Nintendo Wi-Fi Connection

NITRO-DWC プログラミングマニュアル

Ver 1.4.3

任天堂株式会社発行

目次

表

図

コード

改訂履歴

版	改訂日	改 訂 内 容	承認者	担当者
1.4.3	2007-07-21	「**コード 7-3 マッチメイク候補プレイヤーの評価**」誤記を修正（s_int_key→&s_int_key）		
1.4.2	2007-02-15	「**コード 6-1 DSワイヤレス通信による友達情報の交換**」誤記を修正（s_friendData→ownFriendData）		
1.4.1	2006-08-09	「**8.3 DWC_InitFriendsMatch関数で指定するバッファサイズの目安**」と 「**表 8-2 通信データ内訳**」の誤記を修正		
1.4.0	2006-06-19	「**10.1 エラー処理**」エラーコードの表示条件を変更		
1.3.0	2006-06-06	「**2.1.3 ゲーム毎のプレイヤー情報：ログインID**」【仮ログインIDの重複例】を修正 「**3 NITRO-DWCの初期化**」使用メモリサイズを 200kbyte→230kbyteに変更 「**7.6 マッチメイク指標キーに使用できないキー名**」を追加 その他の変更（語句の統一、修正等）		
1.2.0	2006-03-10	「**2 NITRO-DWCにおけるユーザー管理**」を追加 「**7.5 マッチメイクの高速化**」を追加 「**8.5 データ送受信量の目安**」を追加 その他の変更（段落の見直し、語句の変更等）		
1.1.0	2006-01-30	「**コード 6-3 友達リストの同期処理**」更新 「**コード 6-4 友達情報タイプの取得**」誤植を修正（stablished→established） 「**コード 7-3 マッチメイク候補プレイヤーの評価**」誤植を修正（"anymatch"→"anymatch_test"） 「**11ネットワークストレージ対応**」データロード関数を、新規追加関数に変更		
1.0.0	2005-12-27	初版作成。		

1 はじめに

NITRO-DWC(以下 DWC)は、Nintendo Wi-Fi Connection の「カンタン・あんしん・無料」の理念を実現するため、下記の内容を目標に構築されています。

- インターネットに接続するための煩雑な情報をユーザーから遮蔽し、ユーザーが簡単に使用できる
- インターネットに接続していない状態で、あらかじめ「ワイヤレス通信」または「友達コード交換」等で友達関係を構築しておくとインターネットに接続したときに友達と簡単に通信できる
- ユーザーが DS カードを手放し、他のユーザーの手に渡った場合、インターネットに関する情報を他のユーザーが簡単に参照できないようにセキュリティを保つ

2 NITRO-DWC におけるユーザー管理

2.1 Wi-Fi ユーザー情報の管理

Nintendo Wi-Fi Connectionの認証（以下、Wi-Fi認証）に必要な情報には、ユーザーID、プレイヤーID、パスワードがあります。これらの情報はDS本体とDSカードを対にして管理します（図 2-1）。

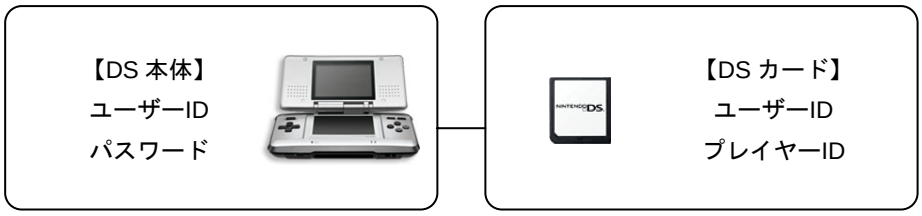

図 2-1 DS 本体と DS カードのユーザーID の保存状態

- DS 本体には、Wi-Fi 認証に使用するユーザーID とパスワードを保存します。
- DS カードには、Wi-Fi 認証に使用するユーザーID とプレイヤーID を保存します。

Nintendo Wi-Fi Connectionでは、これらの情報を使って、インターネット上のサーバに接続します。DSカードに保存されたユーザーIDがDS本体に保存されているものと違う場合、インターネット上に保存されたデータにはアクセス出来ません。こうすることで、不正なデータアクセスを防ぎます（図 2-2）。

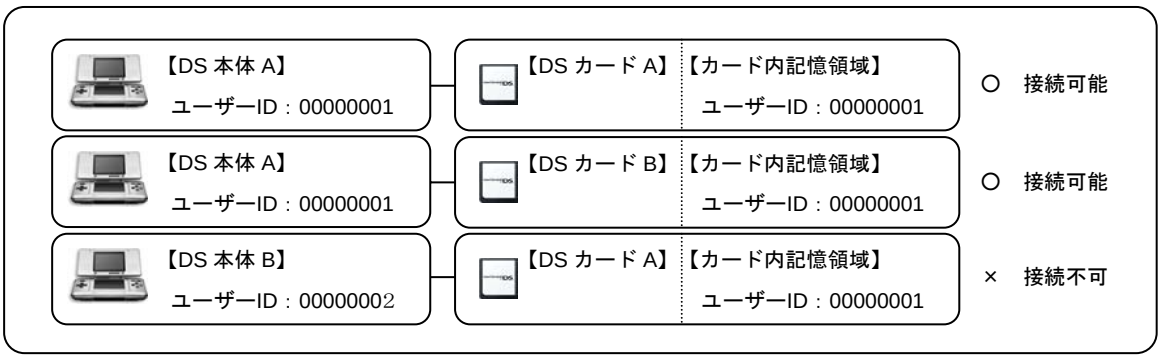

図 2-2 複数の DS 本体と複数の DS カードで使用したイメージ

2.1.1 ユーザーID とプレイヤーID

ユーザーID は、出来る限り固有になるようにインターネットに接続されていないときに生成されます。その後、インターネットに接続し、認証サーバに確認して登録されると正式なユーザーID となります。もし認証サーバに確認した結果、ユーザーID が重複していて使用できない場合は、重複していないユーザーID が割り当てられます。

※ユーザーIDをできるだけ固有にするために、本体のMACアドレスの一部を利用します。そのため、違う本体間で同じユーザーIDが生成されることはありませんが、ユーザーIDを移動させた場合[1]・ユーザーIDを再

[1] 本体に格納されているユーザー情報は、DWC で提供される「Wi-Fi コネクション設定」で移動可能です。

生成した場合などに重複の可能性があります。

プレイヤーID は、32bits の乱数です。インターネットのサーバ上のデータは「ユーザーID＋プレイヤーID＋イニシャルコード」単位で管理されますので、プレイヤーID は、同一ユーザーID 及び同一イニシャルコードに対して固有であれば問題ありません。これも、もし重複した場合は、認証時に重複していないプレイヤーID が割り当てられます。

2.1.2 ユーザーID とプレイヤーID の違い

ユーザーID は、DS 本体に対して発行するので、同じ DS 本体を使用するユーザーは、いくつものゲームで一つのユーザーID を使用することになります。

プレイヤーIDは、DSカードに対して発行するので、同じDS本体（ユーザーID）かつ同じイニシャルコードのゲームに対して、違うプレイヤーIDを使用することになります。（図 2-3）

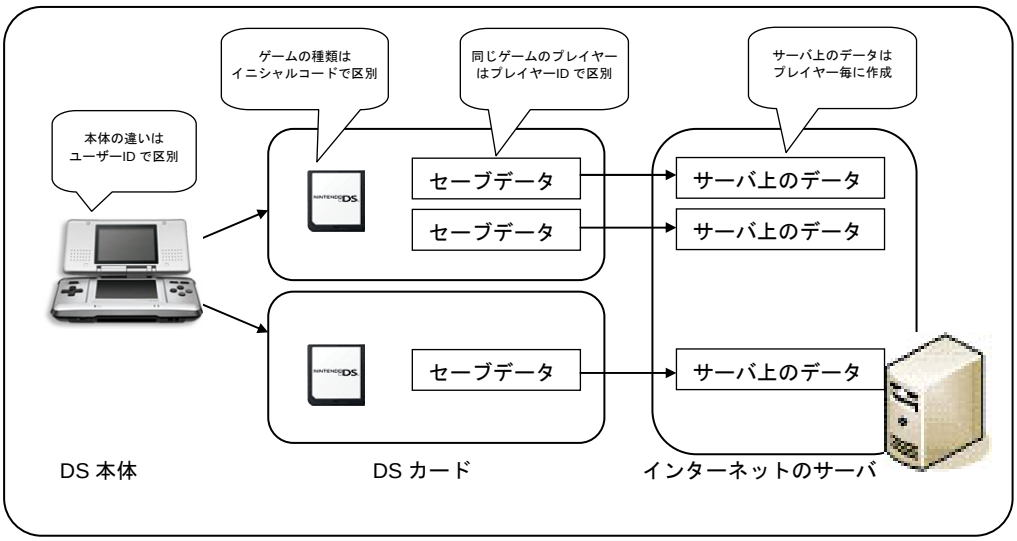

図 2-3 インターネット上のデータの持ち方の関係

2.1.3 ゲーム毎のプレイヤー情報：ログイン ID

「ユーザーID＋プレイヤーID＋イニシャルコード」をまとめたものを「**ログインID**」と呼びます（図 2-4）。また、インターネットのサーバ上に保存するユーザー情報の単位を「**プロファイル**」、プロファイルをサーバ上で管理するためのIDを「**プロファイルID**」と呼びます。

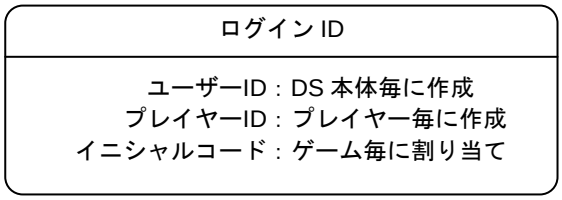

図 2-4 ログイン ID の構成

DWC 内部では、インターネットのサーバ上で他のユーザーのプロファイルを検索するために、ログイン ID またはプロファイル ID を使用します。

前述のとおり、ログイン ID はインターネットに接続していない状態で生成されます（仮ログイン ID）。高い確率でそのまま使用可能ですが、仮ログイン ID がそのまま使えない場合もあります。その場合は、重複して

いない承認されたログイン ID（認証済ログイン ID）が生成されます。認証済ログイン ID に対してプロファイル ID が 1 対 1 で割り当てられます。

【仮ログイン ID の重複例】

- 認証されていないユーザーID でログイン ID を作成した場合、既に他者がそのユーザーID を認証サーバに登録していて、かつ同じゲーム、同じプレイヤーID でログイン ID を作成していた場合。
- 複数の DS 本体で、同じ未認証ユーザーID を使って、同じゲーム、同じプレイヤーID でログイン ID を作成した場合。

2.1.4 ゲームで保存する Wi-Fi 認証用の情報

ゲームでは、これらの Wi-Fi 認証用の情報を DS カードのバックアップに保存しなければなりません。

この認証用の情報のサイズは、64 バイトです。

Wi-Fi 認証用の情報には、仮のログイン ID、認証済ログイン ID、プロファイル ID などが含まれています。この情報は、DWC で作成・更新されますので、開発者が内容の詳細を把握する必要はありません。

また、一つの DS カードで複数のプレイヤーがプレイヤーデータを保存できる場合、プレイヤーごとに Wi-Fi 認証用の情報を保存しなければなりません。

これまでの、Wi-Fi認証に関する用語は、図 2-5のようになります。

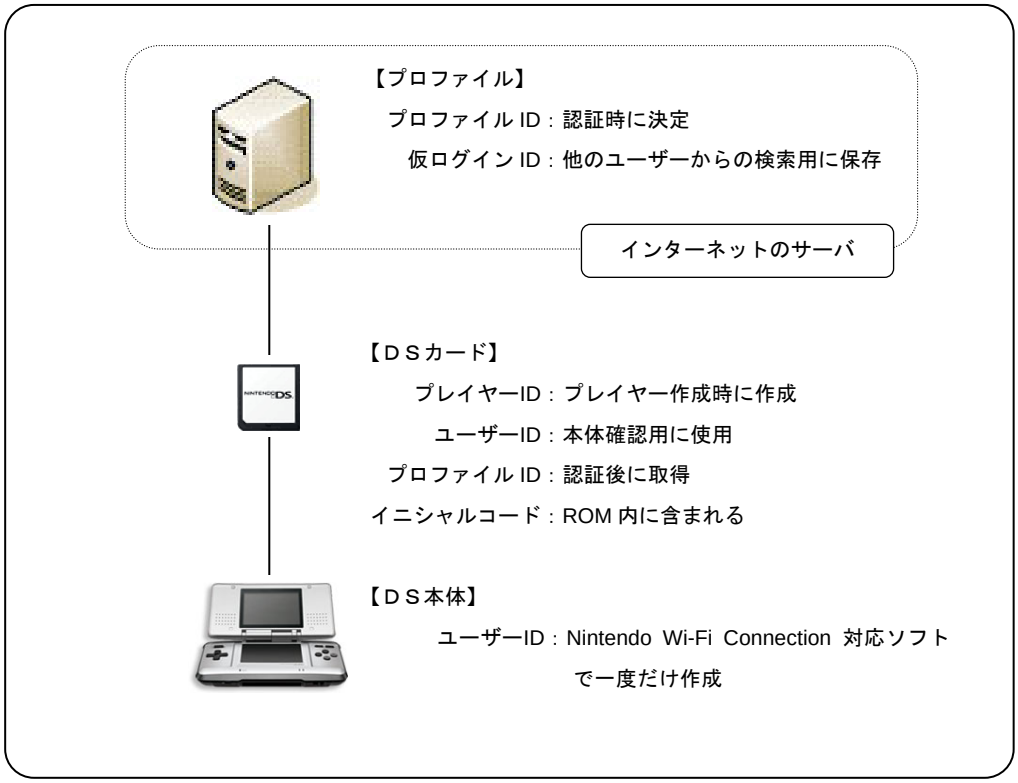

図 2-5 Wi-Fi 認証用の用語の全体図

2.2 友達管理概要

2.2.1 友達関係の構築

DWC では、友達と簡単に通信を開始できるようにするために、友達関係の構築をインターネット上のサーバで行います。友達関係の構築は、ユーザー情報を交換することで実現します。構築された友達関係は、各ユーザーのプロファイルに保存されます。

友達関係を作成するために交換するユーザー情報として、以下の方法があります。

- DS ワイヤレス通信を用いる方法
  ログイン ID またはプロファイル ID を交換します。

  一度もログインしていない場合は、ログイン ID を使用します。それぞれがローカルで作ったままの状態でも、非常に一意性が高いため、そのまま使える確率は高いのですが、確実ではありません。しかし、現時点では、$1/2^{75}$ 以下の確率であり、特別な対策は必要ありません。

  既に一度でもログインしている場合は、プロファイル ID を用います。必ず相手を特定できるため、確実に友達関係を作成することが出来ます。

- 友達コードを用いる方法
  プロファイル ID にエラーチェック用の情報を入れた「友達コード」を交換します。

  プロファイル ID を用いるためには、必ず一度はインターネットに接続する必要があります。また、入力を間違う可能性があるため、入力の確認ややり直しができるインタフェースを作成する必要があります。

これらの交換用の情報は、DWC で作成することができます。DWC には、DS カードに保存された Wi-Fi 認証用の情報から、もっとも適切と思われる情報を自動的に作成する機能があります。

2.2.2 DS ワイヤレス通信での友達関係の構築

DSワイヤレス通信を行った際に相手と情報を交換しておくことで、自動的に友達関係が結べる仕組みを提供します。交換する情報は、ユーザーデータに格納されているログインIDまたはプロファイルIDから作成します。（図 2-6）**ただし、DWCではこの情報をDSワイヤレス通信で交換することはサポートしていません。作成した情報の交換は、アプリケーション側で行ってください。**

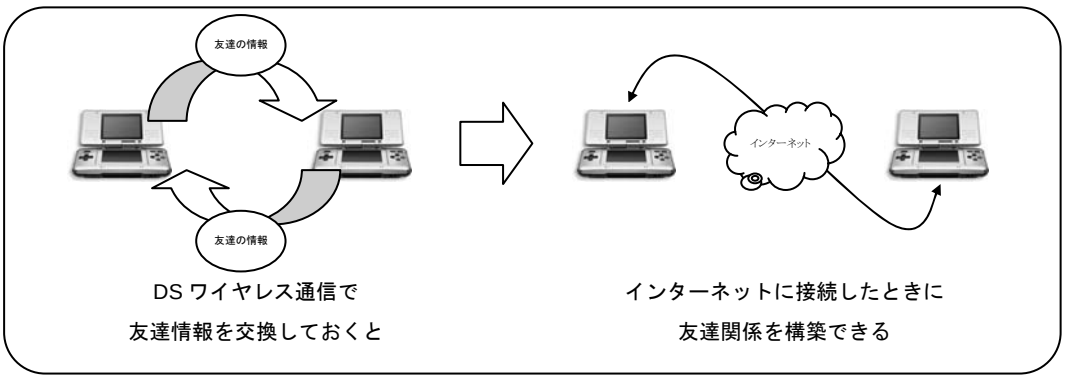

図 2-6 DS ワイヤレス通信での友達関係作成のイメージ

2.2.3 友達コードでの友達関係の構築

友達関係を結ぶための相手ユーザーが特定できる情報を「友達コード」と呼びます。

この友達コードを交換する事で、友達関係を作成できる仕組みを提供します。（図 2-7）

この友達コードは、人が扱うため不必要に長くては問題があります。そのため、ログイン ID ではなく、一度インターネットに接続することで得られるプロファイル ID を使って作成します。

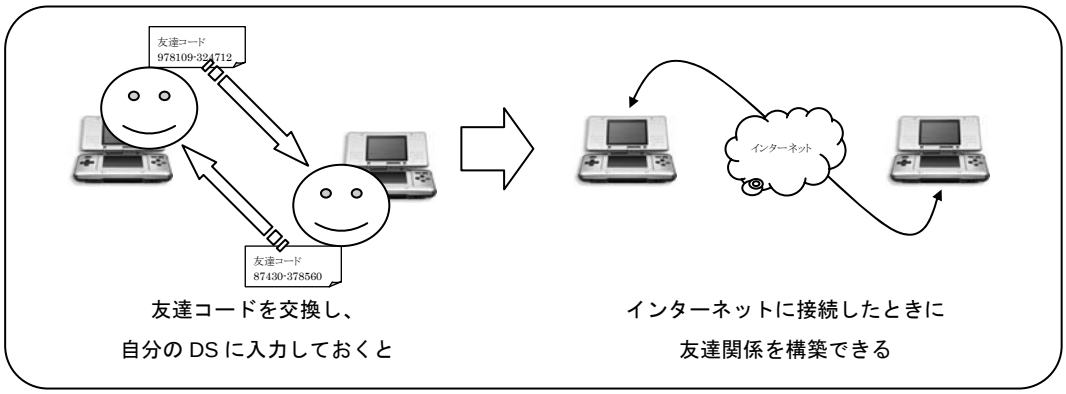

図 2-7 友達コードでの友達関係作成のイメージ

友達コードは、12 桁の数字で表現されます。ゲームでは、以下の点に注意してください。

- 友達コード発行のためのユーザーインタフェースを作成する必要があります。インターネットに一度も接続されていない場合は発行できませんので、メッセージを表示する必要があります。
- 友達コード入力のためのユーザーインタフェースを作成する必要があります。このユーザーインタフェースでは、友達コードが間違えている場合も想定して、入力されたものを保存し、何度でも修正できるようにする必要があります。

2.2.4 ゲームで保存する友達の情報

ゲームでは、交換した友達情報をゲームで友達関係として管理したい最大人数分、バックアップに保存する必要があります。これは、インターネットに接続していない状態で、ユーザーが友達関係を編集できるようにするためです。また、実際のゲームでは管理している友達に関連する情報（ニックネームや対戦戦績など）もあわせて保存する必要があります。

DWC では、ログイン ID、プロファイル ID、友達コードの種類を意識せず、友達情報として扱うようにしています。

DWC で使用する友達情報の保存には、一人につき 12 バイト必要です。

2.3 例外処理

2.3.1 DS 本体と DS カードの関連付けの消去

Nintendo Wi-Fi Connection では、セキュリティのために DS 本体と DS カードをセットにして使用します。そのため、DS 本体を買い換えたり、故障したときに、Nintendo Wi-Fi Connection に接続できなくなり、ユーザーの利便性が著しく落ちる場合があります。

この問題に対応するために DWC では、プロファイルに保存している情報を破棄することで、DS カードの関連付け情報を削除できる仕組みを提供しています。この場合、インターネット上の友達関係はすべて削除されますので、ユーザーに警告し、ユーザーが誤って情報を削除してしまわないようなインタフェースを作成してください。

もし、インターネット上の友達関係を削除した場合でも、削除したユーザーの DS カードには相手の友達情報が残っています。この情報を利用して、相手に新しい友達コードを伝えることで、友達関係を復帰させることもできます。この場合、削除したユーザーが友達にもう一度登録をしてもらうように促す必要がありますので、各アプリケーションで手順を具体的にユーザーに伝えるメッセージを加えてください。

具体的な処理としては、現在 DS カードに保存している関連付けを消去し、新たな関連付けを作成したい場合は、新規にユーザーデータを作成し、以前まで使用していたユーザーデータを破棄することで対応してください。また、ユーザーデータが更新されても、DS カードに保存している友達リストの友達関係は成立したままの状態になります。もし、友達リストを残す仕様にしていて、かつ友達関係が成立した状態であることをユーザーにわかるようにしている場合、友達情報の友達関係成立フラグをクリアするようにしてください。

※ フロー図を「Nintendo Wi-Fi Connection プログラミングガイドライン」に載せていますので、ご確認ください。

3 NITRO-DWC の初期化

DWC の初期化は、すべての DWC 関数を呼び出す前に必ず行ってください。

DWC の初期化を行う DWC_Init 関数では、以下の処理を行います。

- DS 本体に格納されるユーザー認証のための情報を生成
- DS 本体のバックアップに格納されている接続先情報などの正当性のチェックと修正

また、第 4 章以降のインターネット、Wi-Fiコネクションへの接続と、マッチメイク／友達関連処理の内部で使用するメモリ確保／解放関数を、DWC_SetMemFunc関数を使って設定して下さい（コード 3-1）。

DWC が必要とするメモリは、四人でマッチメイクを行う場合で 230KByte 程度です。

マッチメイクの最大人数を一人減らすたびに必要なメモリは 20Kbyte（DWC_InitFriendsMatch 関数の引数、sendBufSize, recvBufSize がデフォルトの 8Kbyte づつの場合）程度減ります。

```
void init_dwc( void )
{
    u8 work[ DWC_INIT_WORK_SIZE ] ATTRIBUTE_ALIGN( 32 );

    // DWC初期化
    if ( DWC_Init( work ) == DWC_INIT_RESULT_DESTROY_OTHER_SETTING )
        disp_init_warning_msg();  // 警告メッセージ表示

    // メモリ確保／解放関数を設定
    DWC_SetMemFunc( AllocFunc, FreeFunc );
    :
}

// メモリ確保関数
void* AllocFunc( DWCAllocType name, u32 size, int align )
{
    void * ptr;
    OSIntrMode old;
    (void)name;
    (void)align;

    old = OS_DisableInterrupts();

    ptr = OS_AllocFromMain( size );

    OS_RestoreInterrupts( old );

    return ptr;
}

// メモリ解放関数
void FreeFunc( DWCAllocType name, void* ptr, u32 size )
{
    OSIntrMode old;
    (void)name;
    (void)size;

    if ( !ptr ) return;

    old = OS_DisableInterrupts();
```

```
    OS_FreeToMain( ptr );

    OS_RestoreInterrupts( old );
}
```

コード 3-1　DWC 初期化

DS カードのバックアップデータを削除するシーケンスを実装する場合は、それと同じタイミングで呼び出すことを推奨します。

4 ユーザーデータの作成

DWC では、ユーザーデータを基にして下記の典型的な処理を行います。

- ユーザー認証
- 友達関係の作成

※インターネットに接続しない場合でも、DS ワイヤレス通信での友達関係を構築するためにユーザーデータは必要になります。

ユーザーデータがまだ作成されていない場合やユーザーデータが破損している場合 DWC_CreateUserData 関数を用いてユーザーデータを作成し DS カードのバックアップに保存してください。

DWCUserData 構造体を保存するメモリはアプリケーションで確保してください。ひとつの DS カードで複数のプレイヤーをサポートする場合、人数分のユーザーデータが必要になります。

既に作成している場合は、バックアップからメモリに読み込んだ後、DWC_CheckUserData関数を用いて、ユーザーデータの正当性を確認してください（コード 4-1）。

```
BOOL create_userdata( void )
{
    // バックアップデータがあって、かつそこにユーザーデータがあれば、
    // 全てロードしてTRUEを返す
    if ( DTUDs_CheckBackup() )
    {
        (void)DTUD_LoadBackup( 0, &s_PlayerInfo, sizeof(DTUDPlayerInfo) );

        OS_TPrintf("Load From Backup¥n");

        if ( DWC_CheckUserData( &s_PlayerInfo.userData ) )
        {
            DWC_ReportUserData( &s_PlayerInfo.userData );
            return TRUE;
        }
    }

    // 有効なユーザーデータがセーブされていなかった場合
    OS_TPrintf("no Backup UserData¥n");

    // ユーザーデータを作成する
    DWC_CreateUserData( &s_PlayerInfo.userData, DTUD_INITIAL_CODE );

    OS_TPrintf("Create UserData.¥n");
    DWC_ReportUserData( &s_PlayerInfo.userData );

    return FALSE;
}
```

コード 4-1　ユーザーデータの作成

DWC_CheckDirtyFlag関数を使用すると、DSカードへユーザーデータを保存する必要があるかどうかを判定できます。ユーザーデータをバックアップへ保存する前に、必ずDWC_ClearDirtyFlag関数でDirtyFlagをクリアするようにしてください（コード 4-2）。

```
void check_and_save_userdata( void )
{
    if ( DWC_CheckDirtyFlag( &s_PlayerInfo.userData ) )
```

```
    {
        DWC_ClearDirtyFlag( &s_PlayerInfo.userData );
        DTUD_SaveBackup( 0, &s_PlayerInfo.userData, sizeof(DWCUserData) );
    }
}
```

コード 4-2　ユーザーデータのセーブ

インターネットに接続する前に以下の手順でユーザーデータの確認を行ってください。

- DWC_CheckHasProfile 関数で、ユーザーデータがインターネットに接続しプロファイルを取得しているかどうかを確認してください。取得していない場合はユーザーデータが更新され、DS 本体と DS カードがセットで扱われるようになります。
- DWC_CheckValidConsole 関数で、DS 本体と DS カードが正しく使用されているかを確認してください。DS 本体と DS カードが正しくない場合、認証に失敗するためインターネットに接続できません。

※　フロー図を「Nintendo Wi-Fi Connection プログラミングガイドライン」に載せていますので、ご確認ください。

5 接続処理

DWC では、インターネットに接続する処理を下記の二つのフェーズに分けて行えるようにしています。

- インターネットに接続する（Wi-Fi 接続し IP アドレスを取得する）
- Wi-Fi コネクションサーバ（以下サーバ）へ接続する

初めてインターネットに接続する際に、任天堂の認証サーバから DS 本体に対してユーザーID を発行します。このユーザーID は、発行された DS 本体のバックアップに保存されます。

その後、サーバへ接続する際に、DWC で作成したユーザーデータにユーザーID とプレイヤーID を保存し、プロファイルを作成します。作成されたプロファイルに対応する GS プロファイル ID がユーザーデータに格納されます。

5.1 インターネットに接続する

初めてインターネットに接続し、IP アドレスを取得する際に、任天堂の認証サーバから DS 本体に対してユーザーID を発行します。また、インターネット接続時に接続テストサーバへ TCP での通信ができることを確認し、インターネットへの接続が正常に行われているかを確認します。

これらの処理は、DWC_*Inet関数で自動的に行われます（コード 5-1）。

```
static DWCInetControl s_ConnCtrl;  // インターネット接続が切断されるまで保持する
BOOL connect_to_inet( void )
{
    // インターネット接続初期化処理
    DWC_InitInet( &s_ConnCtrl );

    // 接続開始
    DWC_SetAuthServer( DWC_CONNECTINET_AUTH_RELEASE );
    DWC_ConnectInetAsync();

    // 接続処理
    while ( !DWC_CheckInet() )
    {
        DWC_ProcessInet();
        // Vブランク待ち処理
        // 接続処理中は、メインスレッドよりも優先順位の低いスレッドに
        // 処理時間を渡す必要があるため、OS_WaitIrq関数を使用してくだ
        // さい。
        GameWaitVBlankIntr();
    }

    // 接続結果確認
    if ( DWC_GetInetStatus() != DWC_CONNECTINET_STATE_CONNECTED )
    {
        handle_error();
        return FALSE;
    }
    // 接続完了
    :
}
```

コード 5-1　インターネットへの接続

5.2 インターネットから切断する

DWC_CleanupInet*関数を呼び出すことで、インターネットから切断されます（コード 5-2）。

もし、通信エラーが発生し自動的に切断された場合でも、ライブラリ用のメモリを解放する必要があるため、必ず呼び出してください。

```
void disconnect_func( void )
{
   while ( !DWC_CleanupInetAsync() )
   {
      GameWaitVBlankIntr();
   }
   :
}
```

コード 5-2 インターネットからの切断

5.3 Wi-Fi コネクションサーバへ接続する

Wi-Fiコネクションサーバ（以下サーバ）へ接続する際には、DWC_InitFriendsMatch関数を使用して、マッチメイク／友達関連機能の初期化を行います（コード 5-3）。

この関数の引数には、これらの機能の制御オブジェクトへのポインタ、ユーザーデータ、GameSpy 社から付与されるプロダクト ID とゲーム名とシークレットキー、DS 同士の通信で使用される送受信バッファサイズ、友達リストと友達リストの最大要素数を渡します。

ここで指定された制御オブジェクトは、DWC_ShutdownFriendsMatch 関数が呼び出されるまで、DWC 内で使用されます。

送受信バッファサイズの詳細については、「8 データ送受信」に記載しています。下記のサンプルプログラムのように 0 を指定した場合は、デフォルトの 8Kbyteが使用されます。

友達リストは、DWCFriendData構造体で友達情報の配列です。友達リストと友達情報の詳細については、「6 友達リストの作成／友達情報の作成」に記載しています。

次に、実際にサーバへ接続するために、DWC_LoginAsync関数を使用します（コード 5-3）。

この関数の最初の引数はゲーム内スクリーンネームです。ここでゲーム中に使用するプレイヤー名があれば必ずそれを指定してください。ゲーム内スクリーンネームは認証サーバに送信され、不適切な名前でないかどうかチェックされます。

チェックの結果はDWC_GetIngamesnCheckResult関数で取得できます（コード 5-3）。

2 つ目の引数は現在は使われていないので、NULL を渡してください。残りの引数はログイン完了コールバックとそのパラメータです。

この関数を呼出した後は、ログイン処理を進めるために毎ゲームフレーム程度の頻度でDWC_ProcessFriendsMatch関数を呼び出してください（コード 5-3）。

DWC_ProcessFriendsMatch 関数は、以後 DWC_ShutdownFriendsMatch 関数を呼ぶまで、マッチメイク／友達関連機能の全ての通信処理を実行します。

ログイン完了後、他のホストへの接続を開始する間などのアプリケーションで意図したネットワーク処理がない場合でも、友達リストの更新などの通信処理が発生する可能性があるので、必ず呼び出すようにしてください。

```
static BOOL s_logined = FALSE;
static DWCFriendsMatchControl s_FMCtrl;

void connect_to_wifi_connection( void )
{
   DWC_InitFriendsMatch( &s_FMCtrl, DTUD_GetUserData(),
                         GAME_PRODUCTID, GAME_NAME, GAME_SECRET_KEY,
                         0, 0,
                         DTUD_GetFriendList(), FRIEND_LIST_LEN );

   // 認証用関数を使ってログイン
   s_logined = FALSE;
   if ( !DWC_LoginAsync( L"なまえ", NULL, cb_login, NULL ) )
   {
      // 接続処理開始の失敗。
      return;
   }

   // 接続完了のポーリング
   while ( !s_logined )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // エラー発生。
         handle_error();
         return;
      }

      GameWaitVBlankIntr();
   }

   // 接続処理完了
   if ( DWC_GetIngamesnCheckResult() == DWC_INGAMESN_INVALID )
   {
      // 不適切なゲーム内スクリーンネームを検出した場合の特別な処理
      disp_ingamesn_warning();
   }
   :
}

// ログイン完了コールバック
void cb_login( void )
{
   if (error == DWC_ERROR_NONE)
   {
      check_and_save_userdata();
      s_logined = TRUE;
   }
}
```

コード 5-3　Wi-Fi コネクションサーバへの接続

DWC_ShutdownFriendsMatch 関数は、マッチメイク／友達関連機能を終了し、ライブラリ内部で確保していたメモリを解放します。

DWC_InitFriendsMatch 関数で設定したユーザーデータで初めてサーバに接続するときに、DS 本体と DS カードはペアとして扱われるようになります。ペアとして扱われるようになった場合、ユーザーデータを保存した DS カードと他の DS 本体を使い、サーバへ接続しようとした場合、接続に成功しなくなりますので、注意してください。

また、初めてサーバに接続するときに、ユーザーデータは必ず更新されます。アプリケーションでは、ログインコールバックを使用して、ログイン完了時にユーザーデータの更新を DWC_CheckDirtyFlag 関数を使用して確認し、もし必要がある場合は、DS カードに保存するようにしてください。

6 友達リストの作成／友達情報の作成

DWC では、プレイヤー同士の友達関係を構築するために、以下の二つの手順を用意しています。

- DS ワイヤレス通信で友達情報を交換する
- 友達コードを交換する

6.1 DS ワイヤレス通信で友達情報を交換する

DSワイヤレス通信では、自分のユーザーデータを元にDWC_CreateExchangeToken関数を用いて交換用の友達情報を作成し、それを他のプレイヤーと交換します（コード 6-1）。

受信した友達情報は、アプリケーションで友達リストに保存して下さい。

```
DWCUserData   s_userData;
DWCFriendData s_friendList[ FRIEND_LIST_LEN ];

// 友達情報を交換する
void exchange_friend_data( void )
{
    int i, j;

    DWCFriendData ownFriendData;
    DWCFriendData recvFriendList[ FRIEND_LIST_LEN ];

    // 自分のユーザーデータから送信用の友達情報を作成する
    DWC_CreateExchangeToken( s_userData, &ownFriendData );

    // MP通信で友達情報を送受信する
    MP_start( (u16 *)&ownFriendData, (u16 *)recvFriendList );
    :

    // 受信した友達情報を友達リストの空いているところへ保存する。
    // 既に持っている同じ友達情報ならば保存しない。
    for ( i = 0; i < num_recv_data; ++i )
    {
        int index;
        for ( j = 0, index = -1; j < FRIEND_LIST_LEN; ++j )
        {
            if ( DWC_IsValidFriendData( &s_friendList[ j ] ) )
            {
                // 友達リストが有効なデータなら、受信した友達情報と
                // 同じでないか調べ、同じなら保存しない
                if ( DWC_IsEqualFriendData( &recvFriendList[ i ],
                                            &s_friendList[ j ] ) )
                    break;
            }
            else
            {
                // 友達リストの空きインデックスを記録しておく
                if ( index == -1 ) index = j;
            }
        }

        // 有効な被りのない友達情報を、友達リストに保存する
        if ( j >= FRIEND_LIST_LEN && index >= 0 )
        {
```

```
            s_friendList[ index ] = recvFriendList[ i ];
        }
    }
    :
}
```

コード 6-1　DS ワイヤレス通信による友達情報の交換

6.2　友達コードを交換する

一度でも Wi-Fi コネクションサーバに接続したプレイヤーは、そのプレイヤーに固有の GS プロファイル ID が割り当てられ、ユーザーデータ内に保存されでいます。

GSプロファイルIDを持っているプレイヤーは、GSプロファイルIDにエラーチェック用の情報を加えた「友達コード」を作成することができます（コード 6-2）。これを 10 進 12 桁の数値として表示し、他のプレイヤーと交換し、入力することで友達データの交換を簡単に行うことができます。

入力された友達コードは、DWC_CreateFriendKeyToken関数で友達情報に変換し、友達リストに保存して下さい（コード 6-2）。

入力された友達コードの正当性はDWC_CheckFriendKey関数でチェックすることができますが、それでも間違っている可能性はあるため、何度でも修正できるようにするユーザーインタフェースを用意して下さい（コード 6-2）。

```
// 友達コードを表示する
void disp_friend_key( void )
{
    u64 friend_key;

    // 自分のユーザーデータから友達コードを作成する
    if ( ( friend_key = DWC_CreateFriendKey( &s_userData ) ) != 0 )
    {
        // 友達コードを表示する
        disp_message( "FRIEND CODE : %lld", friend_key );
    }
    else
    {
        // 友達コードがないことを表示する
        disp_message( "FRIEND CODE : not available" );
    }
    :
}

// 友達コードから友達情報を作成し、友達リストに登録する
BOOL register_friend_key( void )
{
    u64  friend_key;
    DWCFriendData friendData;

    while ( 1 )
    {
        char friend_key_string[ 13 ];

        // 友達コードをユーザーに手入力してもらう
        input_friend_key( friend_key_string );

        // 入力された友達コードの文字列をu64の数値に変換する
        friend_key = charToU64( friend_key_string );
```

```
        // 友達コードの正当性をチェックし、正しければ先に進む
        // 間違っていたらメッセージを表示して、もう一度入力させる
        if ( DWC_CheckFriendKey( s_userData, friend_key ) ) break;
        else disp_warning_message();
    }

    // 正当な友達コードから友達情報を作成する
    DWC_CreateFriendKeyToken( &friendData, friend_key );

    {
        int index;
        // MP通信と同じやり方で友達リストの空きと被りを探し、
        // 友達情報を登録する
        :
        s_friendList[ index ] = friendData;
        :
    }
}
```

コード 6-2　友達コードの交換

6.3　友達リストの同期処理

アプリケーションが保持している友達リスト（以下ローカル友達リスト）をインターネット上で有効にするためには、DWC_UpdateServersAsync関数を呼び出してGameSpy社のサーバ上に格納されている友達リスト（以下サーバ友達リスト）を更新する必要があります（コード 6-3）。

この同期処理を行うためには、まず DWC_LoginAsync 関数によるログインまでを完了させておく必要があります。

引数には、プレイヤー名（過去の仕様です。NULL を指定して下さい）、友達リスト同期処理完了コールバックとそのパラメータ、後述する友達状態変化通知コールバックとそのパラメータ、友達リスト削除コールバックとそのパラメータを指定します。

友達リストの同期処理の主な内容は、ローカル友達リストにあってサーバ友達リストにない友達に友達関係構築要求を送信することと、サーバ友達リストにあってローカル友達リストにない友達情報をサーバ友達リストから削除することです。

友達関係構築要求を送った相手がその時オフライン状態だったとしても、この要求は GameSpy サーバ上に保存され、相手が次回 DWC_LoginAsync 関数によるログインが完了した直後に届けられます。そして、相手もこちらの情報をローカル友達リストに持っていた場合のみ、友達関係が成立します。

ただしこれは、こちらが相手を友達として登録できただけです。友達関係構築要求を受信した相手は、自動的に同様の手順を踏んで、こちらを友達として登録します。

注意しなければならないのは、友達リスト同期処理完了コールバックが呼び出されるのは、ローカル／サーバ両友達リストを全てチェックし、必要な友達関係構築要求の送信と、不要な友達情報の削除を全て終えた時であるということです。コールバックが返ってきたからといって、全ての友達関係が成立したというわけではありません。

友達リスト同期処理完了コールバックの引数 isChanged が TRUE の場合は、ローカル友達リスト中のいずれかの友達情報が更新されたことを示しており、ローカル友達リストをセーブする必要があります。友達リスト同期処理中以外に友達関係が成立した場合は、DWC_SetBuddyFriendCallback 関数で設定した友達関係成立コールバックが呼び出されます。

また、友達リストの同期処理の過程では、リスト内に同じ友達の友達情報を複数発見した場合には、1 つを残

して自動的に削除します。そして削除した友達情報の友達リスト内でのインデックスと、同じ情報と判定された友達のインデックスを引数として、削除するごとにコールバックが呼び出されます。

```
BOOL s_update     = FALSE;
BOOL s_updateFriendList = FALSE;

void sync_friend_list( void )
{
   // 友達関係成立コールバック設定
   DWC_SetBuddyFriendCallback( cb_buddyFriend, NULL );

   // ローカル友達リストとサーバ友達リストの同期処理
   if ( !DWC_UpdateServersAsync( NULL,
                                 cb_updateServers, NULL,
                                 NULL, NULL,
                                 cb_deleteFriend, NULL ) )
   {
      // 同期処理開始の失敗
      return;
   }

   while ( !s_update )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // エラー発生。
         handle_error();
         return;
      }

      GameWaitVBlankIntr();
   }
   :

   while ( 1 )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // エラー発生。
         handle_error();
         return;
      }

      // 友達リストは不定期に更新されるため、適当な時点で以下の処理を行ない、
      // 更新されたローカル友達リストをまとめてセーブしてください。
      if ( s_updateFriendList )
      {
         // 友達リストが更新されていたらセーブする
         s_updateFriendList = FALSE;
         save_friendList();
      }

      game_loop();

      GameWaitVBlankIntr();
   }
   :
}
```

```
// 友達リスト同期処理完了コールバック
void cb_updateServers( DWCError error, BOOL isChanged, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        // 友達リスト同期処理成功
        s_update = TRUE;

        // 友達リストが変更されていたらセーブする必要がある
        if ( isChanged ) s_updateFriendList = TRUE;
    }
}

// 友達リスト削除コールバック
void cb_deleteFriend( int deletedIndex, int srcIndex, void* param )
{
    OS_TPrintf( "friend[%d] was deleted (equal friend[%d]).¥n",
                deletedIndex, srcIndex );
    s_updateFriendList = TRUE;
}

// 友達関係成立コールバック
void cb_buddyFriend( int index, void* param )
{
    OS_TPrintf( "Got friendship with friend[%d].¥n", index );
    s_updateFriendList = TRUE;
}
```

コード 6-3　友達リストの同期処理

6.4　友達情報タイプ

友達情報にはデータタイプが設定されており、DWC_GetFriendDataType関数でそれを取得することができます（コード 6-4）。データタイプは下記の通りです。

- DWC_FRIENDDATA_NODATA　　友達情報は格納されていない
- DWC_FRIENDDATA_LOGIN_ID　　一度も Wi-Fi コネクションに接続していない状態の ID
- DWC_FRIENDDATA_FRIEND_KEY　　友達コード
- DWC_FRIENDDATA_GS_PROFILE_ID　　GS プロファイル ID

データタイプが DWC_FRIENDDATA_LOGIN_ID の時は、相手がまだ GS プロファイル ID を取得していない時に DS ワイヤレス通信で取得した友達情報であることを示しています。

その後、相手が GS プロファイル ID を取得したのちに、自分が友達リスト同期処理を完了させると、データタイプが DWC_FRIENDDATA_GS_PROFILE_ID に変化します。

データタイプが DWC_FRIENDDATA_FRIEND_KEY の時は、友達コードで登録された GS プロファイル ID で、まだ友達関係が成立していないことを示しています。友達関係が成立すれば、データタイプが DWC_FRIENDDATA_GS_PROFILE_ID に変化します。

また、友達情報から友達関係が成立しているかどうかを、DWC_IsBuddyFriendData関数を用いて調べることができます（コード 6-4）。

```
void disp_friendList( void )
{
    int i;

    for ( i = 0; i < FRIEND_LIST_LEN; ++i )
```

```
    {
        // 友達情報タイプ取得
        int type = DWC_GetFriendDataType( &s_friendList[ i ] );
        OS_TPrintf( "friend[%d] type %d.¥n", type );

        if ( type == DWC_FRIENDDATA_GS_PROFILE_ID )
        {
            // GSプロファイルIDの場合は友達関係を表示する
            if ( DWC_IsBuddyFriendData( &s_friendList[ i ] ) )
            {
                OS_TPrintf( "Friendship is established.¥n" );
            }
            else
            {
                OS_TPrintf( "Friendship is not yet established.¥n" );
            }
        }
    }
    :
}
```

コード 6-4　友達情報タイプの取得

6.5　友達の状態を取得する

Wi-Fi コネクションに参加するプレイヤーは、全員自分の状態というものを保持し、それは GameSpy 社のサーバによって管理されています。アプリケーションから参照できるプレイヤーの状態には以下の二種類があります。

- 通信状態
- ステータス文字列、もしくはバイナリデータ

通信状態は、DWC_STATUS_*定数で定義されており、DWC が自動的に設定します。

ステータス文字列、もしくはバイナリデータは、DWC_SetOwnStatusString /DWC_SetOwnStatusData関数を用いて、アプリケーションがセットすることができます（コード 6-5）。

セットすることができる文字列は NULL で終端されている必要があり、NULL 終端も含めて 256 文字までに制限されています（バイナリデータは関数内で文字列に変換され、データサイズ×1.5 倍ほどの文字数になります）。

'/'と'¥¥'は識別文字としてライブラリが使用するため、文字列中には使用しないでください。

友達関係が成立していれば、友達の現在の状態を取得することができます。DWC_UpdateServersAsync関数の引数で友達状態変化通知コールバックを指定していれば、友達の状態に変化がある度にコールバックが呼び出され、友達の状態を知ることができます（コード 6-5）。

また、友達の状態取得用にDWC_GetFriendStatus*関数群も用意されています（コード 6-5）。

これらの関数群は、DWC が保持している友達の状態リストにアクセスするため、通信は発生しません。しかし、数 100us 単位の処理時間がかかるため、短い期間に何度も呼び出す場合はご注意ください。

プレイヤーの状態は、通信中に突然電源を切った場合などは、数分間は前の状態が残ります。

```
void sync_friend_list( void )
{
    int i;

    // ローカル友達リストとサーバ友達リストの同期処理
```

```
    if ( !DWC_UpdateServersAsync( NULL,
                                  cb_updateServers, NULL,
                                  cb_friendStatus, NULL,
                                  NULL, NULL ) )
    {
        // 同期処理開始の失敗
        return;
    }
    :

    // 友達リスト同期処理完了
    :

    // 自分のステータス文字列をセットする
    DWC_SetOwnStatusString( "location=city,level=1" );
    :

    for ( i = 0; i < FRIEND_LIST_LEN; ++i )
    {
        if ( DWC_IsValidFriendData( &friendList[ i ] )
        {
            u8    status;
            char* statusString;

            // 有効な友達情報ならその友達の状態を取得する
            status = DWC_GetFriendStatus( &friendList[ i ], statusString );

            // 友達の状態を表示する
            disp_friend_status( status, statusString );
        }
    }
    :
}

// 友達状態変化通知コールバック
void cb_friendStatus( int index, u8 status, const char* statusString, void*
param )
{
    OS_TPrintf( "Friend[%d] status -> %d (statusString : %s).¥n",
                index, status, statusString );
}
```

コード 6-5　友達の状態取得

7 マッチメイク

DWC ではマッチメイクを行うための仕組みとして、ピア型とサーバクライアント型の二つを用意しています。

ピア型マッチメイクは、各 DS をサーバ、クライアントと分けずにマッチメイクを行う方法で、更に下記の二つに分類されます。

- 友達無指定ピアマッチメイク
- 友達指定ピアマッチメイク

7.1 友達無指定ピアマッチメイク

不特定多数のプレイヤーとのマッチメイクを行ないます。

友達無指定ピアマッチメイクを開始するためには、DWC_ConnectToAnybodyAsync関数を使用します（コード 7-1）。この関数の引数には、自分を含めた接続希望人数、マッチメイクに条件を付加するためのフィルタ文字列、マッチメイク完了コールバックとそのパラメータ、後述するプレイヤー評価コールバックとそのパラメータを渡します。

フィルタ文字列は、マッチメイクの候補として検索するプレイヤーを絞り込むために用います。このフィルタ文字列内で使用するためのマッチメイク指標キー（以下の例では、キー名"str_key"と"int_key"）は、予めDWC_AddMatchKey*関数を使って登録しておく必要があります（コード 7-1）。キー名はライブラリ内で保存しますが、キーの値はライブラリ内ではポインタのみ保存するので、マッチメイクが完了するまでは保持するようにして下さい。また、キー名には使用できないものがあります。詳しくは「7.6 マッチメイク指標キーに使用できないキー名」を参照してください。

```
static BOOL s_matched  = FALSE;
static BOOL s_canceled = FALSE;
static const char* s_str_key = "anymatch_test";
static const int s_int_key = 10;

void do_anybody_match( void )
{
   // マッチメイク指標キーをセット
   DWC_AddMatchKeyString( 0, "str_key", s_str_key );
   DWC_AddMatchKeyInt( 0, "int_key", &s_int_key );

   // 友達無指定マッチメイク開始
   DWC_ConnectToAnybodyAsync( 4,
                              "str_key = 'anymatch_test' and int_key = 10",
                              cb_anymatch, NULL,
                              NULL, NULL );

   // マッチメイク完了のポーリング
   while ( !s_matched )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // エラー発生。
         handle_error();
         return;
      }
```

```
        GameWaitVBlankIntr();
    }

    // マッチメイク完了
    :
}

// マッチメイク完了コールバック
void cb_anymatch( DWCError error, BOOL cancel, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( cancel ) s_canceled = TRUE;
        else          s_matched  = TRUE;
    }
}
```

コード 7-1　友達無指定ピアマッチメイク

7.2　友達指定ピアマッチメイク

友達リストに登録された友達とのマッチメイクを行ないます。

友達指定ピアマッチメイクを開始するためには、DWC_ConnectToFriendsAsync関数を使用します（コード7-2）。

この関数の引数には、マッチメイクしたい友達の友達リストインデックス配列（以下インデックスリスト）、インデックスリストの要素数、自分を含めた接続希望人数、友達の友達とのマッチメイクを許可するかどうか、マッチメイク完了コールバックとそのパラメータ、後述するプレイヤー評価コールバックとそのパラメータを渡します。

インデックスリストに NULL を指定すれば、友達リスト中の全ての友達をマッチメイクの候補とします。

友達指定ピアマッチメイク時に参照される友達リストは、DWC_InitFriendsMatch 関数で指定されたものになります。

また、各人はおのおの異なる友達リストを持っているか、もしくは異なるインデックスリストを指定している可能性が高いため、友達の友達とのマッチメイクを許可しない場合は、マッチメイクの成功率が極端に下がってしまいます。

```
static BOOL s_matched  = FALSE;
static BOOL s_canceled = FALSE;

void do_friend_match( void )
{
    // 友達指定マッチメイク開始
    DWC_ConnectToFriendsAsync( NULL, 0, 4, TRUE,
                               cb_friendmatch, NULL,
                               NULL, NULL );

    // マッチメイク完了のポーリング
    while ( !s_matched )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // エラー発生。
```

```
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // マッチメイク完了
    :
}

// マッチメイク完了コールバック
void cb_friendmatch( DWCError error, BOOL cancel, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( cancel ) s_canceled = TRUE;
        else          s_matched  = TRUE;
    }
}
```

コード 7-2　友達指定ピアマッチメイク

7.3　マッチメイク候補プレイヤーを評価する

ピア型マッチメイクにおいては、マッチメイク候補として検索された複数のプレイヤーをゲーム独自の指標で評価し、マッチメイク候補としての優先順位付けを行うことができます。

ピア型マッチメイク開始関数の引数に評価コールバックを指定すると、マッチメイク時に候補プレイヤーを見つける度に、指定された評価コールバックが呼び出されます。

このコールバック内では、DWC_AddMatchKey*関数で登録されているマッチメイク指標キーを、DWC_GetMatch*Value関数を用いて参照することができます（コード 7-3）。その値を元にプレイヤーを評価し、評価値を戻り値として返してください。

評価値が 0 以下のプレイヤーはマッチメイクの対象から外されます。

ただし、常に評価値が最高のプレイヤーがマッチメイクの相手として選ばれるわけではなく、評価値が高いプレイヤーほど選ばれやすくなります。

```
static const char* s_str_key = "anymatch_test";
static const int s_int_key = 10;

void do_anybody_match( void )
{
    // マッチメイク指標キーをセット
    DWC_AddMatchKeyString( 0, "str_key", s_str_key );
    DWC_AddMatchKeyInt( 0, "int_key", &s_int_key );

    // 友達無指定マッチメイク開始
    DWC_ConnectToAnybodyAsync( 4,
                               "str_key = 'anymatch_test'",
                               cb_anymatch, NULL,
                               cb_eval, NULL );
    :
}

// プレイヤー評価コールバック
int cb_eval( int index, void* param )
{
```

```
    int eval_int;

    // マッチメイク指標キー int_key の値を取得する
    eval_int = DWC_GetMatchIntValue(index, "int_key", -1);

    if ( eval_int >= 0 )
    {
        // 自分とどれだけ近い値かを評価値とする
        return MATH_ABS( s_int_key - eval_int ) + 1;
    }
    else
    {
        // キー int_key を持たないプレイヤーとはマッチメイクしない
        return 0;
    }
}
```

コード 7-3　マッチメイク候補プレイヤーの評価

7.4　サーバクライアント型マッチメイク

サーバクライアント型マッチメイクは、各 DS をサーバとクライアントに役割を明確に分ける友達同士のマッチメイク方法です。完成したネットワークがメッシュ型ネットワークである点は、ピア型マッチメイクと同じです。

サーバDSは、自分を含めた最大接続希望人数、マッチメイク完了コールバックとそのパラメータ、新規接続クライアント通知コールバックとそのパラメータを引数に指定して、DWC_SetupGameServer関数を呼び出し、クライアントDSが接続に来るのを待ちます（コード 7-4）。

クライアントDSは、接続したい友達の友達リストインデックス、マッチメイク完了コールバックとそのパラメータ、新規接続クライアント通知コールバックとそのパラメータを引数に指定して、DWC_ConnectToGameServerAsync関数を呼び出せば、その友達がサーバDSとしてマッチメイクを開始していれば接続に行くことができます（コード 7-4）。

サーバクライアント型マッチメイク完了時には、サーバ DS からは接続された全クライアント DS が相互に友達の関係ですが、クライアント DS 同士はサーバ DS を介した友達の友達の関係である可能性があります。

マッチメイク完了コールバックは、自分がサーバ DS との接続に成功した時に呼ばれる他、自分が所属しているメッシュ型ネットワークに新たにクライアント DS が加わった場合にも呼ばれます。

新規接続クライアント通知コールバックは、自分が所属しているメッシュ型ネットワークに、新たにクライアント DS が接続を開始した時に呼び出されます。

```
static BOOL s_matched = FALSE;

void do_server_match( void )
{
    // サーバDSとしてマッチメイクを開始する
    DWC_SetupGameServer( 4,
                         cb_sc_match, (void *)CB_CONNECT_SERVER,
                         cb_sc_new, NULL );

    while ( 1 )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
```

```
            // エラー発生。
            handle_error();
            return;
        }

        if ( s_matched )
        {
            // 新規接続クライアントとの接続が完了した場合
            init_new_connection();
            s_matched = FALSE;
        }

        GameWaitVBlankIntr();
    }
    :
}

void do_client_match( void )
{
    // クライアントDSとしてマッチメイクを開始する
    DWC_ConnectToGameServerAsync( 0,
                                  cb_sc_match, (void *)CB_CONNECT_CLIENT,
                                  cb_sc_new, NULL );

    // マッチメイク完了のポーリング
    while ( !s_matched )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // エラー発生。
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // マッチメイク完了
    :
}

// マッチメイク完了コールバック
void cb_sc_match( DWCError error, BOOL cancel, BOOL self, BOOL isServer, int
index, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( !cancel )
        {
            // 接続成功
            s_matched = TRUE;
        }
        else if ( self || isServer )
        {
            // 自分がマッチメイクをキャンセルした、もしくは自分が
            // クライアントDSで、サーバDSがマッチメイクをキャンセルした
            s_canceld = TRUE;
        }
        // 新規接続クライアントがマッチメイクをキャンセルしても何もしない
    }
}
```

```
// 新規接続クライアント通知コールバック
void cb_sc_new( int index, void* param )
{
    OS_TPrintf( "Newcomer : friend[%d].¥n", index );
}
```

コード 7-4　サーバクライアント型マッチメイク

サーバクライアント型マッチメイクでも、ピア型マッチメイクと同じくメッシュ型ネットワークを構築しているので、サーバ DS が接続を切断しても残りのクライアント DS 同士の通信はできます。

しかし、サーバ DS が抜けた状態で引き続きサーバクライアント型マッチメイクを行うことはできないので、サーバ DS が抜けた時点で全 DS が接続を切断するよう実装することを推奨します。

また、マッチメイク中に DWC_StopSCMatchingAsync 関数を呼ぶ事で、サーバ DS はクライアントの受付を締め切る事ができます。

7.5　マッチメイクの高速化

友達無指定ピアマッチメイクにおいて、マッチメイキングサーバよりマッチメイク候補リストを取得する際にフィルタ機能を用いる事でマッチメイクの高速化を図ることができます（コード 7-1）。

マッチメイキングサーバに置かれているマッチメイク候補リストは、様々な条件が混在したリストとなっています。そのため無条件にリストを取得し、評価コールバック内でのマッチメイク候補の絞込みを行うと、マッチメイク不成立の可能性が高くなり、リストの取り直しとマッチメイクの繰り返しでタイムロスが発生します。

フィルタ機能を使用することによって、取得するマッチメイク候補リストをマッチメイク成立可能な候補リストにする事ができるため、不成立の可能性が下がり、マッチメイクの高速化を図ることができます。

また、同様に評価コールバック内での極端な絞込み（同レベル対戦、同地区対戦等の候補自体が少ないと考えられる条件）を行うと、マッチメイクの成功率が下がり、タイムロスが発生します。

マッチメイクの高速化を図る場合、

- フィルタ機能を使用し、取得するマッチメイク候補リストを成立可能な候補リストにする
- 評価コールバック内では極端な絞込みをせず、極力成立するような仕様とする

この点を考慮するようにしてください。

7.6 マッチメイク指標キーに使用できないキー名

DWC_AddMatchKey*関数によって登録されるマッチメイク指標キーには、ライブラリとサーバで使用されているため使用できないキー名が存在します。以下のキー名は使用しないようにしてください。

表 7-1 マッチメイク指標キーに使用できないキー名一覧

country	region	hostname	gamename	gamever	hostport
mapname	gametype	gamevariant	numplayers	numteams	maxplayers
gamemode	teamplay	fraglimit	teamfraglimit	timeelapsed	timelimit
roundtime	roundelapsed	password	groupid	player_	score_
skill_	ping_	team_	deaths_	pid_	team_t
score_t	dwc_pid	dwc_mtype	dwc_mresv	dwc_mver	dwc_eval

8 データ送受信

8.1 ピア・ツー・ピアのデータ送受信

マッチメイクが完了したら、各 DS 同士が相互にコネクションを確立した状態、いわゆるメッシュ型ネットワークが構築されています。このネットワークを介して各 DS と直接通信を行う前に、いくつか準備が必要です。

まず各DSからのデータを受信するための受信バッファの設定です。これにはDWC_SetRecvBuffer関数を使用しますが、引数のaidには、各DSの識別番号であるAIDを指定します（コード 8-1）。

AID には 0～（ネットワーク構成台数-1）の数値が用いられます。例えば 4 人でマッチメイクを完了した場合なら、0, 1, 2, 3 の 4 台が存在し、AID = 1 の人がネットワークから抜ければ、残りは 0, 2, 3 となります。

受信バッファを設定する前に届いたデータは破棄されます。

次に送受信コールバックの設定を、DWC_SetUserSendCallback関数、DWC_SetUserRecvCallback関数で行うことができます（コード 8-1）。

受信コールバックは、他の DS からデータを受信した時に呼び出されます。送信コールバックは、送信指定したデータの送信が完了した直後に呼び出されます。

ここで言う送信の完了とは、低レイヤーの送信関数にデータを渡し終えたというだけで、相手側にデータが到着したという意味ではありません。

もう 1 つ、自分、もしくは他のDSが正式なコネクション切断の手続きを踏んで、ネットワークから離脱した場合に呼び出される、コネクションクローズコールバックを設定することができます。これにはDWC_SetConnectionClosedCallback関数を用います（コード 8-1）。

以上の設定は、DWC_ShutdownFriendsMatch 関数が呼ばれるまではクリアされないので、必ずしもマッチメイク完了直後に設定する必要はありません。

```
static u8 s_RecvBuffer[ 3 ][ SIZE_RECV_BUFFER ];

void prepare_communication( void )
{
    u8* pAidList;
    int num = DWC_GetAIDList( &pAidList );
    int i, j;

    for ( i = 0, j = 0; i < num; ++i )
    {
        if ( pAidList[i] == DWC_GetMyAID() )
        {
            j++;
            continue;
        }

        // 自分のAID以外の受信バッファをセットする
        DWC_SetRecvBuffer( pAidList[i], &s_RecvBuffer[i-j], SIZE_RECV_BUFFER
);
    }
```

```
    // 送信コールバックの設定
    DWC_SetUserSendCallback( cb_send );

    // 受信コールバックの設定
    DWC_SetUserRecvCallback( cb_recv );

    // コネクションクローズコールバックの設定
    DWC_SetConnectionClosedCallback( cb_closed, NULL );
}

// データ送信コールバック
void cb_send( int size, u8 aid )
{
    OS_TPrintf( "to aid = %d  size = %d¥n", aid, size );
}

// データ受信コールバック
void cb_recv( u8 aid, u8* buffer, int size )
{
    OS_TPrintf( "from aid = %d  size = %d  buffer[0] = %X¥n",
                aid, size, buffer[0] );
}

// コネクションクローズコールバック
void cb_closed( DWCError error, BOOL isLocal, BOOL isServer, u8  aid, int
index, void* param)
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( isLocal )
        {
            OS_TPrintf( "Closed connection to aid %d
                        (friendListIndex = %d).¥n", aid, index );
        }
        else
        {
            OS_TPrintf( "Connection to aid %d
                        (friendListIndex = %d) was closed.¥n", aid, index );
        }
    }
}
```

コード 8-1　データ送受信のための準備

データの送信には、Reliable 送信、Unreliable 送信の二種類があり、どちらも UDP 通信を用いていますが、Reliable 送信は TCP 通信のようにパケットロスがなく、パケットの到着順が入れ替わることもない代わりに、パケットの到着を送信ごとに確認するので送信完了に時間がかかります。

Unreliable 送信は UDP 通信そのもので、上記のどちらの問題も起こりえますが、データの到着確認も再送も行わないので高速です。

DWC よりも低位のレイヤーでデータ送信が滞った場合、DWC_InitFriendsMatch 関数でサイズを指定した送信バッファにデータが溜まっていきます。そして送信バッファに空きが足りない時に Reliable 送信しようとすると、送りきれないデータはそのまま保留され、送信バッファに空きができ次第 DWC_ProcessFriendsMatch 関数内から送信されます。

またこれとは別に、一度に送信できる最大データサイズ（デフォルト 1465Byte)というものも決まっています。このサイズ以上のデータを送信しようとした場合も、送信データが分割され、送信が保留されます。

この最大サイズは DWC_SetSendSplitMax 関数で変更することができますが、様々な設定の通信機器に対応

するためには、これ以上のサイズには設定しないでください。

このように送信データが保留されている間は、送信バッファを破棄しないでください。また、保留されている間は次のデータを送信できません。

送信バッファの空きや、送信先AIDが有効かどうかを含めて、Reliable送信が可能かどうかは、DWC_IsSendableReliable関数を用いてチェックすることができます（コード 8-2）。

Unreliable 送信時は、前述の最大データサイズを越えて一度に送信しようとすると、送信できずに FALSE を返します。

```
static u8  s_SendBuffer[ SIZE_SEND_BUFFER ];

void send_data( void )
{

    // 接続中の全てのDSにデータをUnreliable送信する
    // 自分のAIDを渡しても無視されます。
    DWC_SendUnreliableBitmap( DWC_GetAIDBitmap(), s_SendBuffer,
SIZE_SEND_BUFFER );
    :

    // 今AID=0のDSにReliable送信が可能かどうか調べる
    if ( !DWC_IsSendableReliable( 0 ) ) return;

    // 特定のDSにデータをReliable送信する。
    DWC_SendReliableBitmap( 0, s_SendBuffer, SIZE_SEND_BUFFER );
    :
}
```

コード 8-2　データ送信

8.2　コネクションを切断する

メッシュ型ネットワーク中の全ての DS とコネクションを切断するには、DWC_CloseAllConnectionsHard 関数を呼び出してください。クローズ処理が実行されたら、この関数を抜ける前に DWC_SetConnectionClosedCallback 関数で設定したコネクションクローズコールバックが呼び出されます。

同時にクローズの通知が接続されていた他の DS にも通知され、コネクションクローズコールバックが呼び出されます。

サーバクライアント型マッチメイクのサーバ DS においては、既に接続中の DS がない場合でもこの関数を呼び出せます。この場合は、マッチメイクに使用したメモリ領域が残っていればそれを解放し、通信状態をオンライン状態に戻します。

この関数を呼び出しても、Wi-Fi コネクションサーバとの接続は切断されません。

また、AID を指定してコネクションを切断する DWC_CloseConnectionHard 関数と AID のビットマップを指定して複数のコネクションを同時に切断できる DWC_CloseConnectionHardBitmap 関数も用意されています。

これらの関数は、電源を切るなどの理由で通信不能になったホストに対し、コネクションをクローズするというような、異常状態処理の用途を想定しています。

8.3 DWC_InitFriendsMatch 関数で指定するバッファサイズの目安

DWC_InitFriendsMatch 関数で指定するバッファサイズは DWC が内部で使用するバッファサイズとなります。送信バッファは Reliable 通信で送信したデータのうち ACK が返ってきていないものを保持するために使用されます。受信バッファは正しい順番で到着しなかったデータを保持しておくために使用されます。

Reliable 通信の場合、ネットワークの瞬断に対応できるだけの容量が必要になるためゲームの仕様上許容される瞬断時間分の送受信バッファが必要となります。Unreliable 通信では、基本的に送受信バッファは使われませんが、DWC がピア･ツー・ピア接続時に内部で Reliable 通信を行っているため、送信バッファに最低 1KByte、受信バッファに最低 128Byte が必要となります。

表 8-1 通信内容とバッファサイズの目安

通信内容		バッファサイズの目安	備考
Reliable 通信	送信バッファサイズ	ゲームの仕様上許容される瞬断時間（秒）x 1 秒あたりの Reliable データ数＋Reliable データサイズ（7 x 送信データの分割数＋送信データサイズ＋15）	最低 1KByte
	受信バッファサイズ		最低 128Byte
Unreliable 通信	送信バッファサイズ	Unreliable 通信の最大データサイズ＋2Byte	最低 1KByte
	受信バッファサイズ	最低 128Byte	

※ 送信データの分割数は、一度に送信できる最大データサイズ（DWC_SetSendSplitMax 関数で設定される、デフォルトは 1465Byte）を超えた場合に送信データが分割される数の事を示します。

仕様上許容される瞬断時間が 1 秒、通信頻度が 3 フレームに 1 回、一度に送信できる最大データサイズ 64Byte のゲームにおいて、100Byte のデータを Reliable 通信で送る場合、必要な送受信バッファサイズは

1（秒）x（60（フレーム）÷3）x（7 x 2（分割）＋100（Byte）+15）=2580（Byte）

となります。

8.4 遅延とパケットロスのエミュレーション

DWC では、送受信データの遅延とパケットロスをエミュレーションすることができます。

ただし送信遅延に関しては、送信データを指定時間、別バッファにコピーして置いておくため、コネクションをクローズした時にそのデータが破棄され、相手に届きません。そのため受信遅延のみの使用を推奨します。

それぞれ、下記のようにパケットロス率（単位は%）と、遅延時間（単位はミリ秒）と、対象とするDSのAIDを指定します（コード 8-3）。

```
void set_trans_emulation( void )
{
    DWC_SetSendDrop( 30, 0 );
    DWC_SetRecvDrop( 30, 0 );

    DWC_SetSendDelay( 300, 0 );
    DWC_SetRecvDelay( 300, 0 );
    :
}
```

コード 8-3 遅延とパケットロスのエミュレーション

8.5 データ送受信量の目安

Reliable 通信と Unreliable 通信における通信量は下表の内容になります。

表 8-2 通信データ内訳

<table>
<tr><th>送信データ項目</th><th colspan="4">送信データサイズ</th></tr>
<tr><td>Preamble</td><td colspan="4">192bit（24Byte）</td></tr>
<tr><td>MAC</td><td colspan="4">24Byte</td></tr>
<tr><td>LLC</td><td colspan="4">8Byte</td></tr>
<tr><td>IP</td><td colspan="4">20Byte</td></tr>
<tr><td>UDP</td><td colspan="4">8Byte</td></tr>
<tr><td rowspan="3">DATA</td><td colspan="3">Reliable 通信</td><td>Unreliable 通信</td></tr>
<tr><td>ヘッダ送信</td><td>データ送信</td><td>受信確認</td><td>データ送信</td></tr>
<tr><td>15Byte</td><td>7 + XXXByte</td><td>5Byte</td><td>XXXByte</td></tr>
<tr><td>FCS</td><td colspan="4">4Byte</td></tr>
<tr><td>B（パケットの衝突回避用ランダム時間）</td><td colspan="4">MAX 600usec</td></tr>
</table>

※ Reliable 通信にはデータ送信の前後にヘッダ送信と受信確認が送受信されます。

各送信におけるデータ送信時間は Preamble + (MAC + LLC + IP + UDP + DATA + FCS) x 4 + B [usec]から求める事ができますが、帯域の状況による再送や送信パケット数、衝突回避のための送信待機等、様々な要因によって送信時間が変化するため、正確なデータ送受信量を計算する事は困難です。

そのため、本項目では実験により得られたデータ送受信量を目安として記載しています。

実験は、Reliable 通信と Unreliable 通信、AP の機種やメーカー、電波の使用率、送信サイズ、送信頻度等の条件を変化させ、スループットや CPU 負荷、パケットロス率を測定する事で行いました。その結果、おおよそ下記の内容が判明しています。

- ヘッダ分や無線通信のバックオフ時間（通信間の空き時間、パケットの衝突回避用ランダム時間等を含む）があるため、送信頻度（発生させるパケット数）の影響が大きい
- 4 台メッシュ型、送信頻度 3 フレーム、ノイズ電波 10%以下の条件では送信サイズ 120～150byte が限界となる
- 4 台メッシュ型、送信頻度 3 フレーム、ノイズ電波 50%程度の条件では送信サイズ 100～120byte が限界となる
- Reliable 通信ではトラフィック混雑状態（ネットワークの混雑によって繰り返される再送で混雑が続く状態）が起こりやすく、1 度起こると復帰までに時間がかかる。

※ ノイズ電波は、他の DS から WMTestTool を使用して発生させています。

以上の実験結果を元に、弊社発売のタイトルでは下記のように通信を行っています。

- 「4 台メッシュ型、Unreliable 通信」
  N フレーム目　　：相手 1 に送信
  N＋1 フレーム目：送信しない
  N＋2 フレーム目：相手 2 に送信
  N＋3 フレーム目：送信しない
  N＋4 フレーム目：相手 3 に送信
  N＋5 フレーム目：送信しない
  ：（以下繰り返し）
  各 60～104byte の通信
- 「4 台サーバクライアント型、Reliable 通信」
  送信頻度を 3 フレーム、通信サイズが通常 1～40byte、最大 256byte

9 HTTP 通信機能

DWC では、HTTP によるデータのアップロード／ダウンロードを行う GHTTP ライブラリを用意しており、これはマッチメイク／友達関連機能とは独立して使うことができます。

9.1 GHTTP ライブラリを使用するための準備をする

GHTTPライブラリを使用する前に、必ず一度DWC_InitGHTTP関数を呼び出して下さい（コード 9-1）。

引数には NULL を指定してください。返り値は必ず TRUE になります。

DWC_InitGHTTP 関数が呼ばれており、インターネットへの接続が完了していれば、GHTTP ライブラリの機能を使用することができます。

```
void init_ghttp( void )
{

    // DWC初期化
    init_dwc();

    // インターネットへの接続を行う
    if ( connect_to_inet() ) return;

    // GHTTPを初期化する
    DWC_InitGHTTP( NULL );
}
```

コード 9-1 GHTTP ライブラリの初期化

9.2 データをアップロードする

GHTTPライブラリを用いてデータをHTTPサーバにアップロードするためには、まずDWC_GHTTPNewPost関数でDWCGHTTPPost型オブジェクトを作成する必要があります（コード 9-2）。

その後、アップロードしたいデータをDWC_GHTTPPostAddString関数でこのオブジェクトに追加していきます（コード 9-2）。

DWC_GHTTPPostAddString 関数の引数には、DWCGHTTPPost 型オブジェクトへのポインタ、データを特定するための key 文字列へのポインタ、実際に追加したいデータ（value 文字列）へのポインタを指定します。

key／value 文字列ともに、コピーしてライブラリ内で保持されます。

また、両文字列ともに NULL 終端されている必要があります。value 文字列に NULL を指定した場合は、NULL 終端のみの文字列""を指定したことになります。

実際にデータのアップロードを開始するためには、DWC_PostGHTTPData関数を使います（コード 9-2）。

この関数の引数には、アップロード先の URL、DWCGHTTPPost 型オブジェクトへのポインタ、完了コールバックとそのパラメータを渡します。

アップロード開始後の通信処理は全てDWC_ProcessGHTTP関数内で行われるため、毎ゲームフレーム程度の頻度でこの関数を呼ぶようにしてください（コード 9-2）。

データのアップロードが完了すれば、完了コールバックが呼び出されます。

DWCGHTTPPost 型オブジェクトは、アップロードが完了して、完了コールバックを抜けた直後に解放されます。

実際にHTTPサーバに送信されるデータは、下記サンプルプログラム（コード 9-2）の例では以下のような文字列になります。

"key1=value1&key2=value2"

同一 DWCGHTTPPost 型オブジェクトに対して更にデータが追加されれば、以下のように文字列が追加されていきます。

"key1=value1&key2=value2&key3=value3&key4=value4…"

```
static int s_send_cb_level = 0;

void post_ghttp_data( void )
{
    int req;
    DWCGHTTPPost post;

    // DWCGHTTPPost型オブジェクトを作成する
    DWC_GHTTPNewPost( &post );

    // DWCGHTTPPost型オブジェクトにアップロードするデータをセットする
    DWC_GHTTPPostAddString( &post, "key1", "value1" );
    DWC_GHTTPPostAddString( &post, "key2", "value2" );

    // データのアップロードを開始する
    s_send_cb_level++;
    req = DWC_PostGHTTPData( "http://www.test.net", &post, cb_post, NULL );

    if ( req < 0 )
    {
        // エラー発生。
        handle_error();
        return;
    }

    while ( s_send_cb_level )
    {
        // アップロード処理を進める
        DWC_ProcessGHTTP();

        GameWaitVBlankIntr();
    }

    if ( DWC_GetLastErrorEx( NULL, NULL ) )
    {
        // エラー発生。
        handle_error();
        return;
    }

    // データアップロード成功
    :
}

// アップロード完了コールバック
void cb_post( const char* buf, int buflen, DWCGHTTPResult result, void*
param)
```

```
{
    s_send_cb_level--;
}
```

コード 9-2　データのアップロード

9.3　データをダウンロードする

HTTPサーバからデータをダウンロードするための関数は、シンプルなDWC_GetGHTTPData関数と、機能拡張版のDWC_GetGHTTPDataEx関数が用意されています（コード 9-3）。

DWC_GetGHTTPDataEx 関数の引数には、データダウンロード先の URL、受信バッファサイズ、ダウンロード完了後に受信バッファを解放するかどうか、通信状況取得コールバック、完了コールバック、コールバック用パラメータを渡します。

受信バッファサイズに 0 が指定された場合は、始め 2048Byte 分のメモリを確保し、受信データのサイズに応じて 2048Byte 分ずつ確保するメモリを追加していき、アプリケーションで確保したヒープ領域の限界までデータを受信することができます。

通信状況取得コールバックを指定した場合は、リクエスト送信中、データ受信中など、ダウンロードシーケンスの状態が変化するたびにコールバックが呼び出されます。また、データ受信中であれば、受信済みのデータサイズも確認できます。

データのダウンロードが完了すれば、完了コールバックが呼び出されます。

ダウンロード完了後に受信バッファを解放する設定を行っていた場合は、完了コールバックを抜けた直後に受信バッファが解放されるため、受信データをコピーして使用してください。

受信バッファを解放しない設定の場合は、GHTTP ライブラリは受信バッファを解放しないので、完了コールバックの引数で渡される受信バッファへのポインタは、アプリケーションで都合の良い時に解放して下さい。受信バッファの解放には DWC_Free 関数を使用してください。

DWC_GetGHTTPData 関数は、DWC_GetGHTTPDataEx 関数の引数 bufferlen を 0、buffer_clear を TRUE、progressCallback を NULL に指定した場合と同じ挙動をします。

ダウンロード開始後の通信処理は全てDWC_ProcessGHTTP関数内で行われるため、毎ゲームフレーム程度の頻度でこの関数を呼ぶようにしてください（コード 9-3）。

```
static char s_recvBuffer[ 2 ][ SIZE_RECV_BUFFER ];
static int  s_get_cb_level = 0;

void get_ghttp_data( void )
{
    // 簡易関数でデータのダウンロードを開始する
    s_get_cb_level++;
    req = DWC_GetGHTTPData( "http://www.test.net", cb_get, GET_TYPE_NORMAL );

    if ( req < 0 )
    {
        // エラー発生。
        handle_error();
        return;
    }

    // 拡張関数でデータのダウンロードを開始する
    s_get_cb_level++;
    req = DWC_GetGHTTPDataEx( "http://www.test.net",
```

```
                                RECV_SIZE, TRUE,
                                NULL, cb_get, GET_TYPE_EX );

    if ( req < 0 )
    {
        // エラー発生。
        handle_error();
        return;
    }

    while ( s_get_cb_level )
    {
        // ダウンロード処理を進める
        DWC_ProcessGHTTP();

        GameWaitVBlankIntr();
    }

    if ( DWC_GetLastErrorEx( NULL, NULL ) )
    {
        // エラー発生。
        handle_error();
        return;
    }

    // データダウンロード成功
    :
}

// ダウンロード完了コールバック
void cb_get( const char* buf, int buflen, DWCGHTTPResult result, void* param)
{
    s_get_cb_level--;

    if ( result == DWC_GHTTP_SUCCESS )
    {
        if ( (int)param == GET_TYPE_NORMAL )
        {
            MI_CpuCopy8( buf, s_recvBuffer[ 0 ], SIZE_RECV_BUFFER );
        }
        else if ( (int)param == GET_TYPE_EX )
        {
            MI_CpuCopy8( buf, s_recvBuffer[ 1 ], SIZE_RECV_BUFFER );
        }
    }
}
```

コード 9-3　データのダウンロード

9.4　GHTTP ライブラリを終了する

DWC_ShutdownGHTTP 関数を呼んで GHTTP ライブラリを終了します。

DWC_InitGHTTP 関数を複数回呼んだ場合は、同じ回数だけ DWC_ShutdownGHTTP 関数を呼び出さなければ、GHTTP ライブラリが確保したメモリが解放されずに残ってしまいます。

10 通信エラー

DWC では、各 DWC モジュールのエラー処理を一括して行うためのシステムを用意しています。

これにより、DWC のエラーとアプリケーションのエラーを同じように扱うことができます。

10.1 エラー処理

DWCのエラーの状態は、DWC_GetLastErrorEx関数で取得することができます（コード 10-1）。

戻り値はエラーの種別、引数はエラーコードとエラー処理タイプ格納先のポインタとなります。

エラーコードは 0 または負の数になります。表示する場合、符号を反転して正の数にした値を表示してください。ただし、復帰可能なエラーで Wi-Fi コネクションから切断しない場合、表示の必要はありません。

エラー処理タイプはエラー発生後に必要な復帰処理を示しており、この値ごとに定型的なエラー処理を作成することができます。

エラー状態に入るとDWCは、ほとんど全ての関数を受け付けなくなります。エラー状態から復帰するためには、DWC_ClearError関数を呼び出してください（コード 10-1）。

```
void main_loop( void )
{
    while ( 1 )
    {
        DWC_ProcessFriendsMatch();

        handle_error();  // エラー処理

        GameWaitVBlankIntr();
    }
    :
}

// エラー処理
void handle_error( void )
{
    int dwcError, gameError;

    dwcError  = handle_dwc_error();
    gameError = handle_game_error();
    :
}

int handle_dwc_error( void )
{
    int errcode;
    DWCError err;
    DWCErrorType errtype;

    // エラーを取得する
    err = DWC_GetLastErrorEx( &errcode, &errtype );

    // エラーがなければ何もせずに戻る
    if ( err == DWC_ERROR_NONE ) return 0;
```

```
    // エラーをクリア
    DWC_ClearError();

    // エラーメッセージを表示する
    disp_error_message( err );
    // エラーコードが0以下なら正の数にして表示する
    if ( errcode <= 0 ) disp_message( "%d", -1*errcode );

    if ( errtype == DWC_ETYPE_SHUTDOWN_FM )
    {
        // FriendsMatch処理を終了する
        DWC_ShutdownFriendsMatch();
    }
    else if ( errtype == DWC_ETYPE_DISCONNECT )
    {
        // FriendsMatch処理の終了とインターネット接続のClearupを行なう
        DWC_ShutdownFriendsMatch();
        disconnect_func();
    }
    else if ( errtype == DWC_ETYPE_FATAL )
    {
        // Fatal Errorなので電源切断を促した後は何もできなくする
        while (1) ;
    }
    // 軽いエラーの場合はエラークリアだけでFriendsMatch処理を再開できる

    return err;
}
```

コード 10-1　エラー処理

10.2　エラーコード一覧

マッチメイク／友達関連処理において発生する主要なエラーコードは下記の通りです。

下三桁が 010, 020 のエラーは、GameSpy 社のサーバがメンテナンスなどで不安定な状態にある時に発生しやすくなっています。

- 61010　GameSpy 社の GP サーバへのログイン中に、GP サーバとの間で通信エラーが発生しました。
- 61020　GameSpy 社の GP サーバへのログイン中に、GP サーバとの間で通信エラーが発生しました。
- 61070　GameSpy 社の GP サーバへのログイン中に、ログインのタイムアウトエラーが発生しました。
- 71010　友達リスト同期処理中に、GameSpy 社の GP サーバとの間で通信エラーが発生しました。
- 80430　サーバクライアントマッチメイクのクライアント DS において、接続先のサーバ DS、もしくはそれに接続されているクライアント DS が電源切断するなどして、接続に失敗しました。
- 81010　マッチメイク中に GameSpy 社の GP サーバとの間で通信エラーが発生しました。
- 81020　マッチメイク中に GameSpy 社の GP サーバとの間で通信エラーが発生しました。
- 84020　マッチメイク中に GameSpy 社のマスターサーバからの通信が途絶えました。マスターサーバがダウンしているか、ファイヤウォールが UDP をブロックしています。
- 85020　マッチメイク中に GameSpy 社のマスターサーバとの間で通信エラーが発生しました。
- 85030　マッチメイク中に GameSpy 社のマスターサーバの DNS に失敗しました。下三桁が 030 のエラーは全て DNS エラーを表しています。
- 86420　1 回のマッチメイク中に一定回数 NAT ネゴシエーションに失敗しました。ルータなどに問題がある可能性があります。サーバクライアントマッチメイクでは接続を開始したクライアント DS のみに発生し、1 回でも NAT ネゴシエーションに失敗するとこのエラーになります。
- 97003　マッチメイク完了後、DWC より低位のレイヤーでソケットエラーが発生しました。

下四桁が 1010, 1020 のエラーと 85020 のエラーは、NitroWiFi 1.0 RC2 以前の WiFi ライブラリにおいて GameSpy 社のサーバとの間の TCP の送信が滞ると多発することが確認されています。

11 ネットワークストレージ対応

DWC では、GameSpy 社によってネットワーク上に用意されているストレージサーバに、データを格納することができます。

このストレージサーバにアクセスするには、まずDWC_LoginAsync関数によるログインまでを完了させておく必要があります。その上で、更にストレージサーバへのログインをDWC_LoginToStorageServerAsync関数を用いて行います（コード 11-1）。

ストレージサーバにセーブできるデータは、Public か Private の属性を持たせることができます。

DWC_SavePublicDataAsync関数を用いてデータをセーブすれば、Public属性となり、他のプレイヤーからも参照できるデータとなります（コード 11-1）。

一方、DWC_SavePrivateDataAsync関数を用いてデータをセーブすれば、Private属性となり、他のプレイヤーからは参照できないデータとなります（コード 11-1）。

ストレージサーバからデータをロードする場合は、自分の Public データをロードするDWC_LoadOwnPublicDataAsync関数とPrivateデータをロードするDWC_LoadOwnPrivateDataAsync関数、友達リスト内の友達のデータをロードするDWC_LoadOthersDataAsync関数を使うことができます（コード 11-1）。友達は友達リストのインデックスで指定します。

セーブ・ロード完了時には、DWC_SetStorageServerCallback関数で設定したコールバック関数が呼び出されます（コード 11-1）。

各コールバック関数は必ずセーブ・ロード関数を呼び出した順に呼び出されます。

セーブデータとして指定できるのは、key/value の組み合わせの文字列で、"¥¥name¥¥mario¥¥stage¥¥3"のように'¥¥'で区切ってキーとその値の組を繰り返します。

このデータを指定した場合、ストレージサーバ上のデータベースには、"name"というキーの値が"mario"、"stage"というキーの値が"3"として、全て文字列で登録されます。

データをロードする時は、取得したいキーを"¥¥name¥¥stage"のように'¥¥'で区切って指定します。

この場合、ロードコールバックで取得できる文字列は"¥¥name¥¥mario¥¥stage¥¥3"という形になります。

存在しないキーや、友達が Private 属性でセーブしたキーのみをロードしようとした場合は、コールバックの引数 success は FALSE になります。ただし、複数のキーを指定したうちの、一部だけが上記のようなキーであった場合は、ロードデータにそのキーが含まれないだけで、引数 success は TRUE になります。

ストレージサーバの処理が全て終了したら、DWC_LogoutFromStorageServer関数でストレージサーバからログアウトしてください（コード 11-1）。

```
static int  s_cb_level = 0;
static BOOL s_storage_logined = FALSE;

void access_net_storage( void )
{
    // ストレージサーバへのログイン
    if ( !DWC_LoginToStorageServerAsync( cb_storage_login, NULL ) )
    {
        OS_TPrintf( "DWC_LoginToStorageServerAsync() failed.¥n" );
        return;
```

```
    }

    // ストレージサーバへのログイン完了待ち
    while ( !s_storage_logined )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // エラー発生。
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // ストレージサーバへのセーブ・ロード完了コールバック設定
    DWC_SetStorageServerCallback( cb_save_storage, cb_load_storage );

    // パブリックデータをセーブする
    s_cb_level++;
    if ( !DWC_SavePublicDataAsync( "¥¥name¥¥mario¥¥stage¥¥3", NULL ) )
    {
        OS_TPrintf( "DWC_SavePublicDataAsync() failed.¥n" );
        return;
    }

    // プライベートデータをセーブする
    s_cb_level++;
    if ( !DWC_SavePrivateDataAsync( "¥¥id¥¥100", NULL ) )
    {
        OS_TPrintf( "DWC_SavePrivateDataAsync() failed.¥n" );
        return;
    }

    // セーブ完了待ち
    while ( s_cb_level > 0 )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // エラー発生。
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // 自分のPublicセーブデータをロードする
    s_cb_level++;
    if ( !DWC_LoadOwnPublicDataAsync( "¥¥name", NULL ) )
    {
        OS_TPrintf( "DWC_LoadOwnPublicDataAsync() failed.¥n" );
        return;
    }

    // 自分のPrivateセーブデータをロードする
    s_cb_level++;
    if ( !DWC_LoadOwnPrivateDataAsync( "¥¥id", NULL ) )
    {
```

```
        OS_TPrintf( "DWC_LoadOwnPrivateDataAsync() failed.¥n" );
        return;
    }

    // 他人のセーブデータをロードする
    s_cb_level++;
    if ( !DWC_LoadOthersDataAsync( "¥¥name", 0, NULL ) )
    {
        OS_TPrintf( "DWC_LoadOthersDataAsync() failed.¥n" );
        return;
    }

    // ロード完了待ち
    while ( s_cb_level > 0 )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // エラー発生。
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // ストレージサーバからのログアウト
    DWC_LogoutFromStorageServer();
    :
}

// ストレージサーバ  ログイン完了コールバック
void cb_storage_login( DWCError error, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        s_storage_logined = TRUE;
        s_cb_level        = 0;
    }
}

// ストレージサーバ  セーブ完了コールバック
void cb_save_storage( BOOL success, BOOL isPublic, void* param )
{
    OS_TPrintf( "result %d, isPublic %d.¥n", success, isPublic );
    s_cb_level--;
}

// ストレージサーバ  ロード完了コールバック
void cb_load_storage( BOOL success, int index, char* data, int len, void*
param )
{
    OS_TPrintf( "result %d, index %d, data '%s', len %d¥n",
                success, index, data, len );
    s_cb_level--;
}
```

コード 11-1　ストレージサーバへのアクセス