



FILE I/O

RTX allows two options for file input and output (I/O):

1. Win32 API supported functions (fast, low-overhead)
2. I/O functions in the C run-time library.

File I/O **does not include** support for **asynchronous** file operations.



FILE I/O

RTSS applications have no current-directory notion. RTSS file I/O requests must use **fully** qualified path names.

example:

```
c:\log\report.txt
```

Internal Windows names and Win32 names (for access to a device object) are allowed.

example:

```
\Device\beep
```

```
\Device\Harddisk0\Partition1\log\report.txt
```

```
\\. \Beep
```

With boot-time file I/O, only device paths are allowed:

example:

```
\\Device\\Harddisk0\\Partition1\log\report.txt
```



FILE I/O Win32 API Supported Functions

Function	Description	Duration
CreateFile	creates or opens a file or open a directory	?
CreateDirectory	creates a new directory	?
ReadFile	reads data from a file	?
WriteFile	writes data to a file	?
SetFilePointer	moves the file pointer of an open file	?
SetEndOfFile	moves the end-of-file (EOF) position to the current position of the file pointer	?
RemoveDirectory	deletes an existing empty directory	?
DeviceIoControl	sends a control code directly to a specified device driver	?
DeleteFile	deletes an existing file	?



FILE I/O C Library Supported Functions

Function	Description	Duration
<code>fflush</code>	flushes a stream	?
<code>fclose</code>	closes a stream	?
<code>fgets</code>	get a string from a stream	?
<code>fopen</code>	open a stream	?
<code>fprintf</code>	print formatted data to a stream	?
<code>fputc</code>	writes a character to a stream	?
<code>fputs</code>	write a string to a stream	?
<code>fread</code>	reads data from a stream	?
<code>fseek</code>	moves the file pointer to a specified location	< 5 μ s
<code>ftell</code>	gets the current position of a file pointer	?
<code>fwrite</code>	writes data to a stream	?



CreateFile(...)

CreateFile creates or opens files; opens directories.

Prototype

HANDLE

```
CreateFile(  
    LPCTSTR lpFileName,  
    DWORD DesiredAccess,  
    DWORD ShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD CreationDisposition,  
    DWORD FlagsAndAttributes,  
    HANDLE hTemplateFile  
)
```



CreateFile(...)

Parameters

lpFileName

a pointer to a null-terminated string that specifies the name of the object to create or open. If **lpFileName* is a path, the string size limit is `RTX_MAX_PATH` characters. This limit is related to how `CreateFile` parses paths. The `CreateFile` API **cannot** be used to create or open files **over a network**.

DesiredAccess

the type of access to the object. An application can obtain read access, write access, read-write access, or device query access.

This parameter can be any combination of the following values:

`0` Specifies device query access to the object.

An application can query device attributes without accessing the device.

`GENERIC_READ` Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with `GENERIC_WRITE` for read-write access.

`GENERIC_WRITE` Specifies write access to the object. Data can be written to the file and the file pointer can be moved. Combine with `GENERIC_READ` for read-write access.



CreateFile(...)

Parameters

ShareMode

specifies how the object can be shared. If *ShareMode* is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

To share the object, use a combination of one or more of the following values:

FILE_SHARE_DELETE subsequent open operations on the object will succeed only if delete access is requested.

FILE_SHARE_READ subsequent open operations on the object will succeed only if read access is requested.

FILE_SHARE_WRITE subsequent open operations on the object will succeed only if write access is requested.



CreateFile(...)

Parameters

CreationDisposition

specifies which action to take on files that exist, and which action to take when files do not exist. This parameter must be one of the following values:

<i>CREATE_NEW</i>	creates a new file. CreateFile fails if the specified file already exists.
<i>CREATE_ALWAYS</i>	creates a new file. If the file exists, CreateFile overwrites the file and clears the existing attributes.
<i>OPEN_EXISTING</i>	opens the file. CreateFile fails if the file does not exist.
<i>OPEN_ALWAYS</i>	opens the file, if it exists. If the file does not exist, CreateFile creates the file as if <i>CreationDisposition</i> were <i>CREATE_NEW</i> .
<i>TRUNCATE_EXISTING</i>	opens the file. Once opened, the file is truncated so that its size is zero bytes. The calling process must open the file with at least <i>GENERIC_WRITE</i> access. CreateFile fails if the file does not exist.



CreateFile(...)

Parameters

FlagsAndAttributes

the file attributes and flags for the file.

FILE_ATTRIBUTE_ARCHIVE

the file should be archived. Applications use this attribute to mark files for backup or removal.

FILE_ATTRIBUTE_HIDDEN

the file is hidden. It is not to be included in an ordinary directory listing.

FILE_ATTRIBUTE_NORMAL

the file has no other attributes set. This attribute is valid **only** if used **alone**.

FILE_ATTRIBUTE_OFFLINE

the data of the file is not immediately available. Indicates that the file data has been physically moved to offline storage.

FILE_ATTRIBUTE_READONLY

the file is read only. Applications can read the file but cannot write to it or delete it.

FILE_ATTRIBUTE_SYSTEM

the file is part of or is used exclusively by the operating system.

FILE_ATTRIBUTE_TEMPORARY

the file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.



CreateFile(...)

Parameters

<i>FlagsAndAttributes</i>	the file attributes and flags for the file.
<i>FILE_FLAG_WRITE_THROUGH</i>	instructs the system to write through any intermediate cache and go directly to disk. Windows can still cache write operations, but cannot lazily flush them.
<i>FILE_FLAG_NO_BUFFERING</i>	instructs the system to open the file with no intermediate buffering or caching. An application must meet certain requirements when working with files opened with <i>FILE_FLAG_NO_BUFFERING</i> : <ol style="list-style-type: none">1. File access must begin at byte offsets within the file that are integer multiples of the volume's sector size.2. File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.3. Buffer addresses for read and write operations must be aligned on addresses in memory that are integer multiples of the volume's sector size.



CreateFile(...)

Parameters

<i>FILE_FLAG_RANDOM_ACCESS</i>	indicates that the file is accessed randomly. The system can use this as a hint to optimize file caching.
<i>FILE_FLAG_SEQUENTIAL_SCAN</i>	indicates that the file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.
<i>FILE_FLAG_DELETE_ON_CLOSE</i>	indicates that the operating system is to delete the file immediately after all of its handles have been closed, not just the handle for which you specified <i>FILE_FLAG_DELETE_ON_CLOSE</i> . Subsequent open requests for the file will fail, unless <i>FILE_SHARE_DELETE</i> is used.
<i>hTemplateFile</i>	(ignored)



CreateFile(...)

Return Values

If CreateFile succeeds, the return value is an open handle to the specified file.

If the specified file exists before the function call and *CreationDisposition* is *CREATE_ALWAYS* or *OPEN_ALWAYS*, a call to `GetLastError` returns *ERROR_ALREADY_EXISTS* (even though the function has succeeded).

If the file does not exist before the call, **GetLastError** returns *ERROR_SUCCESS*.

If CreateFile fails, the return value is *INVALID_HANDLE_VALUE*. To get extended error information, call `GetLastError`.

Use **CloseHandle** to close an object handle returned by CreateFile.



ReadFile(...)

ReadFile reads data from a file.

Prototype:

```
BOOL ReadFile(  
    HANDLE hFile,  
    LPVOID lpBuffer,  
    DWORD nNumberOfBytesToRead,  
    LPDWORD lpNumberOfBytesRead,  
    LPOVERLAPPED lpOverlapped  
)
```



ReadFile(...)

Parameters

<i>hFile</i>	the file to be read. The file handle must have been created with GENERIC_READ access to the file.
<i>lpBuffer</i>	a pointer to the buffer that receives the data read from the file.
<i>nNumberOfBytesToRead</i>	the number of bytes to be read from the file.
<i>lpNumberOfBytesRead</i>	a pointer to the number of bytes read. ReadFile sets this value to zero before doing any work or error checking.
<i>lpOverlapped</i>	not supported by RTX; this parameter must be set to NULL.



ReadFile(...)

Return Values

If the function succeeds, the return value is TRUE.

If the return value is TRUE and the number of bytes read is zero

(*lpNumberOfBytesRead*), the file pointer was beyond the current end of the file at the time of the read operation.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

ReadFile returns when the number of bytes requested has been read, or an error occurs. If part of the file is locked by another process and the read operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the input buffer that a read operation is using until the read operation completes. A premature access to the input buffer may lead to corruption of the data read into that buffer.



WriteFile(...)

WriteFile writes data to a file.

Prototype

```
BOOL WriteFile(  
    HANDLE hFile,  
    LPCVOID lpBuffer,  
    DWORD nNumberOfBytesToWrite,  
    LPDWORD lpNumberOfBytesWritten,  
    LPOVERLAPPED lpOverlapped  
)
```



WriteFile(...)

Parameters

<i>hFile</i>	the file to be written to. The file handle must have been created with <i>GENERIC_WRITE</i> access to the file.
<i>lpBuffer</i>	a pointer to the buffer containing the data to be written to the file.
<i>nNumberOfBytesToWrite</i>	the number of bytes to write to the file.
<i>lpNumberOfBytesWritten</i>	a pointer to the number of bytes written by this function. WriteFile sets this value to zero before doing any work or error checking.
<i>lpOverlapped</i>	not supported by RTX; this parameter must be set to NULL.



WriteFile(...)

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.

Comments

RTX does not support asynchronous operations.

If part of the file is locked by another process and the write operation overlaps the locked portion, this function fails.

Applications must not read from nor write to the output buffer that a write operation is using until the write operation completes. Premature access of the output buffer may lead to corruption of the data written from that buffer.



CreateFile(...), WriteFile(...)

```
/* *****  
 * LogFileMessage(...)  
 *  
 * Description:  
 * Log specific message to a LOG file  
 * *****/  
void LogFileMessage(char message[]) {  
    RTX_TIME rtx_time;  
    HANDLE logfile;  
    char c_time_stamp[40];  
    char c_message[300];  
    DWORD length, written;  
  
    /* check if it is necessary to create new log file */  
    GetCurrentDateTime(&rtx_time);  
    if ((rtx_time.day + rtx_time.month + rtx_time.year) != day_of_last_log_file)  
        CreateLogFile();  
  
    /* if log file is NOT created, then return right now */  
    if (!logging_enabled) return;  
  
    sprintf(c_time_stamp, "%02d:%02d:%02d:%03d", rtx_time.hour, rtx_time.min, rtx_time.sec, rtx_time.msec);  
    /* open existing log file */  
    logfile = CreateFile(current_log_file, GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, 0);  
    if (logfile == INVALID_HANDLE_VALUE) {  
        #ifdef SCREEN_LOG_RTX  
            sprintf(c_message, "ERROR\tLogFileMessage::CreateFile()\tCan not open LOG file '%s'!  
                Error = %d.", current_log_file, GetLastError());  
            LogScreenMessage(c_message);  
        #endif  
        return;  
    }  
    sprintf(c_message, "%s\t%s\r\n", c_time_stamp, message);  
    length = strlen(c_message);  
    SetFilePointer(logfile, 0, NULL, FILE_END);  
    WriteFile(logfile, c_message, length, &written, 0);  
    CloseHandle(logfile);  
}
```