

Multiple Channel Stream ライブラリ

TWL, ニンテンドーDS と複数 Windows アプリケーションとの通信

2009-07-31

任天堂株式会社発行

このドキュメントの内容は、機密情報であるため、**厳重な取り扱い、管理を行ってください。**

目次

1	はじめに	5
2	TWLおよびニンテンドーDS用プログラムとWindowsアプリケーションとの通信	6
2.1	TWLまたはニンテンドーDS側	6
2.1.1	mcsライブラリの初期化	6
2.1.2	データの受信方法の設定	7
2.1.2.1	コールバック関数の登録	7
2.1.2.2	バッファの登録	7
2.1.3	デバイスのオープン	8
2.1.4	割り込みの設定	8
2.1.5	ポーリング	9
2.1.6	データの読み取り	10
2.1.6.1	コールバック関数を登録した場合	10
2.1.6.2	受信用バッファを登録した場合	10
2.1.7	データの書き込み	10
2.1.8	オープンしたデバイスがIS-NITRO-UICの場合	11
2.2	Windows側	12
2.2.1	DLLの読み込みと関数のアドレスの取得	12
2.2.2	ストリームのオープン	12
2.2.3	ストリームからの読み込み	13
2.2.4	ストリームへの書き込み	14
2.2.5	ストリームのクローズ	14
3	ファイルの検索とファイルの読み書き	15
3.1	mcsファイル入出力ライブラリの初期化	15
3.2	ファイルの読み書き	15
3.2.1	ファイルのオープン	15
3.2.2	ファイルからの読み込み	16
3.2.3	ファイルへの書き込み	17
3.2.4	ファイルのクローズ	17
3.2.5	ファイルポインタの移動	17
3.3	ファイルの検索	18
3.3.1	ファイル検索の開始	18
3.3.2	ファイル検索の続き	19
3.3.3	ファイル検索の終了	19
4	コンソールへの文字列の出力	20
4.1	関数OS_Printfによる出力	20
4.2	mcsの文字列出力関数による出力	20
4.2.1	文字列出力ライブラリの初期化	20
4.2.2	文字列の出力	20
5	mcsサーバについて	21

5.1	一般的な操作の流れ	21
5.1.1	通信するハードウェアの選択	21
5.1.2	接続	21
5.1.3	ROMファイルの読み込み(IS-NITRO-EMULATORまたはIS-TWL-DEBUGGERハードウェアの場合)	21
5.1.4	切断	21
5.1.5	リセット(IS-NITRO-EMULATORまたはIS-TWL-DEBUGGERハードウェアの場合)	21
5.2	特別な場合	22
5.2.1	共有モードと専有モード	22
5.2.2	コマンドラインオプション	22
5.2.3	IS-NITRO-EMULATORのDSカード差込口/GBAカートリッジ差込口の電源ONについて	22
5.2.4	TWLまたはニンテンドーDSからのデータを取得する間隔について	23

コード

コード 2-1	mcsライブラリの初期化	6
コード 2-2	コールバック関数の登録	7
コード 2-3	受信用バッファの登録	8
コード 2-4	デバイスのオープン	8
コード 2-5	割り込みの設定	9
コード 2-6	ポーリング関数の呼び出し	10
コード 2-7	受信データの読み取り	10
コード 2-8	データの書き込み	11
コード 2-9	mcsサーバ接続の待機	11
コード 2-10	DLLの読み込みと関数のアドレスの取得	12
コード 2-11	ストリームのオープン	13
コード 2-12	ストリームからの読み込み	13
コード 2-13	ストリームへの書き込み	14
コード 2-14	ストリームのクローズ	14
コード 3-1	ファイルのオープン	16
コード 3-2	ファイルからの読み込み	16
コード 3-3	ファイルへの書き込み	17
コード 3-4	ファイルのクローズ	17
コード 3-5	ファイルポインタの移動	18
コード 3-6	ファイル検索の開始	18
コード 3-7	ファイル検索の続き	19
コード 3-8	ファイル検索の終了	19
コード 4-1	文字列出力ライブラリの初期化	20
コード 4-2	文字列の出力	20



図 2-1	TWLまたはニンテンドーDS用プログラムとWindowsアプリケーションとの通信	6
図 3-1	ファイルの検索とファイルの読み書き	15

改訂履歴

改訂日	改訂内容
2009-07-31	IS-TWL-DEBUGGER ソフトウェアをインストールしていない環境でビルドされた場合についての注意点の説明を修正。
2009-06-22	IS-TWL-DEBUGGER ソフトウェアをインストールしていない環境でビルドされた場合についての注意点を追加。
2008-07-14	<ul style="list-style-type: none"> •NITRO-System の名称変更のよる修正 (NITRO-System を TWL-System に変更)。 •TWL 対応についての記述を追加。 •TWL-TS ボードに関する記述を IS-TWL-DEBUGGER に変更。
2008-04-08	<ul style="list-style-type: none"> •改訂履歴の書式を変更。 •TWL-TS ボードについて追加。
2007-11-26	初期化関数の変更による説明の修正。
2007-03-14	DS カード差込口の電源を ON にする機能を追加。
2005-03-18	<ul style="list-style-type: none"> •現在のファイルポインタの位置を変更する関数を追加。 •mcs サーバのニンテンドーDS からの読み取り時間間隔の変更機能の追加。
2005-01-18	初版。

1 はじめに

mcs ライブラリは、TWL またはニンテンドーDS 用(実機用)プログラムと複数の Windows アプリケーションとの通信を可能にするライブラリおよびツールプログラム群の総称です。具体的には次のような機能を提供します。

- 実機用プログラムと複数の Windows アプリケーションとの間で通信を行えるようにする機能。
- 実機用プログラムから PC 上のファイルにアクセスする機能。
- 実機用プログラムからの文字列出力の表示。

TWL またはニンテンドーDS 用プログラムが動作するハードウェアの中で、mcs ライブラリが対応しているものは次の通りです。

- IS-NITRO-EMULATOR
- IS-TWL-DEBUGGER ハードウェア
- DS 本体 + IS-NITRO-UIC
- ソフトウェアエミュレータ ensata

IS-NITRO-EMULATOR や IS-NITRO-UIC を利用する場合は”ISNITRO.dll”がシステムにインストールされている必要があります。

”ISNITRO.dll”は、IS-NITRO-DEBUGGER をインストールすることで、システムにインストールされます。

IS-TWL-DEBUGGER ハードウェアを利用する場合は”ISTWL.dll”がシステムにインストールされている必要があります。

”ISTWL.dll”は、IS-TWL-DEBUGGER ソフトウェアをインストールすることで、システムにインストールされます。

IS-NITRO-DEBUGGER をインストールし、IS-TWL-DEBUGGER ソフトウェアをインストールしていない環境で SRL ファイルをビルドした場合、あるいは MCS ライブラリ自体をビルドしなおした場合は、IS-TWL-DEBUGGER ハードウェア上での実行時に MCS ライブラリが機能しなくなります。IS-TWL-DEBUGGER ハードウェアで実行させる場合は IS-TWL-DEBUGGER ソフトウェアをインストールするようにして下さい。

2 TWLおよびニンテンドーDS用プログラムとWindowsアプリケーションとの通信

mcs ライブラリの基本的な目的として、1 つの TWL またはニンテンドーDS 用 (実機用) プログラムと PC 上で動作する複数の Windows アプリケーションとの通信を可能にすることがあります。次にその概念を示します。

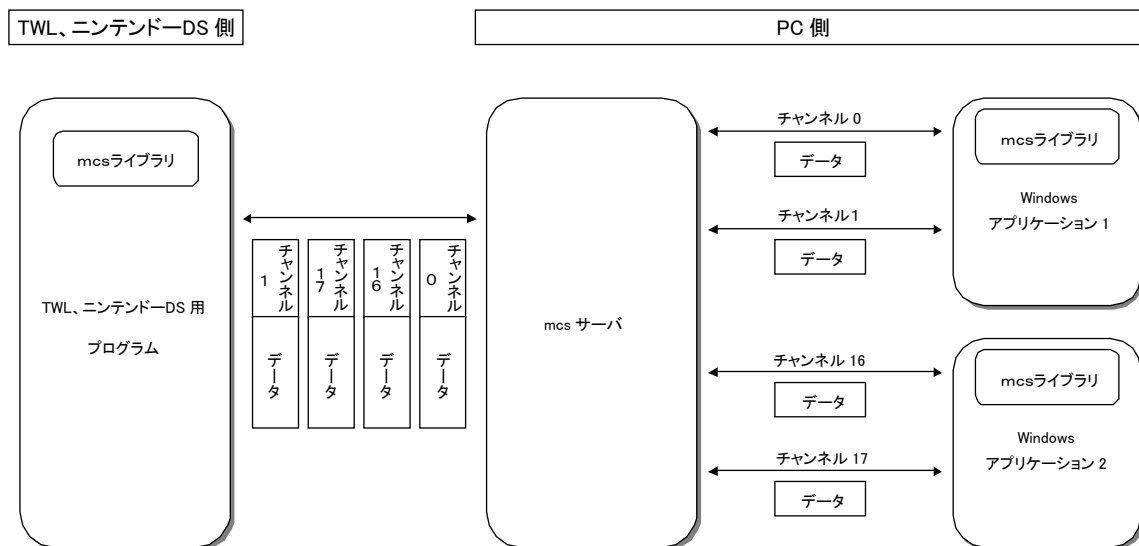


図 2-1 TWL またはニンテンドーDS 用プログラムと Windows アプリケーションとの通信

実機用プログラムと Windows アプリケーションの両方に通信するための手順を必要としますが、それぞれ行う内容が異なりますので、それぞれ個別に説明します。

2.1 TWLまたはニンテンドーDS側

2.1.1 mcsライブラリの初期化

mcs ライブラリを使用するには、最初に関数 `NNS_McsInit` を呼び出して初期化しておく必要があります。`NNS_McsInit` を呼び出す際に、mcs ライブラリが内部で使用するワーク用メモリを指定します。ワーク用メモリは `NNS_MCS_WORKMEM_SIZE` のバイト数で、4 バイトアライメント境界にある必要があります。

コード 2-1 mcs ライブラリの初期化

```
// mcsライブラリが使用するワーク用メモリ
static u32 sMcsWork
    [(NNS_MCS_WORKMEM_SIZE + sizeof(u32) - 1) / sizeof(u32)];

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

2.1.2 データの受信方法の設定

データの受信方法は、データを受信したときにコールバックで呼び出す方法と、プログラムの任意のタイミングで読み出すことが出来るようにする方法とがあります。どちらの場合も、あらかじめチャンネル毎に設定しておく必要があります。

2.1.2.1 コールバック関数の登録

データを受信したときにコールバックで呼び出すようにするには、コールバック関数を登録しておきます。あらかじめ、構造体 `NNSMcsRecvCBInfo` の変数を確保しておき、この変数へのポインタと、Windows アプリケーションとの識別で用いるチャンネル値、コールバック関数、コールバック関数に渡すユーザ定義の値を引数として関数 `NNS_McsRegisterRecvCallback` を呼び出します。登録した内容は関数呼び出し時に指定した `NNSMcsRecvCBInfo` 型の変数にセットされます。

コード 2-2 コールバック関数の登録

```
#define MCS_CHANNEL_ID 10    // チャンネル値

// PC側からデータを受信したときに呼ばれるコールバック関数
static void
DataRecvCallback(
    const void* pRecv,      // データバッファへのポインタ
    u32         recvSize,   // 受信データサイズ
    u32         userData,   // ユーザ定義値
    u32         offset,     // 受信データ全体に対してのオフセット値
    u32         totalSize  // 受信データ全体のサイズ
)
{
}

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

    // コールバック関数の登録
    NNS_McsRegisterRecvCallback(
        &sRecvCBInfo,      // NNSMcsRecvCBInfo型変数
        MCS_CHANNEL_ID,    // チャンネル値
        DataRecvCallback,  // コールバック関数
        0);                // ユーザ定義値
    ...
}
```

2.1.2.2 バッファの登録

任意のタイミングでデータを読み出すことが出来るようにするには、データの受信用のバッファを登録する必要があります。受信用のメモリをあらかじめ確保しておき、チャンネル値とともに関数 `NNS_McsRegisterStreamRecvBuffer` を呼び出します。

受信用バッファの管理用メモリをここで指定したバッファ用メモリから確保するため、少なくとも 48 バイト以上のサイズである必要があります。また、受信したデータが読み出されること無くバッファに蓄積されてあふれ出した場合は、その受信データは捨てられます。従って、チャンネル毎に使用目的に応じて適切なサイズのバッファを割り当てる必要があります。

コード 2-3 受信用バッファの登録

```
#define MCS_CHANNEL_ID 10    // チャンネル値

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

...

NNS_McsRegisterStreamRecvBuffer(
    MCS_CHANNEL_ID,          // チャンネル値
    sRecvBuf,                // 受信用バッファへのポインタ
    sizeof(sRecvBuf));       // 受信用バッファのサイズ
```

2.1.3 デバイスのオープン

通信を行うためのデバイスをオープンします。まず、関数 `NNS_McsGetMaxCaps` を呼び出し、通信可能なデバイスの総数を取得します。総数が 0 の場合、デバイスが見つからないことを示しています。

デバイスが 1 つ以上見つかったら、関数 `NNS_McsOpen` を使ってデバイスをオープンします。引数には、あらかじめ確保してある `NNSMcsDeviceCaps` 型の変数へのポインタを指定します。この変数にはオープンしたデバイスに関する情報が入ります。

コード 2-4 デバイスのオープン

```
NNSMcsDeviceCaps deviceCaps;

if (NNS_McsGetMaxCaps() == 0)
{
    OS_Panic("デバイスが見つかりません。");
}

if (! NNS_McsOpen(&deviceCaps))
{
    OS_Panic("デバイスのオープンに失敗しました。");
}
```

2.1.4 割り込みの設定

オープンしたデバイスの種類によって、定期的に特定の関数を呼び出す必要があります。デバイスがどの関数呼び出しを必要とするかは、`NNS_McsOpen` の呼び出しのときに指定した `NNSMcsDeviceCaps` 型の変数の `maskResource` メンバ変数にセットされています。この変数とマスクを取り、必要な関数が呼び出されるように、割り込みハンドラの設定を行います。

例えば、`maskResource` 変数と `NITROMASK_RESOURCE_VBLANK` とのビット AND の結果が 0 でないなら、デバイスが毎フレーム関数 `NNS_McsVBlankInterrupt` の呼び出しを必要としているので、V ブランクの割り込みハンドラを設定し、割り込みハンドラ内で `NNS_McsVBlankInterrupt` を呼び出すようにします。

同様に、`maskResource` 変数と `NITROMASK_RESOURCE_CARTRIDGE` とのビット AND の結果が 0 でないなら、デバイスがカートリッジ割り込みが発生するたびに関数 `NNS_McsCartridgeInterrupt` の呼び出しを必要としているので、カートリッジ割り込みハンドラを設定し、割り込みハンドラ内で `NNS_McsCartridgeInterrupt` を呼び出すようにします。

コード 2-5 割り込みの設定

```
...

if (deviceCaps.maskResource & NITROMASK_RESOURCE_VBLANK)
{
    // VBlank割り込みを有効にし、VBlank割り込み内で
    // NNS_McsVBlankInterrupt()が呼ばれるようにする

    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)
{
    // カートリッジ割り込みを有効にし、カートリッジ割り込み内で
    // NNS_McsCartridgeInterrupt()が呼ばれるようにする。

    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();
}
```

もともと、関数 `NNS_McsVBlankInterrupt` や `NNS_McsCartridgeInterrupt` はデバイスがオープンされて必要とされるまでは、呼び出されても何も行いません。そのため、デバイスのオープンの前に、デバイスの種類に関係なく割り込みの設定を先に行うようにしても問題ありません。

2.1.5 ポーリング

上記で説明した割り込みの設定とは別に、定期的に関数 `NNS_McsPollingIdle` も呼び出すようにしてください。例えば、メインループの中などで毎回 `NNS_McsPollingIdle` を呼び出すようにします。

コード 2-6 ポーリング関数の呼び出し

```
// メインループ
while (TRUE)
{
    SVC_WaitVBlankIntr();

    ...

    // ポーリング処理
    NNS_McsPollingIdle();
}
```

2.1.6 データの読み取り

2.1.6.1 コールバック関数を登録した場合

コールバック関数を登録した場合は、データを受信したときに登録した関数が呼び出されます。

2.1.6.2 受信用バッファを登録した場合

受信用バッファを登録した場合は、受信したデータは一旦登録された受信用バッファに蓄積されます。バッファからデータを読み込むには関数 `NNS_McsReadStream` を使用します。関数 `NNS_McsGetStreamReadableSize` を使用すると、一度の `NNS_McsReadStream` 呼び出しで読み込み可能なサイズを取得できます。また、関数 `NNS_McsGetTotalStreamReadableSize` を使用すると、バッファに蓄積されている読み込み可能なデータのサイズの合計を取得できます。

コード 2-7 受信データの読み取り

```
static u8 sBuf[1024];

u32 nLength = NNS_McsGetStreamReadableSize(MCS_CHANNEL_ID);

if (nLength > 0)
{
    u32 readSize;
    BOOL result = NNS_McsReadStream(
        MCS_CHANNEL_ID, // チャンネル値
        sBuf,           // 読み込み用バッファへのポインタ
        sizeof(sBuf),   // 読み込み用バッファのサイズ
        &readSize);      // 実際に読み込まれたサイズを格納する変数へのポインタ
    if (result)
    {
        // 読み込みOK
    }
    else
    {
        // 読み込み失敗
    }
}
```

2.1.7 データの書き込み

データを書き込むには、関数 `NNS_McsWriteStream` を使用します。関数 `NNS_McsGetStreamWritableLength` を使用して、その時点での書き込み可能なサイズを取得できます。`NNS_McsWriteStream` で書き込むデータのサイズが `NNS_McsGetStreamWritableLength` で取得できるサイズ以下の場合は、`NNS_McsWriteStream` はすぐに終了します。`NNS_McsGetStreamWritableLength` で取得できるサイズを超えている場合は、指定したサイズの書き

込みが完了するまで NNS_McsWriteStream 呼び出しはブロックします。

コード 2-8 データの書き込み

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

...

// 書き込み可能サイズを取得
if (NNS_McsGetStreamWritableLength(&nLength))
{
    // ブロックしないで書き込みできるなら書き込む
    if (sizeof(sendBuf) <= nLength)
    {
        // 書き込み
        if (! NNS_McsWriteStream(
            MCS_CHANNEL_ID,
            sendBuf,
            sizeof(sendBuf)))
        {
            // 書き込み成功
        }
        else
        {
            // 書き込み失敗
        }
    }
}
```

2.1.8 オープンしたデバイスがIS-NITRO-UICの場合

オープンしたデバイスが IS-NITRO-UIC の場合は、mcs サーバ側が IS-NITRO-UIC と接続していない状態で関数 NNS_McsWriteStream を呼び出すと、mcs サーバが接続するまで NNS_McsWriteStream は制御を返しません。このことが問題となる場合は、関数を NNS_McsIsServerConnect を呼び出して、mcs サーバが接続状態にあるかどうかをチェックしてください。NNS_McsIsServerConnect は mcs サーバが接続状態にあると真を返します。

mcs サーバの通信状態の確認は、mcs の通信の機能を利用して判断しています。そのため、実際の mcs サーバの接続状態が反映されるまで若干のタイムラグが生じます。

コード 2-9 mcs サーバ接続の待機

```
NNSMcsDeviceCaps deviceCaps;

...

if (NNS_McsOpen(&deviceCaps))
{
    // mcsサーバが接続してくるのを待つ
    while (! NNS_McsIsServerConnect())
    {
        SVC_WaitVBlankIntr();
    }
}
```

2.2 Windows側

2.2.1 DLLの読み込みと関数のアドレスの取得

Windows 用のライブラリは”nnsms.dll”というダイナミックリンクライブラリの形式で提供しています。このファイルは TWL-System がインストールされているディレクトリの下の”tools¥win¥mcserver”ディレクトリにあります。

このライブラリでエクスポートしている関数は後述しますストリームのオープン用の NNS_McsOpenStream および NNS_McsOpenStreamEx です。必要に応じて関数のアドレスを取得します。

コード 2-10 DLL の読み込みと関数のアドレスの取得

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

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

// nnsms.dllの絶対パスを求める
writtenChars = ExpandEnvironmentStrings(
    _T( "%NITROSYSTEM_ROOT%¥¥tools¥¥win¥¥mcserver¥¥nnsms.dll" ),
    modulePath,
    MAX_PATH);
if (writtenChars > MAX_PATH)
{
    // パスが長すぎる
    return 1;
}

hModule = LoadLibrary(modulePath);
if (NULL == hModule)
{
    // モジュールの読み込みに失敗
    return 1;
}

// 関数のアドレスを取得
pfOpenStream = (NNSMcsPFOpenStream)GetProcAddress(
    hModule,
    NNS_MCS_API_OPENSTREAM);
```

2.2.2 ストリームのオープン

Windows 側ではチャンネル毎にストリームをオープンします。ストリームのオープンには関数 NNS_McsOpenStream か関数 NNS_McsOpenStreamEx を使用します。NNS_McsOpenStreamEx は NNS_McsOpenStream の機能に加えて、接続したデバイスの情報が取得できるようになっています。

ストリームは、実際には Win32 システムの名前付きパイプです。関数 NNS_McsOpenStream(Ex)は、名前付パイプをメッセージタイプでオープンし、指定したチャンネルを mcs サーバに登録します。

コード 2-11 ストリームのオープン

```
HANDLE hStream;

// ストリームのオープン
hStream = pfOpenStream(
    MCS_CHANNEL_ID,    // チャンネル値
    0);               // フラグ
if (hStream == INVALID_HANDLE_VALUE)
{
    // オープン失敗
    return 1;
}
```

2.2.3 ストリームからの読み込み

ストリームからの読み込みには、Win32 API の ReadFile あるいは ReadFileEx を使用します。読み込み可能なサイズを取得するには PeekNamedPipe を使用します。

コード 2-12 ストリームからの読み込み

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

fSuccess = PeekNamedPipe(
    hStream,          // ストリームのハンドル
    NULL,
    0,
    NULL,
    &totalBytesAvail, // 利用可能なバイト数
    NULL);
if (! fSuccess)
{
    // Peek失敗
    return 1;
}

// 読み込み可能なデータがあるとき
if (totalBytesAvail > 0)
{
    DWORD readBytes;

    fSuccess = ReadFile(
        hStream,      // ストリームのハンドル
        buf,          // 読み込み用バッファへのポインタ
        sizeof(buf),  // 読み込むバイト数
        &readBytes,    // 実際に読み込んだバイト数
        NULL);
    if (! fSuccess)
    {
        // 読み込み失敗;
        return 1;
    }
}
```

2.2.4 ストリームへの書き込み

ストリームへの書き込みには、Win32 API の WriteFile あるいは WriteFileEx を使用します。

コード 2-13 ストリームへの書き込み

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

fSuccess = WriteFile(
    hStream,          // ストリームのハンドル
    buf,              // 書き込み用バッファへのポインタ
    sizeof(buf),      // 書き込むバイト数
    &writtenBytes,     // 実際に書き込んだバイト数
    NULL);
if (! fSuccess)
{
    // 書き込み失敗
    return 1;
}
```

2.2.5 ストリームのクローズ

ストリームのクローズには、Win32 API の CloseHandle を使用します。

コード 2-14 ストリームのクローズ

```
// ストリームのクローズ
CloseHandle(hStream);
```

3 ファイルの検索とファイルの読み書き

mcs ライブラリでは、TWL またはニンテンドーDS 用のプログラムから PC 上のファイルに対して読み書きを行ったり、PC 上のファイルを検索する機能を提供しています。次にその概念を示します。

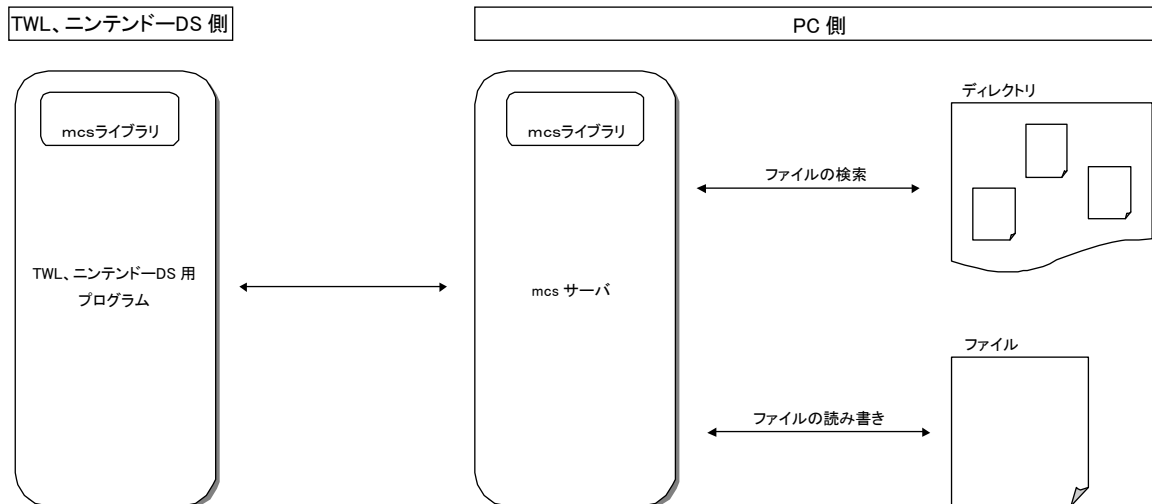


図 3-1 ファイルの検索とファイルの読み書き

この機能についての Windows 用のライブラリは無く、mcs サーバが TWL もしくはニンテンドーDS のデバイスと接続されていれば利用可能となります。次にファイルの検索とファイルの読み書きを行う方法について説明します。

3.1 mcsファイル入出力ライブラリの初期化

ファイルの検索とファイルの読み書きの機能を利用するには、関数 `NNS_McsInit` にて mcs ライブラリの初期化を行った後、関数 `NNS_McsInitFileIO` を呼び出してファイル入出力ライブラリを初期化しておく必要があります。`NNS_McsInitFileIO` を呼び出す際に、ファイル入出力ライブラリが内部で使用するワーク用メモリを指定します。ワーク用メモリは `NNS_MCS_FILEIO_WORKMEM_SIZE` のバイト数で、4 バイトアライメント境界にある必要があります。

```
// ファイル入出力ライブラリが内部で使用するワーク用メモリ
static u32 sMcsFileIOWork
    [(NNS_MCS_FILEIO_WORKMEM_SIZE + sizeof(u32) - 1) / sizeof(u32)];

NNS_McsInit( ... ); // mcsライブラリの初期化
...
NNS_McsInitFileIO(sMcsFileIOWork); // ファイル入出力機能の初期化
```

3.2 ファイルの読み書き

3.2.1 ファイルのオープン

PC 上のファイルをオープンするには、関数 `NNS_McsOpenFile` を呼び出します。引数には、あらかじめ確保してある `NNSMcsFile` 型の変数へのポインタ、オープンするファイル名、読み書きのフラグを指定します。オープンに成功すれば 0 が返り、`NNSMcsFile` 型の変数にオープンしたファイルに関する情報が入ります。失敗すれば 0 以外の値が返ります。

コード 3-1 ファイルのオープン

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

// 読み込み用オープン
errCode = NNS_McsOpenFile(
    &infoRead,
    "c:¥¥testApp¥¥test.txt",    // ファイル名
    NNS_MCS_FILEIO_FLAG_READ);  // 読み込みモード
if (errCode != 0)
{
    // ファイルオープン失敗
    return 1;
}

// 書き込み用オープン
errCode = NNS_McsOpenFile(
    &infoWrite,
    "c:¥¥testApp¥¥outTest.txt",
    NNS_MCS_FILEIO_FLAG_WRITE);
if (errCode != 0)
{
    // ファイルオープン失敗
    return 1;
}
```

3.2.2 ファイルからの読み込み

ファイルから読み込みには関数 `NNS_McsReadFile` を使用します。ファイルのサイズは関数 `NNS_McsGetFileSize` で取得できます。

コード 3-2 ファイルからの読み込み

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

// ファイルサイズの取得
fileSize = NNS_McsGetFileSize(&infoRead);

if (fileSize <= sizeof(buf))
{
    // ファイル全体を一挙に読み込む
    errCode = NNS_McsReadFile(
        &infoRead,
        buf,          // 読み込むバッファへのポインタ
        fileSize,     // 読み込みバイト数
        &readSize);    // 実際に読み込んだバイト数
    if (errCode != 0)
    {
        // ファイル読み込み失敗
        return 1;
    }
}
```


3.2.3 ファイルへの書き込み

ファイルへの書き込みには関数 `NNS_McsWriteFile` を使用します。

コード 3-3 ファイルへの書き込み

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

// bufの内容全てを書き込む
errCode = NNS_McsWriteFile(
    &infoWrite,
    buf,          // 書き込みバッファへのポインタ
    sizeof(buf)); // 書き込みバイト数
if (errCode != 0)
{
    // ファイル書き込み失敗
    return 1;
}
```

3.2.4 ファイルのクローズ

ファイルのクローズには関数 `NNS_McsCloseFile` を使用します。

コード 3-4 ファイルのクローズ

```
u32 errCode;

errCode = NNS_McsCloseFile(&infoRead);
if (errCode)
{
    // ファイルクローズ失敗
    return 1;
}
```

3.2.5 ファイルポインタの移動

現在のファイルポインタを移動するには関数 `NNS_McsSeekFile` を使用します。`u32` 型の変数のポインタを渡すことで、移動後のファイルポインタの位置を取得することもできます。

コード 3-5 ファイルポインタの移動

```
u32 errorCode;
u32 filePointer; // ファイルポインタの位置を格納するための変数

// ファイルの先頭から100バイト目に移動。
errorCode = NNS_McsSeekFile(&infoRead, 100, NNS_MCS_FILEIO_SEEK_BEGIN, NULL);
...
// 現在のファイルポインタの位置から200バイト移動。
// 移動後のファイルポインタの位置を取得。
errorCode = NNS_McsSeekFile(&infoRead, 200, NNS_MCS_FILEIO_SEEK_CURRENT,
                            &filePointer);
...
// 現在のファイルポインタの位置を取得。
// ファイルポインタは移動しない。
errorCode = NNS_McsSeekFile(&infoRead, 0, NNS_MCS_FILEIO_SEEK_CURRENT,
                            &filePointer);
```

3.3 ファイルの検索

3.3.1 ファイル検索の開始

ファイル検索を行うには、最初に関数 `NNS_McsFindFirstFile` を呼び出します。引数には、あらかじめ確保してある `NNSMcsFile` 型の変数へのポインタ、あらかじめ確保してある `NNSMcsFileFindData` 型の変数へのポインタ、検索するファイルのパターン文字列を指定します。

一致するファイルが見つかった場合は 0 が返り、`NNSMcsFile` 型の変数に検索に関する情報がセットされ、`NNSMcsFileFindData` 型の変数に見つかったファイルに関する情報がセットされます。パターンに一致するファイルが見つからなかった場合は `NNS_MCS_FILEIO_ERROR_NOMOREFILES` が返ります。

コード 3-6 ファイル検索の開始

```
NNSMcsFile info;
NNSMcsFileFindData findData;
u32 errorCode;

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

// パターンに一致するファイルが見つからない
if (errorCode == NNS_MCS_FILEIO_ERROR_NOMOREFILES)
{
    OS_Printf("no match *.txt .¥n");
    return 0;
}

if (errorCode != 0)
{
    // ファイル検索の失敗
    return 1;
}
```

3.3.2 ファイル検索の続き

次にパターンに一致するファイルを検索するには、関数 `NNS_McsFindNextFile` を呼び出します。引数には `NNS_McsFindFirstFile` の呼び出し時に指定した `NNSMcsFile` 型の変数へのポインタ、あらかじめ確保してある `NNSMcsFileFindData` 型の変数へのポインタを指定します。一致するファイルが見つかった場合は `0` が返り、`NNS_McsFindFirstFile` の時と同様に、`NNSMcsFile` 型の変数に検索に関する情報がセットされ、`NNSMcsFileFindData` 型の変数に見つかったファイルに関する情報がセットされます。パターンに一致するファイルが見つからなかった場合は `NNS_MCS_FILEIO_ERROR_NOMOREFILES` が返ります。

コード 3-7 ファイル検索の続き

```
do
{
    // ファイル名の表示
    OS_Printf("find filename %s¥n", findData.name);

    // 次にパターンに一致するファイルの検索
    errCode = NNS_McsFindNextFile(&info, &findData);
}while (errCode == 0);

if (errCode != NNS_MCS_FILEIO_ERROR_NOMOREFILES)
{
    // ファイル検索の失敗
}
```

3.3.3 ファイル検索の終了

ファイルの検索を終了するには、関数 `NNS_McsCloseFind` を呼び出します。

コード 3-8 ファイル検索の終了

```
errCode = NNS_McsCloseFind(&info);
if (errCode != 0)
{
    // ファイル検索終了の失敗
    return 1;
}
```

4 コンソールへの文字列の出力

mcs ライブラリでは、mcs サーバのコンソールに文字列を出力する機能を提供しています。文字列の出力の方法には 2 つあり、1 つは TWL-SDK の関数 `OS_Printf` であり、もう 1 つは mcs の文字列出力関数で行う方法で行う方法です。それぞれに利点と欠点があるため状況に応じて使い分けるようにしてください。

4.1 関数 `OS_Printf` による出力

関数 `OS_Printf` による出力では、接続しているデバイスが IS-NITRO-EMULATOR の場合のみ mcs のコンソールに表示されます。接続しているデバイスが IS-NITRO-UIC の場合や ensata の場合は mcs サーバのコンソールに表示されません。

この方法の利点としては、IS-NITRO-DEBUGGER のように他の `OS_Printf` による出力に対応しているアプリケーション等と同じ方法で文字列の出力が可能であることです。

4.2 mcs の文字列出力関数による出力

mcs の文字列出力機能の場合は、mcs の通信が確立されていれば、接続しているデバイスによらず文字列の出力が可能です。ただし、出力は mcs サーバのコンソールに限定されます。

次に mcs の文字列出力関数の使い方を説明します。

4.2.1 文字列出力ライブラリの初期化

文字列出力機能を利用するには、最初に関数 `NNS_McsInitPrint` を呼び出して初期化しておく必要があります。ただし、`NNS_McsInitPrint` を呼び出す前に、関数 `NNS_McsInit` にて mcs ライブラリの初期化が行われている必要があります。

コード 4-1 文字列出力ライブラリの初期化

```
NNS_McsInit();           // mcsライブラリの初期化
...
NNS_McsInitPrint ();     // 文字列出力機能の初期化
```

4.2.2 文字列の出力

単純に文字列を出力する場合は関数 `NNS_McsPutString` を使用します。書式指定による文字列出力を行う場合は関数 `NNS_McsPrintf` を使用します。

コード 4-2 文字列の出力

```
u32 val = 16;

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

5 mcsサーバについて

mcs サーバは、複数の PC 上の Windows アプリケーションと TWL またはニンテンドーDS 用(実機用)プログラムが同時に通信を行うことが出来るようにするために、双方の間に入って通信の橋渡しを行うプログラムです。また、実機用プログラムから PC 上のファイルにアクセスする機能と mcs サーバのコンソールに文字列を出力する機能も提供します。

5.1 一般的な操作の流れ

5.1.1 通信するハードウェアの選択

Windows アプリケーションと通信する TWL またはニンテンドーDS 用プログラムが動作するハードウェアを選択します。

- IS-NITRO-EMULATOR または IS-NITRO-UIC と通信する場合は、[デバイス]メニューの[Nitro]を選択します。
- IS-TWL-DEBUGGER ハードウェアと通信する場合は、[デバイス]メニューの[Twl]を選択します。
- ensata と通信する場合は、[デバイス]メニューの[ensata]を選択します。

5.1.2 接続

Windows アプリケーションと TWL またはニンテンドーDS 用(実機用)プログラムとの通信や、実機用プログラムから PC 上のファイルのアクセス、mcs サーバのコンソールへの文字列出力の機能を行うには、最初に実機用プログラムが動作するハードウェアに接続する必要があります。[デバイス]メニューの[接続]を選択することでハードウェアに接続します。

[デバイス]メニューで[ensata]が選択されている場合は、[デバイス]メニューの[接続]を選択した際に ensata が起動します。

IS-NITRO-EMULATOR、IS-NITRO-UIC の両方が PC 上に接続されている場合は、IS-NITRO-UIC に接続します。また、同じ種類のデバイスが複数存在する場合は、最初に見つかったデバイスに接続します。

5.1.3 ROM ファイルの読み込み (IS-NITRO-EMULATOR または IS-TWL-DEBUGGERハードウェアの場合)

IS-NITRO-EMULATOR または IS-TWL-DEBUGGER ハードウェアと接続している場合は、接続後 ROM ファイルを読み込みます。[ファイル]メニューの[開く]を選択します。ファイルダイアログが表示されますので、読み込みたいファイルを選択してください。ファイルを読み込んだ後、しばらくして実機用プログラムが起動します。

IS-NITRO-UIC と接続している場合は、ROM ファイルを読み込むことは出来ません。

5.1.4 切断

通信を終了するには、[デバイス]メニューの[切断]を選択します。

5.1.5 リセット (IS-NITRO-EMULATOR または IS-TWL-DEBUGGERハードウェアの場合)

接続しているデバイスが IS-NITRO-EMULATOR または IS-TWL-DEBUGGER ハードウェアの場合は、リセットを行うことが出来ます。リセットを行うには [デバイス]メニューの[リセット]を選択します。

IS-NITRO-UIC と接続している場合は、リセットを行うことは出来ません。

5.2 特別な場合

5.2.1 共有モードと専有モード

mcs サーバには専有モードと共有モードという 2 つの状態があります。[リソース]メニューの[共有モード]にチェックが入っていない状態だと専有モード、チェックが入っていると共有モードとなります。専有モードは TWL またはニンテンドー DS 用プログラムと同時に通信する Windows アプリケーションを 1 つに限定したいときを想定しています。具体的な挙動は次のようになります。

- チャンネル値を 16 進数で見たときに、上位 12bit をグループ値と考えます。そして最初に接続されたチャンネル値と同一のグループ値を持つチャンネルのみが接続することができ、同一グループ外のチャンネルは接続を拒否されます。

共有モードではこのような制限はありません。

5.2.2 コマンドラインオプション

mcs サーバには起動時にパラメータを与えることが出来ます。スイッチの大文字・小文字の違いは無視されます。

mcsserv [/U] [/N または /T または /E] [/D] [/A] [ROMファイル名]

/U	起動時後デバイスに接続します。ROMファイルが指定されていると無効になります。
/N	接続するデバイスをIS-NITRO-EMULATORまたはIS-NITRO-UICにします。
/T	接続するデバイスをIS-TWL-DEBUGGERハードウェアにします。
/E	接続するデバイスをensataにします。
/D	IS-NITRO-EMULATORのDSカード差込口の電源をONにします。IS-NITRO-EMULATORと接続した場合に有効です。
/A	IS-NITRO-EMULATORのGBAカートリッジ差込口の電源をONにします。IS-NITRO-EMULATORと接続した場合に有効です。

ROMファイル名 起動後に接続し、指定されたファイルを読み込みます。IS-NITRO-EMULATORまたはIS-TWL-DEBUGGERハードウェアと接続した場合に有効です。

5.2.3 IS-NITRO-EMULATORのDSカード差込口/GBAカートリッジ差込口の電源ONについて

コマンドラインオプションで"/D"を指定すると、IS-NITRO-EMULATOR と接続するときに DS カード差込口の電源を ON にします。DS カード差込口に対応したハードウェアを同時に利用することが可能になります。

また、コマンドラインオプションで"/A"を指定すると、IS-NITRO-EMULATOR と接続するときに GBA カートリッジ差込口の電源を ON にします。GBA カートリッジ差込口に対応したハードウェアを同時に利用することが可能になります。

電源を ON にしている間にスロットに挿入したり取り外したりしないでください。ハードが破損する恐れがあります。

5.2.4 TWLまたはニンテンドーDSからのデータを取得する間隔について

mcs サーバは、TWL またはニンテンドーDS 用プログラムが動作するハードウェアと接続している間は、これらのハードウェアから Windows アプリケーションに送られるデータが無いかどうかをある一定の時間間隔で確認しています。この時間間隔はオプションダイアログにて変更することができます。例えば、ニンテンドーDS から Windows アプリケーションに対して大量のデータを送信することで、ニンテンドーDS 用プログラムの動作が遅くなったりする場合は、この時間間隔を短くすることで改善される場合があります。ただし、時間間隔を短くしたするとその分 Windows 側の処理の負担が増えます。

Microsoft、Windows は米国 Microsoft Corporation の米国およびその他の国における登録商標または商標です。

その他、記載されている会社名、製品名等は、各社の登録商標または商標です。

© 2005–2009 Nintendo

任天堂株式会社の許諾を得ることなく、本書に記載されている内容の一部あるいは全部を無断で複製・複写・転写・頒布・貸与することを禁じます。