
N I N T E N D O
NITRO-System
Multiple Channel Stream Library

Communications Between Nintendo DS
and Multiple Windows Applications

Version 1.1.0

The contents of this document are strictly
confidential and the document should be
handled accordingly.

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo of America Inc. and/or Nintendo Company Ltd. and are protected by Federal copyright law. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

1	Introduction	7
2	Communications Between Nintendo DS Programs and Windows Applications	8
2.1	Procedures on the Nintendo DS	8
2.1.1	Initialize the mcs Library	8
2.1.2	Configure the Way to Receive Data.....	9
2.1.2.1	Register a Callback Function.....	9
2.1.2.2	Register a Buffer.....	10
2.1.3	Open the Device.....	10
2.1.4	Configure Interrupts.....	11
2.1.5	Polling.....	13
2.1.6	Reading Data.....	13
2.1.6.1	When a Callback Function has been Registered.....	13
2.1.6.2	When a Receiving Buffer has been Registered.....	13
2.1.7	Writing Data.....	14
2.1.8	When the Opened Device is IS-NITRO-UIC	15
2.2	Procedures on the Windows Side.....	16
2.2.1	Read DLL and Get Function Address	16
2.2.2	Open the Stream	17
2.2.3	Read from the Stream.....	18
2.2.4	Write to Stream	18
2.2.5	Close the Stream	19
3	File Search and File Read/Write	20
3.1	Initialize the mcs File Input/Output Library.....	20
3.2	File Reading and Writing.....	21
3.2.1	Open the File	21
3.2.2	Read from File	22
3.2.3	Write to File.....	22
3.2.4	Close the File.....	23
3.2.5	Moving the File Pointer	23
3.3	File Searching.....	24
3.3.1	Start File Search	24
3.3.2	Continue File Search	25
3.3.3	End File Search	25
4	Outputting Character Strings to the Console.....	26
4.1	Output with OS_Printf Function.....	26

4.2	Output with mcs String Output Functions	26
4.2.1	Initialize the Character String Output Library	26
4.2.2	Output Character String	26
5	About the mcs Server	27
5.1	General Operations Flow	27
5.1.1	Connect	27
5.1.2	Load ROM File (if the Device is IS-NITRO-EMULATOR)	27
5.1.3	Disconnect	27
5.1.4	Reset (if the Device is IS-NITRO-EMULATOR)	27
5.2	Special Situations	28
5.2.1	Connecting with <code>ensata</code>	28
5.2.2	Shared Mode and Dedicated Mode	28
5.2.3	Command Line Options	28
5.2.4	Powering ON the IS-NITRO-EMULATOR DS Game Card Slot and GBA Game Pak Slot	29
5.2.5	About the Interval for Obtaining Data from the Nintendo DS	29

Code Samples

Code 2-1	Initilizing the mcs Library	8
Code 2-2	Registering a Callback Function	9
Code 2-3	Registering a Receiving buffer	10
Code 2-4	Opening the Device	10
Code 2-5	Configuring Interrupts	12
Code 2-6	Calling the Polling Function	13
Code 2-7	Reading the Received Data	14
Code 2-8	Writing Data	15
Code 2-9	Waiting for mcs Server Connection	15
Code 2-10	Reading DLL and Getting Function Address	16
Code 2-11	Opening a Stream	17
Code 2-12	Reading from the Stream	18
Code 2-13	Writing to the Stream	19
Code 2-14	Closing the Stream	19
Code 3-1	Opening a File	21
Code 3-2	Reading from a File	22
Code 3-3	Writing to a File	22
Code 3-4	Closing a File	23
Code 3-5	Moving the File Pointer	23
Code 3-6	Starting File Search	24
Code 3-7	Continuing File Search	25
Code 3-8	Ending File Search	25
Code 4-1	Initilzaing the Character String Output Library	26

Code 4-2 Outputting a Character String	26
Code 5-1 Command Line Options.....	28

Figures

Figure 2-1 Communications Between Nintendo DS Program and Windows Application.....	8
Figure 3-1 Searching Files and Reading/Writing to Files	20

Revision History

Version	Revision Date	Details of Revision
1.1.0	2007/03/14	Added a feature for turning on the power to the DS Game Card slot.
1.0.2	2005/03/18	Added a function that changes the position of the current file pointer. Added a feature to change the load time interval from a Nintendo DS on an mcs server.
1.0.0	2005/01/18	Initial version.

1 Introduction

The mcs (“Multiple Channel Stream”) library is the collective name of the library and a group of tool programs that enable Nintendo DS programs to communicate with multiple Windows applications. The library provides the following features.

- Ability to communicate between Nintendo DS programs and Windows applications
- Ability to access files on the PC from the Nintendo DS program
- Display of text strings output from the Nintendo DS program

Among the pieces of hardware that run Nintendo DS programs, the following devices support the mcs library.

- IS-NITRO-EMULATOR
- Nintendo DS System + IS-NITRO-UIC
- ensata software emulator

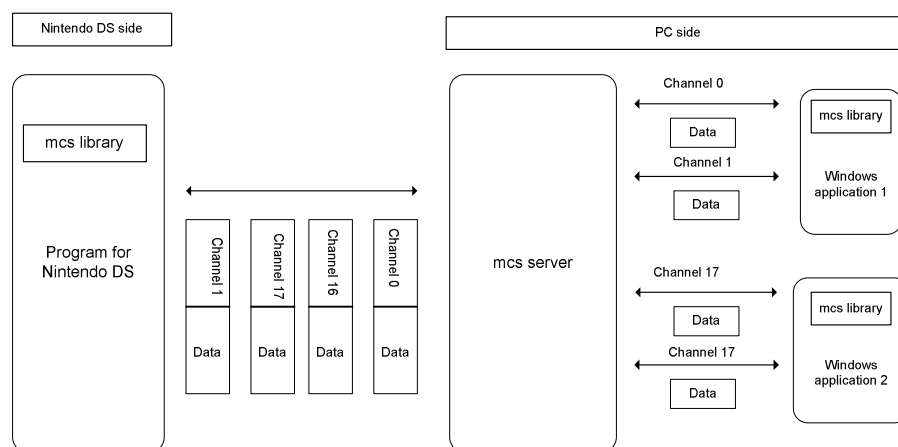
If you are using IS-NITRO-EMULATOR or IS-NITRO-UIC, the `ISNITRO.dll` must be installed on the system.

`ISNITRO.dll` gets installed on the system when you install the IS-NITRO-DEBUGGER software.

2 Communications Between Nintendo DS Programs and Windows Applications

One of the basic purposes of the mcs library is to enable communications between Nintendo DS programs and multiple Windows applications running on a PC. Figure 2-1 provides a schematic diagram of this process.

Figure 2-1 Communications Between Nintendo DS Program and Windows Application



Communications require procedures to be carried out by both the Nintendo DS program and the Windows application. Because the procedures differ, they will be explained separately.

2.1 Procedures on the Nintendo DS

2.1.1 Initialize the mcs Library

To use the mcs library, you must first call the `NNS_McsInit` function and initialize the library.

Code 2-1 Initializing the mcs Library

```
void
NitroMain
{
    OS_Init();
    ...
    NNS_McsInit();
    ...
}
```


2.1.2 Configure the Way to Receive Data

There are two ways to receive data: by calling a callback function when the data is received, or by having the program read the data at a time of its own discretion. For both methods, each channel must be set ahead of time.

2.1.2.1 Register a Callback Function

To call a callback when data have been received, register a callback function. Secure the variable of the `NNSMcsRecvCBInfo` structure ahead of time, and call the function `NNS_McsRegisterRecvCallback` by passing a pointer to this variable as an argument. Other arguments include the channel value used to identify the Windows application, the registered callback function, and the user-defined value passed to this callback function. When `NNS_McsRegisterRecvCallback` is called, the registered information gets set in the specified type `NNSMcsRecvCBInfo` variable.

Code 2-2 Registering a Callback Function

```
#define MCS_CHANNEL_ID 10    // Channel value

// The callback function that gets called when data is received from PC
static void
DataRecvCallback(
    const void* pRecv,      // Pointer to the data buffer
    u32         recvSize,   // Size of received data
    u32         userData,   // User defined value
    u32         offset,     // Offset value to all received data
    u32         totalSize  // Total size of received data
)
{
}

...
void
NitroMain()
{
    ...
    static NNSMcsRecvCBInfo sRecvCBInfo;
    ...

    // Register the callback function
    NNS_McsRegisterRecvCallback(
        &sRecvCBInfo,      // NNSMcsRecvCBInfo type variable
        MCS_CHANNEL_ID,    // Channel value
        DataRecvCallback,  // Callback function
        0);               // User defined value
    ...
}
```

2.1.2.2 Register a Buffer

To have the program read the data at a time of its own discretion, you need to register a buffer that can hold the received data. Memory for the received data must be secured ahead of time. Using this and the channel value as arguments, call the `NNS_McsRegisterStreamRecvBuffer` function to register the buffer.

The memory for managing the Receiving buffer is secured from the memory-use buffer specified here, so make sure you secure at least 48 bytes. If received data accumulates in this buffer without being read and the buffer overflows, that received data will be discarded. Thus, it is essential to allocate a buffer of the appropriate size for your objectives for every channel.

Code 2-3 Registering a Receiving buffer

```
#define MCS_CHANNEL_ID 10 // Channel value

static u32 sRecvBuf[64 * 1024 / sizeof(u32)];

...

NNS_McsRegisterStreamRecvBuffer(
    MCS_CHANNEL_ID,          // Channel value
    sRecvBuf,                // Pointer to Receiving buffer
    sizeof(sRecvBuf));       // Size of Receiving buffer
```

2.1.3 Open the Device

Open the device used for communications. First call the `NNS_McsGetMaxCaps` function to get the total number of devices that are capable of communicating. If the total number is 0, this indicates that no devices were found. If there are 1 or more devices, use the `NNS_McsOpen` function to open a device. The argument for this function is the pointer to the `NNSMcsDeviceCaps` type variable, which was secured ahead of time. Information related to the opened device is placed in this variable.

Code 2-4 Opening the Device

```
NNSMcsDeviceCaps deviceCaps;

if (NNS_McsGetMaxCaps() == 0)
{
    OS_Panic("Could not find device.");
}

if (! NNS_McsOpen(&deviceCaps))
{
    OS_Panic("Failed to open the device.");
}
```

2.1.4 Configure Interrupts

Depending on the type of the device that has been opened, certain functions need to be called periodically. The function that needs to be called for a given device is set in the `maskResource` member variable of the `NNSMcsDeviceCaps` type variable that was specified when the `NNS_McsOpen` function was called. Using this variable and a mask, configure an interrupt handler so the necessary function is called.

For example, if the bitwise AND result of the `maskResource` variable and `NITROMASK_RESOURCE_VBLANK` is not zero, the device needs to call the `NNS_McsVBlankInterrupt` function in every frame. Configure a V-Blank interrupt handler so that `NNS_McsVBlankInterrupt` gets called from inside of the interrupt handler.

Similarly, if the bitwise AND result of the `maskResource` variable and `NITROMASK_RESOURCE_CARTRIDGE` is not zero, the device needs to call the `NNS_McsCartridgeInterrupt` function every time a cartridge interrupt occurs. Configure a cartridge interrupt handler so that `NNS_McsCartridgeInterrupt` is called from inside of the interrupt handler.

Code 2-5 Configuring Interrupts

```
...

if (deviceCaps.maskResource & NITROMASK_RESOURCE_VBLANK)
{
    // Enable VBlank interrupts and configure so NNS_McsVBlankInterrupt()
    // gets called from inside VBlank interrupt

    BOOL preIRQ = OS_DisableIrq();
    OS_SetIrqFunction(OS_IE_V_BLANK, VBlankIntr);
    (void)OS_EnableIrqMask(OS_IE_V_BLANK);
    (void)OS_RestoreIrq(preIRQ);

    (void)GX_VBlankIntr(TRUE);
}

if (deviceCaps.maskResource & NITROMASK_RESOURCE_CARTRIDGE)
{
    // Enable cartridge interrupts and configure so
    // NNS_McsCartridgeInterrupt() gets called from inside
    // cartridge interrupt

    BOOL preIRQ = OS_DisableIrq();
    OS_SetIrqFunction(OS_IE_CARTRIDGE, CartIntrFunc);
    (void)OS_EnableIrqMask(OS_IE_CARTRIDGE);
    (void)OS_RestoreIrq(preIRQ);
}

...

static void
VBlankIntr(void)
{
    OS_SetIrqCheckFlag(OS_IE_V_BLANK);

    NNS_McsVBlankInterrupt();
}

static void
CartIntrFunc(void)
{
    OS_SetIrqCheckFlag(OS_IE_CARTRIDGE);

    NNS_McsCartridgeInterrupt();
}
```

Until it becomes necessary to open the device, nothing happens when the `NNS_McsVBlankInterrupt` or the `NNS_McsCartridgeInterrupt` function gets called. Thus, interrupts can be configured at any time before opening the device, regardless of the device type.

2.1.5 Polling

In addition to configuring interrupts explained above, call the `NNS_McsPollingIdle` function periodically. For example, call `NNS_McsPollingIdle` every time in the main loop.

Code 2-6 Calling the Polling Function

```
// Main loop
while (TRUE)
{
    SVC_WaitVBlankIntr();

    ...

    // Polling process
    NNS_McsPollingIdle();
}
```

2.1.6 Reading Data

2.1.6.1 When a Callback Function has been Registered

If you have registered a callback function, that function will get called when data are received.

2.1.6.2 When a Receiving Buffer has been Registered

If you have registered a Receiving buffer, the received data will get stored in that buffer. As shown in the code sample below, to read the data stored in the buffer, call the `NNS_McsReadStream` function. Use the `NNS_McsGetStreamReadableSize` function to get the size of data that can be read with a single call to `NNS_McsReadStream`. Use the `NNS_McsGetTotalStreamReadableSize` function to get the total size of data available in the buffer for reading.

Code 2-7 Reading the Received Data

```
static u8 sBuf[1024];

u32 nLength = NNS_McsGetStreamReadableSize(MCS_CHANNEL_ID);

if (nLength > 0)
{
    u32 readSize;
    BOOL result = NNS_McsReadStream(
        MCS_CHANNEL_ID,    // Channel value
        sBuf,              // Pointer to the buffer for reading
        sizeof(sBuf),      // Size of the buffer for reading
        &readSize);         // Pointer to the variable that stores the
                          // size actually read

    if (result)
    {
        // Read OK
    }
    else
    {
        // Read failure
    }
}
```

2.1.7 Writing Data

Use the `NNS_McsWriteStream` function to write data. Use the `NNS_McsGetStreamWritableLength` function to get the amount of data that can be written at a given time. If the amount of data to write with `NNS_McsWriteStream` is less than the writable amount obtained by `NNS_McsGetStreamWritableLength`, the `NNS_McsWriteStream` function will quit immediately. If the amount of data to write is larger than the writable amount, calls to `NNS_McsWriteStream` will be blocked until writing of the specified amount of data has completed.

Code 2-8 Writing Data

```
u8 sendBuf[32];
u32 nLength;

...

// Get the writable size of data
if (NNS_McsGetStreamWritableLength(&nLength))
{
    // Write if can write without blocking
    if (sizeof(sendBuf) <= nLength)
    {
        // Write
        if (! NNS_McsWriteStream(
            MCS_CHANNEL_ID,
            sendBuf,
            sizeof(sendBuf)))
        {
            // Write succeeds
        }
        else
        {
            // Write fails
        }
    }
}
```

2.1.8 When the Opened Device is IS-NITRO-UIC

When the opened device is IS-NITRO-UIC and the `NNS_McsWriteStream` function is called while the mcs server is not connected to IS-NITRO-UIC, control will not return to this function until the mcs server connects to the device. If this is going to be a problem, call the `NNS_McsIsServerConnect` function to check if the mcs server is connected. If the server is connected to IS-NITRO-UIC, `NNS_McsIsServerConnect` will return TRUE.

The communications state of the mcs server is checked by using the mcs communications functionality. Therefore, there may be a slight time lag before the actual connection state of the mcs server gets reflected.

Code 2-9 Waiting for mcs Server Connection

```
NNSMcsDeviceCaps deviceCaps;

...

if (NNS_McsOpen(&deviceCaps))
{
    // Wait for connection from mcs server
    while (! NNS_McsIsServerConnect())
    {
        SVC_WaitVBlankIntr();
    }
}
```

2.2 Procedures on the Windows Side

2.2.1 Read DLL and Get Function Address

The library for Windows is provided in the form of the dynamic link library `nnsmcsl.dll`. This file can be found in the `tools\win\mcserver` directory, under the directory where NITRO-System was installed.

The `NNS_McsOpenStream` and `NNS_McsOpenStreamEx` functions exported with this library are used for opening the stream. Get the addresses for these functions as needed.

Code 2-10 Reading DLL and Getting Function Address

```
#include <nnsys/mcs.h>

_TCHAR modulePath[MAX_PATH];
DWORD writtenChars;
HMODULE hModule;
NNSMcsPFOpenStream pfOpenStream;

// Obtain the absolute path for nnsmcsl.dll
writtenChars = ExpandEnvironmentStrings(
    _T("%NITROSYSTEM_ROOT%\tools\win\mcserver\%nnsmcsl.dll"),
    modulePath,
    MAX_PATH);
if (writtenChars > MAX_PATH)
{
    // Path is too long
    return 1;
}

hModule = LoadLibrary(modulePath);
if (NULL == hModule)
{
    // Reading of module fails
    return 1;
}

// Get address of function
pfOpenStream = (NNSMcsPFOpenStream)GetProcAddress(
    hModule,
    NNS_MCS_API_OPENSTREAM);
```


2.2.2 Open the Stream

On the Windows side, a stream gets opened for every channel. Open the stream using the `NNS_McsOpenStream` or `NNS_McsOpenStreamEx` functions. `NNS_McsOpenStreamEx` has the same features as `NNS_McsOpenStream`, plus the ability to get information about the connected device.

A stream is actually a Win32 System named pipe. The `NNS_McsOpenStream(Ex)` function opens the named pipe as a message type and registers the specified channel to the mcs server.

Code 2-11 Opening a Stream

```
HANDLE hStream;

// Open the stream
hStream = pfOpenStream(
    MCS_CHANNEL_ID,    // Channel value
    0);               // Flag
if (hStream == INVALID_HANDLE_VALUE)
{
    // Open fails
    return 1;
}
```

2.2.3 Read from the Stream

To read the stream, use the Win32 `ReadFile` or `ReadFileEx` functions. To get the readable size, use `PeekNamedPipe`.

Code 2-12 Reading from the Stream

```
static BYTE buf[1024];
DWORD totalBytesAvail;
BOOL fSuccess;

fSuccess = PeekNamedPipe(
    hStream,          // Stream's handle
    NULL,
    0,
    NULL,
    &totalBytesAvail, // Number of bytes available
    NULL);
if (! fSuccess)
{
    // Peek fails
    return 1;
}

// When there is readable data:
if (totalBytesAvail > 0)
{
    DWORD readBytes;

    fSuccess = ReadFile(
        hStream,          // Stream's handle
        buf,              // Pointer to Reading buffer
        sizeof(buf),      // Number of bytes to read
        &readBytes,        // Number of bytes actually read
        NULL);
    if (! fSuccess)
    {
        // Read fails
        return 1;
    }
}
```

2.2.4 Write to Stream

To write to the stream, use the Win32 `WriteFile` or `WriteFileEx` functions.

Code 2-13 Writing to the Stream

```
static BYTE buf[1024];
BOOL fSuccess;
DWORD writtenBytes;

fSuccess = WriteFile(
    hStream,          // Stream's handle
    buf,              // Pointer to Writing buffer
    sizeof(buf),      // Number of bytes to write
    &writtenBytes,     // Number of bytes actually written
    NULL);
if (! fSuccess)
{
    // Write fails
    return 1;
}
```

2.2.5 Close the Stream

To close the stream, use the Win32 `CloseHandle` function.

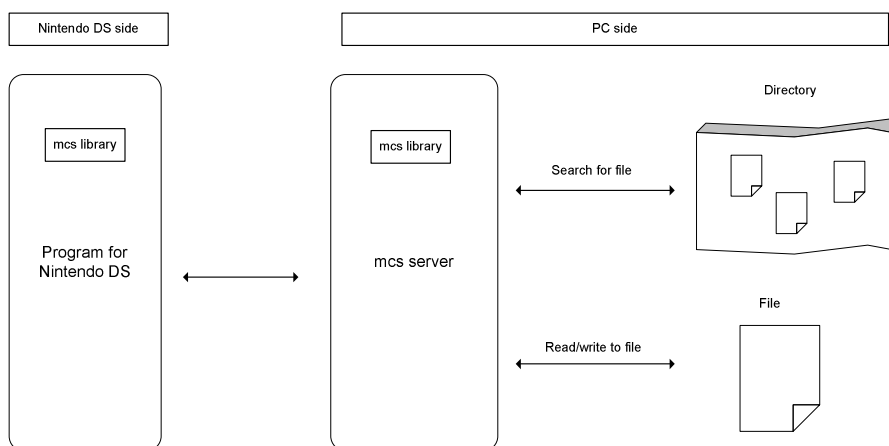
Code 2-14 Closing the Stream

```
// Close the stream
CloseHandle(hStream);
```

3 File Search and File Read/Write

The mcs library has features for reading and writing to PC files from the Nintendo DS program, and for searching for files on the PC from the Nintendo DS program. The following diagram illustrates the concept.

Figure 3-1 Searching Files and Reading/Writing to Files



There is no Windows library for these features. They become available when the mcs server is connected to a Nintendo DS system.

The following sections explain the procedures for file searching and for file reading/writing.

3.1 Initialize the mcs File Input/Output Library

To use the features for file searching and file reading/writing, call the `NNS_McsInit` function to initialize the mcs library, then call and initialize the `NNS_McsInitFileIO` function.

```
NNS_McsInit();           // Initialize the mcs library
...
NNS_McsInitFileIO();     // Initialize the file I/O features
```

3.2 File Reading and Writing

3.2.1 Open the File

To open a file on the PC, call the `NNS_McsOpenFile` function. For the arguments of this function, specify the pointer to the previously secured `NNSMcsFile` type variable, the name of the file to open, and the read/write flag. If the file is opened successfully, the function returns 0 and the information pertaining to the opened file gets placed in the `NNSMcsFile` type variable. If the process fails, the function returns a non-zero value.

Code 3-1 Opening a File

```
NNSMcsFile infoRead;
NNSMcsFile infoWrite;
u32 errCode;

// Open file for reading
errCode = NNS_McsOpenFile(
    &infoRead,
    "c:\\testApp\\test.txt",    // File name
    NNS_MCS_FILEIO_FLAG_READ); // Reading mode
if (errCode != 0)
{
    // File fails to open
    return 1;
}

// Open file for writing
errCode = NNS_McsOpenFile(
    &infoWrite,
    "c:\\testApp\\outTest.txt",
    NNS_MCS_FILEIO_FLAG_WRITE);
if (errCode != 0)
{
    // File fails to open
    return 1;
}
```

3.2.2 Read from File

To read the file, use the `NNS_McsReadFile` function. To get the size of the file, use the `NNS_McsGetFileSize` function.

Code 3-2 Reading from a File

```
static u8 buf[1024];
u32 errCode;
u32 fileSize;
u32 readSize;

// Get the size of the file
fileSize = NNS_McsGetFileSize(&infoRead);

if (fileSize <= sizeof(buf))
{
    // Read entire file at once
    errCode = NNS_McsReadFile(
        &infoRead,
        buf,           // Pointer to the Reading buffer
        fileSize,      // Number of bytes to read
        &readSize);    // Number of bytes actually read
    if (errCode != 0)
    {
        // Reading from file fails
        return 1;
    }
}
```

3.2.3 Write to File

To write to the file, use the `NNS_McsWriteFile` function.

Code 3-3 Writing to a File

```
static u8 buf[1024];
u32 errCode;
u32 fileSize;
u32 readSize;

// Write everything in buf
errCode = NNS_McsWriteFile(
    &infoWrite,
    buf,           // Pointer to the Writing buffer
    sizeof(buf));  // Number of bytes to write
if (errCode != 0)
{
    // Writing to file fails
    return 1;
}
```

3.2.4 Close the File

To close the file, use the `NNS_McsCloseFile` function.

Code 3-4 Closing a File

```
u32 errCode;

errCode = NNS_McsCloseFile(&infoRead);
if (errCode)
{
    // Closing of file fails
    return 1;
}
```

3.2.5 Moving the File Pointer

Use the `NNS_McsSeekFile` function to move the current file pointer. Passing a `u32` type variable pointer allows the position of the moved file pointer to be obtained.

Code 3-5 Moving the File Pointer

```
u32 errorcode;
u32 filePointer // variable for storing the file pointer position

// Move to the 100th byte from the start of the file
errCode = NNS_McsSeekFile(&infoRead, 100, NNS_MCS_FILEIO_SEEK_BEGIN, NULL);
...
// Move 200 bytes from the current file pointer position
// Get the position of the moved file pointer
errCode = NNS_McsSeekFile(&infoRead, 200, NNS_MCS_FILEIO_SEEK_CURRENT,
&filePointer);
...
// Get the current file pointer position
// Do not move the file pointer
errCode = NNS_McsSeekFile(&infoRead, 0, NNS_MCS_FILEIO_SEEK_CURRENT,
&filePointer);
```

3.3 File Searching

3.3.1 Start File Search

To conduct a file search, first call the `NNS_McsFindFirstFile` function. For its arguments, use the pointer to the previously secured `NNSMcsFile` type variable, the pointer to the previously secured `NNSMcsFileFindData` type variable, and the pattern character string for which you want to search files.

If the function finds a matching file, it returns 0 and sets the information related to the search in the `NNSMcsFile` type variable, and the information related to the found file in the `NNSMcsFileFindData` type variable. If the file that matches the pattern is not found, `NNS_MCS_FILEIO_ERROR_NOMOREFILES` is returned.

Code 3-6 Starting File Search

```
NNSMcsFile info;
NNSMcsFileFindData findData;
u32 errCode;

errCode = NNS_McsFindFirstFile(
    &info,
    &findData,
    "c:¥¥testApp¥¥*.txt");

// File with matching pattern was not found
if (errCode == NNS_MCS_FILEIO_ERROR_NOMOREFILES)
{
    OS_Printf("no match *.txt .¥n");
    return 0;
}

if (errCode != 0)
{
    // File search fails
    return 1;
}
```


3.3.2 Continue File Search

To search for the next matching pattern, call the `NNS_McsFindNextFile` function. For its arguments, use the pointer to the `NNSMcsFile` type variable that was specified when `NNS_McsFindFirstFile` was called, and the pointer to the previously secured `NNSMcsFileFindData` type variable. If the function finds a matching file, it returns 0 and sets the information related to the search in the `NNSMcsFile` type variable and the information related to the found file in the `NNSMcsFileFindData` type variable, just as the `NNS_McsFindFirstFile` function does. If the function cannot find a file that matches the pattern, it returns `NNS_MCS_FILEIO_ERROR_NOMOREFILES`.

Code 3-7 Continuing File Search

```
do
{
    // Display the file name
    OS_Printf("find filename %s¥n", findData.name);

    // Search for the next file with a matching pattern
    errCode = NNS_McsFindNextFile(&info, &findData);
}while (errCode == 0);

if (errCode != NNS_MCS_FILEIO_ERROR_NOMOREFILES)
{
    // File search fails
}
```

3.3.3 End File Search

To end the file search, call the `NNS_McsCloseFind` function.

Code 3-8 Ending File Search

```
errCode = NNS_McsCloseFind(&info);
if (errCode != 0)
{
    // Failed to end file search
    return 1;
}
```

4 Outputting Character Strings to the Console

The mcs library provides features for outputting strings to the mcs server's console. There are two ways to output these character strings: by using the NITRO-SDK function `OS_Printf`, or by using one of the mcs library's string output functions. Both of these methods have advantages and disadvantages, so use them according to the situation.

4.1 Output with `OS_Printf` Function

If you output using the `OS_Printf` function, the string will only display on the mcs console if the mcs server is connected to IS-NITRO-EMULATOR. The string will not display on the console if the connected device is IS-NITRO-UIC or `ensata`.

The advantage of this method is that the same procedure can be used to output strings to other applications that support `OS_Printf`, such as IS-NITRO-DEBUGGER.

4.2 Output with mcs String Output Functions

When the mcs library is used to output character strings, the strings can be output no matter what connected device is used, as long as mcs communications have been established. However, the output can only go to the console of the mcs server.

The following is an explanation of how to use the mcs library functions to output character strings.

4.2.1 Initialize the Character String Output Library

To use the features for outputting character strings, you must first call the `NNS_McsInit` function to initialize the mcs library. Next, initialize the features by calling the `NNS_McsInitPrint` function.

Code 4-1 Initilzaing the Character String Output Library

```
NNS_McsInit();           // Initialize the mcs library
...
NNS_McsInitPrint ();    // Initialize the string output feature
```

4.2.2 Output Character String

To output a plain character string, use the `NNS_McsPutString` function. To output a formatted string, use the `NNS_McsPrintf` function.

Code 4-2 Outputting a Character String

```
u32 val = 16;

NNS_McsPutString("print string\n");
NNS_McsPrintf("val = %d\n", val);
```

5 About the mcs Server

The mcs server is a program that provides a communications bridge enabling simultaneous communications between Nintendo DS applications and multiple Windows applications on a PC. The mcs server also provides features that allow Nintendo DS applications to access files on the PC and to output character strings to the console of the mcs server.

5.1 General Operations Flow

5.1.1 Connect

Before communications can proceed between Windows applications and a Nintendo DS application, and before a Nintendo DS application can access PC files or output character strings to the mcs server console, the mcs server must connect to a hardware device that is running a Nintendo DS application.

If an IS-NITRO-EMULATOR device and an IS-NITRO-UIC device are both connected on the PC, the mcs server will connect to the IS-NITRO-UIC device. If two or more devices of the same kind exist, the mcs server will connect to the first device that it discovers.

5.1.2 Load ROM File (if the Device is IS-NITRO-EMULATOR)

If the mcs server is connected to an IS-NITRO-EMULATOR device, load the ROM file after the connection is established. Select **Open** from the **File** menu. In the File dialog, select the file you want to read. After the file has been loaded, the Nintendo DS application will start.

If the mcs server is connected to an IS-NITRO-UIC device, you cannot load a ROM file.

5.1.3 Disconnect

To end communications, select **Disconnect** from the **Device** menu.

5.1.4 Reset (if the Device is IS-NITRO-EMULATOR)

If the connected device is an IS-NITRO-EMULATOR, you can reset the system by selecting **Reset** from the **Device** menu.

If the mcs server is connected to an IS-NITRO-UIC device, you cannot perform a reset.

5.2 Special Situations

5.2.1 Connecting with ensata

To connect the mcs server to **ensata**, select **ensata** from the **Devices** menu and place a check mark next to "ensata". Next, select **Connect** from the **Devices** menu. This starts **ensata**. Loading a ROM file after this enables communications with a Nintendo DS application running on **ensata**.

5.2.2 Shared Mode and Dedicated Mode

The mcs server has two modes: shared and dedicated. When **Share Mode** in the **Resource** menu is checked, the server is in the shared mode. Otherwise it is in the dedicated mode.

When the mcs server is in the dedicated mode, the Nintendo DS application can only communicate with one Windows application at a time. In this state, when the channel value is seen in hexadecimal, the upper 12 bits are taken as the group value. Connections are allowed only to channels with the same group value as that of the first connected channel. Connections to channels in other groups are denied. In share mode, there are no such restrictions.

5.2.3 Command Line Options

You can use command line options to set parameters when starting the mcs server. The entries are not case sensitive.

Code 5-1 Command Line Options

```
mcsserv [/U] [/E] [/D] [/A] [ROM filename]

/U          Connect to device after startup. Invalid
            if ROM file has been specified.
/E          Connect to ensata.
/D          Turn on power to IS-NITRO-EMULATOR DS Game Card slot.
            Valid when mcs server connected to IS-NITRO-EMULATOR.
/A          Turn on power to IS-NITRO-EMULATOR GBA Game Pak slot.
            Valid when connected to IS-NITRO-EMULATOR.

ROM filename After startup, connect and load specified file. Valid
            when mcs server connected to IS-NITRO-EMULATOR.
```

5.2.4 Powering ON the IS-NITRO-EMULATOR DS Game Card Slot and GBA Game Pak Slot

When the command line option "/D" is specified, power will be turned on to the DS Game Card slot when the mcs server connects to the IS-NITRO-EMULATOR device. This enables simultaneous use of the hardware that supports the DS Game Card slot.

When the command line option "/A" is specified, power will be turned on to the GBA Game Pak slot when the mcs server connects to the IS-NITRO-EMULATOR device. This enables simultaneous use of the hardware that supports the GBA Game Pak slot.

Do not insert or remove a game device while the power is ON, as this could damage the device.

5.2.5 About the Interval for Obtaining Data from the Nintendo DS

While the mcs server is connected to the hardware that is run by an application for the Nintendo DS, for a fixed time interval the server will be checking whether any data need to be sent from the Nintendo DS to the Windows application. This time interval can be changed in the Options dialog box. For example, if the operations of the application for the Nintendo DS start to slow down when a large amount of data are sent to the Windows application, shortening this time interval will improve the operations in some cases. However, if the time interval is shortened, the processing load on the equivalent Windows-side processes will increase.

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries.

All other company names and product names mentioned in this document are the registered trademarks or trademarks of those other companies.

© 2005-2007 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed or loaned in whole or in part without the prior approval of Nintendo Co. Ltd.