TWL-System
Multiple Channel Stream Library

Communications Between TWL or Nintendo DS Programs and Multiple Windows Applications

2009/07/31

The content of this document is highly confidential and should be handled accordingly.

TWL-06-0088-001-B
Released: September 11, 2009

Table of Contents

Code

Figures

Revision History

Revision Date	Description
2009/07/31	Revised note about building in an environment where IS-TWL-DEBUGGER software is not installed.
2009/06/22	Added a note about building in an environment where IS-TWL-DEBUGGER software is not installed.
2008/07/14	Made revisions in line with the NITRO-System name change (from NITRO-System to TWL-System). Added a description about TWL support. Changed mentions of the TWL-TS board to IS-TWL-DEBUGGER.
2008/04/08	Changed the format of the Revision History. Added information about the TWL-TS board.
2007/11/26	Revised text according to changes in the initialization function.
2007/03/14	Added a feature for turning on the power to the DS Game Card slot.
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 program on an MCS server.
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 TWL or Nintendo DS programs to communicate with multiple Microsoft Windows applications. The library provides the following features.

- Ability to communicate between TWL or Nintendo DS programs and Windows applications
- Ability to access files on the PC from TWL or Nintendo DS programs
- Ability to display text strings output from TWL or Nintendo DS programs

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

- IS-NITRO-EMULATOR
- IS-TWL-DEBUGGER (hardware)
- Nintendo DS System and IS-NITRO-UIC
- ensata software emulator

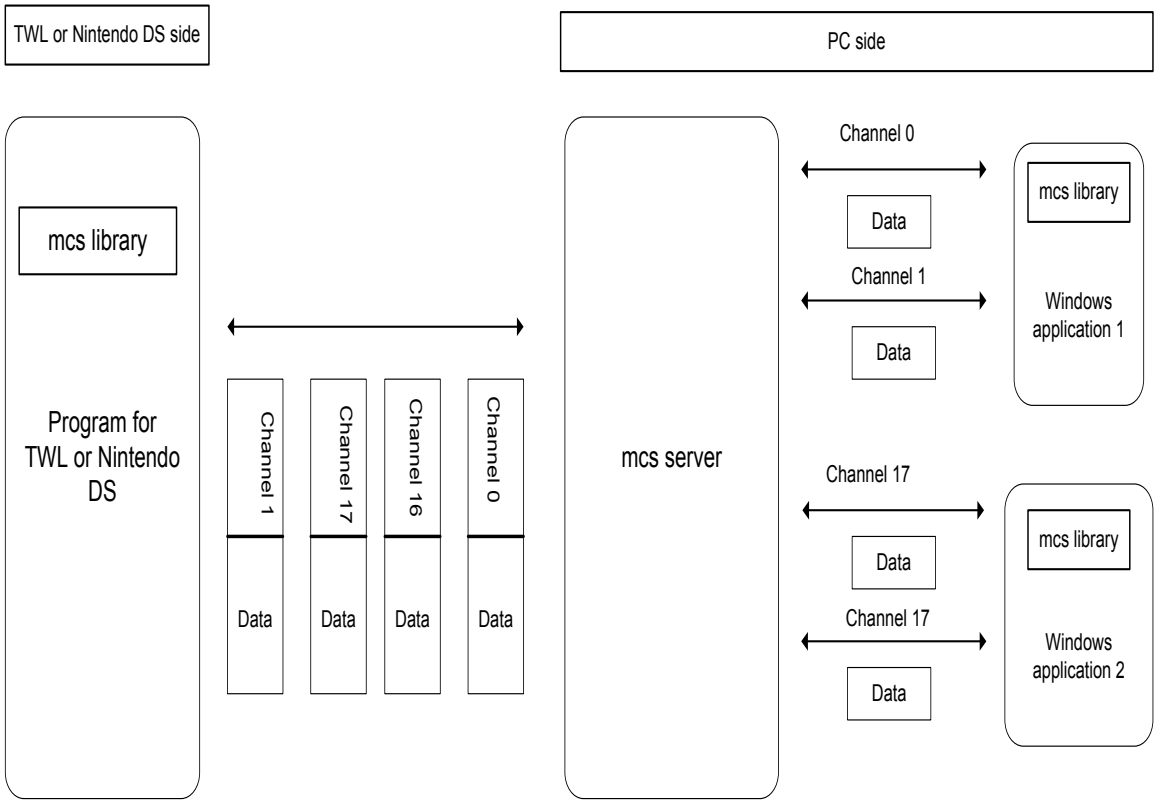
Notes:

- If you are using IS-NITRO-EMULATOR or IS-NITRO-UIC, the `ISNITRO.dll` must be installed on the system. `ISNITRO.dll` is installed on the system when you install the IS-NITRO-DEBUGGER software.
- If you are using IS-TWL-DEBUGGER (hardware), `ISTWL.dll` must be installed on the system. `ISTWL.dll` is installed on the system when IS-TWL-DEBUGGER (software) is installed.
- If the SRL files have been built or the MCS library itself has been rebuilt in an environment where IS-NITRO-DEBUGGER is installed but the IS-TWL-DEBUGGER software is not installed, the MCS library will not work when run on IS-TWL-DEBUGGER hardware. To run the library on IS-TWL-DEBUGGER hardware, be sure that IS-TWL-DEBUGGER is installed in the build environment.

2 Communications Between TWL and Nintendo DS Programs and Windows Applications

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

Figure 2-1 Communications Between TWL or Nintendo DS Programs and Windows Applications



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

2.1 TWL or Nintendo DS Procedures

2.1.1 Initializing the MCS Library

To use the MCS library, you must first call the `NNS_McsInit` function and initialize the library. When doing so, you specify the working memory that the MCS library will use internally. That memory requires a number of bytes equivalent to `NNS_MCS_WORKMEM_SIZE` and must be 4-byte aligned.

Code 2-1 Initializing the MCS Library

```
// Working memory to be used by the MCS library
static u32 sMcsWork
[(NNS_MCS_WORKMEM_SIZE + sizeof(u32) - 1) / sizeof(u32)];

void
NitroMain
{
  OS_Init();
  …
  NNS_McsInit(sMcsWork);
  …
```

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 specified time. For both methods, each channel must be set in advance.

2.1.2.1 Register a Callback Function

To call a callback when data has been received, register a callback function. Secure the variable of the `NNSMcsRecvCBInfo` structure in advance and call the `NNS_McsRegisterRecvCallback` function 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 `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 specified time, you need to register a buffer to hold the received data. Memory for the received data must be secured in advance. 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 allocated from the specified memory-use buffer. Be sure to allocate at least 48 bytes. If received data accumulates in this buffer without being read and the buffer overflows, received data will be discarded. Thus, it is essential to allocate appropriately sized buffers 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, no devices were found. If there are one 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 previously set. 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 that 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 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 the previously explained interrupts, 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 Callback Function Has Been Registered

If you have registered a callback function, that function is called when data is received.

2.1.6.2 Receiving Buffer Has Been Registered

If you have registered a Receiving buffer, the received data is stored in that buffer. As shown in Code 2-7, call the NNS_McsReadStream function to read the data stored in the buffer. 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 size obtained by NNS_McsGetStreamWritableLength, NNS_McsWriteStream quits immediately. If the amount of data to write is larger than the writable amount, calls to NNS_McsWriteStream are blocked until the specified amount of data is written.

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 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. Call the `NNS_McsIsServerConnect` function to check if the MCS server is connected. If the server is connected to IS-NITRO-UIC, `NNS_McsIsServerConnect` returns `TRUE`.

The communications state of the MCS server is checked via MCS communications. Therefore, there may be a slight time lag before the actual connection state of the MCS server is 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 Windows Procedures

2.2.1 Read the DLL and Get the Function Address

The library for Windows procedures is the dynamic link library `nnsmcs.dll`. This file is in the `tools\win\mcsserver` directory, under the directory where TWL-System is 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 the DLL and Getting the Function Address

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

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

// Get the absolute path for nnsmcs.dll
writtenChars = ExpandEnvironmentStrings(
  _T("%NITROSYSTEM_ROOT%\\tools\\win\\mcsserver\\nnsmcs.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 is 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_McsOpenStreamEx` 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 the `PeekNamedPipe` function.

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 the 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.

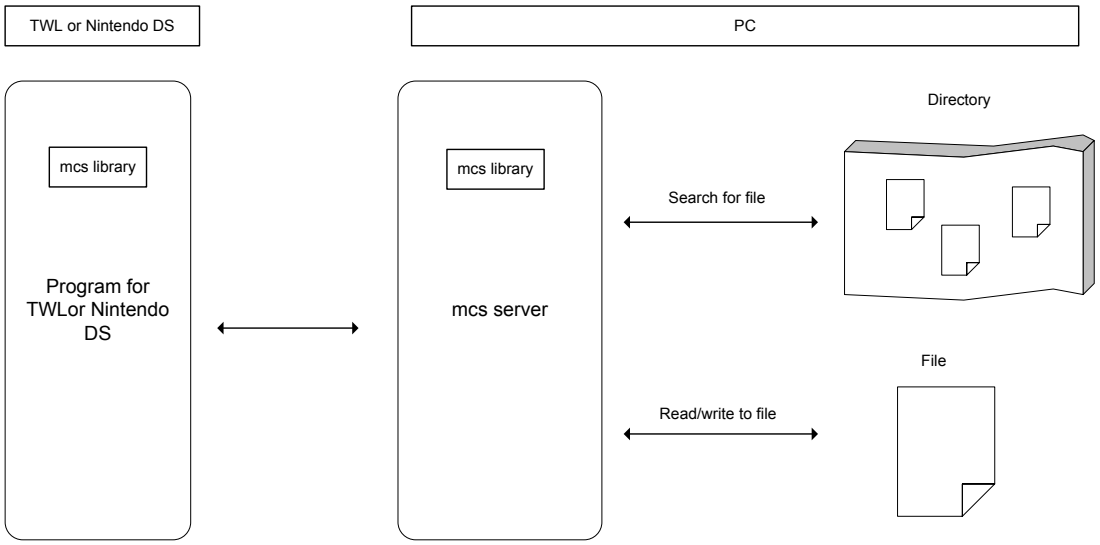
Code 2-14 Closing the Stream

```
// 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 TWL or Nintendo DS programs, and for searching for files on the PC from TWL or Nintendo DS programs. 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 TWL or Nintendo DS system.

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

3.1 Initializing the MCS File Input/Output Library

To use the file search and read/write features, you must call the `NNS_McsInitFileIO` function to initialize the file input/output library after calling the `NNS_McsInit` function to initialize the MCS library. When calling `NNS_McsInitFileIO`, you specify the working memory that the file input/output library will use internally. That memory requires a number of bytes equivalent to `NNS_MCS_FILEIO_WORKMEM_SIZE` and must be 4-byte aligned.

Code 3-1 Initializing the MCS File Input/Output Library

```
// Working memory to be used by the file I/O library
static u32 sMcsFileIOWork
  [(NNS_MCS_FILEIO_WORKMEM_SIZE + sizeof(u32) - 1) / sizeof(u32)];

NNS_McsInit( ... );                     // Initialize the MCS library
…
NNS_McsInitFileIO(sMcsFileIOWork);      // Initialize file I/O features
```

3.2 File Reading and Writing

3.2.1 Opening 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 set `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 is placed in the `NNSMcsFile` type variable. If the process fails, the function returns a nonzero value.

Code 3-2 Opening a File

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

// Open file for reading
errCode = NNS_McsOpenFile(
  &infoRead,
  "c:\\testApp\\test.txt",       // Filename
  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 Reading from the File

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

Code 3-3 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 Writing to the File

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

Code 3-4 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 Closing the File

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

Code 3-5 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 3-6 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 set NNSMcsFile type variable, the pointer to the previously set NNSMcsFileFindData type variable, and the search string.

If the function finds a matching file, it returns 0, sets the information related to the search in the NNSMcsFile type variable, and sets 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 3-7 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 set `NNSMcsFileFindData` type variable. If the function finds a matching file, it returns 0, sets the information related to the search in the `NNSMcsFile` type variable, and sets the information related to the found file in the `NNSMcsFileFindData` type variable. (This is the same as a successful call to the `NNS_McsFindFirstFile` function.) If the function cannot find a file that matches the pattern, it returns `NNS_MCS_FILEIO_ERROR_NOMOREFILES`.

Code 3-8 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 3-9 Ending File Search

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

4 Directing Output to the Console

The MCS library provides features for directing output to the MCS server's console. There are two ways to direct the output: by using the TWL-SDK function `OS_Printf` or by using one of the MCS library's string output functions. The differences between these methods are explained in the following sections.

4.1 Output with the OS_Printf Function

If you output using the `OS_Printf` function, the string only displays on the MCS console if the MCS server is connected to IS-NITRO-EMULATOR. The string is not displayed 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 direct output, the strings can be output no matter what connected device is used, as long as MCS communications have been established. However, the output can go only to the console of the MCS server.

The following sections explain how to use the MCS library functions to direct output.

4.2.1 Initialize the String Output Library

To use the features to direct output, 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 Initializing the String Output Library

```
NNS_McsInit();            // Initialize the MCS library
…
NNS_McsInitPrint();       // Initialize the string output feature
```

4.2.2 Output a String

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

Code 4-2 Outputting a 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 TWL or Nintendo DS programs and multiple Windows applications on a PC. The MCS server also provides features that allow TWL or Nintendo DS programs to access files on the PC and to direct output to the console of the MCS server.

5.1 General Operations Flow

5.1.1 Selecting Hardware for Communication

Select the hardware that will be running the TWL or Nintendo DS program that will be communicating with the Windows application.

- When communicating with IS-NITRO-EMULATOR or IS-NITRO-UIC, select **Nitro** from the **Device** menu.
- When communicating with IS-TWL-DEBUGGER hardware, select **Twl** from the **Device** menu.
- When communicating with ensata, select **ensata** from the **Device** menu.

5.1.2 Connecting

Before communications can proceed between Windows applications and a TWL or Nintendo DS program, and before a TWL or Nintendo DS program can access PC files or output strings to the MCS server console, the MCS server must connect to a hardware device that is running a TWL or Nintendo DS program. Connect to the hardware by selecting **Connect** from the **Device** menu.

If **ensata** is selected from the **Device** menu, ensata starts up by selecting **Connect** from the **Device** menu.

Note: 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.

5.1.3 Loading ROM Files with IS-NITRO-EMULATOR or IS-TWL-DEBUGGER Hardware

If the MCS server is connected to IS-NITRO-EMULATOR or IS-TWL-DEBUGGER hardware, load the ROM file after the connection is established. Select **Open** from the **File** menu. In the **Open** dialog box, select the file you want to read. After the file is loaded, the TWL or Nintendo DS program starts.

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

5.1.4 Disconnecting

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

5.1.5 Resetting IS-NITRO-EMULATOR or IS-TWL-DEBUGGER Hardware

If the connected device is IS-NITRO-EMULATOR or IS-TWL-DEBUGGER hardware, 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 reset the system.

5.2 Special Situations

5.2.1 Shared Mode and Dedicated Mode

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

When the MCS server is in the dedicated mode, the TWL or Nintendo DS program can communicate with only 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 shared mode, there are no such restrictions.

5.2.2 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.
/N              Set IS-NITRO-EMULATOR or IS-NITRO-UIC as the device to connect to.
/T              Set IS-TWL-DEBUGGER hardware as the device to connect to.
/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 or IS-TWL-DEBUGGER (hardware).
```

5.2.3 Powering the IS-NITRO-EMULATOR DS Game Card and GBA Game Pak Slots

When the command-line option `/D` is specified, the DS Game Card slot is powered on when the MCS server connects to the IS-NITRO-EMULATOR device. This enables simultaneous use of hardware that supports the DS Game Card slot.

When the command-line option `/A` is specified, the GBA Game Pak slot is powered on when the MCS server connects to the IS-NITRO-EMULATOR device. This enables simultaneous use of hardware that supports the GBA Game Pak slot.

Note: Do not insert or remove a game device while the power is on. Doing so could damage the device.

5.2.4 Setting the Interval to Obtain Data from the TWL or Nintendo DS System

When the MCS server is connected to hardware run by a program for the TWL or a Nintendo DS system, the server periodically checks whether data needs to be sent from the handheld device to the Windows application. This time interval can be changed in the **Options** dialog box. For example, if the application on the device slows down when a large amount of data is sent to the Windows application, shortening this time interval may improve the performance on the device. However, if the time interval is shortened, the processing load on the equivalent Windows-side processes increases.