FILE_OPEN Example with error checking:

```
FILE_OPEN_RESULT=FILE_OPEN('textfile.txt',2)
IF(FILE_OPEN_RESULT>0)
{
    FILE_HANDLE=FILE_OPEN_RESULT
    SEND_COMMAND dvTP,"'@TXT',1,'Success - handle:',ITOA(FILE_HANDLE)"
}
ELSE
{
    SWITCH(ABS_VALUE(FILE_OPEN_RESULT))
    {
        CASE 2:
        {
            SEND_STRING 0,'Invalid file path or name'"
        }
        CASE 3:
        {
            SEND_STRING 0,'Invalid value supplied for IOFlag'"
        }
        CASE 5:
        {
            SEND_STRING 0,'Disk I/O Error'"
        }
        CASE 14:
        {
            SEND_STRING 0,'Maximum number of files are already open'"
        }
    }
```

After a file has been read or written it should be closed using the FILE_CLOSE function.

```
FILE_CLOSE(FileHandle)
```

Where:

- *FileHandle* is a long integer containing the handle to the file opened by the FILE_OPEN function.

This function returns a non-zero signed integer value representing the status of the function:

- >0: The close function was successful.
- -1: Invalid file handle.
- -5: Disk I/O Error.
- -7: File is already closed.

Since the NetLinx system has a limited amount of file handles, if files are not closed as they are used there may not be a file handle available to open another file.

Reading and Writing Files

After a file is opened the file can be read using the FILE_READ or FILE_READ_LINE functions.

```
FILE_READ(FileHandle,Char_Var,Bytes_to_Read)
FILE_READ_LINE(FileHandle,Char_Var,Bytes_to_Read)
```

Where:

- *FileHandle* is a long integer containing the handle to the file opened by the FILE_OPEN function.
- *Char_Var* is a character variable to hold the data to be read.
- *Bytes_to_Read* is a long integer to indicate the maximum number of bytes to read. (Must be greater than zero.)

This function returns a non-zero signed integer value representing the status of the function:

- >0: The read function was successful. The value returned is the actual number of bytes read.
- -1: Invalid file handle.
- -5: Disk I/O Error.
- -6: Invalid parameter
- -9: End-of-file reached

The FILE_READ function reads from the current location of the file pointer the number of bytes specified by *Bytes_to_Read* (or fewer bytes if the end of file is reached).

The FILE_READ_LINE function reads from the current location of the file pointer the number of bytes specified by *Bytes_to_Read* up to the next carriage return (<CR>) or to the end-of-file (EOF), whichever comes first. A complete line will not be read if the *Bytes_to_Read* length is exceeded before a carriage return (or EOF) is encountered. The <CR> or <CR><LF> pair will not be stored in Buffer.

The bytes are read from the file identified by *FileHandle* and are stored in *Char_Var* variable. The file pointer will automatically be advanced the correct number of bytes so the next read operation continues where the last operation ended.

Similar to reading from a file, it can be written using the FILE_WRITE or FILE_WRITE_LINE functions.

```
FILE_WRITE(FileHandle,Char_Var,Bytes_to_Write)
FILE_WRITE_LINE(FileHandle,Char_Var,Bytes_to_Write)
```

Where:

- *FileHandle* is a long integer containing the handle to the file opened by the FILE_OPEN function.
- *Char_Var* is a character variable containing the data to write.
- *Bytes_to_Write* is a long integer to indicate the maximum number of bytes write. (Must be greater than zero.)

This function returns a non-zero signed integer value representing the status of the function:

- >0: The write function was successful. The value returned is the actual number of bytes written.
- -1: Invalid file handle.
- -5: Disk I/O Error.
- -6: Invalid parameter
- -11: Disk full

The FILE_WRITE function writes beginning at the current location of the file pointer the number of bytes specified by *Bytes_to_Write* from the *Char_Var* variable.

The FILE_WRITE_LINE function writes beginning at the current location of the file pointer the number of bytes specified by *Bytes_to_Write* from the *Char_Var* variable, then appends a carraige return and line feed pair (<CR><LF>).

If the *Bytes_to_Write* exceed the length of the *Char_Var* only the number of bytes in *Char_Var* will be written.

The bytes are written to the file identified by *FileHandle* and are read from the *Char_Var* variable. The file pointer will automatically be advanced the correct number of bytes so the next write operation continues where the last operation ended.

It may be necessary to set the file pointer to a specfic position before reading or writing. To do this NetLinx provides the FILE_SEEK function.

```
FILE_SEEK(FileHandle,Pos)
```

Where:

- *FileHandle* is a long integer containing the handle to the file opened by the FILE_OPEN function.
- *Pos* is a signed long integer containing the byte position to set the file pointer. (0 = beginning of the file, -1 = end of the file

This function returns a non-zero signed integer value representing the status of the function:

- >0: The seek function was successful. The value returned is the current file pointer value.
- -1: Invalid file handle.
- -5: Disk I/O Error.
- -6: Invalid parameter. (If *Pos* points beyond the end of the file the position is set to the end of the file.)

Copying, Deleting and Renaming Files

Just as with any file system, the NetLinx program provides file manipulation commands to copy, delete, and rename files.

```
FILE_COPY(SrcFilePath, DstFilePath)
FILE_RENAME(SrcFilePath, RenFilename)
```

Where:

- *SrcFilePath*: File path of the file to copy or rename
- *DstFilePath*: File path of the resulting file.
- *RenFilename*: File name of the resulting file. (Cannot include a path name.)

These functions will return a signed integer value representing the status of the function:

- 0: Operation was successful.
- -2: Invalid file name.
- -5: Disk I/O error.
- -8: Filename exists (FILE_RENAME only).
- -11: Disk full (FILE_COPY only).

If either path name fails to specify a directory, the current directory is assumed.

Example of copying a file:

```
// copy FILE.TXT in the current directory to NEWFILE.TXT
Result = FILE_COPY('FILE.TXT', 'NEWFILE.TXT')
```

Example of renaming a file

```
// rename FILE.TXT in the \DATA_FILES\ directory to RENAMED.TXT
Result = FILE_RENAME('\DATA_FILES\FILE.TXT', 'RENAMED.TXT')
```

Deleting files from the NetLinx flash memory is a very easy process, however the FILE_DELETE function does not prompt for confirmation when performing the delete function.

```
FILE_DELETE(FilePath)
```

Where:

- *FilePath*: File path of the file to delete.

The FILE_DELETE function returns a signed integer value representing the status of the function:

- 0: Operation was successful.
- -2: Invalid file name.
- -5: Disk I/O error.

If either path name fails to specify a directory, the current directory is assumed. If only a path name is supplied all files in that directory will be deleted.

Example of deleting a file: