NITRO-DWC Programming Manual

Nintendo Wi-Fi Connection

Version 1.4.1

The contents in this document are highly confidential and should be handled accordingly.

NTR-06-0429-001-C
Released: September 12, 2006

Table of Contents

Code

Tables

Figures

Revision History

Version	Revision Date	Description
1.4.1	08/09/2006	• Revised text within 8.3, Yardstick for Buffer Size Specified by `DWC_InitFriendsMatch` and Table 8 2, Communication Data Breakdown.
1.4.0	06/19/2006	• Changed the conditions for displaying error codes.
1.3.0	06/06/2006	• Revised the section "Examples of When a Temporary Login ID May be Duplicated" in 2.1.3 Player Information by Game: Login ID. • Changed the memory size to 230 kbytes from 200 kbytes in Chapter 3, Initializing NITRO-DWC. • Added 7.6 Names that Cannot be Used for Matchmaking Index Keys • Miscellaneous changes (unified terminology, made corrections, etc.)
1.2.0	3/10/2006	• Added “2 User Management Under NITRO-DWC” • Added “7.5 Increasing Matchmaking Speed” • Added “8.4 Amount of Data Sent/Received” • Miscellaneous changes (review of text, changes in terminology, etc.)
1.1.0	1/30/2006	• Updated “Code 5-3 Synchronizing Friend Rosters” • Corrected error in “Code 5-4 Friend Information Types” (“stablished” -> “established”) • Corrected error in “Code 6-3 Evaluating Candidate Players for Matchmaking” (“anymatch” -> “anymatch test”) • Changed data load function in “11 Accessing the Storage Server” to a newly added function.
1.0.0	12/28/2005	Initial version.

1 Introduction

The NITRO-DWC library (DWC library) is designed with the goal of making the Nintendo Wi-Fi Connection "easy to use, free of worries, and free of charge.” Specific benefits include:

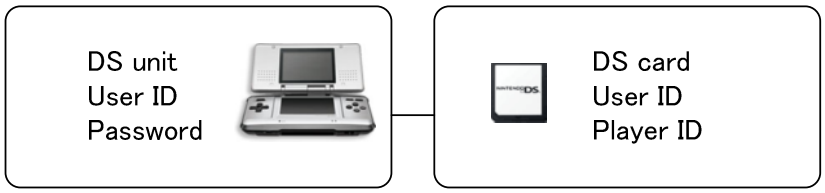
- Making it easy to connect by sheltering users from complicated and detailed Internet settings.
- Making it easy to communicate with friends with whom friendships were established by using wireless communications or by exchanging friend registration keys when not connected to the Internet.
- Making it easy to remain secure by ensuring that one user cannot easily access another user’s Internet-related information when a DS changes hands.

2 User Management Under NITRO-DWC

2.1 Managing Wi-Fi User Information

Information required for Nintendo Wi-Fi Connection authentication (Wi-Fi authentication) includes a User ID, Player ID, and password. This information is managed while treating the DS unit and DS card as a pair. (Figure 2-1)

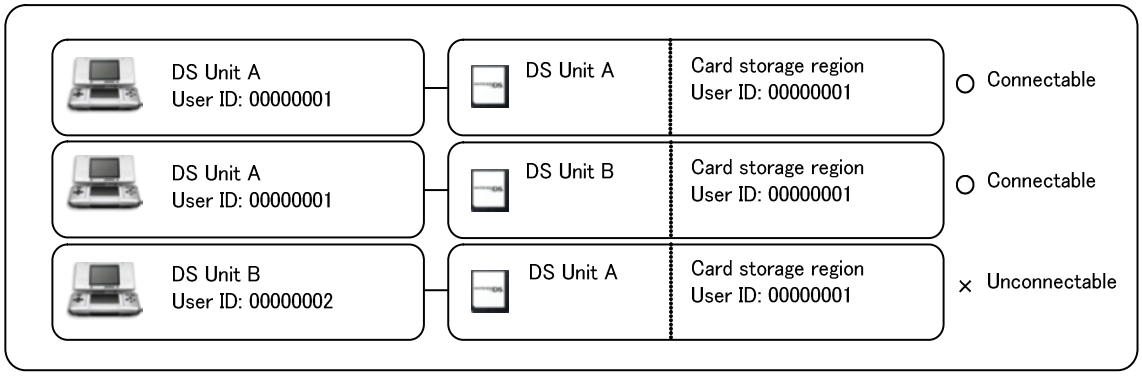
Figure 2-1 Save State of the User ID on the DS and DS Card



- The User ID and password used for Wi-Fi authentication are saved on the DS unit.
- The User ID and Player ID used for Wi-Fi authentication are saved on the DS card.

This information is used by the Nintendo Wi-Fi Connection for authentication. If the User ID saved on the DS card differs from the User ID saved on the DS unit, data saved on the Nintendo Wi-Fi Connection cannot be accessed. This prevents the unauthorized access of data. (Figure 2-2)

Figure 2-2 Using Multiple Nintendo DS systems and DS Cards



2.1.1 User ID and Player ID

The User ID is generated offline and is designed to be as unique as possible. After it is generated, it becomes the User ID for connecting to the Internet, authenticating, and registering with the system. If the ID is found to already be in use during authentication, a new, unique User ID will be assigned.

To ensure that the User ID is unique, part of the DS Unit's MAC address is used. Although this

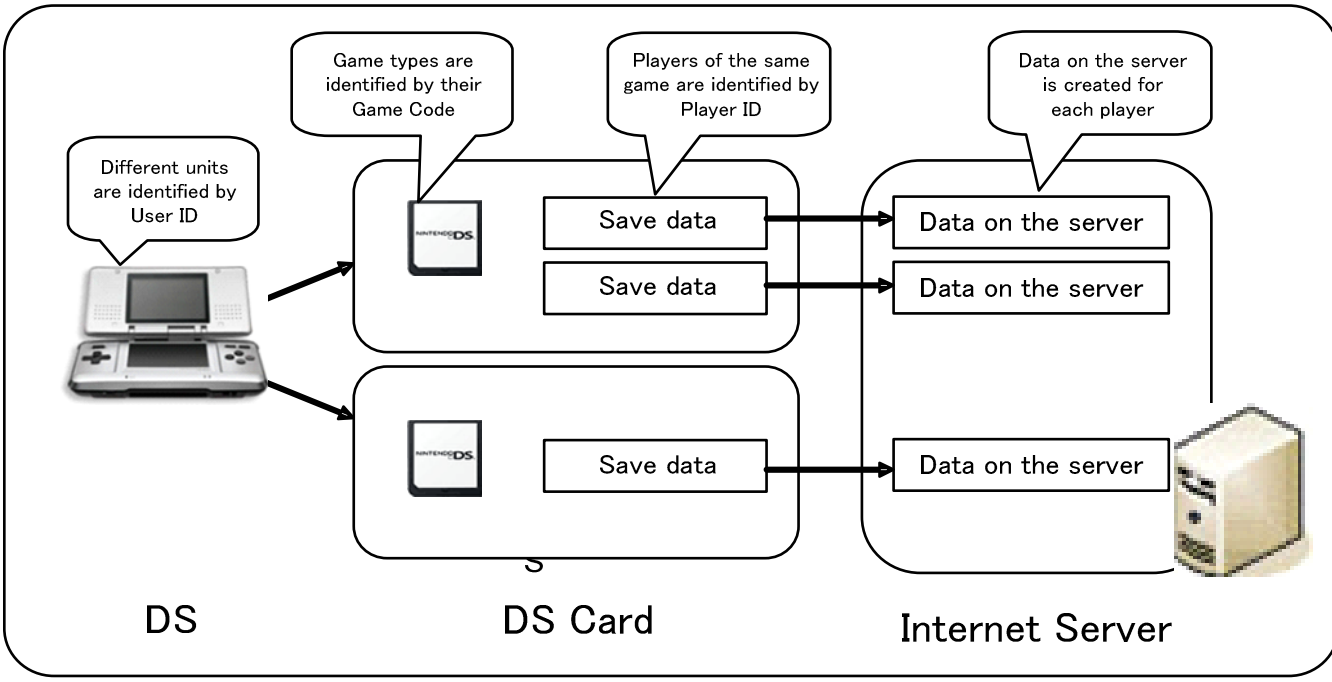
prevents the same User ID from being used on different DS units, duplication might occur when a User ID is moved[1] or regenerated.

The Player ID is a random 32-bit ID. Because data on the Internet server is managed using the combined User ID, Player ID, and Initial Code, a Player ID only needs to be unique with respect to the User ID and Initial Code. If the Player ID is duplicated, a unique Player ID will be assigned during authentication.

2.1.2 The Difference Between a User ID and Player ID

Because a User ID is issued to each DS, a user that uses the same DS must use a single User ID for all games. Since Player IDs are issued to DS Cards, you can use different Player IDs when using the same DS (User ID) and the same Game Code (Figure 2-3).

Figure 2-3 How Data is Stored on the Internet

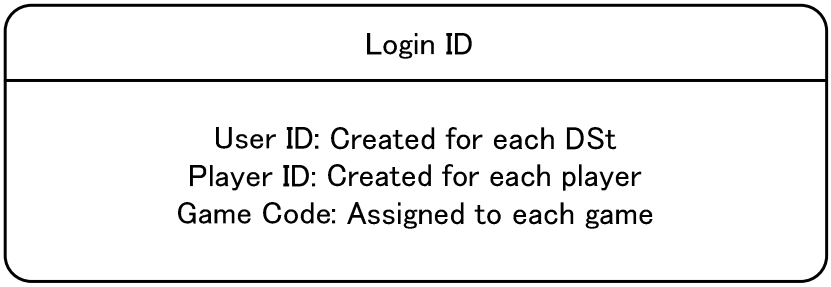


2.1.3 Player Information by Game: Login ID

The combined User ID + Player ID + Initial Code are called the “Login ID” (Figure 2-4). User information saved on the Internet server is called a “Profile,” while the ID used to manage profiles on the server is called a “Profile ID.”

[1] User Information stored on the DS can be moved using the “Nintendo Wi-Fi Connection Setup” feature provided by DWC.

Figure 2-4 Configuration of a Login ID



Inside the DWC library, the Login ID or Profile ID is used to search for the profiles of other users on the Internet server.

The Login ID is generated when not connected to the Internet and becomes a temporary Login ID. Although a user is likely to use this Login ID as is, it might not be available. In this case, a unique, approved Login ID (authenticated Login ID) is generated. There is a one-to-one correspondence between authenticated Login IDs and assigned Profile IDs.

A temporary Login ID may be duplicated under the following circumstances:

- The Login ID is created with a User ID that was not authenticated, the same User ID already is registered in the Authentication server by another person, and the Login ID was created with the same player ID for the same game.
- Multiple DS systems created Login IDs with the same player ID for the same game using the same unauthenticated User ID.

2.1.4 Information for Wi-Fi Authentication Saved by Games

Games must save this information for Wi-Fi authentication as backup on the DS Card.

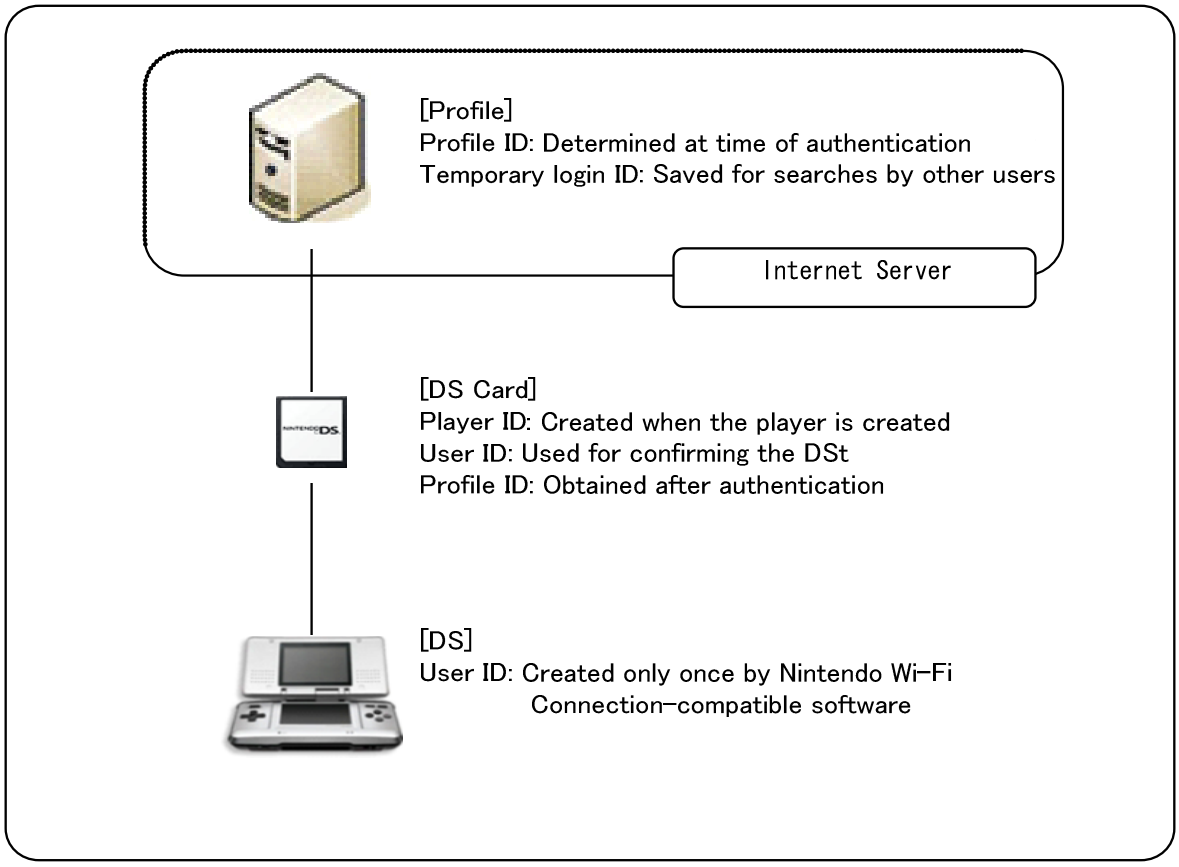
The size of the information used for authentication is 64 bytes.

The Wi-Fi authentication information includes the temporary Login ID, the authenticated Login ID, and the Profile ID. Developers do not need to fully understand the details because this information is created and updated by the DWC library.

Information for Wi-Fi authentication must also be saved for each player when multiple players can use the same DS Card.

Figure 2-5 shows the Wi-Fi authentication terminology covered so far.

Figure 2-5 Comprehensive Diagram of Terminology for Wi-Fi Authentication



2.2 Friend Management Overview

2.2.1 Building Friend Relationships

To be able to easily start communication with friends using DWC, friend relationships are built by an Internet server. Friendships are built by exchanging user information. Established friendships are saved in the profile of each user.

There are two methods of exchanging the user information used to create a friendship.

- Using DS Wireless Communication
  Using this method, the players exchange Login or Profile IDs. The Login ID is used if the player in question has never logged in before. Even though each of these was created locally, it is highly likely that they are unique but not guaranteed. However, because the probability of duplication is less than 2^{-75}, no special countermeasure against duplication is required. The Profile ID is used for players who have logged in at least once before. This creates friendships with certainty, because a particular party can always be specifically identified.

- Exchanging Friend Registration Keys
  Using this method, the players exchange friend registration keys included in the Profile ID as information used for error checking. A player must have connected to the Internet at least once to use a Profile ID. You must create an interface that allows input to be confirmed and re-entered in case it is incorrectly input.

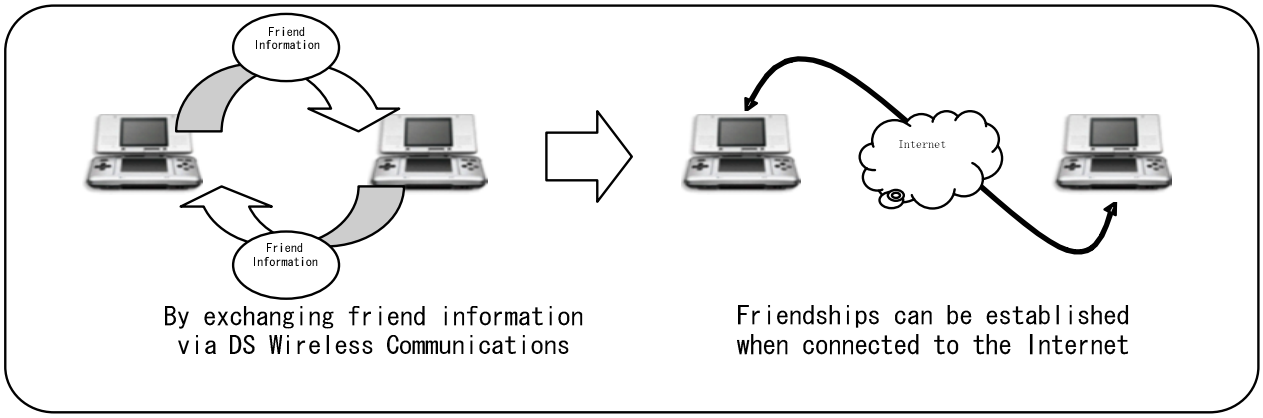
The information exchanged can be created using DWC. DWC includes functions for automatically creating the most applicable information possible based on information used for Wi-Fi authentication saved on the DS Card.

2.2.2 Building Friendships Using DS Wireless Communications

A mechanism is provided that allows friendships to automatically be established later on the Internet when information is exchanged with another party during DS Wireless Communications. The information exchanged is created from the Login ID or Profile ID included in user data.

Note: The exchange of this information via DS Wireless Communications is not supported by DWC.. Be sure that applications handle the exchange of created information.

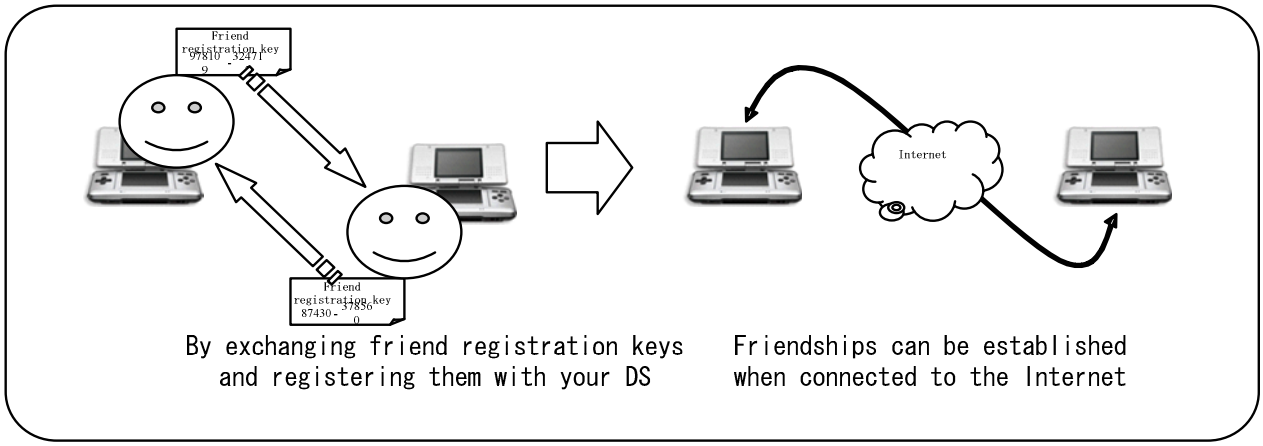
Figure 2-6 Creating Friendships Using DS Wireless Communications



2.2.3 Building Friendships Using Friend Registration Keys

The term “Friend Registration Key” refers to information that can be used to specifically identify another user when establishing a friendship. A mechanism is provided that allows friendships to be created by exchanging this friend registration key (see Figure 2-7). Because the friend registration key is manually entered by users, it should not be unnecessarily long. It is created using the Profile ID obtained by connecting at least once to the Internet rather than using the Login ID.

Figure 2-7 Creating Friendships Using Friend Registration Keys



The friend registration key is a 12-digit number.

Pay attention to the following points when developing games.

- You must create a user interface for issuing friend registration keys. Since a key cannot be issued unless a player connects to the Internet at least once, a message to this effect must be displayed.
- You must create a user interface for entering the friend registration key. The user interface must allow the user to correct an incorrectly input friend registration key. It should also allow users to save and edit the entered data as many times as necessary.

2.2.4 Friend Information Saved by Games

Games must save exchanged friend information for the maximum number of players to be managed as friends in a backup area. This is required so users can edit friendships when they are not connected to the Internet. Friend-related information used by the actual game (such as nicknames, win-loss record, etc.) must also be saved. DWC treats all of this as friend information without regard to the type of data (Login ID, Profile ID, and friend registration key).

To store friend information used by DWC, 12 bytes per player are required.

2.3 Exception Handling

2.3.1 Removing the Association Between a DSt and a DS Card

For security reasons, Nintendo Wi-Fi Connection treats the DS and DS Card as a set. This can be inconvenient for a user if the DS is resold or broken as the ability to connect to Nintendo Wi-Fi Connection is lost.

To solve this problem, there is a DWC mechanism that allows the user to delete the data that associates a DS Card with a given DS by destroying information stored in the profile. Because this deletes all Internet friendships, you must create an interface to warn the user before deleting the data.

Even if Internet friendships are deleted, friend information for other parties remains on the DS Card of the deleted user. This allows friendships to be restored by using this information and sending a new friend registration key to the other party. Since it is necessary in these cases to prompt to the user to register the deleted user as a friend again, each application needs to include a message for notifying the user of the required procedure.

With regard to specific processing, the currently saved association on the DS Card is deleted. If a user wants to create a new association, it must be handled by creating new user data and destroying the previous user data. Furthermore, even if user data is updated, friendships on the friend roster saved on the DS Card remain established. If a specification where the friend roster remains intact is used, be sure to clear the friendship established flag included in the friend information when letting the user know that friendships remain established.

Note: Refer to the flow diagram in the Nintendo Wi-Fi Connection Programming Guidelines.

3 Initializing Nitro DWC

Before calling any of its library functions, you must initialize the DWC library as shown in Code 3-1 using the `DWC_Init` function that performs the following processes:

- Generates information for user authentication stored in the DS
- Checks if the connection target information stored in the DS's backup memory is valid

Also use the `DWC_SetMemFunc` function to configure the Internet and Nintendo Wi-Fi Connection and the functions that allocate and free internal memory used for matchmaking and friend relationship processing. (These topics are covered in Chapter 4, Creating User Data, and subsequent chapters.)

For four player matchmaking, the DWC library requires approximately 230 kilobytes of memory. Removing one player from the maximum matchmaking number reduces the required memory by approximately 20 kilobytes. (This is true for when the `sendBufSize` and `recvBufSize` arguments of the `DWC_InitFriendsMatch` function are both set to the default value of 8 kilobytes.)

Code 3-1 DWC Initialization

```
void init_dwc( void )
{
    u8 work[ DWC_INIT_WORK_SIZE ] ATTRIBUTE_ALIGN( 32 );
    // Initialize the DWC library

    if ( DWC_Init( work ) == DWC_INIT_RESULT_DESTROY_OTHER_SETTING )
        disp_init_warning_msg();  // Display warning message

    // Set functions for allocating and freeing memory
    DWC_SetMemFunc( AllocFunc, FreeFunc );
    :
}
// Function for allocating memory
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;
}
```

```
// Function for freeing memory
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 );
}
```

To read more about user authentication and other related topics, see the *Nintendo Wi-Fi Connection Guidelines*.

4 Creating User Data

The DWC library performs typical processes based on user data:

- Authenticating users
- Creating friend relationships

Even when the DS is not connected to the Internet, it requires user data to create the friend information that is exchanged to create friend relationships via wireless communication.

If user data is not yet created or the user data is damaged, create the user data with the `DWC_CreateUserData` function and store the user data in the DS Card backup memory.

Be sure the application allocates memory for saving the `DWCUserData` structure. User data for several people is required when a single DS Card supports multiple players.

If player data is already created, be sure to check its validity using the `DWC_CheckUserData` function after loading it from backup into memory (Code 4-1).

Code 4-1 Creating User Data

```
BOOL create_userdata( void )
{
   // If there is backup data and user data in that backup data, load all and
   // return TRUE.
   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;
       }
   }

   // If valid user data has not been saved
   OS_TPrintf("no Backup UserData\n");

   // Create user data
   DWC_CreateUserData( &s_PlayerInfo.userData, DTUD_INITIAL_CODE );
```

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

    return FALSE;
}
```

The DWC_CheckDirtyFlag function can be used to check whether it is necessary to save user data to the DS Card. Always use the DWC_ClearDirtyFlag function to clear DirtyFlag before saving the user data to backup memory as shown in Code 4-2.

Code 4-2 Saving User Data

```
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) );
    }
}
```

Before connecting to the Internet, be sure to check user data according to the following procedure:

- Connect to the Internet and get the user profile using the DWC_CheckHasProfile function. If the profile cannot be obtained, user data is updated and the DS and DS Card are treated as a set.
- Check whether the DS and DS Card are being used correctly using the DWC_CheckValidConsole function. It is impossible to connect to the Internet if the DS and DS Card are not correct because authentication will fail.

Note: Be sure to check flow charts included in NINTENDO_Wi-Fi_Connection_Programming_Guidelines.pdf.

5 Connection Process

The DWC library performs a two-phase process when connecting to the Internet:

- Connects to the Internet (making a Wi-Fi connection to get an IP address)
- Connects to the Nintendo Wi-Fi Connection server (referred to as "server")

When a DS connects to the Internet for the first time, the Nintendo Authentication server issues a user ID for that DS. This user ID is stored in the DS backup memory.

After this initial connection is established, the DWC library stores this user ID and the player ID in the previously created user data to generate a profile. The GS profile ID that corresponds to this generated profile is stored in the user data.

5.1 Connecting to the Internet

When the DS first connects to the Internet to obtain the IP address, Nintendo's authentication server issues a user ID to that DS. Tests are also performed to confirm that the DS can connect to the connection test server using TCP communication and that the Internet connection is functioning normally.

All these processes are performed automatically by calling the `DWC_*Inet` functions as shown in Code 5-1.

Code 5-1 Connecting to the Internet

```
static DWCInetControl s_ConnCtrl;  // Retain until the Internet connection is
disconnected
BOOL connect_to_inet( void )
{
   // Initialization process for Internet connection
   DWC_InitInet( &s_ConnCtrl );

   // Start establishing connection
   DWC_SetAuthServer( DWC_CONNECTINET_AUTH_RELEASE );
   DWC_ConnectInetAsync();

   // The connection process
   while ( !DWC_CheckInet() )
   {
      DWC_ProcessInet();

      // V-Blank wait process
```

```
        // During the connection process you need to pass the
        // process time to threads that have lower priority than
        // the main thread. Use the OS_WaitIrq function for this.
        GameWaitVBlankIntr();
    }

    // Confirm the connection result
    if ( DWC_GetInetStatus() != DWC_CONNECTINET_STATE_CONNECTED )
    {
        handle_error();
        return FALSE;
    }
    // Connected
    :
}
```

5.2 Disconnecting from the Internet

Call the `DWC_CleanupInet*` functions as shown in Code 5-2 to disconnect the DS from the Internet.

Even if a communication error occurs and the DS is disconnected automatically, you must call this function because the library memory needs to be freed.

Code 5-2 Disconnecting from the Internet

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

5.3 Connecting to the Nintendo Wi-Fi Connection Server

To connect to the Nintendo Wi-Fi Connection server, use the `DWC_InitFriendsMatch` function shown in Code 5-3 to initialize matchmaking and friend relationship features.

The arguments to this function are:

- Pointers to the control objects of these features

- User data
- Product ID
- Game name and secret key provided by GameSpyS
- Send and receive buffer sizes used for communication between Nintendo DS systems
- Friend roster
- Maximum number of friends in the friend roster

The specified control objects are used in the DWC library until the `DWC_ShutdownFriendsMatch` function is called.

Chapter 8 talks about the sizes of the Send and Receiver buffers in detail. When 0 is specified, as is the case in the sample program below, the buffers use 8 kbytes by default.

The friend roster is an array of friend information in the `DWCFriendData` structure. Chapter 6, Creating Friend Rosters and Information, discusses friend rosters and friend information in detail.

Next, call the `DWC_LoginAsync` function to make the connection to the server (see Code 5-3).

The first argument of this function is the player's screen name. If players use names in your game application, you must specify the screen name in this argument. The screen name used in the game is sent to the authentication server to confirm and check for inappropriate names.

You can check the results of this function by calling the `DWC_GetIngamesnCheckResult` function (see Code 5-3).

The second argument of the `DWC_LoginAsync` function is not currently used. Pass `NULL` for this argument. The remaining arguments represent the callback to use after login completes and the parameters of the callback.

After calling this function, call the `DWC_ProcessFriendsMatch` function repeatedly to advance the login process, approximately once per game frame.

Next, the `DWC_ProcessFriendsMatch` function executes all matchmaking and friend-related processing until the `DWC_ShutdownFriendsMatch` function is called. After login completes, be sure to call `DWC_ProcessFriendsMatch` function to make sure that network processes (for example, updating the friend roster) do not start while the DS is connected to another client.

Code 5-3 Connecting to the Nintendo Wi-Fi Connection Server

```
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 );

    // Login using function for authentication
    s_logined = FALSE;
    if ( !DWC_LoginAsync( L"name", NULL, cb_login, NULL ) )
    {
        // Connection process fails to start.
        return;
    }

    // Polling to see if connected
    while ( !s_logined )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // Error occurs
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // Connection process completed
    if ( DWC_GetIngamesnCheckResult() == DWC_INGAMESN_INVALID )
    {
        // Special process performed when inappropriate in-game screenname was detected
        disp_ingamesn_warning();
    }
    :
}

// Callback when logged in
void cb_login( void )
{
    if (error == DWC_ERROR_NONE)
    {
        check_and_save_userdata();
```

```
        s_logined = TRUE;
    }
}
```

The DWC_ShutdownFriendsMatch function ends the matchmaking and friend relationship features and frees the memory reserved internally by the library.

When the DS connects to the server for the first time using the user data specified by the DWC_InitFriendsMatch function, the DS and the DS Card are treated as a pair. When they are treated as a pair, the DS Card that stores the specified user data cannot be used with another DS to connect.

Furthermore, the user data is always updated when the first connection is made. Once the login completes, the application should call a login callback and the DWC_CheckDirtyFlag function to check the updated user data. If necessary, save the updated data to the DS Card.

6 Creating Friend Rosters and Information

The DWC library has two procedures for establishing friend relationships among players:

- Exchanging friend information using DS Wireless Communications
- Exchanging friend registration keys

6.1 Exchanging Friend Information via DS Wireless Communications

During DS Wireless Communications, the `DWC_CreateExchangeToken` function is used to create friend information based on the local user data for exchange with other players (shown in Code 6-1).

Friend information that the DS receives should be saved in the friend roster using the application.

Code 6-1 Exchanging Friend Information Using DS Wireless Communications

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

// Exchange friend information
void exchange_friend_data( void )
{
   int i, j;

   DWCFriendData ownFriendData;
   DWCFriendData recvFriendList[ FRIEND_LIST_LEN ];

   // Create friend information from local user data to send
   DWC_CreateExchangeToken( s_userData, &s_friendData );

   // Send & receive friend information via MP communication
   MP_start( (u16 *)&s_friendData, (u16 *)recvFriendList );
   :

   // Save the received friend information in an open slot
   //  in the friend roster.
   // Do not save if the same friend information already exists.
   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 ] ) )
```

```
            {
                /* If the friend roster has valid data, check if it is the same as
                   the received friend information and do not save if it is the
                   same. */
                if ( DWC_IsEqualFriendData( &recvFriendList[ i ],
                                            &s_friendList[ j ] ) )
                    break;
            }
            else
            {
                // Records an available friend roster index
                if ( index == -1 ) index = j;
            }
        }

        // Save valid friend information that does not overlap in friend roster
        if ( j >= FRIEND_LIST_LEN && index >= 0 )
        {
            s_friendList[ index ] = recvFriendList[ i ];
        }
    }
    :
}
```

6.2 Exchanging Friend Registration Keys

A player that has connected at least once to Nintendo Wi-Fi Connection is assigned a GS profile ID that is saved in the user data. Any player that has a GS profile ID can create a friend registration key that adds special error checking information to the GS profile ID. This friend registration key is a 12-digit decimal number that players can exchange. Once this friend registration key has been entered, friend data can be exchanged.

After the friend registration key is entered, the `DWC_CreateFriendKeyToken` function is called to convert the key into friend information and save the friend information to the friend roster. (See Code 6-2.)

Use the `DWC_CheckFriendKey` function to check if the entered friend registration key is valid as shown in Code 6-2. Even if this function is called, the error does not correct itself, so prepare a user interface so that the user can enter the key until the key information is correct.

Code 6-2 Exchanging Friend Registration Keys

```
// Display Friend Registration Key
void disp_friend_key( void )
{
   u64 friend_key;

   // Create friend registration key from local user data
   if ( ( friend_key = DWC_CreateFriendKey( &s_userData ) ) != 0 )
   {
      // Display friend registration key
      disp_message( "FRIEND CODE : %lld", friend_key );
   }
   else
   {
      // Display message that there is no friend registration key
      disp_message( "FRIEND CODE : not available" );
   }
   :
}

/* Create friend information from friend registration key and register in
   friend roster */
BOOL register_friend_key( void )
{
   u64  friend_key;
   DWCFriendData friendData;

   while ( 1 )
   {
      char friend_key_string[ 13 ];

      // Get user to manually enter friend registration key
      input_friend_key( friend_key_string );

      /* Convert entered friend registration key string into u64 numerical
         value */
      friend_key = charToU64( friend_key_string );

      // Check that friend registration key is correct and proceed if OK.
      // If there is a problem, display message and have it entered again
      if ( DWC_CheckFriendKey( s_userData, friend_key ) ) break;
```

```
        else disp_warning_message();
    }

    // Create Friend information from correct Friend Registration Key
    DWC_CreateFriendKeyToken( &friendData, friend_key );

    {
        int index;
        /* Using same method as MP communication, search for open slot and
           overlaps in friend roster and register friend information. */
        :
        s_friendList[ index ] = friendData;
        :
    }
}
```

6.3 Synchronizing Friend Rosters

For a friend roster stored in the application (local friend roster) to be valid on the Internet, you need to call the `DWC_UpdateServersAsync` function and update the friend roster stored on the GameSpy server (server friend roster) as shown in Code 6-3.

To synchronize the friend rosters, you must first complete the login process with the `DWC_LoginAsync` function.

Specify the following function arguments: the player name (the old specification — specify NULL); the callback and its parameters when the friend roster completes synchronization; the callback and its parameters for a change notification in friend status (discussed later); and the callback and its parameters when the friend roster is deleted.

The friend roster synchronization process involves two main tasks: sending requests to establish friend relationships for friends that are on the local but not the server friend roster and deleting friends that are on the server but not the local friend roster.

If a request to establish a friend relationship is sent while the other party is offline, call the `DWC_LoginAsync` function to save the request on the server and immediately deliver the request the next time the contacted partner logs in. The friend relationship is only established after the information is saved in the local friend roster of the other party.

Note that this process only registers the other party as your friend. When the other party receives the request to establish a friend relationship, the contacted partner follows the same process to register the initiating partner as a friend.

After the friend roster synchronization process completes, the callback is called after the local and server friend rosters are checked, needed requests to establish friend relationships are sent, and unneeded friend information is deleted. Be aware that even if the callback has returned, this state does not indicate that all friend relationships are established. If the `isChanged` argument of the callback is set to TRUE, this indicates that the friend information in the local friend roster is updated and needs to be saved. If a friend relationship is established at a time other than during the friend roster synchronization process, the callback for an established friend relationship specified by the `DWC_SetBuddyFriendCallback` function is called.

If multiple sets of friend information for the same friend are discovered during the friend roster synchronization process, all but one set are automatically deleted. A callback is called for each deleted set by comparing the friend roster index of the deleted friend information and the friend roster index of the matching friend.

Code 6-3 The Friend Roster Synchronization Process

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

void sync_friend_list( void )
{
   // Set the callback for establishment of friend relationship
   DWC_SetBuddyFriendCallback( cb_buddyFriend, NULL );

   // Synchronize local Friend roster and server Friend roster
   if ( !DWC_UpdateServersAsync( NULL,
                                 cb_updateServers, NULL,
                                 NULL, NULL,
                                 cb_deleteFriend, NULL ) )
   {
      // Synchronization process fails to start
      return;
   }

   while ( !s_update )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // Error generation
         handle_error();
         return;
```

```
        }

        GameWaitVBlankIntr();
    }
    :

    while ( 1 )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // Error generation
            handle_error();
            return;
        }

         // To update the friend list asynchronously, perform the following
        // processing when appropriate and collect the updated local friend list
        // and save.
        if ( s_updateFriendList)
        {
            // Save the friend list if it has been updated
            s_updateFriendList = FALSE;
            save_friendList();
        }

        game_loop();

        GameWaitVBlankIntr();
    }
    :
}

// Callback for when Friend roster synchronization has completed
void cb_updateServers( DWCError error, BOOL isChanged, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        // Friend roster synchronization successful
        s_update = TRUE;
```

```
        // Must be saved if Friend roster has been changed
       if ( isChanged ) s_updateFriendList = TRUE;
    }
}

// Callback for when there is a Friend roster deletion
void cb_deleteFriend( int deletedIndex, int srcIndex, void* param )
{
    OS_TPrintf( "friend[%d] was deleted (equal friend[%d]).\n",
                deletedIndex, srcIndex );
    s_updateFriendList = TRUE;
}

// Callback for when friend relationship has been established
void cb_buddyFriend( int index, void* param )
{
    OS_TPrintf( "Got friendship with friend[%d].\n", index );
    s_updateFriendList= TRUE;
}
```

6.4 Obtaining Friend Information Types

Code 6-4 shows how you can obtain the data type set in the friend information using the `DWC_GetFriendDataType` function.

The possible data types are:

- `DWC_FRIENDDATA_NODATA` No stored Friend information
- `DWC_FRIENDDATA_LOGIN_ID` ID for the state when a connection to Nintendo Wi-Fi Connection has never been made
- `DWC_FRIENDDATA_FRIEND_KEY` Friend Registration Key
- `DWC_FRIENDDATA_GS_PROFILE_ID` GS Profile ID

When the contacted partner has not yet obtained a GS profile ID, the data type `DWC_FRIENDDATA_LOGIN_ID` indicates that friend information was downloaded via DS Wireless Communications.

Once the contacted partner has obtained a GS profile ID and initiating partner has completed the friend roster synchronization process, the data type changes to `DWC_FRIENDDATA_GS_PROFILE_ID`.

The data type `DWC_FRIENDDATA_FRIEND_KEY` indicates that the friend relationship is not yet established for the GS profile ID registered using the friend registration key. Once the friend relationship is established, the data type changes to `DWC_FRIENDDATA_GS_PROFILE_ID`.

You can use the DWC_IsBuddyFriendData function to determine whether a friend relationship has been established from the friend information.

Code 6-4 Obtaining Friend Information Types

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

    for ( i = 0; i < FRIEND_LIST_LEN; ++i )
    {
        // Get the friend information type
        int type = DWC_GetFriendDataType( &s_friendList[ i ] );
        OS_TPrintf( "friend[%d] type %d.\n", type );

        if ( type == DWC_FRIENDDATA_GS_PROFILE_ID )
        {
            // Show friend relationship if GS profile ID
            if ( DWC_IsBuddyFriendData( &s_friendList[ i ] ) )
            {
                OS_TPrintf( "Friendship is established.\n" );
            }
            else
            {
                OS_TPrintf( "Friendship is not yet established.\n" );
            }
        }
    }
    :
}
```

6.5 Obtaining Friend Status

All players maintain their own status when using Nintendo Wi-Fi Connection. Nintendo Wi-Fi Connection is managed by a server operated by GameSpy.

There are two player states that the application can reference:

- The communication state
- A status string or binary data

The communication state is defined by the DWC_STATUS_* constants, which are set automatically by the DWC library.

The application sets the status string with the `DWC_SetOwnStatusString` function and the binary data with the `DWC_SetOwnStatusData` function as shown in Code 6-5.

Status strings must terminate with NULL and can be up to 256 text characters long, including the NULL terminator. Binary data are converted inside the function into a string, and the approximate number of text characters will be data size x 1.5. The string should not include '/' or '\\' because these text characters are used by the library as identifiers.

The current status of a friend can be obtained if a friend relationship has been established. Specify a friend status change callback as the argument in the `DWC_UpdateServersAsync` function to enable a user to receive notices whenever friend status changes.

To obtain friend status, use the `DWC_GetFriendStatus*` function group. For this group of functions, communication doesn't occur while accessing the friend status list maintained by the DWC library. However, processing of these functions takes several hundred microseconds, so take care when calling the functions frequently over a short period of time.

Furthermore, if there is a sudden loss of power during communication, the player's status will remain in the previous state for a few minutes.

Code 6-5 Getting a Friend's Status

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

   // Synchronize local friend roster and server friend roster
   if ( !DWC_UpdateServersAsync( NULL,
                                 cb_updateServers, NULL,
                                 cb_friendStatus, NULL,
                                 NULL, NULL ) )
   {
      // Synchronization process fails to start
      return;
   }
   :

   // Friend roster synchronization completed
   :

   // Set lcoal status test string
   DWC_SetOwnStatusString( "location=city,level=1" );
   :
```

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

            // If friend information is valid, get the status of that friend
            status = DWC_GetFriendStatus( &friendList[ i ], statusString );

            // Display the status of friend
            disp_friend_status( status, statusString );
        }
    }
    :
}

// Callback notifying change in friend's status
void cb_friendStatus( int index, u8 status, const char* statusString, void*
param )
{
    OS_TPrintf( "Friend[%d] status -> %d (statusString : %s).\n",
                index, status, statusString );
}
```

7 Matchmaking

The DWC library provides two methods of matchmaking: peer matchmaking and server/client matchmaking.

In peer matchmaking, the Nintendo DS systems are not distinguished as servers and clients. There are two implementation methods:

- Not specifying friends
- Specifying friends

7.1 Peer Matchmaking Without Specifying Friends

This method performs matchmaking for players in the general public.

Call the `DWC_ConnectToAnybodyAsync` function to begin peer matchmaking without specifying friends. The function's arguments are: the desired number of connected players including the local player; a filter string for matchmaking conditions; a matchmaking completion callback and its parameters when matchmaking completes; and a player evaluation callback and its parameters. (This last callback is explained later.)

Use the filter string to narrow the search for matchmaking candidates. The matchmaking index keys (in Code 7-1, the key names are `str_key` and `int_key`) need to be registered in advance using the `DWC_AddMatchKey*` function. The key names are saved inside the library, but only pointers to the key values are stored in the library. Consequently, you should retain key values until matchmaking completes.

Note: There are certain names that cannot be used as Matchmaking index keys. For details, see Chapter 7.6, Names that Cannot be Used for Matchmaking Index Keys.

Code 7-1 Peer Matchmaking Without Specifying Friends

```
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 )
{
    // Set the matchmaking index keys
    DWC_AddMatchKeyString( 0, "str_key", s_str_key );
```

```
    DWC_AddMatchKeyInt( 0, "int_key", s_int_key );

    // Start matchmaking without specifying friends
    DWC_ConnectToAnybodyAsync( 4,
                               "str_key = 'anymatch_test' and int_key = 10",
                               cb_anymatch, NULL,
                               NULL, NULL );

    // Poll to see if matchmaking has completed
    while ( !s_matched )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // Error generation
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // Matchmaking has completed
    :
}

//  Callback for when matchmaking has completed
void cb_anymatch( DWCError error, BOOL cancel, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( cancel ) s_canceled = TRUE;
        else          s_matched  = TRUE;
    }
}
```

7.2 Peer Matchmaking by Specifying Friends

This method performs matchmaking for friends registered in friend rosters.

Use the `DWC_ConnectToFriendsAsync` function to begin peer matchmaking by specifying friends,

as shown in Code 7-2 . The function's arguments are: the friend roster index array (the index list) of friends to perform matchmaking; the number of elements in the index list; the desired number of connected players including the host player; whether to allow matchmaking with friends from friend rosters of other friends; a matchmaking completion callback and its parameters; and a player evaluation callback and its parameters. (This callback is explained later.)

If NULL is specified for the index list, all friends in a friend roster are treated as matchmaking candidates.

Peer matchmaking by specifying friends uses the `DWC_InitFriendsMatch` function to specify the friend roster.

Furthermore, because each player has a different friend roster and there is a high probability that a different index list is specified, the success rate of matchmaking drops dramatically when you disallow matchmaking with friends of friends

Code 7-2 Peer Matchmaking by Specifying Friends

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

void do_friend_match( void )
{
   // Start matchmaking with specifying friends
   DWC_ConnectToFriendsAsync( NULL, 0, 4, TRUE,
                              cb_friendmatch, NULL,
                              NULL, NULL );

   // Poll to see if matchmaking has completed
   while ( !s_matched )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // Error generation
         handle_error();
         return;
      }

      GameWaitVBlankIntr();
   }
```

```
    // Matchmaking has completed
    :
}

// Callback for when matchmaking has completed
void cb_friendmatch( DWCError error, BOOL cancel, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        if ( cancel ) s_canceled = TRUE;
        else          s_matched  = TRUE;
    }
}
```

7.3 Evaluating Candidate Players for Matchmaking

During peer matchmaking, players who have been identified as matchmaking candidates can be evaluated using game-specific criteria listed in order of preference.

When an evaluation callback is set as an argument of the function that starts peer matchmaking, that callback is called every time a player is identified as a possible matchmaking candidate during matchmaking. Use the `DWC_GetMatch*Value` function inside this callback to reference the matchmaking index keys that were registered by the `DWC_AddMatchKey*` function as shown in Code 7-3. Evaluate each player based on these values and use the evaluated value as the return value. Players whose evaluated value is less than zero are removed as matchmaking candidates.

Note that this method is designed to make selecting players with the highest evaluated values easier, but this method does not guarantee that players with the highest evaluated values will be selected for matchmaking.

Code 7-3 Evaluating Candidate Players for Matchmaking

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

void do_anybody_match( void )
{
    // Set matchmaking index keys
    DWC_AddMatchKeyString( 0, "str_key", s_str_key );
    DWC_AddMatchKeyInt( 0, "int_key", s_int_key );

    // Start matchmaking by specifying friends
    DWC_ConnectToAnybodyAsync( 4,
```

```
                                "str_key = 'anymatch_test'",
                                cb_anymatch, NULL,
                                cb_eval, NULL );
    :
}

// Player evaluation callback
int cb_eval( int index, void* param )
{
    int eval_int;

    // Get the value for the matchmaking index key int_key
    eval_int = DWC_GetMatchIntValue(index, "int_key", -1);

    if ( eval_int >= 0 )
    {
        // Sees which are close to local value and takes it as evaluated value
        return MATH_ABS( s_int_key - eval_int ) + 1;
    }
    else
    {
        // Does not match make players that do not have the int_key key
        return 0;
    }
}
```

7.4 Server/Client Matchmaking

In server/client matchmaking among friends, the Nintendo DS systems take on clearly defined roles as servers and clients. Server/client matchmaking is the same as peer matchmaking to the extent that the completed network is a mesh network.

The server DS specifies the number of players allowed to connect (this number includes the server DS), a matchmaking completion callback and its parameters, and a notify newly connected clients callback and its parameters. The server DS calls the `DWC_SetupGameServer` function and then waits for the client Nintendo DS systems to connect. The code for this process is shown in Code 7-4.

The client Nintendo DS systems specify an index list of friends allowed to connect, a matchmaking completion callback and its parameters, and a notify newly connected clients callback and its parameters. The client DS calls the `DWC_ConnectToGameServerAsync` function. With this function configuration, the client Nintendo DS systems will try to connect if matchmaking has started with the friend established as the server DS.

When server/client matchmaking completes, the server DS has a friend relationship with every connected client DS. However, the client Nintendo DS systems may have friend relationships through their friends via their connection to the server DS.

The matchmaking completion callback is called when the client DS successfully connects to the server DS, and also when a new client DS is added to the mesh network to which it belongs. The newly connected client notification callback is called when a new client DS starts the connection to the mesh network to which it belongs.

Code 7-4 Server/Client Matchmaking

```
static BOOL s_matched = FALSE;

void do_server_match( void )
{
    // Start matchmaking as server DS
    DWC_SetupGameServer( 4,
                         cb_sc_match, (void *)CB_CONNECT_SERVER,
                         cb_sc_new, NULL );

    while ( 1 )
    {
        DWC_ProcessFriendsMatch();

        if ( DWC_GetLastErrorEx( NULL, NULL ) )
        {
            // Error generation.
            handle_error();
            return;
        }

        if ( s_matched )
        {
            // If connection has been made with new client
            init_new_connection();
            s_matched = FALSE;
        }

        GameWaitVBlankIntr();
    }
    :
}
```

```
void do_client_match( void )
{
   // Start matchmaking as client DS
   DWC_ConnectToGameServerAsync( 0,
                                 cb_sc_match, (void *)CB_CONNECT_CLIENT,
                                 cb_sc_new, NULL );

   // Poll to see if matchmaking completed
   while ( !s_matched )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // Error generation.
         handle_error();
         return;
      }

      GameWaitVBlankIntr();
   }

   // Matchmaking completed
   :
}

// Callback for when matchmaking completed
void cb_sc_match( DWCError error, BOOL cancel, BOOL self, BOOL isServer, int
index, void* param )
{
   if ( error == DWC_ERROR_NONE )
   {
      if ( !cancel )
      {
         // Connection successful
         s_matched = TRUE;
      }
      else if ( self || isServer )
      {
         // If local system cancels matchmaking, or if the local system is a
         // client DS and the server DS has cancelled matchmaking
         s_canceld = TRUE;
```

```
        }
        /* Do nothing even if some other newly connecting client cancels
           matchmaking */
    }
}

// Callback to notify a newly connected client
void cb_sc_new( int index, void* param )
{
    OS_TPrintf( "Newcomer : friend[%d].\n", index );
}
```

Because server/client matchmaking creates a mesh network similar to peer matchmaking, the client Nintendo DS systems can continue to communicate even after the server DS disconnects. However, because server/client matchmaking cannot continue to function without a server DS, it is recommended that you implement a way to disconnect all Nintendo DS systems when the server DS disconnects. The server DS can also filter client connection requests by calling the `DWC_StopSCMatchingAsync` function during matchmaking.

7.5 Increasing Matchmaking Speed

During peer matchmaking without specifying friends, you can increase the speed of matchmaking using filters when getting a list of matchmaking candidates from the matchmaking server. (See Code 7-1). The matchmaking candidate list stored on the matching server has various conditions attached.

Matchmaking is more likely to fail when this list is obtained unconditionally and matchmaking candidates are filtered inside the evaluation callback. This also takes more time by repeatedly re-obtaining the list and performing matchmaking. You can reduce matchmaking failures and increase matchmaking speed using a filter function to form the obtained matchmaking candidate list into a list of acceptable matchmaking candidates.

Conversely, matchmaking efficiency can drop and time may be lost if excessive filtering is performed inside the evaluation callback in situations where the number of candidates is likely to be low (such as seeking players of the same skill level or in the same geographical region).

Consider the following when seeking to increase the matchmaking speed.

- Use a filter function to form a list of available candidates from the obtained matchmaking candidate list.
- Adopt a specification where matches are made aggressively without too much filtering inside the evaluation callback.

7.6 Names that Cannot be Used for Matchmaking Index Keys

There are certain key names that cannot be registered as Matchmaking Index Keys by the `DWC_AddMatchKey*` function because the key names are used by the library and the server. Do not use any of the names listed in Table 7-1.

Table 7-1 Key Names That Cannot Be Used for Matchmaking Index Keys

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 Sending and Receiving Data

8.1 Peer-to-Peer Data Exchange

Once matchmaking completes and the Nintendo DS system connections are established (that is, a mesh network is formed), you must set up for data exchange before the Nintendo DS systems can communicate.

First, set up a receive buffer so each DS can receive data from other Nintendo DS systems. Call the `DWC_SetRecvBuffer` function. For the `aid` argument, specify the AID that serves as the ID number of each DS. The AID accepts values between 0 and *N*, where *N* is one less than the number of Nintendo DS systems in the network.. Therefore, if matchmaking four players completes, the four Nintendo DS systems are assigned the AID numbers 0, 1, 2, and 3. If the DS system assigned AID = 1 leaves the network, the remaining systems maintain the assigned AID numbers 0, 2, and 3. Any data that arrives before setting up the receive buffer is deleted.

Next, configure the send and receive callbacks using the `DWC_SetUserSendCallback()` and `DWC_SetUserRecvCallback` functions. Call the receive callback when a DS receives data from another DS. Call the send callback immediately after transmission of specified data completes.

In this context, note that "transmission completes" means that the data has been passed to the lower layer transmission function; it does not indicate that the partner DS has received the data.

To configure the connection close callback, call the `DWC_SetConnectionClosedCallback` function when the local or partner DS leaves the network by the procedure to officially disconnect (shown in Code 8-1).

These settings are not cleared until the `DWC_ShutdownFriendsMatch` function is called.; Hence, it is not always necessary to set them immediately after matchmaking completes.

Code 8-1 Setup for Data Exchange

```
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;
        }

        // Set the receive buffer for AIDs other than local AID
        DWC_SetRecvBuffer( pAidList[i], &s_RecvBuffer[i-j], SIZE_RECV_BUFFER );
    }

    // Set the send callback
    DWC_SetUserSendCallback( cb_send );

    // Set the receive callback
    DWC_SetUserRecvCallback( cb_recv );

    // Set the connection close callback
    DWC_SetConnectionClosedCallback( cb_closed, NULL );
}

// Callback for sent data
void cb_send( int size, u8 aid )
{
    OS_TPrintf( "to aid = %d  size = %d\n", aid, size );
}

// Callback for received data
void cb_recv( u8 aid, u8* buffer, int size )
{
    OS_TPrintf( "from aid = %d  size = %d  buffer[0] = %X\n",
                aid, size, buffer[0] );
}

//  Connection close callback
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 );
        }
    }
}
```

There are two kinds of data transmission: *reliable* transmission and *unreliable* transmission. Both use UDP communication, but, just like TCP communication, reliable transmission does not experience packet loss and maintains packet order. However, the tradeoff is that reliable transmission takes longer to complete because every sent packet is checked upon receipt.

Because unreliable transmission uses UDP communication, problems with packet loss and packet order can occur. However, transmission is very fast because no packets are checked or resent.

If data transmission occurs at a layer lower than the DWC library, the data accumulates in the send buffer that has a size specified by the `DWC_InitFriendsMatch` function. If the send buffer does not have enough free space when *reliable* transmissions are attempted, any unsent data are retained as-is; they are sent from inside the `DWC_ProcessFriendsMatch` function as soon as space is freed in the send buffer.

Note that the default maximum amount of data that can be sent at once is 1465 bytes. If you try to send more than this maximum amount of data, the data is divided up and the send is suspended. You can change the maximum size of the send buffer using the `DWC_SetSendSplitMax` function. However, because communication devices with various settings need to be accommodated, do not set a maximum size larger than the default maximum.

Do not delete the send buffer if data for transmission is retained and suspended in this way. Also be aware that the next data set cannot be sent while data is retained and suspended.

Use the `DWC_IsSendableReliable` function to check if space is available in the send buffer, the send target AID is valid, and reliable transmission is possible (See Code 8-2.)

If you attempt to send more than the maximum amount of data using unreliable transmission, the transmission will fail and FALSE will be returned.

Code 8-2 Sending Data

```
static u8  s_SendBuffer[ SIZE_SEND_BUFFER ];

void send_data( void )
{
```

```
    // Send data using unreliable transmission to all connected DS systems.
    // Ignore local AID if passed.
    DWC_SendUnreliableBitmap( DWC_GetAIDBitmap(),
                              s_SendBuffer, SIZE_SEND_BUFFER );
    :

    // Check whether reliable transmission is possible for DS with AID=0
    if ( !DWC_IsSendableReliable( 0 ) ) return;

    // Send data using reliable transmission to a specific DS
    DWC_SendReliableBitmap( 0, s_SendBuffer, SIZE_SEND_BUFFER );
    :
}
```

8.2 Closing Connections

Call the `DWC_CloseAllConnectionsHard` function to close the connection with all Nintendo DS systems in the mesh network. When the close process is executed, the connection close callback set by `DWC_SetConnectionClosedCallback()` is called before exiting this function. The close notification also notifies other Nintendo DS systems that were connected and the connection close callback is called.

The server DS in server/client matchmaking calls this `DWC_CloseAllConnectionsHard` function even if there are no other connected systems at the time. This function call frees any remaining regions of memory that were used for matchmaking and restores the communication state to the online state. Calling this function does not close the connection with Nintendo Wi-Fi Connection server.

The following functions are also provided: the `DWC_CloseConnectionHard` function closes a connection by specifying an AID and the `DWC_CloseConnectionHardBitmap` function closes multiple connections by specifying an AID bitmap. These functions are designed for use in unusual circumstances, such as closing connections for a DS that becomes unavailable for communication because the power is turned off.

8.3 Yardstick for Buffer Size Specified by DWC_InitFriendsMatch

The buffer sizes specified by the `DWC_InitFriendsMatch` function become the buffer sizes adopted internally by the DWC. When data is sent using reliable communication, the Send buffer stores data for which ACK is not returned. The Receive buffer stores data that did not reach the Receive buffer in the correct order.

With reliable communication, you need as much capacity as possible to deal with instantaneous network interruptions. The Send and Receive buffers both need to be large enough to handle as much interruption time as the game's specifications permit.

Although the Send and Receive buffers are generally not used with unreliable communication, you still need a Send buffer of at least 1 kbyte and a Receive buffer of at least 128 bytes because DWC uses reliable communication internally when connecting peer-to-peer.

Table 8-1 Yardstick for buffer sizes

Kind of Communication		Yardstick for Buffer Size	Comments
Reliable Communication	Send buffer size	Compute buffer size as: (allowable duration in seconds of instantaneous interruption as per the game specs) x (amount of reliable data per second) + (total size of reliable data). Total size of reliable data = 7 x (number of divisions in the data being sent) + (size of data being sent) + 15	Minimum of 1 kbyte
	Receive buffer size		Minimum of 128 bytes
Unreliable Communication	Send buffer size	(Max. data size for unreliable communication)＋ 2 bytes	Minimum of 1 kbyte
	Receive buffer size	Minimum of 128 bytes	

Note: The number of divisions in the data sent indicates the number into which the data is divided when the total data size exceeds the maximum amount of data that can be sent at any one time. This is specified by the `DWC_SetSendSplitMax` function (default size: 1,465 bytes).

The following shows an example of how to calculate the required size of the Send and Receive buffers.

Assume that:

- The game spec allows an instantaneous interruption to last for as long as 1 second
- Communication is performed once every 3 frames
- The maximum amount of data that can be sent at one time is 64 bytes
- The game is sending 100 bytes of data using reliable communication.

In this case, the required size of the Send and Receive buffers is:

1 (second) x (60 (frames) ÷3) x (7 x 2 (divisions) + 100 (bytes) + 15) = 2580 (bytes)

8.4 Emulating Delays and Packet Loss

The DWC library can emulate delays and packet loss for sending and receiving data. For send delays, the send data is copied to another buffer and kept for a specified amount of time; this data will not be

sent to the partner because the data is deleted when the connection is closed. For this reason, using only the receive delay is recommended.

The packet loss rate (in units of percent), the delay time (in units of milliseconds), and the AID of the receiving DS are specified in Code 8-3.

Code 8-3 Emulating Delays and Packet Loss

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

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

8.5 Amount of Data Sent and Received

Table 8-1 shows the amount of data transmitted during reliable and unreliable communication.

Table 8-1 Communication Data Breakdown

<table>
<tr><th>Send Data Items</th><th colspan="4">Send Data Size</th></tr>
<tr><td>Preamble</td><td colspan="4">192 bits (24 bytes)</td></tr>
<tr><td>MAC</td><td colspan="4">24 bytes</td></tr>
<tr><td>LLC</td><td colspan="4">8 bytes</td></tr>
<tr><td>IP</td><td colspan="4">20 bytes</td></tr>
<tr><td>UDP</td><td colspan="4">8 bytes</td></tr>
<tr><td rowspan="3">DATA</td><td colspan="3">Reliable Communications</td><td>Unreliable Communications</td></tr>
<tr><td>Header send</td><td>Data send</td><td>Receive check</td><td>Data send</td></tr>
<tr><td>15 bytes</td><td>7 + XXX bytes</td><td>5 bytes</td><td>XXX bytes</td></tr>
<tr><td>FCS</td><td colspan="4">4 bytes</td></tr>
<tr><td>B (random time for avoiding packet collision)</td><td colspan="4">MAX 600 µs (microseconds)</td></tr>
</table>

Note: The header send and receive check are sent before and after data send during reliable communication.

Although you can find the data send time for each transmission based on the formula Preamble + (MAC + LLC + IP + UDP + DATA + FCS) x 4 + B [µs], it is difficult to accurately calculate the amount

of data sent and received due to the fact that the transmission time varies depending on factors such as the number of retries required due to bandwidth conditions, the number of sent packets, the amount of transmission standby to avoid packet collisions, etc.

This section provides figures obtained in experiments for the amount of data sent/received.

Experiments were conducted by measuring throughput, CPU load, and the packet loss ratio while varying conditions such as the use of reliable or unreliable communication, the AP model and manufacturer, the amount of radio usage, the send size, and the send frequency. As a result, the following became clear:

- Send frequency (the number of packets issued) is greatly affected by the presence of back-off time (including empty intervals between communication and random time for avoiding packet collisions) of the header part and wireless communication.
- In a four-unit mesh network where data is sent at a rate of once every three frames and the radio noise is under 10%, the upper limit of send size is in the range 120-150 bytes.
- In a four-unit mesh network where data is sent at a rate of once every three frames and the radio noise is around 50%, the upper limit of send size is in the range 100-120 bytes.
- When using reliable communication, traffic congestion occurs easily because congestion is exacerbated by the need to repeatedly resend data when the network is busy. Once this occurs, recovery time is extended.

Note: Radio noise is generated by using `WMTestTool` from another DS.

Based on the experimental results above, Nintendo titles communicate as listed below.

- Four-unit mesh network, unreliable communication
  Nth frame: Send to Party 1

  (N＋1)th frame: Do not send

  (N＋2)th frame: Send to Party 2

  (N＋3)th frame: Do not send

  (N＋4)th frame: Send to Party 3

  (N＋5)th frame: Do not send

  (Repeats from this point on)

  Communication every 60 to 104 bytes
- Four-unit server-client type connection, reliable communication
  Send frequency is three frames with a usual send size of 1 to 40 bytes (up to 256 bytes).

9 HTTP Communication

The DWC library provides the GHTTP library to upload and download data using HTTP. You can use this feature alone without the matchmaking and friend relationship features.

9.1 Preparing to Use the GHTTP Library

You need to initialize the GHTTP library before using it by calling the `DWC_InitGHTTP` function as shown in Code 9-1.

Specify `NULL` for the argument. The returned value will always be `TRUE`.

If the `DWC_InitGHTTP` function has been called and the connection to the Internet has been established, the GHTTP library features are available for use.

Code 9-1 Initializing the GHTTP Library

```
void init_ghttp( void )
{

   // Initialize DWC library
   init_dwc();

   // Make connection to Internet
   if ( connect_to_inet() ) return;

   // Initialize GHTTP
   DWC_InitGHTTP( NULL );
}
```

9.2 Uploading Data

Code 9-2 shows the uploading data process. To upload data to the HTTP server using the GHTTP library, you must first call the `DWC_GHTTPNewPost` function and create a `DWCGHTTPPost` type object. Next, use the `DWC_GHTTPPostAddString` function to add the data you want to upload to this object.

For the arguments of the `DWC_GHTTPPostAddString` function, specify the pointer to the `DWCGHTTPPost` type object, the pointer to the key string that specifies the data, and the pointer to the actual data (the value string) that you want to add.

The key and value strings are both copied and saved in the library.

Both strings must terminate with NULL. When NULL is specified for the value string, the string "" that contains only the NULL terminator is specified.

To begin the actual data upload, use the DWC_PostGHTTPData function. For the arguments of this function, pass the upload URL destination, the pointer to the DWCGHTTPPost type object, and the completion callback and its parameters.

After the upload starts, all of the communication processes are carried out inside the DWC_ProcessGHTTP function. Call this function approximately once per game frame.

The DWCGHTTPPost type object is released immediately after the upload completes and the completion callback returns.

In Code 9-2, the actual data sent to the HTTP server is a string similar to the following:

```
"key1=value1&key2=value2"
```

If data has been already added to identical DWCGHTTPPost type objects, the following string is added :

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

Code 9-2 Uploading Data

```
static int s_send_cb_level = 0;

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

    // Create the DWCGHTTPPost type object
    DWC_GHTTPNewPost( &post );

    // Set the data to upload to the DWCGHTTPPost type object
    DWC_GHTTPPostAddString( &post, "key1", "value1" );
    DWC_GHTTPPostAddString( &post, "key2", "value2" );

    // Start uploading data
    s_send_cb_level++;
    req = DWC_PostGHTTPData( "http://www.test.net", &post, cb_post, NULL );

    if ( req < 0 )
    {
```

```
        // Error generation.
        handle_error();
        return;
    }

    while ( s_send_cb_level )
    {
        // Proceed with the upload process
        DWC_ProcessGHTTP();

        GameWaitVBlankIntr();
    }

    if ( DWC_GetLastErrorEx( NULL, NULL ) )
    {
        // Error generation.
        handle_error();
        return;
    }

    // Data upload has succeeded
    :
}

// Callback for when upload has completed
void cb_post( const char* buf, int buflen, DWCGHTTPResult result, void* param)
{
    s_send_cb_level--;
}
```

9.3 Downloading Data

The library provides two functions for downloading data from the HTTP server: the simple `DWC_GetGHTTPData` function and the expanded feature `DWC_GetGHTTPDataEx` function shown in Code 9-3.

For the arguments of the `DWC_GetGHTTPDataEx` function, pass the data download URL target, the size of the receive buffer, whether to release the receive buffer after the download completes, a callback to obtain the communication state and its parameters, and a completion callback and its parameters.

If the receive buffer size is set to zero, 2048 bytes are initially secured for the memory region;

additional 2048 byte chunks are then secured for receive data up to the limit of the heap region allocated by the application.

If a callback for getting the communication status has been specified, the callback will be called when the download sequence status changes (for example, when requests are being sent and data is being received). If data is being received, you can also check the received data size.

When the download completes, the completion callback is called. If the settings are configured to release the receive buffer after the download completes, the buffer is released immediately after the process has exited from this completion callback. Consequently, ensure received data is copied for use.

If the settings are configured not to release the receive buffer, the GHTTP library will not release the receive buffer. At a convenient time, release the pointer to the receive buffer passed by the completion callback argument in the application. To release the receive buffer, use the `DWC_Free` function.

The `DWC_GetGHTTPData` function has the same behavior as `DWC_GetGHTTPDataEx` function with the arguments `bufferlen` set to 0, `buffer_clear` set to `TRUE`, and `progressCallback` set to `NULL`.

After downloading starts, all communication processes occur inside the `DWC_ProcessGHTTP` function. Call this function approximately once every game frame.

Code 9-3 Downloading Data

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

void get_ghttp_data( void )
{
    // Start data download using simple function
    s_get_cb_level++;
    req = DWC_GetGHTTPData( "http://www.test.net", cb_get, GET_TYPE_NORMAL );

    if ( req < 0 )
    {
        // Error generation.
        handle_error();
        return;
    }

    // Start data download using expanded function
    s_get_cb_level++;
    req = DWC_GetGHTTPDataEx( "http://www.test.net",
```

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

    if ( req < 0 )
    {
        // Error generation
        handle_error();
        return;
    }

    while ( s_get_cb_level )
    {
        // Proceed with the download process
        DWC_ProcessGHTTP();

        GameWaitVBlankIntr();
    }

    if ( DWC_GetLastErrorEx( NULL, NULL ) )
    {
        // Error generation.
        handle_error();
        return;
    }

    //  Data download has succeeded
    :
}

// Callback for when download has completed
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 );
        }
    }
}
```

9.4 Closing the GHTTP Library

Call the `DWC_ShutdownGHTTP` function to close the GHTTP library.

You must call the `DWC_InitGHTTP()` and `DWC_ShutdownGHTTP` function the same number of times. If you do not call these functions the same number of times, memory secured by the GHTTP library will not be freed.

10 Communication Errors

The DWC library provides an error handling system for all DWC modules. In this system, DWC errors are treated like application errors.

10.1 Error Handling

You can obtain the error status in the DWC library using `DWC_GetLastErrorEx` function as shown in Code 10-1. The error classification is the return value. The arguments are the error code and the pointer to the storage location for the error handling type.

The error code is 0 or a negative number. If you are going to show the error code, be sure to invert the sign so the value is shown as a positive number. However, if it is a recoverable error and the DS was not disconnected from Nintendo Wi-Fi Connection, you do not need to display the error code.

The error process type indicates the recovery process required after the error occurs, and a routine error process can be created for each value.

Once the error state has been entered, the DWC library will reject most functions. To return from the error state, call the `DWC_ClearError` function.

Code 10-1 Error Handling Process

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

      handle_error();  // Error-handling process

      GameWaitVBlankIntr();
   }
   :
}

// Error-handling process
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;

    // Get the error
    err = DWC_GetLastErrorEx( &errcode, &errtype );

    // If there is no error, return without doing anything
    if ( err == DWC_ERROR_NONE ) return 0;

    // Clear the error
    DWC_ClearError();

    // Display an error message
    disp_error_message( err );
    // If error code is -10000 or lower, display the code as a positive number
    if ( errcode <= -10000 ) disp_message( "%d", -1*errcode );

    if ( errtype == DWC_ETYPE_SHUTDOWN_FM )
    {
        // End the FriendsMatch process
        DWC_ShutdownFriendsMatch();
    }
    else if ( errtype == DWC_ETYPE_DISCONNECT )
    {
        /* End the FriendsMatch process and perform cleanup on Internet
           connection */
        DWC_ShutdownFriendsMatch();
        disconnect_func();
    }
    else if ( errtype == DWC_ETYPE_FATAL )
    {
        // Fatal Error, so nothing can be done after prompting to turn power off
        while (1) ;
    }
    /* If only a minor error, you can just clear the error and resume the
```

```
        FriendsMatch process */

    return err;
}
```

10.2 List of Error Codes

This list provides the main error codes that occur during the matchmaking and friend relationship process.

If the last three digits of an error code are 010 or 020, these errors are likely to occur if the GameSpy server is in an unstable state (for example, during maintenance).

- 61010 A communication error occurred with the GameSpy GP server during GP server login.
- 61020 A communication error occurred with the GameSpy GP server during GP server login.
- 61070 A login timeout error occurred during GP server login.
- 71010 A communication error occurred with the GameSpy GP server while synchronizing friend rosters.
- 80430 Connection to the client DS failed for server/client matchmaking because the server DS that the Client DS was attempting to connect with or the client DS connected to the server DS was powered off.
- 81010 A communication error occurred with the GameSpy GP server during matchmaking.
- 81020 A communication error occurred with the GameSpy master server during matchmaking.
- 84020 Communication from the GameSpy master server were interrupted during matchmaking. Either the master server is down or the firewall is blocking UDP.
- 85020 A communication error occurred with the GameSpy master server during matchmaking.
- 85030 The GameSpy master server DNS failed during matchmaking. All error codes with 030 as the last three digits indicate DNS errors.
- 86420 NAT negotiations failed the set number of times during one matchmaking session. There may be a problem with the router. In server/client matchmaking, this error only occurs when the client DS that has started connecting and NAT negotiation has failed even one time.
- 97003 A socket error has occurred in a lower layer than the DWC library after matchmaking completes.

Error codes with 1010 or 1020 as the last four digits and error code 85020 are known to occur frequently in the Wi-Fi library for NitroWiFi version 1.0 RC2 and earlier when TCP transfers with the GameSpy server are delayed.

11 Network Storage Support

The DWC library can store data onto the network storage server provided by GameSpy. Code 11-1 shows you how to use this feature.

To access this storage server, complete the process up to the login using the `DWC_LoginAsync` function. Next, log in to the storage server using the `DWC_LoginToStorageServerAsync` function.

The data to save on the storage server can have public or private attributes. If the data is saved using the `DWC_SavePublicDataAsync` function, the data attributes are public and other players can reference the data.

If the data is saved using the `DWC_SavePrivateDataAsync` function, the data attributes are private and other players cannot reference the data.

To load data from the storage server, call the `DWC_LoadOwnPublicDataAsync` function to load your own public data, `DWC_LoadOwnPrivateDataAsync` function to load your own private data, and the `DWC_LoadOthersDataAsync` function to load the friend data saved in your friend roster. Friends are specified by the friend roster index.

When saving or loading data completes, the appropriate callback set by the `DWC_SetStorageServerCallback` function is called. These callbacks are always called in the order that the save and load functions were called.

A string that combines key and value can be specified as saved data. The key/value combinations are repeated by delimiting with `\\`, as in `\\name\\mario\\stage\\3`. If this example data is specified, "`mario`" will be registered in the key value name and "`3`" is registered in the key value stage as a string.

To load data saved on the storage server, specify the keys that you want to retrieve as `\\name\\stage`, separating the name and stage with `\\`.

In this case, the string that you can get with a load callback would be in the format of `\\name\\mario\\stage\\3`.

If you attempt to load a key that does not exist on the storage server or a key that was saved by a friend who used the private attribute, the `success` argument of the callback function will be `FALSE`. If you specify multiple keys to load and only some of the keys fall into these two categories, the `success` argument will be `TRUE`, but these keys will not be included with the loaded data.

After storage server processing completes, call the `DWC_LogoutFromStorageServer` function to log out from the storage server (as in Code 11-1).

Code 11-1 Accessing the Storage Server

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

void access_net_storage( void )
{
   // Login to the storage server
   if ( !DWC_LoginToStorageServerAsync( cb_storage_login, NULL ) )
   {
      OS_TPrintf( "DWC_LoginToStorageServerAsync() failed.\n" );
      return;
   }

   // Wait for login to storage server to complete
   while ( !s_storage_logined )
   {
      DWC_ProcessFriendsMatch();

      if ( DWC_GetLastErrorEx( NULL, NULL ) )
      {
         // Error generation
         handle_error();
         return;
      }

      GameWaitVBlankIntr();
   }

   // Set callbacks for the time when saving and loading complete
   DWC_SetStorageServerCallback( cb_save_storage, cb_load_storage );

   // Save public data
   s_cb_level++;
   if ( !DWC_SavePublicDataAsync( "\\name\\mario\\stage\\3", NULL ) )
   {
      OS_TPrintf( "DWC_SavePublicDataAsync() failed.\n" );
      return;
   }

   // Save private data
   s_cb_level++;
   if ( !DWC_SavePrivateDataAsync( "\\id\\100", NULL ) )
   {
      OS_TPrintf( "DWC_SavePrivateDataAsync() failed.\n" );
      return;
```

```
}

// Wait for saving to complete
while ( s_cb_level > 0 )
{
    DWC_ProcessFriendsMatch();

    if ( DWC_GetLastErrorEx( NULL, NULL ) )
    {
        // Error generation
        handle_error();
        return;
    }

    GameWaitVBlankIntr();
}

// Load local saved data
s_cb_level++;
if ( !DWC_LoadOwnDataAsync( "\\id\\stage", NULL ) )
{
    OS_TPrintf( "DWC_LoadOwnDataAsync() failed.\n" );
    return;
}
// Load ones own private save data
s_cb_level++;
if ( !DWC_LoadOwnPrivateDataAsync( "\\id", NULL ) )
{
  OS_TPrintf( "DWC_LoadOwnPrivateDataAsync() failed.\n" );
  return;
}
// Load another player's saved data
s_cb_level++;
if ( !DWC_LoadOthersDataAsync( "\\name", 0, NULL ) )
{
    OS_TPrintf( "DWC_LoadOthersDataAsync() failed.\n" );
    return;
}

// Wait for loading to complete
while ( s_cb_level > 0 )
{
    DWC_ProcessFriendsMatch();

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

```
            // Error generation
            handle_error();
            return;
        }

        GameWaitVBlankIntr();
    }

    // Log out from storage server
    DWC_LogoutFromStorageServer();
    :
}

// Callback for the tune when logged in to storage server
void cb_storage_login( DWCError error, void* param )
{
    if ( error == DWC_ERROR_NONE )
    {
        s_storage_logined = TRUE;
        s_cb_level        = 0;
    }
}

// Callback for when data is saved to storage server
void cb_save_storage( BOOL success, BOOL isPublic, void* param )
{
    OS_TPrintf( "result %d, isPublic %d.\n", success, isPublic );
    s_cb_level--;
}

// Callback for the time when data loaded from storage server
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--;
}
```