Combining the C Library and Native File System APIs in a Single NET+OS Application

1 Document History

Date	Version	Change Description	
7/29/09	V1.0	Initial entry	
7/31/09	V1.1	First Edits	
7/31	V1.2	Add in edits	

2 Table of Contents

3 Introduction

This document describes creating files in the NET+OS file system using C library API calls and later accessing those same files using the native NET+OS file system API calls.

3.1 Problem Solved

NET+OS provides two sets of APIs for manipulating (creating, deleting, reading, writing…et al) files in the NET+OS file system. One of these API sets is the C library file system APIs. The C library file system APIs are calls such as fopen, open, fread, read, fwrite, and close (for example). These calls should be familiar to any C programmer as they are operating environment agnostic. The other set of APIs is specific to the NET+OS development environment. An example set of these APIs might be NAFSopen_file, NAFSclose_file, NAFScreate_file….). These APIs would not be familiar to all C programmers but might be familiar to experienced NET+OS developers.

The question is whether these two sets of APIs are proper sets. The answer is no. The C library set of APIs does not give the developer access to information such as file size, whether a file is a directory or a file. The problem is that many developers are more comfortable using the C library calls for their file system code and only want to use NET+OS native mode calls for special purposes. But can a file that was created using C library APIs be accessed by native NET+OS APIs? The answer is yes. This paper includes an example application that demonstrates this use.

3.2 Audience

This paper is geared for developers who are familiar with both C library-based file system calls and NET+OS native file system calls. In addition, we assume that the reader is familiar with developing system within the NET+OS environment. Further, we expect that users of this document have some familiarity with manipulation of files in the C programming language.

3.3 Assumptions

This document assumes that the user is developing a system or systems using Digi International's NET+OS development environment. Further the attached application assumes that the BSP is creating the file system within the application by setting the manifest constant *BSP_INCLUDE_FILESYSTEM_FOR_CLIBRARY to TRUE*.

3.4 Scope

This paper describes creating files using the C library APIs for file manipulation and subsequently accessing those same files using NET+OS native mode file access APIs.

This paper does not describe the following:

- Developing under the NET+OS environment (command line or ESP)
- Developing in C
- General concepts of file manipulation in C

- Accessing files using ftp, AWS, telnet or other NET+OS components
- A full tutorial of the file system capabilities of the NET+OS environment

To further understand the full set of APIs and the NET+OS environment in general, please refer to the NET+OS API reference guide.

3.5 Theory of Operation

The C library provides a high level generally operating system agnostic method for creating, accessing and closing files. By operating system agnostic I mean that the API calls can (generally) be ported from the NET+OS environment to Windows, and to UNIX. There are probably some C library-based file operations that can't be performed in a NET+OS environment that your application might still need to perform. You might then look at NET+OS's file system API for "filling in the gaps" of operations that you can't perform using the C library file system APIs.

So why not use the NET+OS file system APIs for your entire application? First it is not portable. Second it involves a learning curve that you might not want to go through. Third it is possible that for many applications, the C library file system APIs are adequate for your application. Additionally, the NET+OS native file system APIs are asynchronous. That is you make a request (for example open a file) and then your application must query the NET+OS file system to ascertain whether that operation has completed. The C library, on the contrary, from the C perspective, are atomic (loosely used here). You make a call and when the call returns, the operation is complete (be it successful or unsuccessful). You could look at the NET+OS file system API as having an advantage as you can make a request, go do something else and then check back later to see whether your operation is complete. On the other hand, if your operation is dependent on a file operation being complete, then doing something else might not be an advantage. This is highly application dependent, so it is up to you.

4 Basics

Since this paper describes using APIs for accomplishing a set of tasks, the remainder of this document describes and follows the contents of the attached sample application.

5 Example Application Explanation

I have included a fairly simple application that demonstrates the creation of files using the C library file system APIs and then accessing those same files using the NET+OS native file system APIs. You might want to familiarize yourself with both the C library file system APIs and the NET+OS native mode file system APIs before continuing.

At a high level, the sample application uses C library calls to open four files, write some data to those files and closes the files. Additionally three directories are also created using C library calls. After that the four files are opened using NET+OS native file system calls. The size of each file is accessed and the files are closed. Last, NET+OS native file system calls are used to take a directory of the files and directories.

Copyright 2009 Digi International

We'll now dive into the sample application. Please open (in an editor) the root.c file of the attached sample application before continuing. Example Code

First you'll notice that I have created two tables containing file paths. One contains the file name and the volume, while the other contains just the file name. The C library APIs require the entire file name and path in the string used as a parameter to the API. So for example a call to fopen might look like "**fopen**("RAM0/digi_data.txt", "w+");". While in the NET+OS file system APIs the volume is separated from the file name. So for the example application I use two tables. I could have used a single table of file names and a second with the volume but I chose to implement the example application in the aforementioned way.

Function applicationStart, creates a thread whose entry point is function theFileManipulationThread(). Function theFileManipulationThread() is where the real work is performed.

We'll now look into function theFileManipulationThread(). As an aside, I have added a lot of debug statements that are "ifdefed" out with the manifest constant WANT_DEBUG. Define WANT_DEBUG if you choose to view the additional debug output.

The first portion of function thefileManipulationThread(), in turn, creates a file (fopen), writes some data to the file (fwrite) and closes the file (fclose). Clearly this section is using C library file system APIs.

The second portion creates 3 directories using the C library file system API mkdir.

At this point, all the operations using the C library APIs are complete. The remainder of the sample application uses NET+OS file system APIs.

For all NET+OS file system APIs that perform work (we are excluding the API NAFSio_request_cb_status) you need to "create" an io request block structure. In this application, we use a stack variable as the operations are done serially. Whether you use a stack variable, or allocate memory for this is application dependant and beyond the scope of this paper. You'll notice that I zero out the structure (always a good idea) and then call API **`NAFSinit_io_request_cb`** to initialize the structure (required). Next I call the NET+OS API NAFSopen_file. NAFSopen_file does what the name describes. It opens a file allowing other NET+OS file system APIs to access the file. Notice the parameter theFileHandle is passed as a pointer. NAFSopen_file fills in this pointer with a file handle that other calls will use for references to the now opened file. It is important to note that the status returned by NAFSopen_file (and all other NET+OS file system APIs) *does not* mean that the operation was complete. All the status signifies is that the file manipulation request was (successfully) handed off to the file system thread (the file system operates in a separate thread). Whether or not the actual operation (in this case open) completed is explained next.

After calling NAFSopen_file and checking its return status, you'll notice that the sample application calls the function checkForCompleteStatus. You'll need to move to the bottom of the file root.c to see this function. checkForCompleteStatus, requests from the file system thread, whether an operation was completed. You'll want to take notice of a couple of items. First I am passing the io request block, that was passed to the NAFSopen_file call. Also you will notice that I did not zero out the io request block and I did not call function NAFSinit_io_request_cb. There is information placed in the io request block by NAFSopen_file that will be used in checkForCompleteStatus to tie the operation with the status request. Zeroing out the io request block at this time will keep the file system thread from being able to find your operation and tell you whether or not it completed. We also pass in a constant that describes the type of operation whose status we are seeking. In this first case `NAFS_REQUEST_OPEN_FILE`.

In function checkFor CompleteStatus, we call function NAFSio_request_cb_status. In this simplified sample application, we are doing file operations serially, so we loop continually calling NAFSio_request_cb_status until either we complete successfully or the operation fails with an error. If your application required that you not sit in a function polling file system status, then you'd have to (re)call checkFor CompleteStatus and check status. You'll notice that there is also a callback function called user_file_callback. The file system thread calls your call back function when either your file operation is complete or it has failed. You could let the callback function let your application know that the operation is complete. In this sample, we have chosen to do both. The callback function is set in the call to NAFSinit_io_request_cb when you initialized your io request block.

A note about the callback function. The callback function is optional. I used both a callback function and a call to NAFSio_request_cb_status in order to show the "whole story". For a real application you'd generally want to either use a callback function or get the status of an operation using NAFSio_request_cb_status.

The next chunk of code gets the size of a file. The sample application uses API NAFSopen_file_size for this. Again, notice that first NAFSinit_io_request_cb is called. If NAFSopen_file_size returns a successful status, checkFor CompleteStatus is called.

The next chunk of code closes a previously opened file using NAFSclose_file. As before, call NAFSinit_io_request_cb, followed by NAFSclose_file and if the call to NAFSclose_file completes successfully, call checkFor CompleteStatus to see whether or not the file actually closed.

The last two chunks of code are related. NAFSdir_entry_count returns the number of files and or directories (so the number of objects) on a volume or in a directory that is on a volume. Then NAFSlist_dir returns detailed information about each of the objects in that volume and/or directory. NAFSdir_entry_count is called first because you need to malloc a buffer large enough to hold some number of file information structures, the number defined by the return value from NAFSdir_entry_count. You'll notice that for both of these APIs we call NAFSinit_io_request_cb, call the actual API and then call checkFor

CompleteStatus. Also notice that the number of objects variable returned by NAFSdir_entry_count is invalid until the file system reports that the operation is complete. Also notice that the buffer of file information structures is invalid until the file system reports that the operation is complete. That is, you must (either or both) get a call from the callback or a report from NAFSio_request_cb_status showing operational completion, before accessing the output from the initial API.

Last, in the callback, notice that the io request block is passed as a void *. You must then typecast it before accessing it.

6 Conclusion

For many applications involving the manipulation of files, the C library APIs are an adequate means for performing the file system operations required by the application. But in other applications, some of NET+OS's native mode file system APIs are required. In those cases, applications can mix NET+OS APIs and C library APIs for accessing the same files. You will want to complete C library operations and close the files before accessing those files using the NET+OS native mode APIs.

7 Appendix

7.1 Glossary of terms

- API – application programming interface. – An interface that requests that an operating system perform some operation(s).
- AWS – Advanced Web Server – The name of the embedded web server that ships with Digi International's NET+OS embedded operating system.
- C library API – an operating system agnostic interface to commands that manipulate something. In this document, these APIs manipulate files.
- ESP – An IDE (integrated development environment) shipped as part of Digi International's NET+OS embedded operating system.
- FTP – file transport protocol. One of the TCPIP protocols generally used for moving files from machine to machine.
- NET+OS – An embedded operating system developed and sold by Digi International
- NET+OS native mode file system API – A set of APIs that are provide lower level file system access to Digi International NET+OS's file system.
- telnet – teletype network – One of the TCPIP protocols used for accessing remote computers