

ワイヤレス通信チュートリアル

Ver 1.1.1

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	ワイヤレスマネージャの概要	5
1.1	ワイヤレスマネージャの位置付け	5
1.2	MP通信プロトコル	5
1.3	データシェアリング	6
2	ワイヤレスマネージャの動作	7
2.1	ワイヤレスマネージャの構成	7
2.2	内部状態の遷移	8
3	dataShare-Modelの実装	9
3.1	初期化	10
3.2	接続処理	12
3.2.1	親機モードでの接続	12
3.2.2	子機モードでの接続	13
3.3	同期処理	15
3.4	切断と終了処理	18
4	WHライブラリ	19
4.1	関数リファレンス(初期化・終了・リセット)	20
4.1.1	WH_Initialize関数	20
4.1.2	WH_Finalize関数	20
4.1.3	WH_Reset関数	20
4.2	関数リファレンス(接続)	21
4.2.1	WH_ParentConnect関数	21
4.2.2	WH_ChildConnect関数	21
4.3	関数リファレンス(MP通信)	22
4.3.1	WH_SetReceiver関数	22
4.3.2	WH_SendData関数	22
4.4	関数リファレンス(データシェアリング)	23
4.4.1	WH_StepDS関数	23
4.4.2	WH_GetSharedDataAdr関数	23
4.5	関数リファレンス(キーシェアリング)	23
4.5.1	WH_GetKeySet関数	23
4.6	関数リファレンス(状態取得)	24
4.6.1	WH_GetAllowedChannel関数	24
4.6.2	WH_GetConnectMode関数	24
4.6.3	WH_GetBitmap関数	24
4.6.4	WH_GetSystemState関数	25
4.6.5	WH_GetLastError関数	25
5	付録	26
5.1	WH_StateInXXXX,WH_StateOutXXXX関数	26

5.1.1	WM関数対応表(親機／子機共通関数)	26
5.1.2	WM関数対応表(親機用関数)	27
5.1.3	WM関数対応表(子機用関数)	28

表

表 5-1	WM関数対応表(親機／子機共通関数)	26
表 5-2	WM関数対応表(親機用関数)	27
表 5-3	WM関数対応表(子機用関数)	28

図

図 2-1	無線通信の内部状態遷移図	8
-------	--------------------	---

改訂履歴

版	改訂日	改 訂 内 容	担当者
1.1.1	2008-09-16	全般 TWL 追加に伴う修正(ハードウェアおよび SDK の名称変更)	岡島
1.1.0	2005-11-21	3 現状の dataShare-Model に内容が添うように更新 4.1 WH の仕様変更に伴う修正	安達
1.0.1	2005-04-18	3 記述訂正 (wh_config.h に関する記述とサンプルソースの説明を追加) 3.1 記述修正 (wh_config.h の設定に関する項を追加 内部での動的メモリ確保に関する記述を削除)	吉崎
1.0.0	2004-11-24	初版	照井

1 ワイヤレスマネージャの概要

1.1 ワイヤレスマネージャの位置付け

ワイヤレスマネージャ(WM)は、NITRO 互換無線ハードウェアとアプリケーションの中間で、情報を受け取ってハードウェアと直接やりとりする、比較的低レベルな部分を受け持っているライブラリです。

ワイヤレスマネージャライブラリの内容は主としてゲームに特化した無線通信方法の実装で、MP 通信プロトコルと呼ばれる独自のプロトコルを提供し、そのプロトコル上で動作するデータシェアリング機能などの枠組みもライブラリ化して含んでいます。

本文書では、このワイヤレスマネージャを使用するにあたって必要な基礎知識の解説を行い、サンプルプログラムを例にとって実際のアプリケーションでの実装についても解説を加えていきます。

1.2 MP通信プロトコル

プログラミングマニュアルにあるように、NITRO 及び TWL の無線機能を使用するやり方は目的別に3つあり、各々「インフラストラクチャモード」「DS ワイヤレスプレイモード」「DS ダウンロードプレイモード」と呼ばれていますが、このチュートリアルでは「DS ワイヤレスプレイモード」のみを扱っています。

「DS ワイヤレスプレイモード」というのは、接続される各機がゲームカードを差した状態で無線による通信を行うもので、「DS ダウンロードプレイモード」はどれか1つのマシンがゲームカードを差し、他のマシンはそこからプログラムをダウンロードして動作するものです。「インフラストラクチャモード」はインターネットを利用した通信を行うモードを指します。

なお、「DS ダウンロードプレイモード」にてダウンロードしたプログラムが起動後に改めて無線通信を行う場合は、「DS ワイヤレスプレイモード」もしくは「インフラストラクチャモード」にて通信を行うことになります。

DS ワイヤレスプレイモードで通常使用されるプロトコルは「MP (Multi Poll) 通信プロトコル」と呼ばれています。このプロトコルは、多くの通信ゲームアプリケーションにおいて一般的に必要とされる「複数台のマシンでリアルタイムにデータを送受信する」機能を提供します。

MP 通信プロトコルの通信方式では、

- (1) 親機から全ての子機へデータを配信する(ブロードキャスト)
- (2) 全ての子機が親機に反応を返す
- (3) 親機が通信サイクルの終了を通知する(ブロードキャスト)

というステップを一つのサイクルとして通信を行います。

各子機が直接通信を行う相手は親機のみであり、他の子機とは直接通信しないという点に注意して下さい。また、リアルタイム性を重視していますので、1 ピクチャーフレーム(1/60 秒)に 1～数サイクルの通信を行うことができる代わりに 1 回のサイクルで送受信可能なデータ量は比較的小さい、という点も特徴の一つです。

1.3 データシェアリング

データシェアリングとは、多くのゲームアプリケーションで頻繁に利用されると考えられる「ひとかたまりのデータを全ての通信しているマシン上でリアルタイムに共有する」という手法を MP 通信プロトコル上で実現させる為の通信手法です。親機が各子機からデータを集め、それをひとまとめの「共有データ」として全ての子機へ配る、という形で実現されます。

これを前掲の囲みと同じようにまとめると、

- | |
|--|
| <ul style="list-style-type: none">(1) 親機から全ての子機へ共有データを配信する(2) 各子機が自分に固有の情報を親機に向け送信する(3) 親機は返ってきた情報を、次の送信に備えて共有データとしてまとめる |
|--|

となります。

各子機が取得する共有データは、1 サイクル前に親機が各子機から収集したデータとなる点に注意して下さい。

共有データとして「各機のキーデータ」を扱うのが「キーシェアリング」です。

※「データシェアリング」は「MP 通信プロトコル」の一つの応用例であり、「キーシェアリング」は「データシェアリング」の使い方の一例で、この3つの言葉は同一のレベルで語られるべきものではありませんが、都合上、マニュアルやサンプルプログラムなどで並列に記述されることがあります。混同しないように注意して下さい。このチュートリアルで主に取り上げるのは、この「データシェアリング」です。

2 ワイヤレスマネージャの動作

2.1 ワイヤレスマネージャの構成

NITRO 及び TWL では無線通信ユニットは ARM7 バスに接続しています(プログラミングマニュアルのハードウェアブロック図を参照)。つまり、サブプロセッサ (ARM7) の制御下にあります。

従って、通常のゲームでメインプロセッサ (ARM9) から通信機能を制御するにはサブプロセッサを通す必要があり、このため WM 関連 API の多くは ARM7 への要求を FIFO に流すための非同期関数として実装されています。要求の結果はこれも FIFO 経由で送られて来るので、メイン側ではそれを受け取り、それによってユーザが登録しておいたコールバック関数が呼び出される、という形で、結果が取得出来るようになっています。

このチュートリアルで扱っているサンプルプログラムでは、この「要求を発行・結果をコールバックで受ける」動作を1セットとして扱っており、連続した処理を行う場合は、基本的に

- (1) 要求を ARM7 へ送るための関数 A を呼びます(この時設定するコールバックを A' とします)。
- (2) A' が呼び出され、処理が完了したことが通知されます。A' は次の要求を送る為の関数 B を呼びます。
- (3) B を呼ぶときに設定した B' が呼び出され、B' は C を呼びます。
- (4) (以下同様に続いています)

のような実装をしています (A と A'、B と B' がそれぞれセットになっている、ということです)。

2.2 内部状態の遷移

無線通信を制御している ARM7 側は、いくつかの内部状態間を行き来する状態マシンになっています。この内部状態のうち主だったものを以下に図示します。

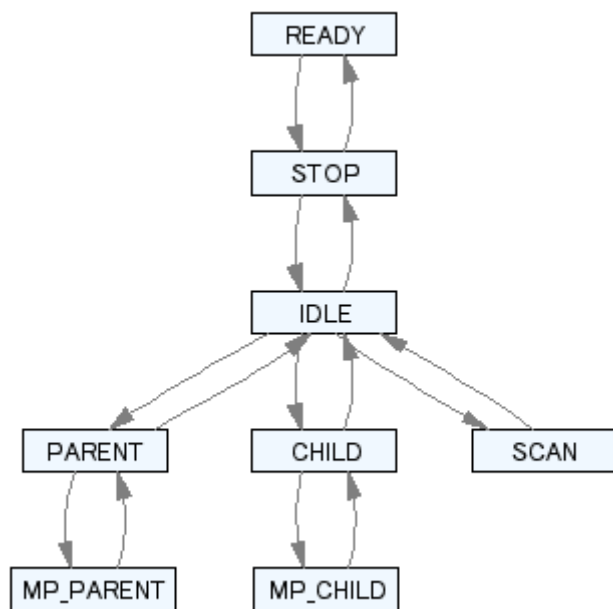


図 2-1 無線通信の内部状態遷移図

ここに掲載しているのはこの解説に必要な状態だけです。より詳細な内部状態の遷移図は、関数リファレンスマニュアルにあります。

上図の矢印で示している遷移には、各々それに対応した関数があり、それらを順番に呼び出すことで処理を進められます。一部の例外を除き、ある状態からは矢印で接続した隣の状態にしか遷移出来ません（例えば IDLE 状態から一足飛びに MP_CHILD 状態へ遷移する、というようなことは出来ません）。

3 dataShare-Modelの実装

これで、サンプルの実装を理解するのに必要な基礎知識はほぼ揃いました。以降では、サンプルの実装に関して解説を加えていきます。

このサンプルプログラムでは WM ライブラリのラッパーライブラリである `wh.h`、`wh_config.h`、および `wh.c` (以降、WH ライブラリと呼びます) を利用してデータシェアリングを実装しています。

WH ライブラリには通常の無線通信プログラムの実装に必要な関数等がまとめられています。

(WH ライブラリはサンプルソースコードとして `$TwlSDK/build/demos/wireless_shared/wh` 以下に収録されています)

この章では以下のようなサンプルプログラムの処理に沿って、WH ライブラリの使用方法を解説します。

1. 初期化
2. エラー! 参照元が見つかりません。
3. 同期処理
4. 切断と終了処理

3.1 初期化

WH ライブラリを使用してデータシェアリングを実現するために必要な手順について説明します。

まず、wh_config.h に記述された各種のワイヤレス設定値をアプリケーション仕様に合わせて変更します。

ワイヤレス通信は「ポート」と呼ばれる仮想的な通信路を設けており、アプリケーションがデータシェアリング以外にも MP 通信でデータを扱う場合、その通信に使用するポート番号とデータシェアリング用のポート番号は重複してはいけません。

また、データシェアリングの最大送受信サイズは接続子機台数に応じて制限があるので、サイズの大きなデータでデータシェアリングを実現させるにはそれらの値を適宜変更する必要があります。

WM ライブラリが使用する DMA チャンネルもここで設定されています。FS ライブラリや GX ライブラリなど、アプリケーションの他の処理で使用させている DMA チャンネルと競合しないように注意して設定値を変更してください。

```
// 無線で使用するDMA番号
#define WH_DMA_NO                2

// 子機最大数（親機を含まない数）
#define WH_CHILD_MAX             15

// シェア出来るデータの最大サイズ
#define WH_DS_DATA_SIZE         12

// 1回の通信で送れるデータの最大サイズ
// データシェアリングに加えて通常の通信をする場合は、その分だけ
// この値を増やしてください。その際は、複数パケット送信による追加の
// ヘッダフッタ分を加算する必要があります。
// 詳しくは docs/TechnicalNotes/WirelessManager.doc を参照してください。
// GUIDELINE : ガイドライン準拠ポイント(6.3.2)
// リファレンスのワイヤレスマネージャ(WM)→図表・情報→無線通信時間計算シート
// で計算した MP 通信1回分の所要時間が 5600 μ秒以下となることを推奨しています。
#define WH_PARENT_MAX_SIZE      (WH_DS_DATA_SIZE * (1 + WH_CHILD_MAX) + 4)
#define WH_CHILD_MAX_SIZE      (WH_DS_DATA_SIZE)

// 通常の MP 通信で使用するポート
#define WH_DATA_PORT            14

// 通常の MP 通信で使用する優先度
#define WH_DATA_PRIO            WM_PRIORITY_NORMAL

// データシェアリングで使用するポート
#define WH_DS_PORT              13
```

次に、共有するデータの型を定義します。最大接続子機数が 15 台のときにデータシェアリングで共有できる最大のデータサイズは 12 バイト(共有可能なデータサイズは最大接続子機数によって変化します)ですので、データサイズが 12 バイトを超過しないようにする必要があります。

```
typedef struct ShareData_ {
    u8  macadr[4]; // MACアドレス
    u32 count;     // フレーム数
    u16 level;     // 電波受信強度
    s16 data;      // グラフ表示用情報
}ShareData;
```

次に、プログラム側で送受信の共有データ領域を確保する必要があります。受信用のバッファには少なくとも、共有データサイズ×(最大接続子機数+1)バイト以上の領域が必要となります。

```
Static u8      sSendBuf[256] ATTRIBUTE_ALIGN(32);  
static u8      sRecvBuf[256] ATTRIBUTE_ALIGN(32);
```

次に、Vブランク割り込みを設定します。ここで設定したVブランク割り込みは「3.3 同期処理」のために必要となります。

```
// 割り込み設定  
OS_SetIrqFunction( OS_IE_V_BLANK , VBlankIntr );  
(void)OS_EnableIrqMask( OS_IE_V_BLANK );  
(void)GX_VBlankIntr( TRUE );  
(void)OS_EnableIrq();  
(void)OS_EnableInterrupts();
```

プログラム側で必要な準備が整いましたので、WH_Initialize 関数を使用して無線通信の初期化を行います。

WH_Initialize 関数は無線通信に必要な送受信データバッファの確保や無線ハードウェアの初期化等に必要な処理をすべて行います。プログラム側で細かな設定を行いたい場合などを除いては WH_Initialize 関数の使用をお勧めします。

3.2 接続処理

サンプルプログラムでは WH_Initialize 関数の終了後に入るメインループ内で WH_GetSystemState 関数により得られた無線通信の状態と、メニュー選択により変更されるステート変数 sSysMode を参照することで各処理へと分岐しています。

サンプルプログラム中の該当箇所を引用します。

```
switch (whstate)
{
case WH_SYSSTATE_ERROR:
    // エラー発生時は WH 状態が優先。
    changeSysMode(SYSMODE_ERROR);
    break;

case WH_SYSSTATE_MEASURECHANNEL:
    {
        ul6 channel = WH_GetMeasureChannel();
        sTgid++;
        (void)WH_ParentConnect(WH_CONNECTMODE_DS_PARENT, sTgid, channel);
    }
    break;

default:
    break;
}

PR_ClearScreen(&sInfoScreen);

// 負荷実験。
forceSpinWait();

switch (sSysMode)
{
case SYSMODE_SELECT_ROLE:
    // 役割（親機・子機）選択画面。
    ModeSelectRole();
    break;

case SYSMODE_SELECT_CHANNEL:
    // チャンネル選択画面。
    ModeSelectChannel();
    break;

case SYSMODE_LOBBY:
    // ロビー画面。
    ModeLobby();
    break;
}
```

初期化に成功していれば、WH_Initialize 関数の終了直後は「WH_SYSSTATE_IDLE」（アイドル状態）となり、sSysMode の初期値は SYSMODE_SELECT_ROLE です。

まず、最初に呼び出されるルーチンは ModeSelectRole となります。ModeSelectRole ルーチンで「Start (Parent mode)」を選択した場合は親機モードの接続処理、「Start (Child mode)」を選択した場合は子機モードでの接続処理がそれぞれ行われます。

3.2.1 親機モードでの接続

親機は通信を開始する前に使用するチャンネルを選択する必要があります。サンプルプログラムでは手動選択・自動選択の

二つの方法によりチャンネルを決定しています。

手動選択は、メニュー画面から `Select channel` を選ぶことにより呼び出される `ModeSelectChannel` ルーチンにより行われます。ここで通信チャンネルとして選択可能となるチャンネルの一覧は `WH_GetAllowedChannel` 関数を使用して取得しています。

自動選択では、まず `WH_StartMeasureChannel` 関数により電波の使用状況を計測します。そして電波使用状況の計測が終了した後に `WH_GetMeasureChannel` 関数を呼び出すことによって最も空いているチャンネルを取得します。電波使用状況の計測が終了したことは、`WH_GetSystemState` 関数が `WH_SYSSTATE_MEASURECHANNEL` を返したかどうかで判別することができます。

その後、データシェアリングの親機モードで接続を開始するには、第一引数に「`WH_CONNECTMODE_DS_PARENT`」を設定して `WH_ParentConnect` 関数を呼び出します。第三引数には選択したチャンネルを設定します。

```
switch (sRoleMenuWindow.selected)
{
case 0:
    if (sForcedChannel == 0)
    {
        // 電波使用率から最適なチャンネルを取得して接続する。
        (void)WH_StartMeasureChannel();

    }
    else
    {
        sTgid++;

        // エントリー受付状態にuserGameInfoを更新する
        updateGameInfo(TRUE);

        // キャッシュされた親機情報を破棄
        MI_CpuClear8(sBssDesc, sizeof(sBssDesc));

        // 手動で選択したチャンネルを使用して接続開始。
        (void)WH_ParentConnect(WH_CONNECTMODE_DS_PARENT, sTgid, sForcedChannel);
    }
    changeSysMode(SYSMODE_LOBBY);
    break;
```

3.2.2 子機モードでの接続

データシェアリングの子機モードで接続を開始するには、親機をスキャンし接続する親機を決定する必要があります。

サンプルプログラムでは `ModeSelectRole` ルーチンで `WH_StartScan` 関数を呼び出すことにより親機スキャンを開始しています。

```

case 1:
{
    // 親機を検索しに行く。
    static const u8 ANY_PARENT[6] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
    enum
    { ALL_CHANNEL = 0 };

    initWindow(&sSelectParentWindow);
    setupWindow(&sSelectParentWindow, 16, 16, WIN_FLAG_SELECTABLE, 8*2, 8,
16);
    (void)WH_StartScan(scanCallback, ANY_PARENT, ALL_CHANNEL);
    changeSysMode(SYSMODE_SCAN_PARENT);
}
break;

```

WH_StartScan 関数の第一引数に指定されるコールバックは、スキャンにより親機が見つかる度に呼び出されます。サンプルプログラムで実際に指定される scanCallback ルーチンでは、見つかった親機をリストに登録する処理を行っています。第三引数には有効なチャンネルのビットマップを指定しますが、指定するチャンネルが有効であることを事前に確認する必要はありません。

親機のスキャン中に呼び出される ModeSelectParent ルーチンでは、スキャンにより見つかった親機を表示しユーザの選択を待ちます。

データシェアリングの子機モードで接続を開始するには、WH_EndScan 関数によりスキャンを終了し WH_GetSystemState 関数が WH_SYSSTATE_IDLE を返すことを確認してから、第一引数に「WH_CONNECTMODE_DS_CHILD」を設定して WH_ChildConnect 関数を呼び出します。

```

// 親機検索画面をユーザが閉じたか
if ((sSelectParentWindow.state == WIN_STATE_CLOSED))
{
    if (WH_GetSystemState() == WH_SYSSTATE_SCANNING)
    {
        // 親機スキャン中であれば一旦スキャンを終了する
        (void)WH_EndScan();
        return;
    }

    if (WH_GetSystemState() == WH_SYSSTATE_IDLE)
    {
        if (sSelectParentWindow.selected < 0)
        {
            WH_Finalize();
            changeSysMode(SYSMODE_SELECT_ROLE);
            return;
        }
        // スキャン中でなく、なおかつ親機をユーザが選択していればデータシェアリング開始
        (void)WH_ChildConnect(WH_CONNECTMODE_DS_CHILD,
                               &(sBssDesc[sSelectParentWindow.selected]));
        changeSysMode(SYSMODE_LOBBYWAIT);
    }
}

```

3.3 同期処理

データシェアリングの親機モード、子機モードともに接続が正常に完了すると `WH_GetSysState` 関数で取得した状態は「`WH_SYSSTATE_DATASHARING`」(データシェアリング)へと遷移しています。

安定した無線通信を行うためには同期処理関数 `WH_StepDS` を、そのフレームの `MP` 通信サイクルの開始より前に呼び出さなければなりません。`WM` ライブラリが `MP` 通信の準備を行う `V` カウントはデフォルトで親機が 260、子機が 240 となっており、`V` ブランク割り込み (`V` カウントは 192) の時点で呼び出した場合に最も効率が良くなるように設計されています。これを踏まえ、サンプルプログラムでは `V` ブランク割り込み開始直後(`OS_WaitVBlankIntr` 関数の直後)の `updateShareData` ルーチン内で `WH_StepDS` 関数を呼び出しています。

```
static void updateShareData(void)
{
    if (WH_GetSystemState() == WH_SYSSTATE_DATASHARING)
    {
        if (WH_StepDS(sSendBuf))
        {
            ul6 i;
            for (i = 0; i < WM_NUM_MAX_CHILD + 1; ++i)
            {
                u8 *adr;
                ShareData *sd;

                adr = (u8 *)WH_GetSharedDataAdr(i);
                sd = (ShareData *) & (sRecvBuf[i * sizeof(ShareData)]);

                if (adr != NULL)
                {
                    MI_CpuCopy8(adr, sd, sizeof(ShareData));
                    sRecvFlag[i] = TRUE;
                }
                else
                {
                    sd->level = 0;
                    sd->data = 0;
                    sRecvFlag[i] = FALSE;
                }
            }

            sNeedWait = FALSE;
        }
        else
        {
            ul6 i;
            for (i = 0; i < WM_NUM_MAX_CHILD + 1; ++i)
            {
                sRecvFlag[i] = FALSE;
            }

            sNeedWait = TRUE;
        }
    }
    else
    {
        ul6 i;
        for (i = 0; i < WM_NUM_MAX_CHILD + 1; ++i)
        {
            sRecvFlag[i] = FALSE;
        }

        sNeedWait = FALSE;
    }
}
```

同期処理によって共有されたデータは WH_GetSharedDataAdr 関数を使用して先頭アドレスを取得し、プログラム側で確保している受信データ領域にコピーします。

メインループでは WH_GetConnectMode 関数を使用してどのモードで接続したのかを判断し、親機モードであれば ModeParent ルーチンを、子機モードであれば ModeChild ルーチンをそれぞれ呼び出しています。それぞれのルーチン

では送受信の結果を表示しています。

その他にも、ガイドラインに従ってリンク強度アイコンの表示を行い、データシェアリングで共有したリンク強度をグラフ表示しています。

3.4 切断と終了処理

ユーザの操作によって親機から特定の子機を切断させたい場合には `WM_Disconnect` 関数を使用してください。一度に複数またはすべての子機を切断したい場合には `WM_DisconnectChildren` 関数を使用してください。

WH ライブラリを使用している場合には、無線通信を終了させたいときに `WH_Finalize` 関数を呼び出すことで、接続モードと現在の WH ライブラリの状態から判断した適切な終了処理を行うことができます。`WH_Finalize` 関数によって無線通信は IDLE 状態に遷移します。そこから `WM_PowerOff` 関数、`WM_Disable` 関数、`WM_Finish` 関数を順番に（またはこの3つを行う `WM_End` 関数を）呼び出すことで、無線通信ハードウェアを含めて完全に終了させることができます。

4 WHライブラリ

WH ライブラリには通常の無線通信プログラムの実装に必要な関数等がまとめられています。

この章では、WH ライブラリにまとめられた関数(サンプルプログラムでは使用されていない関数も含みます)について解説します。

4.1 関数リファレンス(初期化・終了・リセット)

4.1.1 WH_Initialize関数

構文:	<code>int WH_Initialize(void);</code>	
引数:	なし	
返り値:	TRUE	成功
	FALSE	失敗

通常、無線通信を行うために必要な、通信用送受信データバッファの確保や無線通信ハードウェアの初期化などを自動で行い、無線通信ステートをIDLEにまで遷移させます。TRUEが返された時点ではWM_Init関数が成功し、WM_Enable関数の呼び出しに成功しています。初期化処理完了のタイミングはWH_GetSystemState関数の返り値がWH_SYSSTATE_IDLEとなった時点です。

内部で呼び出される OS_Alloc 関数のためにメインメモリ上にヒープを作成する必要があります。

4.1.2 WH_Finalize関数

構文:	<code>int WH_Finalize(void);</code>	
引数:	なし	
返り値:	TRUE	成功
	FALSE	失敗

WHライブラリの状態と接続モードから判断して、適切な終了処理を呼び出します。処理終了後は無線通信ステータがIDLEに遷移します。TRUEが返された時点では終了処理のための関数の呼び出しに成功しています。処理完了のタイミングはWH_GetSystemState関数の返り値がWH_SYSSTATE_IDLEとなった時点です。

無線通信を完全に終了させるには、WM_PowerOff 関数、WM_Disable 関数、WM_Finish 関数の 3 関数もしくは、WM_End 関数を呼び出す必要があります。

4.1.3 WH_Reset関数

構文:	<code>int WH_Reset(void);</code>	
引数:	なし	
返り値:	TRUE	成功
	FALSE	失敗

接続モードなどの現在の状況を考慮せずに無線通信ステータをIDLEに遷移させます。TRUEが返された時点ではWM_Reset関数の呼び出しに成功しています。処理完了のタイミングはWH_GetSystemState関数の返り値がWH_SYSSTATE_IDLEとなった時点です。

4.2 関数リファレンス(接続)

4.2.1 WH_ParentConnect関数

構文: `BOOL WH_ParentConnect(int mode, ul6 tgid, ul6 channel);`
 引数: `mode` 接続モード
 `tgid` 親機通信tgid
 `channel` 親機通信チャンネル
 返り値: `TRUE` 成功
 `FALSE` 失敗

接続モードの定義:

```
enum {
    WH_CONNECTMODE_MP_PARENT, // 親機 MP 接続モード
    WH_CONNECTMODE_MP_CHILD,  // 子機 MP 接続モード
    WH_CONNECTMODE_KS_PARENT, // 親機 key-sharing 接続モード
    WH_CONNECTMODE_KS_CHILD,  // 子機 key-sharing 接続モード
    WH_CONNECTMODE_DS_PARENT, // 親機 data-sharing 接続モード
    WH_CONNECTMODE_DS_CHILD,  // 子機 data-sharing 接続モード
    WH_CONNECTMODE_NUM
};
```

親機モードで無線通信接続を開始します。データシェアリング、キーシェアリングへの遷移を自動で行います。`TRUE`が返された時点では接続処理のための関数の呼び出しに成功しています。処理完了のタイミングは親機／子機にかかわらず、MP接続は`WH_SYSSTATE_CONNECTED`、データシェアリングは`WH_SYSSTATE_DATASHARING`、キーシェアリングは`WH_SYSSTATE_KEYSHARING`が`WH_GetSystemState`関数の返り値として返されるようになった時点です。

4.2.2 WH_ChildConnect関数

構文: `BOOL WH_ChildConnect(int mode, WMBssDesc *bssDesc);`
 引数: `mode` 接続モード
 `bssDesc` 接続する親機のbssDesc
 返り値: `TRUE` 成功
 `FALSE` 失敗

接続モードの定義:

```
enum {
    WH_CONNECTMODE_MP_PARENT, // 親機 MP 接続モード
    WH_CONNECTMODE_MP_CHILD,  // 子機 MP 接続モード
    WH_CONNECTMODE_KS_PARENT, // 親機 key-sharing 接続モード
    WH_CONNECTMODE_KS_CHILD,  // 子機 key-sharing 接続モード
    WH_CONNECTMODE_DS_PARENT, // 親機 data-sharing 接続モード
    WH_CONNECTMODE_DS_CHILD,  // 子機 data-sharing 接続モード
    WH_CONNECTMODE_NUM
};
```

子機モードで無線通信接続を開始します。

4.3 関数リファレンス(MP通信)

4.3.1 WH_SetReceiver関数

構文: void WH_SetReceiver(WHReceiver proc);
 引数: proc WHReceiver型のコールバック関数
 戻り値: なし

WHReceiver型の定義:

```
typedef void (*WHReceiver)(ul6 aid, ul6* data, ul6 size);
```

MP 通信のデータ受信コールバック関数を設定します。データシェアリング、キーシェアリングのときには設定する必要はありません。

コールバック関数には送信元の aid、受信データ、受信データサイズが渡されます。

4.3.2 WH_SendData関数

構文: int WH_SendData(
 void *data, ul6 datasize, WHSendCallbackFunc callback);
 引数: data 送信するデータの先頭アドレス
 datasize 送信するデータのサイズ
 callback WHSendCallbackFunc型のコールバック関数
 戻り値: TRUE 成功
 FALSE 失敗

WHSendCallbackFunc型の定義:

```
typedef void (*WHSendCallbackFunc)(BOOL result);
```

MP 通信のデータ送信を開始します。データシェアリング、キーシェアリングのときには呼び出す必要はありません。TRUE が返された時点では WM_SetMPDataToPortEx 関数の呼び出しに成功しています。処理完了のタイミングはコールバック関数が呼び出された時点です。

コールバック関数には送信結果が渡されます。コールバック関数が呼び出されるまでは送信データバッファの内容を変更してはいけません。

4.4 関数リファレンス(データシェアリング)

4.4.1 WH_StepDS関数

構文:	<code>int WH_StepDS(void *data);</code>
引数:	<code>data</code> 送信するデータの先頭アドレス
戻り値:	TRUE 成功 FALSE 失敗

データシェアリングの同期をひとつ進めます。TRUEが返された時点で処理は完了しています。共有したデータを取得するにはWH_GetSharedDataAdr関数を使用してください。

安定した無線通信を行うためにはそのフレームの MP 通信サイクルの開始より前に呼び出さなければなりません。V ブランク割り込み開始直後に呼び出すことを推奨します。

4.4.2 WH_GetSharedDataAdr関数

構文:	<code>u16 *WH_GetSharedDataAdr(u16 aid);</code>
引数:	<code>aid</code> 共有データを取得したい子機のaid
戻り値:	指定した子機の共有データの先頭アドレス 失敗時にはNULLが返されます

データシェアリングの共有データを子機指定で取得したい場合に呼び出します。

4.5 関数リファレンス(キーシェアリング)

4.5.1 WH_GetKeySet関数

構文:	<code>int WH_GetKeySet(WMKeySet *keyset);</code>
引数:	<code>keyset</code> 共有キーデータを格納するバッファへのポインタ
戻り値:	TRUE 成功 FALSE 失敗

キーシェアリングによって共有したキーデータをバッファに格納します。TRUE が返された時点で処理は完了しています。

安定した無線通信を行うためにはそのフレームの MP 通信サイクルの開始より前に呼び出さなければなりません。V ブランク割り込み開始直後に呼び出すことを推奨します。

4.6 関数リファレンス(状態取得)

4.6.1 WH_GetAllowedChannel関数

構文: u16 WH_GetAllowedChannel(void);
引数: なし
戻り値: 使用を許可されている通信チャンネルのビットパターン

内部で WM_GetAllowedChannel 関数を呼び出しています。

4.6.2 WH_GetConnectMode関数

構文: int WH_GetConnectMode(void);
引数: なし
戻り値: 設定されている接続モード

WH_ChildConnect関数で引数に設定した接続モードを返します。WH_ChildConnect関数が呼び出されるまでの戻り値は不定です。次にWH_ChildConnect関数が呼び出されるまでは以前に設定されていた接続モードが返されます。

4.6.3 WH_GetBitmap関数

構文: u16 WH_GetBitmap(void);
引数: なし
戻り値: 接続されている端末を示すビットパターン

接続されている端末に対応するビットが 1 に設定されています。最下位ビットが親機(aid=0)、最上位ビットが 15 番目の子機(aid=15)に対応しています。

4.6.4 WH_GetSystemState関数

構文: int WH_GetSystemState(void);
引数: なし
戻り値: WHライブラリの内部状態

WHライブラリの内部状態の定義:

```
enum {  
    WH_SYSSTATE_STOP,           // 初期状態  
    WH_SYSSTATE_IDLE,          // 待機中  
    WH_SYSSTATE_SCANNING,      // スキャン中  
    WH_SYSSTATE_BUSY,          // 接続作業中  
    WH_SYSSTATE_CONNECTED,     // 接続完了 (この状態で通信可能)  
    WH_SYSSTATE_DATASHARING,   // data-sharing有効で接続完了  
    WH_SYSSTATE_KEYSHARING,    // key-sharing有効で接続完了  
    WH_SYSSTATE_ERROR,         // エラー発生  
    WH_SYSSTATE_NUM  
};
```

WHライブラリの現在の内部状態を取得します。

4.6.5 WH_GetLastError関数

構文: int WH_GetLastError(void);
引数: なし
戻り値: エラーコード

エラーコードの定義:

```
enum {  
    // 自前のエラーコード  
    WH_ERRCODE_DISCONNECTED = WM_ERRCODE_MAX, // 親から切断された  
    WH_ERRCODE_PARENT_NOT_FOUND,              // 親がいない  
    WH_ERRCODE_NO_RADIO,                       // 無線使用不可  
    WH_ERRCODE_LOST_PARENT,                   // 親を見失った  
    WH_ERRCODE_MAX  
};
```

直前に起こったエラーの詳細を取得します。

5 付録

5.1 WH_StateInXXXX,WH_StateOutXXXX関数

「2.1 ワイヤレスマネージャの構成」のところで述べたように、ワイヤレスマネージャのAPIは要求送信関数(を呼び出す関数)と通知を受け取るコールバック関数の組み合わせで無線通信を行っています。

WH ライブラリでは内部関数として、要求送信関数には「WH_StateInXXXX」、コールバック関数には「WH_StateOutXXXX」という名前がつけられています。

5.1.1 WM関数対応表(親機／子機共通関数)

WH ライブラリ関数名	対応するワイヤレスマネージャの関数
WH_StateInInitialize()	WM_Init()
WH_StateInEnable() WH_StateOutEnable()	WM_Enable()
WH_StateInPowerOn() WH_StateOutPowerOn()	WM_PowerOn()
WH_StateInReset() WH_StateOutReset()	WM_Reset()
WH_StateInSetMPData() WH_StateOutSetMPData()	WM_SetMPDataToPortEx()
WH_StateInPowerOff() WH_StateOutPowerOff()	WM_PowerOff()
WH_StateInDisable() WH_StateOutDisable()	WM_Disable()

表 5-1WM 関数対応表(親機／子機共通関数)

5.1.2 WM関数対応表(親機用関数)

WH ライブラリ関数名	対応するワイヤレスマネージャの関数
WH_StateInMeasureChannel() WH_NextMeasureChannel() WH_StateOutMeasureChannel()	WM_GetAllowedChannel() WM_MeasureChannel()
WH_StateInSetParentParam() WH_StateOutSetParentParam()	WM_SetParentParameter()
WH_StateInStartParent() WH_StateOutStartParent()	WM_StartParent()
WH_StateInStartParentMP() WH_StateOutStartParentMP()	WM_StartMP() データシェアリングモードの場合は WM_StartDataSharing()も呼び出します
WH_StateInStartParentKeyShare() WH_StateOutStartParentKeyShare()	WM_StartKeySharing()
WH_StateInEndParentKeyShare() WH_StateOutEndParentKeyShare()	WM_EndKeySharing()
WH_StateInEndParentMP() WH_StateOutEndParentMP()	WM_EndMP()
WH_StateInEndParent () WH_StateOutEndParent ()	WM_EndParent()
WH_StateInDisconnectChildren() WH_StateOutDisconnectChildren()	WM_DisconnectChildren()

表 5-2WM 関数対応表(親機用関数)

5.1.3 WM関数対応表(子機用関数)

WH ライブラリ関数名	対応するワイヤレスマネージャの関数
WH_StateInStartScan() WH_NextScan () WH_StateOutStartScan()	WM_GetAllowedChannel() WM_StartScan()
WH_StateInEndScan() WH_StateOutEndScan()	WM_EndScan()
WH_StateInStartChild() WH_StateOutStartChild()	WM_StartConnect()
WH_StateInStartChildMP() WH_StateOutStartChildMP()	WM_StartMP() データシェアリングモードの場合は WM_StartDataSharing()も呼び出します
WH_StateInStartChildKeyShare() WH_StateOutStartChildKeyShare()	WM_StartKeySharing()
WH_StateInEndChildKeyShare() WH_StateOutEndChildKeyShare()	WM_EndKeySharing()
WH_StateInEndChildMP() WH_StateOutEndChildMP()	WM_EndMP()
WH_StateInEndChild () WH_StateOutEndChild ()	WM_Disconnect()

表 5-3WM 関数対応表(子機用関数)

© 2003, 2004 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。