blank-filled. You have to use `d_namelen` to retrieve the "good part." 34

The following is a sample program that illustrates the directory APIs

```
        H DFTACTGRP(*NO) BNDDIR('QC2LE')

       FQSYSPRT   O    F  132        PRINTER

        /copy ifsio_h

        d dirh          s            *
        d p_entry       s            *
        d entry         ds               likeds(dirent)
        d                                based(p_Entry)
        D name          s          132A
        D shortname     s           52A
        D msg           s          132A

        D strerror      pr            *   ExtProc('strerror')
        D   errno_val             10I 0 value
        D get_errno     pr            *   ExtProc('__errno')
        D p_errno       s             *
        D errno         s          10I 0 based(p_errno)
```

Notice the following:
- The *dirh* pointer will be used for the directory handle.
- The *entry* data structure is defined with `LIKEDS()` to get the same fields as *dirent*.
- The *entry* data structure is based on the p_entry pointer.
- The definitions for `errno` & `strerror` are included.

35

The following is a sample program that illustrates the directory APIs

```
        /free

          // ----------------------------------------
          //   open a directory.
          // ----------------------------------------

          dirh = opendir('/QIBM/UserData');

          if (dirh = *NULL);
             p_errno = get_errno();
             msg = 'opendir(): ' + %str(strerror(errno));
             except error;
             *inlr = *on;
             return;
          endif;
```

Notice the following:
- If *dirh* is null. An error occurred.
- Like all Unix-type APIs, this uses *errno* and *strerror()* to get info about the error.
- I end the program immediately after printing the error message.

36

Directory Example (3 of 3)

After reading the entries once, I can re-read them with `rewinddir()`

```
// ----------------------------------------
//   read entries and print them
// ----------------------------------------

 p_entry = readdir(dirh);

 dow ( p_entry <> *NULL );

    name = %subst(entry.d_name: 1: entry.d_namelen);
    except print;

    p_entry = readdir(dirh);
 enddo;

// ----------------------------------------
//   read entries again, this time dsply them
// ----------------------------------------

 rewinddir(dirh);

 p_entry = readdir(dirh);
 dow ( p_entry <> *NULL );
    shortname = %subst(entry.d_name: 1: entry.d_namelen);
    dsply shortname;
    p_entry = readdir(dirh);
 enddo;

 callp closedir(dirh);
```

Check if a file exists

The access() API can check if a file exists in the IFS.

It can also check if the current user has read, write or execute authority.

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

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

```
somefile = '/home/scottk/ramble.txt';

if ( access( %trimr(somefile) : F_OK ) = 0 );
   msg = 'File exists!';
endif;

if ( access( %trimr(somefile) : R_OK + W_OK ) = 0 );
   msg = 'You have read & write authority!';
endif;
```

Get File Information (1 of 2)

The stat() API gets information about a file in the IFS

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

D statds          DS                  qualified
D                                     BASED(Template)
D  st_mode                      10U 0
D  st_ino                       10U 0
D  st_nlink                      5U 0
D  st_reserved2                  5U 0
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                   11A
D  st_reserved3                  1A
D  st_codepage                   5U 0
D  st_ccsid                      5U 0
D  st_rdev                      10U 0
D  st_nlink32                   10U 0
D  st_rdev64                    20U 0
D  st_dev64                     20U 0
D  st_reserved1                 36A
D  st_ino_gen_id                10U 0
```

39

Get File Information (2 of 2)

The first parm is the path name of the file you want information for.

The second parm is the data structure that the information will be returned in.

The return value will be 0 if it's successful, or –1 if there's an error.

```
D filename        s            1000A  varying
D filestats       ds                  likeds(statds)

     filename = '/tmp/hello.txt';

     rc = stat(filename: filestats);
     if (rc < 0);
        // check errno
     endif;

     msg = filename + ' is ' + %char(filestats.st_size) +
           ' bytes long.';
```

40

Other Useful APIs

The `chmod()` API can change the authorities on an existing file

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

    rc = chmod('/home/scottk/ramble.txt': M_RDONLY);
    if (rc < 0);
        // check errno
    endif;
```

The `mkdir()` API creates a new directory

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

    rc = mkdir('/home/scottk/testdir': M_RWX);
    if (rc < 0);
        // check errno
    endif;
```

Other Useful APIs, continued…

The `unlink()` API deletes an existing file

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

    somefile = '/QNTC/SERVER4/scott/oldjunk.dat';
    rc = unlink(%trimr(somefile));

    if (rc < 0);
        // check errno
    endif;
```

The `rmdir()` API deletes an existing directory

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

    rc = rmdir('/home/scottk/tempdir');
    if (rc < 0);
        // check errno
    endif;
```

For more information...

The IFS APIs are documented in the IBM Knowledge Center under:
- **Programming / APIs / APIs By Category / Unix-Type APIs / Integrated File System APIs.**
- **The documentation assumes a C programmer, but works fine in RPG**
- **IBM provides some prototypes in QSYSINC library (System Openness Includes)**
- **Others are available from my web page.**

Scott's web site has a tutorial (fixed-format RPG with V4 level code)
- **http://www.scottklement.com/**
- **Click "RPG" then "Reading & Writing from the Integrated File System"**

Who Knew You Could Do That with RPG IV?
> *A Sorcerer's Guide to System Access and More*
- **The RPG redbook contains information on IFS programming and much more.**
- **http://www.redbooks.ibm.com/abstracts/sg245402.html**

43

This Presentation

**You can download a PDF copy of this presentation, and get
the IFSIO_H and ERRNO_H copy members from:**

http://www.scottklement.com/presentations/

Thank you!

44