# NITRO-SDK

## Single-Card Play User Guide

Version 1.0.2

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



NTR-06-0226-002-A3
Released: February 2, 2006

# Table of Contents



# List of Tables



# List of Figures



# List of Code Examples





NTR-06-0226-002-A3
Released: February 2, 2006

# Revision History

| Version | Revision Date | Description |
|---|---|---|
| 1.0.2 | 8/8/2005 | • Updated changes to numbering for code reference (4.1.1). Moved number 4 to following line and moved comment to next line.<br>• No change needed in 2.1.1.1—terminology was already correct. |
| 1.0.1 | 3/11/2005 | • Unified format for describing NITRO-SDK install destination (1)<br>• Deleted text overlaps with following item (1.3)<br>• Changed item names (because of use with `libsyscall.a`) (1.4)<br>Corrected text (Supplement related to previous item)<br>• Corrected text (clearly indicated that startup is same as from Card) (1.5)<br>• Corrected terminology (AID) (2)<br>• Corrected `MB_StartParentFromId` and `MB_EndToIdle` function names (2.1)<br>• Corrected GGID and TGID terminology (2.1.1.2)<br>• Revised description of the maximum number of connected children (2.1.1.3)<br>• Corrected item format (2.1.1.4)<br>Revised text (Supplemented with part about relationship between maximum number of connected children and number of players)<br>Corrected text relating to names for libraries and sample modules<br>Deleted text (old restrictions relating to segment data)<br>• Added text (Supplemented with part about distinguishing multiple communication modes) (2.3.1)<br>• Added text (Supplemented with part relating to build switches) (2.3.3)<br>• Corrected figure and supplemented text about parent-only region (3.1)<br>• Corrected text (Supplemented with reason for data placement) (3.2.1)<br>• Corrected text (Corrected `mb_parent.h` to be `mbp`) (4)<br>Corrected text (To reflect the latest selection of sample code)<br>• Added text (Supplemented with part about changes to procedure when using `MB_StartParentFromIdle` function (4.1)<br>• Added the section for the `cloneboot` sample program (5) |
| 1.0.0 | 10/29/2004 | Initial version. |

# 1 Introduction

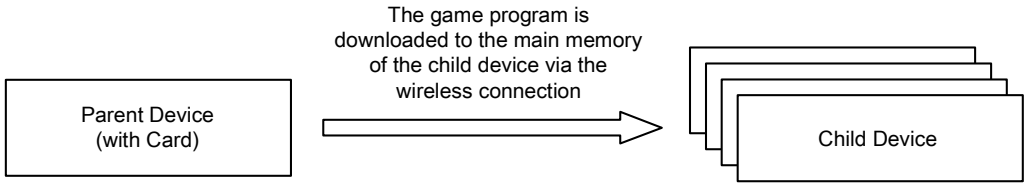
The NITRO-SDK includes a series of APIs for use with the Single-Card Play feature. This document describes how to use the basic Single-Card Play features. (In this document, `$NitroSDK` represents the directory in which NITRO-SDK has been installed.)

## 1.1 Overview

The Nintendo DS (DS) has Single-Card Play capability that allows binary code to be transferred from a Single-Card Play *parent* device to a Single-Card Play *child* device and enables the child device to boot up without a Game Card.

Single-Card Play is also referred to as "Wireless Multiboot" in developer documentation and SDK source code files. This feature can be used to download up to 2.5 MB of binary code from a parent device to the main memory of a child device so that the child device can be booted.

**Figure 1-1 Single-Card Play Schematic**




## 1.2 Single-Card Play Startup Procedure

To start a game using the Single-Card Play feature, the players should start execution according to the following procedure.

1. Start the Single-Card Play parent device.
2. Select Single-Card Play from the start menu on the child device and select the parent program to be downloaded.

However, in order to prevent the execution of illegal code by the IPL, binary code without an attached authentication code will not execute. To start a child device from Single-Card Play, authentication code must be attached to the binary code being transmitted. To promote efficient development, the NITRO-SDK includes `mb_child` to permit the running of binary code without an authentication code. Use `mb_child.srl` included with the NITRO-SDK and follow the procedure below. Use `mb_child` in the same way even when executing under a debugging environment.

1. The following is a list of three pre-built programs that are stored in the NITRO-SDK. Write any one of these programs into the NITRO Flash Card. (If using the debugger, load the binary code for `mb_child.srl` into the debugger.)

```
$NitroSDK/bin/ARM9-TS/Rom/mb_child.srl
$NitroSDK/bin/ARM9-TS/Rom/mb_child_simple.srl
$NitroSDK/bin/ARM9-TS/Release/mb_child_simple.srl
```



NTR-06-0226-002-A3
Released: February 2, 2006

To read about `mb_child.srl` and `mb_child_simple.srl`, see the section titled “Pre-Built Programs" in the Function Reference Manual.

2. Start the Single-Card Play parent device.
3. Start `mb_child` as the Single-Card Play child device and select the parent program to be downloaded.

To actually start the game device, an authentication code must be attached to the binary sent to the child. See section 1.3 for more information.

## 1.3 Attaching an Authentication Code

Under the DS system, an authentication code must be attached to binary code that starts on the child. This convention prevents the execution of invalid binary code transmitted wirelessly.

**Note:** The game device will stop midway through booting (around the time that the Nintendo logo fades from the screen) if an attempt is made to execute binary code that does not have an authentication code.

The following procedure can be used to attach an authentication code to binary code to be transmitted:

1. First, create the binary code to send to the child.
2. Send this binary code to the Nintendo authentication server at Nintendo and obtain an authentication code.
3. Attach the authentication code to the original binary code using `$NitroSDK/tools/bin/attachsign.exe`.
4. Use the Single-Card Play parent device to link the binary code obtained in Step 3 and transmit it to the child device.

This procedure can be used to send code that runs on the parent game device to children. For details on how to obtain an authentication code, please contact support@noa.com.

## 1.4 Using the System Call Library and ROM Header

When creating the production version of a ROM, use the System Call library (`libsyscall.a`) and the ROM headers (`rom_header_****.template.sbin`) provided by Nintendo. However, the binary file for child devices differs from that for parent devices, so in this case it is necessary to use the System Call library and ROM headers that are included in the NITRO-SDK.

## 1.5 Transferable Binary Code Size

The same size restriction that applies to startup from a card applies to binary code for Single-Card Play. The maximum transferable size for resident code is 2.5 MB for the ARM9 and 256 KB for the ARM7. As with startup from a card, if data has been compressed with the `compstatic` tool, the size restriction applies to the compressed data, not the uncompressed data.

To send binary code that exceeds this size restriction, start the child in Single-Card Play mode and then download the necessary additional binary code from the parent. However, be sure to follow the guidelines being given as there are security reasons for restricting the transfer of executable code.

## 1.6 Accessing the Backup Regions in Game Cards and Game Paks

Technically, the backup region of a Game Card or Game Pak plugged into the parent device can be accessed from a child device started with Single-Card Play. However, there are some restrictions in place when doing so. Follow the "*Nintendo DS Programming Guidelines*" when it comes to actual operations.

# 2 Single-Card Play Operations

This section describes the procedures and connection sequence necessary to create a Single-Card Play parent.

The Single-Card Play processes can be implemented using the Single-Card Play (MB) library stored in the NITRO-SDK. The MB library functions by using the Wireless Manager (WM) library internally, but other WM features cannot be used at the same time under current conditions.

## 2.1 Process Flow on the Parent Side

This section describes the preparations made on the parent side before Single-Card Play begins. The parent prepares to send the binary code according to the following procedure:

1. Select a communication channel
2. Set the parent's parameters
3. Start the parent device communication process
4. Register the child binary information
5. Receive a request from the child
6. Send the binary and boot the child.

Once the child's binary information is registered in Step 4, the parent begins disseminating information automatically and enters a child-receptive state.

### 2.1.1 Preparations by the Parent

#### 2.1.1.1 Selecting a Wireless Communication Channel

The recommended method for deciding which wireless communications channel to use is to get the usable channels with the `WM_GetAllowedChannel` function, check the signal traffic level on each channel with the `WM_MeasureChannel` function, and then select the channel with the most available bandwidth.

However, at the present time, you cannot use the `WM_MeasureChannel` function after starting the MB library because the `MB_StartParent` function automatically moves the MB library module from the READY state to the PARENT state when using the MB library, but the `WM_MeasureChannel` function can only execute when the WM library is in the IDLE state. It is therefore necessary to put the WM library module into the IDLE state using the `WM_Initialize` function before checking the signal traffic level of channels.

After the communication channel has been selected, there are two methods to start Single-Card Play:

- Terminate the WM library with the `WM_End` function and then start Single-Card Play.
- Enter the IDLE state using the `MB_StartParentFromIdle` function and then start Single-Card Play.

When using the MB_StartParentFromIdle function, the work buffer size passed to the MB_Init function may be set as small as WM_SYSTEM_BUF_SIZE bytes as long as the WM_Initialize function is called separately. Be sure to call the MB_StartParentFromIdle and MB_EndToIdle functions as well as the MB_StartParent and MB_End functions in pairs.

#### 2.1.1.2 Setting the Parent's Parameters

When starting the Single-Card Play parent device, the GGID and TGID must be set up just as with a normal wireless communication. The following player information on the parent device, such as the nickname to be displayed on IPL child screen during Single-Card Play, must also be set.

- Player Nickname
  A maximum of 10 characters of UTF16-LE. The same format is used as with nicknames obtained with the OS_GetOwnerInfo function.
- Favorite Color
  This is the color-set number representing the player's favorite colors. This makes use of the same color set as favoriteColor obtained with the OS_GetOwnerInfo function. For details, see the reference for the OS_GetFavoriteColorTable function.
- Player Number
  The player number for the parent is always 0.

#### 2.1.1.3 Configuring the Maximum Number of Children

The MB library drives wireless communications using the WM library under the assumption that the default maximum number of devices is 16 (1 parent and 15 children). As a result, if a distributed program is configured for play by less than 16 devices, it may not be possible to achieve the transfer efficiency usually available and a situation may develop in which the number of connection requests from children exceeds the maximum number of players.

If you already know the number of programs distributed from the parent and the maximum number of players allowed by them, then you can use the MB_SetParentCommParam function to set the number of child devices that will be allowed to make connections. The maximum AID value for children to be connected is set using the maxChildren argument of this function. The sendSize argument can be used in conjunction with the maxChildren argument to freely set the send buffer size to use for wireless communication within a predetermined time. The size of this buffer ranges between a minimum of MB_COMM_PARENT_SEND_MIN and a maximum of MB_COMM_PARENT_SEND_MAX.

#### 2.1.1.4 Registering the Child Binary Information

The following information needs to be set when registering the binary that will be sent to the child.

- Pointer to the Distribution Binary Code Data
  When a child starts, only binary code allocated as an ARM9 resident module or as an ARM7 resident module in the ROM specification file is transferred. Code necessary to start the child can be extracted from the binary code using the MB_ReadSegment function. For details on how to configure resident modules (hereafter referred to as Static segments), see the separate reference document for makerom.

All other binary data must be transferred after booting from the parent to the children using wireless communications. The WBT library is provided in the SDK as a data transfer protocol, and it can be used as needed by applications. A sample program in which a child rebuilds its own file system via wireless communications using the WBT library has been prepared as a module in the directory `$NitroSDK/build/demos/wireless_shared/wfs`.

- Game Name
  A maximum of 48 characters of UTF16-LE. The string must fit on one line having a length of 185 dots during the IPL display.
- Game Description
  A maximum of 96 characters of UTF16-LE. The string must fit on two lines having a length of 199 dots during the IPL display.
- Palette and Image Data for Icons Used to Display Downloaded Games on the IPL.
  This is 16-color palette data and 32 dot x 32 dot image data.
- GGID
  This is the Game Group ID for notifying children after bootup. The GGID set here is reflected in the `ggid` member of the structure that the child can obtain from the `MB_GetMultiBootParentBssDesc` function after it boots. The GGID can be used for reconnecting after bootup.
- Maximum Number of Players
  This specifies the maximum number of players (including the parent) displayed on the child's IPL screen. The total number of players including the parent is not the same as the maximum number of children, so be careful not to mistake this with the `maxChildren` argument of the `MB_SetParentCommParam` function. (If both functions are called with the same setting for the number of players, the maximum number of players is equal to `maxChildren` + 1.)
  Also note that this value is only meant for display on the child's IPL screen. The actual number of children that connect may be less than the value set in the `maxChildren` argument of the `MB_SetParentCommParam` function.

Note that it is necessary when using the MB library to register the child binary using the `MB_RegisterFile` function after starting the parent process using the `MB_StartParent` function.

Up to 16 different child binaries can be registered by a single parent when using the MB library. The Single-Card Play menu screen of the child shows the various games being delivered.

### 2.1.2 Sending Data and Starting Children

Once preparations for delivering binary code are complete, the parent waits for a request from a child. For each child it performs processes in the order: Entry → Download → Boot.

In addition to notification of the child device state via the callback function set with the `MB_CommSetParentStateCallback` function, the child device state can also be obtained with the `MB_CommGetParentState` (child AID) function.

**Figure 2-1 Data Reception State Transitions and Parent Requests Used with Single-Card Play Children**

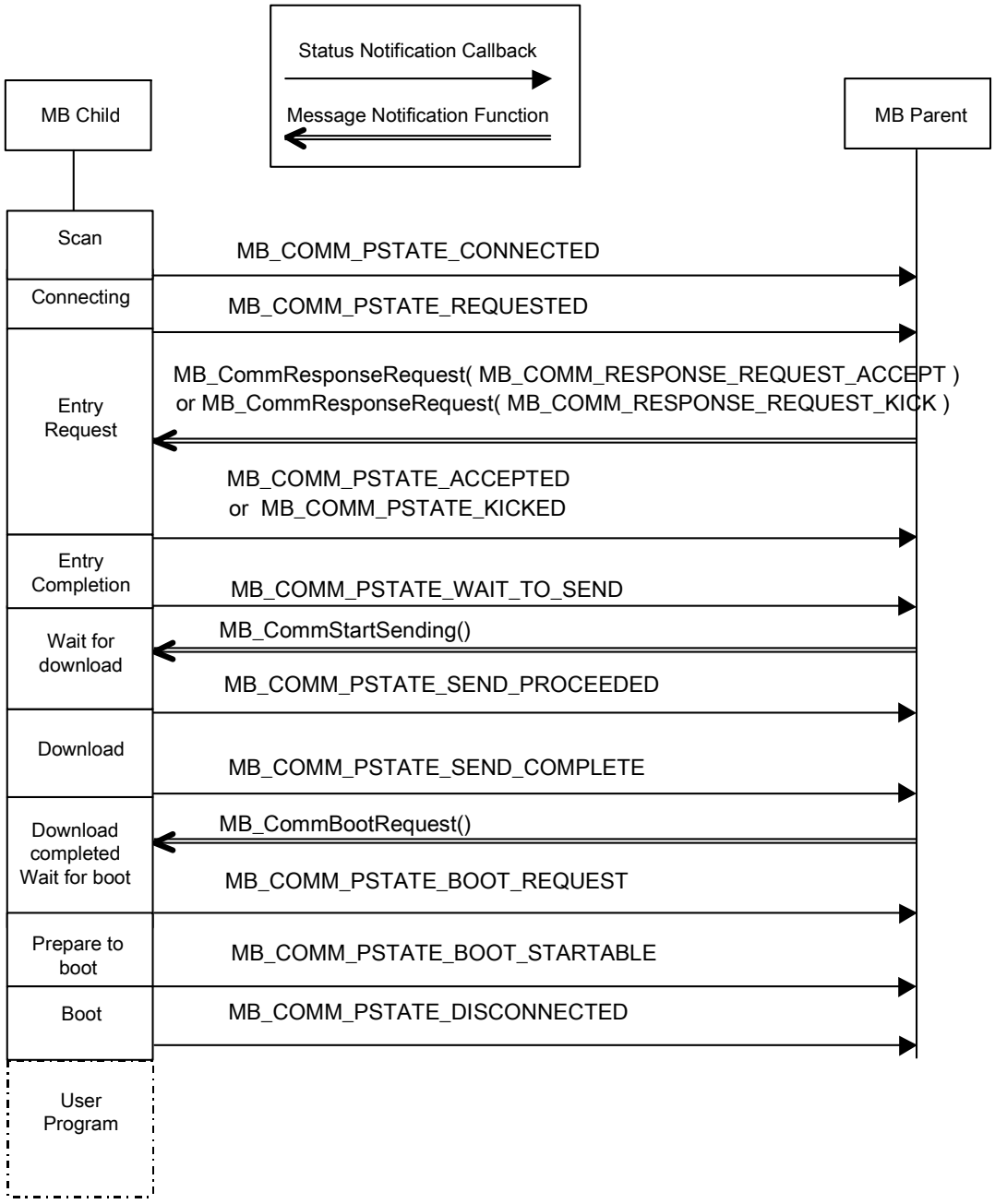



Figure 2-1 depicts the states of the child and the flow of requests from the parent. A callback is generated on the parent side every time the child changes states. Be sure to issue the appropriate command for each state from the parent side based on the state change notification made by this callback or the state obtained by the `MB_CommGetParentState` function.

The connection sequence flow between parent and child devices is shown below.

1. Connect
   When the IPL Single-Card Play child program connects to the parent, the state changes to `MB_COMM_PSTATE_CONNECTED`. The child's MAC address can be obtained with this callback.

2. Entry
   When there is an entry request from a child to the parent device, notification of the MB_COMM_PSTATE_REQUESTED state is sent. The child device then waits for either an MB_COMM_RESPONSE_REQUEST_ACCEPT or MB_COMM_RESPONSE_REQUEST_KICK message to arrive from the parent device. If MB_COMM_RESPONSE_REQUEST_ACCEPT is sent, entry processing is performed and preparations for downloading data are made.
3. Download
   When the child completes preparations for downloading data, the parent is notified that the state has changed to MB_COMM_PSTATE_WAIT_TO_SEND. Once the child is in this state, the parent can begin sending data for the first time. Be careful not to start transmitting data when the state is MB_COMM_PSTATE_ACCEPTED. When data transmissions end, the child sends notification that the state is MB_COMM_PSTATE_SEND_COMPLETE and waits in this state until there is a boot request.
4. Boot
   If the child is in the MB_COMM_PSTATE_SEND_COMPLETE state, it enters the boot process when the parent issues the MB_CommBootRequest command. Once the parent is notified that the state is MB_BOOT_STARTABLE, communications between child and parent are completely severed.

## 2.2 Reconnecting with the Parent

Since the child's communication with the parent is severed once the child boots for Single-Card Play, the connection must be reestablished from the beginning.

Note the following when reestablishing a connection:

- The child's boot timing
  Because MB communication cannot occur at the same time as other WM communication under the current MB library, the parent device must terminate communication using the MB_End function after the child device boots up. (The MB_EndToIdle function is used to return to the IDLE state if the MB_StartParentFromIdle function was used for starting.) In order to reconnect and start communication between the parent device and child devices after booting for Single-Card Play, measures such as adjusting the timing of boot requests sent from the parent to the child are necessary.
- The connection process using parent information
  The child can obtain parent information before booting by using the ReadMultiBootParentBssDesc function. Direct connection to the parent is possible based on the WMBssDesc obtained this way, but the connection cannot be made if the parent's GGID and TGID differ from the GGID and TGID expected by the child device. Furthermore, communications may not be stable after the connection is established if the maximum size or the KS and CS flags differ, so be sure to prepare the application side ahead of time. You can prevent differences in the communication settings between parent and child by specifying the MAC address (bssid) found in WMBssDesc and rescanning for the parent.

- Handling TGIDs
  We recommend changing the parent device TGID when restarting the parent's wireless function to prevent a child device from mistakenly attempting to re-connect to a parent device before that parent's wireless function has been restarted and connecting from an unrelated IPL child device after the parent's wireless function is restarted.
  However, because the TGIDs between parent and child must be synchronized when connecting without rescanning by the child, be sure to set the TGID for the parent and child using a method such as incrementing the shared TGID by a fixed value.
- Parent multiboot flag
  Multiboot flag information is included in the parent information passed as an argument of the `WM_SetParentParameter` function, but do not set this flag under normal circumstances. The multiboot flag does not need to be set even when restarting the parent's wireless function and reconnecting after booting for Single-Card Play

## 2.3 Other Precautions

### 2.3.1 Applications with Multiple Communication Modes

If an application has multiple communication modes for both Multi-Card and Single-Card Play (such as a versus mode for Multi-Card Play and a Single-Card Play mode when using one card), trouble may occur because the parent can be viewed from different communication modes.

In cases where the child detects multiple communications modes, include ID information in userGameInfo set by the parent and have the child reference this ID during scanning. Note, however, that `userGameInfo` cannot be used with the MB library, so be sure to reference the `WM_ATTR_FLAG_MB` flag of `WMBssDesc.gameInfo.gameNameCount_attribute` to check whether or not the MB library is being used.

Another method of handling this is to obtain multiple GGIDs and distinguish different communication modes based on the GGID.

### 2.3.2 About the IRQ Stack

Please note that all callback functions operate in IRQ mode during wireless communication. When processing internal to a callback consumes a large amount of stack, the safe thing to do is set the IRQ stack size slightly larger in the `lcf` file.

The `OS_Printf` function used during debugging particularly consumes a large amount of stack, so be sure to use the `OS_TPrintf` lite version of the function inside callbacks whenever possible.

### 2.3.3 About the Single-Card Play Child Device Program Overlay

When a program running on a Single-Card Play child device uses the overlay feature, the overlay table and overlay segments to be included in the child's binary must be received separately from the parent device. The following points must be observed at this time to ensure the integrity of the received data.

- Specifying the `NITRO_DIGEST` and `NITRO_COMPRESS` build switches

  The build switches `NITRO_DIGEST` and `NITRO_COMPRESS` must be specified in the build of the Single-Card Play child program. This is required so the NITRO-SDK can accurately confirm that the overlay table and individual overlay segments correctly match the child's own. If the overlay feature is used without specifying these build switches, the program will be forced to halt on execution.

  Specifying these build switches is equivalent to calling the `compstatic.exe` tool with the `-a` and `-c` options.

  Note that these build switches are only necessary for applications and are ignored in SDK builds.

- Using the FS library functions

  In addition to the above build switch specifications, you must also use the FS library functions given below for overlay operations to guarantee that the NITRO-SDK has correctly checked the integrity of data.

  - Function always used:

    - `FS_AttachOverlayTable`

  - Function only used when loading is performed synchronously:

    - `FS_LoadOverlay`

  - Functions only used when loading is performed asynchronously:

    - `FS_LoadOverlayInfo`
    - `FS_LoadOverlayImage` or `FS_LoadOverlayImageAsync`
    - `FS_StartOverlay`

# 3 The Clone Boot Feature

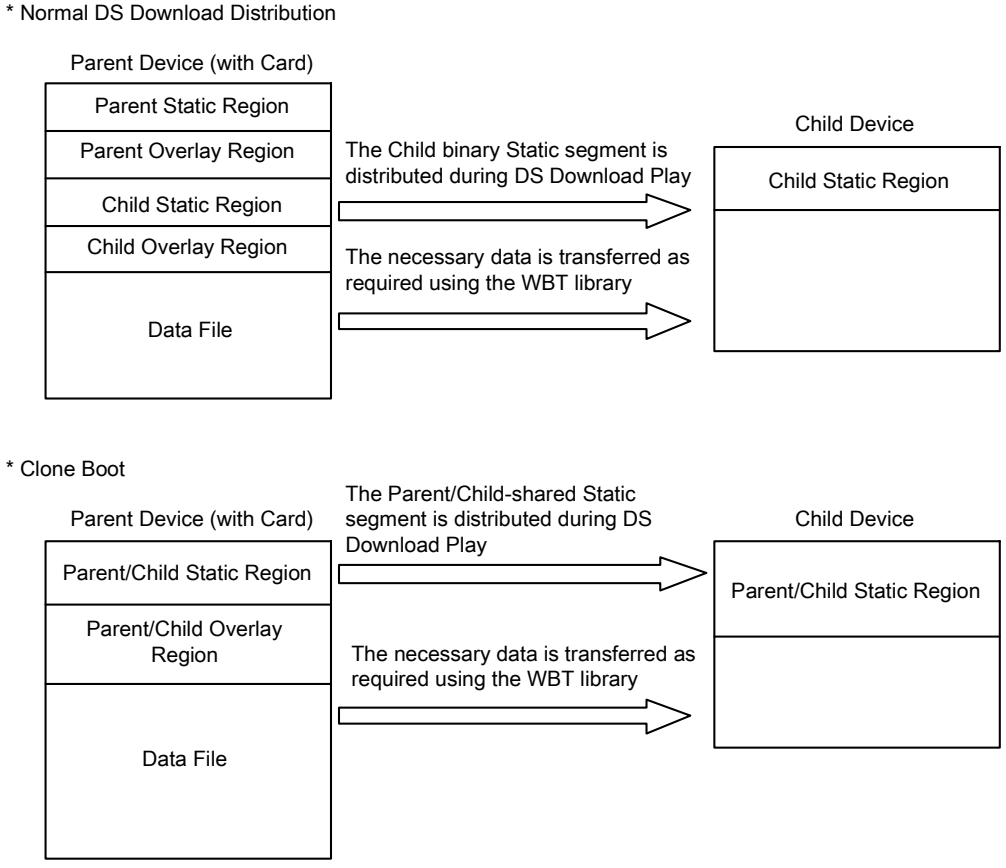
A clone boot feature that sends the Static segment of the parent device without modification to the child device and then boots the child for Single-Card Play is provided in the SDK. This section describes the clone boot procedure.

## 3.1 About Clone Boot

When clone boot is used, Static segments that are the same as the parent's are distributed to children. The parent and the booted children determine whether or not they are a Single-Card Play child device using the `MB_IsMultiBootChild` function. The process then branches. Data that is not included in the Static segment must be obtained by reconnecting to the parent after booting and then using the WBT library.

**Note:** As described in section 3.2, part of the Static segment is for the dedicated use of the parent.

**Figure 3-1 Clone Boot**



## 3.2 Clone Boot Procedure

The procedure for clone boot is described below.

### 3.2.1 Placing Data in ROM

Programs that support the clone boot feature can boot a Single-Card Play child in the same way they are booted from a card. The Multiboot library therefore provides security measures that are meant to avoid the complete reproduction of a game from the delivered data.

Programs that support the clone boot feature treat the data placed in the card's secure region (0x5000–0x6FFF) as data for the dedicated use of the parent and do not include it in the data delivered for Single-Card Play. As a security measure to prevent the reproduction and duplication of commercial programs, please use this region to store data that will definitely be used by the parent but not by any children. For details on configuring this parent-only region and storing data here, see the description of the `cloneboot` sample program in Chapter 5.

For details about the secure region found on cards, see the Programming Manual.

### 3.2.2 Authentication Code Attachment

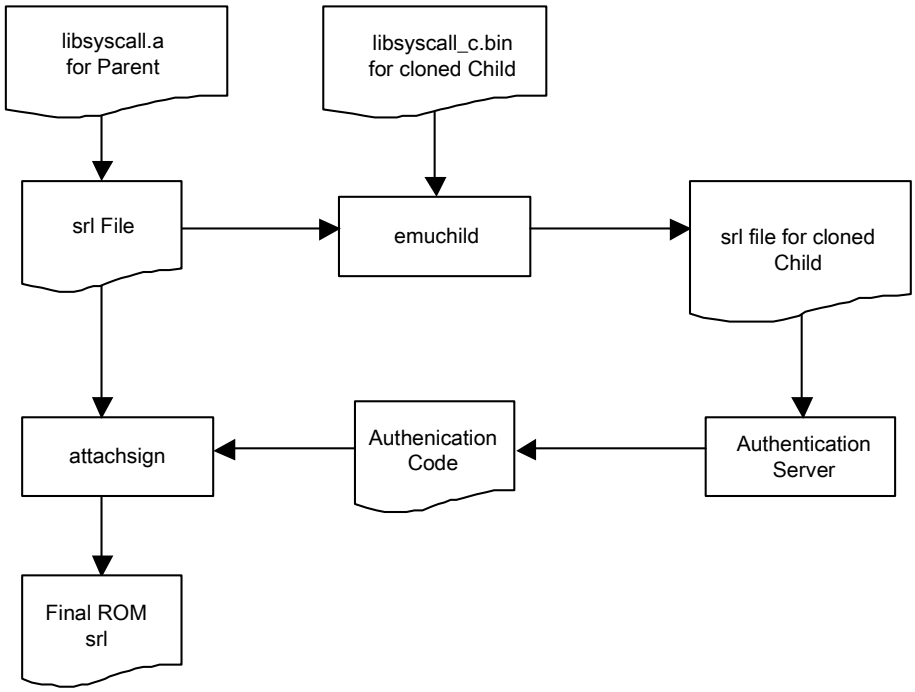
Normal Single-Card Play operations on a DS require that the binary for the child device has an authentication code attached. Clone boot also requires an attached authentication code.

In order to perform clone boot authentication, you must first obtain `libsyscall.a` used on the commercial version of the parent device and then the binary file (called `libsyscall_c.bin` below) corresponding to `libsyscall` for the clone child.

Executing `$NitroSDK/tools/bin/emuchild.exe` on the `srl` file created in the build extracts only the static segment necessary for Single Card Play and adds `libsyscall_c.bin` for children to create a binary file for signatures. (This binary file is henceforth referred to as `srl`.) Perform the same signature procedure on this file as used for normal Single-Card Play authentication, and attach the authentication code obtained to the original `srl` file.

Since the signature is inserted in the proper location with `attachsign when` padding is performed using `RomFootPadding` during ROM creation, the size of the `srl` file will not increase as long as there is enough space to insert the signature.

**Figure 3-2 Clone Boot Binary Authentication Procedure**



### 3.2.3 Clone Boot Binary Registration

Clone boot is activated by passing NULL as the child device binary file pointer when using the `MB_GetSegmentLength` and `MB_ReadSegment` functions in the MB library. Other processing is exactly the same as normal Single-Card Play.

**Code 3-1 Clone Boot Binary Registration Example**

```
// Obtain clone boot data segment size
bufferSize = MB_GetSegmentLength( NULL );
if ( bufferSize == 0 )
{
    return FALSE;
}
// Secure Memory
sFilebuf = OS_Alloc( bufferSize );
if ( sFilebuf == NULL )
{
    return FALSE;
}
// Extract segment information
if ( ! MB_ReadSegment( NULL, sFilebuf, bufferSize ) )
{
    OS_Free( sFilebuf );
    return FALSE;
}
// Register download program
if ( ! MB_RegisterFile( gameInfo, sFilebuf ) )
{
    OS_Free( sFilebuf );
    return FALSE;
}
```

# 4 The Sample Program (Multiboot-Model)

`multiboot-Model` is a sample program in which a parent sends a program to a child using the Single Card Play feature, and then data sharing communications between the parent and the child are performed by the sent program.

This chapter describes the following topics regarding the parent:

1. Preparing for the Single-Card Play feature
2. Initializing the parent
3. Starting parent operations
4. Waiting for connection from a child
5. Sending the program to the child
6. Restarting the child
7. Starting the parent application
8. States of the parent

This chapter then describes the following topics regarding the child:

1. Detecting Single-Card Play children
2. Getting connection information during Single-Card Play
3. Starting the child application

In the sample program, the series of MB library-related processes necessary to the parent for Single-Card Play are collected together in module format under `$NitroSDK/build/demos/wireless_shared/mbp`. Please use this module when actually creating programs that utilize the Single-Card Play feature. Note that you will also need to use `wh.h`, the Wireless Manager's wrapper module, when you use this module. For details about `wh.h`, see the "*Wireless Communications Tutorial*."

## 4.1 Single-Card Play Parent

This section describes the processing required of a parent using the Single-Card Play feature by tracing the control flow of the sample program.

### 4.1.1 Preparing for the Single-Card Play Feature

As mentioned in section 2.1.1.1, an open communication channel must be found before initializing the MB library to use the Single-Card Play feature.

The program code shown below searches for a communication channel. (Comments in the sample program that are unrelated to this description are omitted here.)

**Code 4-1 Search for Communication Channel**

```
static void GetChannelMain( void )
{
    (void)WH_Initialize();                                    1

   while ( TRUE )
   {
       switch ( WH_GetSystemState() )
       {

       //---------------------------------------
       // Initialization complete
       case WH_SYSSTATE_IDLE:                                 2
           (void)WH_StartMeasureChannel();
           break;

        //---------------------------------------
       // Channel search complete
       case WH_SYSSTATE_MEASURECHANNEL:                       3
           {
              sChannel = WH_GetMeasureChannel();
              (void)WH_End();
           }
           break;

       //---------------------------------------
       // End WM
       case WH_SYSSTATE_STOP:                                 4
          /* Go to Multiboot once WM_End is completed */
       return;
       //---------------------------------------
           // Busy
       case WH_SYSSTATE_BUSY:
           break;
           //---------------------------------------
           // Error generation
       case WH_SYSSTATE_ERROR:
           (void)WH_Reset();
           break;
           //---------------------------------------
       default:
           OS_Panic("Illegal State\n");
       }
      SVC_WaitVBlankIntr();        // Wait for V-Blank interrupt
  }}
```

The process begins at **1** using the `WH_Initialize` function to initialize the wireless communication feature. Once the send and receive buffers necessary for wireless communication are secured and initialized and the wireless communications hardware is initialized, the `WH_Initialize` function

changes the WM library state to IDLE.

Once the WM library state becomes IDLE (the state at **2**), the WM_MeasureChannel function can be used to check the signal traffic level on each channel. In the sample program, the WH_StartMeasureChannel function is called to search for the channel with the lowest traffic level.

Once the search for a channel ends (the state at **3**), the result of the search is obtained using the WH_GetMeasureChannel function. Since the search is complete and the communication channel is secured, end processing for the WM library is performed by calling the WH_End function. The WM library must be quit at this point because the MB library and the WM library cannot be used simultaneously.

Once the WM library is closed (the state at **4**), code stops searching for a communication channel and moves on to Single-Card Play processing.

For the rest of the procedure, you can simply move to the IDLE state if the MB_StartParentFromIdle function is being used. The program code is changed as shown below, exiting the process at the state at **3**.

```
//----------------------------------------
// Channel search complete
case WH_SYSSTATE_MEASURECHANNEL:                    3
   /* Move to MultiBoot process while maintaining IDLE  state */
   return;
//----------------------------------------
// Quit WM
...
```

## 4.1.2 The Single-Card Play Feature

Using the wireless channel that was just obtained, the Single-Card Play feature is initialized and other processes are carried out to accept children, deliver the download, and restart the children.

### 4.1.2.1 Initializing the Parent

The information delivered in the download, icon information, Single-Card Play game registration information registered for the GGID, the communication channel obtained in the search process, and the TGID are all used to initialize the parent.

To prevent connections from unexpected child devices, we recommend that a different TGID value be assigned each time the parent device is started.

The following program fragment initializes the parent. (Comments in the sample program unrelated to this description are omitted.)

**Code 4-2 Initialize the Parent**

```
static BOOL ConnectMain( u16 tgid )

{
    MBP_Init( mbGameList.ggid, tgid );

    while ( TRUE )
    {
        --- Omitted ---
    }
}
```

In the sample program, the `MBP_Init` function initializes the parent and sets the necessary information (in step 1 above). The `MBP_Init` function sets the parent player information to be displayed on the screens of the children and initializes the MB library.

**Code 4-3 Set the Parent User Information and Initialize the MB Library**

```
void MBP_Init( u32 ggid, u16 tgid )
{
    /* Set parent information to appear on screens of children */
    MBUserInfo     myUser;

    OSOwnerInfo info;

    OS_GetOwnerInfo( &info );                                          2
    myUser.favoriteColor = info.favoriteColor;
    myUser.nameLength = (u8)info.nickNameLength;
    MI_CpuCopy8( &myUser.name, info.nickName, OS_OWNERINFO_NICKNAME_MAX * 2 );

    myUser.playerNo = 0;      // Parent is number 0                   3

    // Initialize the status information
    mbpState    = (const MBPState) { MBP_STATE_STOP, 0, 0, 0, 0, 0, 0 };

    /* Begin MB parent control. */
    // Secure MB work region.
    sCWork = OS_Alloc( MB_SYSTEM_BUF_SIZE );                           4

    if ( MB_Init( sCWork, &myUser, ggid, tgid, MBP_DMA_NO )
         != MB_SUCCESS )
    {
       OS_Panic( "ERROR in MB_Init\n" );
    }
    MB_CommSetParentStateCallback( ParentStateCallback );              5

    MBP_ChangeState( MBP_STATE_IDLE );
}
```

Using the `MBP_Init`, function, you can set the parent's player information regarding the player's nickname and favorite colors as obtained from the IPL owner information. For more information, see section 2.1.1.2 "Setting the Parent's Parameters."

In step (4), a work region is allocated for use by the MB library and then the `MB_Init` function is used to initialize the MB library.

In step (5), a callback function is set for changing the parent state as notified by the MB library. Processing for the notified parent state is performed inside this callback function.

As for the rest of the procedure, because the IDLE state is maintained when using the `MB_StartParentFromIdle` function as described above in section 4.1.1, the amount of memory allocated can be reduced by changing the previously described program as shown below. (However, there are no particular problems with having a buffer that is too big.)

```
// Secure MB work region.
sCWork = OS_Alloc( MB_SYSTEM_BUF_SIZE - WM_SYSTEM_BUF_SIZE );  4
...
```

#### 4.1.2.2 `The Start of Operations by the Parent

After the MB library is initialized by the `MB_Init` function, the next step is to start a DS device as the Single-Card Play parent and register the file to use for wireless downloads.

The following program code starts the operations of the parent. (Comments in the sample program unrelated to this description have been omitted.)

**Code 4-4 Start Parent Operations**

```
static BOOL ConnectMain( u16 tgid )
{                                                          1
    --- Omitted ---

   while ( TRUE )
   {
      switch ( MBP_GetState() )
      {
      //-----------------------------------------
      // IDLE state
      case MBP_STATE_IDLE :
         {
            MBP_Start( &mbGameList, sChannel );
         }
         break;

    --- Omitted ---

      }
   }
}
```

After processing by the MBP_Init is complete (the state in step 1), the MBP_Start function starts the Single-Card Play feature and registers the information for the program that will be wirelessly downloaded after the parent has accepted connections from children.

**Code 4-5 Start Single-Card Play Parent and Register File**

```
void MBP_Start( const MBGameRegistry *gameInfo, u16 channel )
{
    SDK_ASSERT( MBP_GetState() == MBP_STATE_IDLE );

    MBP_ChangeState( MBP_STATE_ENTRY );
    if ( MB_StartParent( channel ) != MB_SUCCESS )            3
    {
        MBP_Printf("MB_StartParent fail\n");
        MBP_ChangeState( MBP_STATE_ERROR );
        return;
    }

    /* ------------------------------------------------------------------------ *
     * Initialized when MB_StartParent() is called.
     * You must register MB_RegisterFile() after MB_StartParent().
     * ------------------------------------------------------------------------ */

    /*  Register download-program file information. */         4
    if ( ! MBP_RegistFile( gameInfo ) )
    {
        OS_Panic("Illegal Single-Card Play gameInfo\n");
    }
}
```

In step 3, the MB_StartParent function is called with the communication channel specified as an argument to start operations as the Single-Card Play parent.

Because the download program information is initialized when the MB_StartParent function is called, you must call the MB_RegisterFile function to register the download program information after the MB_StartParent function has been called.

In step 4 of the sample program, the MBP_RegistFile function is called to load the binary code to be sent for Single-Card Play into main memory and register download program information.

The download program information used in the sample program is configured as shown below:

**Code 4-6 Load Program in Memory and Register Program Information**

```
/* This is the program information the demo downloads */
const MBGameRegistry mbGameList =
{
    "/child.srl",               // Child binary code
    (u16*)L"DataShareDemo",     // Game name
    (u16*)L"DataSharing demo", // Description of the game contents
    "/data/icon.char",          // Icon character data
    "/data/icon.plt",           // Icon palette data
    WH_GGID,                    // GGID
    MBP_CHILD_MAX + 1,          // Maximum number of players
};
```

If the MB_StartParentFromIdle function is being used, the code at 3 is changed as shown below to handle those changes described in sections 4.1.1. and 4.1.2.

```
    MBP_ChangeState( MBP_STATE_ENTRY );
    if ( MB_StartParentFromIdle( channel ) != MB_SUCCESS )     1
    {
    ...
```

Next, a description is given regarding the registration of download program information by tracing the process flow in the `MBP_RegistFile` function.

In step 5, the File System is used to open the download file to be registered so it can be loaded.

The `MBP_RegistFile` function also supports the clone boot feature (described below). If the file path name received is NULL, software will behave as if a clone boot has been specified.

**Code 4-7 How to Register File: Open the File**

```
static BOOL MBP_RegistFile( const MBGameRegistry* gameInfo )
{
    FSFile file, *p_file;
    u32 bufferSize;
    BOOL ret = FALSE;

    /*
     * In accordance with the specification for this function, if
     * romFilePathp is NULL, it operates as a clone boot. Otherwise,
     * the specified file is treated as the child program.
     */
    if gameInfo->romFilePathp
    {
        p_file = NULL;
    }
    else
    {
        /*
         * The program file must be read by FS_ReadFile().
         * Normally, the program is saved as a file in CARD-ROM, so
         * this is not a problem.  However, if you anticipate there being,
         * a special MultiBoot file system, deal with the situation by
         * using FSArchive to construct an independent archive.
         */
    FS_InitFile( &file );
    if ( ! FS_OpenFile( &file, gameInfo->romFilePathp ) )        5
    {
        /* File cannot be opened */
        OS_Warning("Cannot Register file\n");
        return FALSE;
    }
        p_file = &file;
    }
     --- Omitted ---
}
```

Next, the `MB_GetSegmentLength` function obtains the size of the segment information in step 6 and then memory is allocated for loading the segment information in step 7.

Since only one file is maintained for the segment information in the sample program, you must switch to processing that maintains multiple sets of segment information if you plan to register multiple download files.

**Code 4-8 How to Register File: Get Segment Size and Memory**

```
static BOOL MBP_RegistFile( const MBGameRegistry* gameInfo )
{
    FSFile file, *p_file;
    u32 bufferSize;
    BOOL ret = FALSE;

     --- Omitted ---

    /*
     * Get the size of the segment information.
     * If download program is not legal, 0 is
     * returned for the size.
     */
    bufferSize = MB_GetSegmentLength( &file );              6
    if ( bufferSize == 0 )
    {
        OS_Warning( "specified file may be invalid format.\"%s\"\n",
                    gameInfo->romFilePathp );
    }
    else
    {
        /*
         * Secure memory for loading the download program's segment
         * information.  If file has been registered successfully,
         * this region will be used until MB_End() is called.
         * If the memory size is plenty large enough, it can be
         * prepared statically.
         */
        sFilebuf = (u8*)OS_Alloc( bufferSize );             7
        if ( sFilebuf == NULL )
        {

      /* Failure to secure buffer for storing segment information */
        OS_Warning("can't allocate Segment buffer size.\n");
else      }

     --- Omitted ---

}
```

The segment information is read from the file using the `MB_ReadSegment` function in step 8 and registered using the `MB_RegisterFile` function in step 9. Once the download file is registered, the open download file is closed in step 10 because it is no longer needed.

**Code 4-9 How to Register File: Read and Register Segment Information, Close File**

```
static BOOL MBP_RegistFile( const MBGameRegistry* gameInfo )
{

    --- Omitted ---

   /*
    * Extract segment information from file.
    * This extracted information must remain resident in
    * main memory while the download program is being delivered.
    */
        if ( ! MB_ReadSegment( &file, sFilebuf, bufferSize ) )       8
          {
             /*
              * Segment extraction from illegal file will fail.
              * If size is obtained successfully but the extraction
              * process fails anyway, it may be because some change has
              * been made to the file handle.  (File closed,
              * location seek, etc.)
              */
             OS_Warning(" Can't Read Segment\n" );
          }
          else
          {
             /*
              * Register Download program with extracted segment
              * information and MBGameRegistry.
              */
             if ( ! MB_RegisterFile( gameInfo, sFilebuf ) )          9
             {
                /* Registration fails due to illegal program information */
                OS_Warning(" Illegal program info\n");
             }
             else
             {
                /* Process has ended correctly */
                ret = TRUE;
             }
          }
          if (!ret)
             OS_Free(sFilebuf);
       }
   }
   /* Close file if not a clone boot */
   if (p_file == &file)
   {
       /*
        * Segment extraction from illegal file will fail.
        * If the extraction process fails even though the
        * size has been obtained successfully, it may be
        * because some change has been made to the file handle.
        * (File close, location seek, ...)
        */
       OS_Warning(" Can't Read Segment\n" );
       (void)FS_CloseFile( &file );
       OS_Free( sFilebuf );
       return FALSE;
   }
```

```
        /*
         * Register download program with extracted
         * segment information and MBGameRegistry.
         */
        if ( ! MB_RegisterFile( gameInfo, sFilebuf ) )          9
        {
            /* Registration fails due to illegal program information */
            OS_Warning(" Illegal program info\n");
            (void)FS_CloseFile( &file );
            OS_Free( sFilebuf );
            return FALSE;
        }

        // Close the file
        (void)FS_CloseFile( &file );                            10
        return TRUEret;
    }
```

At this point, the game device begins operating as a Single-Card Play parent and the registered download file is distributed to children via download.

#### 4.1.2.3 Waiting for a Connection from the Child

Once the game device begins operating as a Single-Card Play parent, it processes connection requests from children.

The callback function set with `MB_CommSetParentStateCallback` is notified of connection requests from children as given in the code described in section 4.1.2.1. Since a variety of notifications in addition to connection requests from children are posted to this callback function, processing appropriate for each type of notification is required.

In the sample program, the state in which the parent waits for and accepts connection requests from children is defined as "`MBP_STATE_ENTRY` (accepting connection requests)."

Connection requests from children are denied if the value returned by the `MBP_GetState` function (used to get the parent state) is other than `MBP_STATE_ENTRY`.

There are two states in which children make connection requests:

- `MB_COMM_PSTATE_CONNECTED`, which indicates that the child is connected to the parent
- `MB_COMM_PSTATE_REQUESTED`, which indicates an entry request as a Single-Card Play child

In the sample program, information for managing a child's connection (`mbpState.connectChildBmp`) is updated when the parent receives notification of `MB_COMM_PSTATE_CONNECTED` from the child in question.

**Code 4-10 Parent Receives Child Notification–Update Connection Information**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // Instant notification of child connection
    case MB_COMM_PSTATE_CONNECTED:
        {
            // Parent does not accept connection except in entry reception state
            if ( MBP_GetState() != MBP_STATE_ENTRY )
            {
                break;
            }

            MBP_AddBitmap( &mbpState.connectChildBmp, child_aid );
            // Store child's MacAddress
            WM_CopyBssid( ((WMStartParentCallback*)arg )->macAddress,
                          childInfo[ child_aid - 1 ].macAddress );
            childInfo[ child_aid - 1 ].playerNo = child_aid;
        }
        break;
    }
}
```

When notification of MB_COMM_PSTATE_REQUESTED is posted in a callback function, a decision is made to either accept (2) or deny (1) the entry request.

In the sample program, except in cases where the entry request is denied due to the state of the parent, all entry requests are accepted using the MBP_AcceptChild function and the information for managing child entry requests is updated (mbpState.requestChildBm). The player information of children is obtained using the MB_CommGetChildUser function.

**Code 4-11 Process Connection Request**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // Instant notification of entry request from child
    case MB_COMM_PSTATE_REQUESTED:
        {
            const MBUserInfo* userInfo;

            // If the parent is not in an entry-accept state, the child
            // requesting entry is kicked out without being checked.
            if ( MBP_GetState() != MBP_STATE_ENTRY )
            {
                MBP_KickChild( child_aid );                                    1
                break;
            }

            // Accept child's entry
            mbpState.requestChildBmp |= 1 << child_aid;

            MBP_AcceptChild( child_aid );                                      2

            // The timing of MB_COMM_PSTATE_CONNECTED is such that UserInfo
            // is not set, so MB_CommGetChildUser has no meaning unless it
            // is called after state is REQUESTED.
            userInfo = MB_CommGetChildUser( child_aid );
            if ( userInfo != NULL )
            {
             MI_CpuCopy8(userInfo, &childInfo[ child_aid - 1 ].user, sizeof(MBUserInfo));
            }
            MBP_Printf("playerNo = %d\n", userInfo->playerNo );
        }
        break;
    }
}
```

If the connection request from a child is accepted, the MB_CommResponseRequest function notifies the child by posting MB_COMM_RESPONSE_REQUEST_ACCEPT. If the connection request is denied, the function notifies the child by posting MB_COMM_RESPONSE_REQUEST_KICK.

In the sample program, the information for managing child connections is updated when the notification is posted to the child.

**Code 4-12 Accept or Kick Child Connection**

```
void MBP_AcceptChild( u16 child_aid )                                    1
{
    if ( ! MB_CommResponseRequest( child_aid, MB_COMM_RESPONSE_REQUEST_ACCEPT ) )
    {
        // If a request fails, disconnect that child.
        MBP_DisconnectChild( child_aid );
        return;
    }

    MBP_Printf("accept child %d\n", child_aid);
}

void MBP_KickChild( u16 child_aid )                                      2
{
    if ( ! MB_CommResponseRequest( child_aid, MB_COMM_RESPONSE_REQUEST_KICK ) )
    {
        // If a request fails, disconnect that child.
        MBP_DisconnectChild( child_aid );
        return;
    }

    {
        OSIntrMode enabled = OS_DisableInterrupts();

        mbpState.requestChildBmp &= ~( 1 << child_aid );
        mbpState.connectChildBmp &= ~( 1 << child_aid );

        (void)OS_RestoreInterrupts( enabled );
    }
}
```

Children who receive `MB_COMM_RESPONSE_REQUEST_KICK` from a parent are disconnected from that parent. A callback function posts `MB_COMM_PSTATE_KICKED` to notify the parent that the child received the connection-denied response.

When the parent posts `MB_COMM_RESPONSE_REQUEST_ACCEPT` to a child, the child transits to a state where it can accept download delivery.

First, a callback function posts `MB_COMM_PSTATE_REQ_ACCEPTED` to notify the parent that the child received the connection-accepted response. Then a callback function posts `MB_COMM_PSTATE_WAIT_TO_SEND` to notify the parent that the child entered a state that accepts download delivery. Data transfer to the child will not execute properly if it begins before the parent receives this notification.

In the sample program, nothing happens when MB_COMM_PSTATE_KICKED and MB_COMM_PSTATE_ACCEPTED are posted, but when MB_COMM_PSTATE_WAIT_TO_SEND is posted, the information for managing child connections is updated and, depending on the state of the parent, download delivery to that child begins. (For more details on the MBP_StartDownload function, see section 4.1.2.4.)

**Code 4-13 Determine Child State, Begin Program Download**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // Post ACK to child for ACCEPT
    case MB_COMM_PSTATE_REQ_ACCEPTED:
        // No special process at this point.
        break;
    //---------------------------------------------------------
    // Notification to child when kicked.
    case MB_COMM_PSTATE_KICKED:
        // No particular process is required.
        break;
    //---------------------------------------------------------
    // Notification when download request received from child.
    case MB_COMM_PSTATE_WAIT_TO_SEND:
        {
            // Child's state changes from entry to download-wait.
            // An interrupted process, so changed without
            // prohibiting interrupts.
            mbpState.requestChildBmp  &= ~( 1 << child_aid );
            mbpState.entryChildBmp    |= 1 << child_aid;

            // Calling MBP_StartDownload() from main routine starts data
            // transmission.  If already in the data-transmission state,
            // data transfer also begins to that child.
            if ( MBP_GetState() == MBP_STATE_DATASENDING )
            {
                MBP_StartDownload( child_aid );
            }
        }
        break;
    }
}
```

In the wait portion of the connection process (1), download delivery to a child can begin (4) if there is a child in a state that can accept download delivery (3). Conversely, the Single-Card Play feature can be cancelled (2).

**Code 4-14 Begin Download Delivery or Cancel Single-Card Play**

```
static BOOL ConnectMain( u16 tgid )
{
while ( TRUE )
    {
        switch ( MBP_GetState() )
        {
        //---------------------------------------
        // Waiting for entry from child
        case MBP_STATE_ENTRY :                                              1
            {
                BgSetMessage( PLTT_WHITE, " Now Accepting          ");

                if ( IS_PAD_TRIGGER( PAD_BUTTON_B ) )
                {
                // B Button cancels Single-Card Play
                    MBP_Cancel();                                           2
                    break;
                }

                // Can start if there is at least one child in entry
                if ( MBP_GetChildBmp( MBP_BMPTYPE_ENTRY ) ) ||
                    MBP_GetChildBmp(MBP_BMPTYPE_DOWNLOADING) ||
                    MBP_GetChildBmp(MBP_BMPTYPE_BOOTABLE))                  3
                {
                    BgSetMessage( PLTT_WHITE, " Push START Button to start   ");

                    if ( IS_PAD_TRIGGER( PAD_BUTTON_START ) )
                    {
                        // Start download
                        MBP_StartDownloadAll();                             4
                    }
                }
            }
            break;
        }
    }
}
```

#### 4.1.2.4 Sending the Program to the Child

Once `MB_COMM_PSTATE_WAIT_TO_SEND` is posted, the parent can begin download delivery to the child that posted the notification. Download delivery is started using either the `MB_CommStartSending` or `MB_CommStartSendingAll` function. To use the `MB_CommStartSendingAll` function, first check that all connected children can accept download delivery. Calling the function once may not begin download delivery to all the children.

Because download delivery cannot begin for children that are not in the `MB_COMM_PSTATE_WAIT_TO_SEND` state, be sure to start download delivery separately for each child if a `MB_COMM_PSTATE_WAIT_TO_SEND` notification is received after the `MB_CommStartSendingAll` function has executed.

In the sample program, the `MB_CommStartSending` function is used inside the `MBP_StartDownload` function and the connection state is updated for all children for which download delivery has started.

**Code 4-15 Disable Interrupts, Begin Download**

```
void MBP_StartDownload( u16 child_aid )
{
    if ( ! MB_CommStartSending( child_aid ) )
    {
        // If a request fails, disconnect that child.
        MBP_DisconnectChild( child_aid );
        return;
    }

    {
        OSIntrMode enabled = OS_DisableInterrupts();

        mbpState.entryChildBmp &= ~(1 << child_aid);
        mbpState.downloadChildBmp |= 1 << child_aid;

        (void)OS_RestoreInterrupts( enabled );
    }
}
```

When using the MBP_StartDownloadAll function, after the connection request is accepted and the parent enters the download-delivering state represented by MBP_STATE_DATASENDING (**1**), the connect state of the children is checked and the MBP_StartDownload function begins download delivery to those children that can accept the download (**4**). If the parent accepts a connection request from a child, but the child is not in a state to accept delivery, the download begins later when the child enters the receptive state (**2**). Children in other states are disconnected.

**Code 4-16 Verify Child States, Begin Download**

```
void MBP_StartDownloadAll( void )

{
    u16 i;

    // Entry acceptance completed
    MBP_ChangeState( MBP_STATE_DATASENDING );                          1

    for ( i = 1; i < 16; i++ )
    {
        if ( ! ( mbpState.connectChildBmp & (1 << i) ) )
        {
            continue;
        }

        if ( mbpState.requestChildBmp & (1 << i) )                     2
        {
            // Perform this process when currently entered children are ready
            //  and the MB_COMM_PSTATE_WAIT_TO_SEND notification is received.
            continue;
        }

        // Disconnect children that are not entered
        if ( ! ( mbpState.entryChildBmp & (1 << i) ) )                 3
        {
            MBP_DisconnectChild( i );
            continue;
        }

        // Start download for entered children
        MBP_StartDownload( i );                                        4
    }
}
```

Inside the MB_CommStartSending function, MB_COMM_RESPONSE_REQUEST_DOWNLOAD (start delivery response) is posted to children. The child receives this post and confirms that download delivery has started by posting MB_COMM_PSTATE_SEND_PROCEED in a callback function. When download delivery to the child is complete, MB_COMM_PSTATE_SEND_COMPLETE is posted in a callback function.

In the sample program, nothing happens when MB_COMM_PSTATE_SEND_PROCEED is posted, but information for managing child connections is updated when MB_COMM_PSTATE_SEND_COMPLETE is posted.

**Code 4-17 Notify when Download Begins and Ends**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // Notify when binary transmission to child begins
    case MB_COMM_PSTATE_SEND_PROCEED:
        // None.
        break;
    //---------------------------------------------------------
    // Notify when binary transmission to child ends
    case MB_COMM_PSTATE_SEND_COMPLETE:
        {
            // An interrupted process, so changed without
            // prohibiting interrupts.
            mbpState.downloadChildBmp &= ~( 1 << child_aid );
            mbpState.bootableChildBmp |= 1 << child_aid;
        }
        break;
    }
}
```

#### 4.1.2.5 Restarting the Child

The child can be restarted once download delivery to the child is complete. The `MB_CommIsBootable` function checks whether the child can be rebooted. In the sample program, the `MBP_IsBootableAll` function checks whether all connected children are in a state that allows rebooting.

**Code 4-18 Check Whether Children Are Bootable**

```
BOOL MBP_IsBootableAll( void )
{
    u16 i;

    if ( mbpState.connectChildBmp == 0 )
    {
        return FALSE;
    }

    for ( i = 1; i < 16; i++ )
    {
        if ( ! ( mbpState.connectChildBmp & (1 << i) ) )
        {
            continue;
        }

        if ( ! MB_CommIsBootable( i ) )
        {
            return FALSE;
        }
    }
    return TRUE;
}
```

If download delivery is complete for all children, a reboot request is sent to the children.

**Code 4-19 Reboot Children when Download Is Complete**

```
static BOOL ConnectMain( u16 tgid )
{
     --- Omitted ---

    while ( TRUE )
    {
        //---------------------------------------
        // Process for sending program
        case MBP_STATE_DATASENDING :
            {

                // Can start if all parties have finished downloading.
                if ( MBP_IsBootableAll() )
                {
                    // Start boot
                    MBP_StartRebootAll();
                }
            }
            break;

     --- Omitted ---
        }
    }
}
```

The reboot request sent to children is made using either the `MB_CommBootRequest` or `MB_CommBootRequestAll` function. If you use the `MB_CommBootRequestAll` function, first verify that downloading to all connected children is complete. Calling the function once may not result in a request for all children to reboot.

In the sample program, the child reboot request is made using the MBP_StartRebootAll function. The connection state of all children is checked inside the MBP_StartRebootAll function and the reboot request is made using the MB_CommBootRequest function. The state of the parent is then changed to MBP_STATE_REBOOTING (wait for child reboot).

**Code 4-20 Change Parent State, Continue Booting Children**

```
void MBP_StartRebootAll( void )
{
    u16 i;
    u16 sentChild = 0;

    for ( i = 1; i < 16; i++ )
    {
        if ( ! ( mbpState.bootableChildBmp & (1 << i) ) )
        {
            continue;
        }
        if ( ! MB_CommBootRequest( i ) )
        {
            // If a request fails, disconnect that child.
            MBP_DisconnectChild( i );
            continue;
        }
        sentChild |= ( 1 << i );
    }

    // Error: exit if connection child is 0
    if ( sentChild == 0)
    {
        MBP_ChangeState( MBP_STATE_ERROR );
        return;
    }

    // Change state to child device restart wait state.
    MBP_ChangeState( MBP_STATE_REBOOTING );
}
```

The MB_COMM_RESPONSE_REQUEST_BOOT (reboot request) notification is sent to children from inside the MB_CommBootRequest function. Each child receives the reboot request and posts MB_COMM_PSTATE_BOOT_STARTABLE from inside a callback function when finished rebooting.

Because wireless communications between the parent and the child are disconnected when the child is done rebooting, MB_COMM_PSTATE_DISCONNECTED is posted in a callback function.

In the sample program, if MB_COMM_PSTATE_BOOT_STARTABLE has been posted, the information for managing the connections of children is updated and the Single-Card Play feature is ended using the MB_End function after all children are done rebooting.

**Code 4-21 Verify that Download Is Complete, Disconnect Children**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // Notification that child boot ended correctly
    case MB_COMM_PSTATE_BOOT_STARTABLE:
        {
            // An interrupted process, so changed without
            // prohibiting interrupts.
            mbpState.bootableChildBmp &= ~( 1 << child_aid );
            mbpState.rebootChildBmp |= 1 << child_aid;

            // If all children are done booting, the parent
            // also enters the reconnection process.
            if ( mbpState.connectChildBmp == mbpState.rebootChildBmp )
            {
                MBP_Printf("call MB_End()\n");
                MB_End();
            }
        }
        break;
    //---------------------------------------------------------
    // Notification when child is disconnected
    case MB_COMM_PSTATE_DISCONNECTED:
        {
            // Delete entry if child disconnects in situation
            // other than rebooting.
            if ( MBP_GetChildState( child_aid ) != MBP_CHILDSTATE_REBOOT )
            {
                MBP_DisconnectChildFromBmp( child_aid );
            }
        }
        break;
    }
}
```

If changes have been made to this code so that the MB_StartParentFromIdle function is used, make the following changes to call the MB_EndToIdle function at the end instead of the MB_End function.

```
            // If all children have finished booting, then the
            // parent also enters the reconnection process.
            if ( mbpState.connectChildBmp == mbpState.rebootChildBmp )
            {
                MBP_Printf("call MB_EndToIdle()\n");
                MB_EndToIdle();
            }
            ...
```

When the Single-Card Play feature is ended by the MB_End function, the notification MB_COMM_PSTATE_END is posted by a callback function. In the sample program, the parent is moved to the process-end state (MBP_STATE_COMPLETE) and the work area in memory allocated for download delivery is released.

**Code 4-22 End Single-Card Play, Change Parent State, Clear Buffer**

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------------
    // Notification at end of Single-Card Play
    case MB_COMM_PSTATE_END:
        {
            if ( MBP_GetState() == MBP_STATE_REBOOTING )
            // An end of reboot process, end MB and
            // reconnect with child.
            {
                MBP_ChangeState( MBP_STATE_COMPLETE );
            }
            else
            // Complete shutdown, return to STOP state
            {
                MBP_ChangeState( MBP_STATE_STOP );
            }
            // Release the buffer used for game delivery.
            // The work region is released when MB_COMM_PSTATE_END
            // comes in a callback, so OK to free.
            if ( sFilebuf )
            {
                OS_Free( sFilebuf );
                sFilebuf = NULL;
            }
            if ( sCWork )
            {
                OS_Free( sCWork );
                sCWork = NULL;
            }
            // The registration info is cleared at the same time MB_End is
            // called and work is freed, so MB_UnregisterFile can be omitted
        }
        break;
    }
}
```

In the final step, the code fragment for quitting the Single-Card Play feature (**2**) is referenced from the rebooting process (**1**).

**Code 4-23 End Reboot, Reconnect Wireless Communications**

```
static BOOL ConnectMain( u16 tgid )
{
     --- Omitted ---

    while ( TRUE )
    {
        //---------------------------------------
        // Reboot process
        case MBP_STATE_REBOOTING:                                    1
            {
                BgSetMessage( PLTT_WHITE, " Rebooting now             ");
            }
            break;

        //---------------------------------------
        // Reconnection process
        case MBP_STATE_COMPLETE :                                    2
            {
                // If all parties connect without trouble,
                // quit Single-Card Play process and restart
                // wireless communications as a normal parent.
                BgSetMessage( PLTT_WHITE, " Reconnecting now           ");

                SVC_WaitVBlankIntr();
                return TRUE;
            }
            break;

     --- Omitted ---
        }
    }
}
```

## 4.1.3 Starting the Parent Application

In the multiboot-Model sample program, game software is downloaded to children using the Single-Card Play feature. After the child reboots the wireless-communications parent shares data with the program downloaded to the child. Because the wireless communication connection with the child is cut when the Single-Card Play feature ends, the wireless connection with the child must be reestablished.

In the sample program, the connection information used by the Single-Card Play feature is used for data sharing. The first step is to perform the initialization processes required for data sharing. The `GInitDataShare` function makes the initial settings for the buffer to be used for data-sharing communications. The `WH_Initialize` function initializes the WM library and wireless communications.

If changes have been made in the code in order to use the `MB_StartParentFromIdle` function, it does not need to be called at this point because the IDLE state is being maintained and the `WH_Initialize` function has already been called.

**Code 4-24 Initialize Data Sharing, the WM Library, and Wireless Communications**

```
// Configure the buffer for data-sharing communications
GInitDataShare();
// If MB_StartParent & MB_End have been used, then initialize
// wireless communications at this point
(void)WH_Initialize();
```

Once wireless communications start, the parent may receive connection requests from devices other than the children to which the program has been delivered using the Single-Card Play feature. To handle this possibility, the `WH_SetJudgeAcceptFunc` function sets the function to be used in deciding whether or not to allow the connection.

**Code 4-25 Process Connection Requests**

```
// Configure the function for deciding connection to children
WH_SetJudgeAcceptFunc( JudgeConnectableChild );
```

The `JudgeConnectableChild` function is used to make this determination in the code below. The connection is permitted if the player number (`aid`) used during Single-Card Play can be obtained from the MAC address of the terminal connected in step 1.

**Code 4-26 Process Connection Request–Details**

```
static BOOL JudgeConnectableChild( WMStartParentCallback* cb )
{
    u16 playerNo;

    /*  Search for cb->aid child's multiboot-time aid from MAC address */
    playerNo = MBP_GetPlayerNo( cb->macAddress );                          1

    OS_TPrintf( "MB child(%d) -> DS child(%d)\n", playerNo, cb->aid );

    if ( playerNo == 0 )
    {
        return FALSE;
    }
    sChildInfo[ playerNo ] = MBP_GetChildInfo( playerNo );
    return TRUE;
}
```

Finally, wireless communications with this unit as the parent are started and data sharing begins.

Because the state is `WH_SYSSTATE_IDLE` (1) when the `WH_Initialize` function ends, the `WH_ParentConnect` function is used to start wireless communications. The arguments for the function include `WH_CONNECTMODE_DS_PARENT` (used to indicate data-sharing) and the TGID and communication channel used by the Single-Card Play feature.

Once wireless communications begin, the state changes to `WH_SYSSTATE_DATASHARING` (2) and data sharing begins.

**Code 4-27 Change State and Share Data**

```
/* Main routine */
for ( gFrame = 0 ; TRUE ; gFrame++ )
{
    SVC_WaitVBlankIntr();

    ReadKey();

    BgClear();

    switch ( WH_GetSystemState() )
    {
    case WH_SYSSTATE_IDLE :                                          1
        /* -----------------
         * If you want the child to reconnect to the same parent
         * without rescanning, then tgid and channel must match.
         * In this demo, both the parent and the child use the same
         * channel as that at time of the multiboot, and tgid+1 compared
         * to the tgid at the time of multiboot.  For this reason, the
         * child can reconnect without scanning.
         *
         * If you are going to specify a MAC address and rescan,
         * the tgid and channel values do not need to be the same.
         * ---------------- */
        (void)WH_ParentConnect( WH_CONNECTMODE_DS_PARENT, tgid, sChannel );
        break;

    case WH_SYSSTATE_CONNECTED:
    case WH_SYSSTATE_KEYSHARING:
    case WH_SYSSTATE_DATASHARING:                                    2
        {
            BgPutString( 8 , 1 , 0x2 , "Parent mode" );
            GStepDataShare( gFrame );
            GMain();
        }
        break;
    }
}
```

### 4.1.4 States of the Parent

The MBP_GetState function can obtain the following states of the parent:

**Table 4-1 The Parent States**

| Values Returned by the MBP_GetState Function | The State of the Parent |
|---|---|
| **MBP_STATE_STOP** | The MB_End function was called from the MBP_Cancel function and the Single-Card Play feature was stopped. |
| **MBP_STATE_IDLE** | The MBP_Init function finished, the MBP_Start function was called, and the device can begin operating as the parent. |
| **MBP_STATE_ENTRY** | The MBP_Start function finished and the parent is waiting for a connection from a child. This is the only state in which the parent can accept a connection from a child. |
| **MBP_STATE_DATASENDING** | The MBP_StartDownloadAll function was called and download-delivery to the connected children has begun. |
| **MBP_STATE_REBOOTING** | The MBP_StartRebootAll function was called and connected children are rebooting. |
| **MBP_STATE_COMPLETE** | All connected children received reboot requests and the Single-Card Play feature was ended by the MB_End function. |
| **MBP_STATE_CANCEL** | The MBP_Cancel function was just called. |
| **MBP_STATE_ERROR** | An error has occurred. |

## 4.2 Single-Card Play Children

The user program for a Single-Card Play child starts after Single-Card Play data is transferred from the parent device and the child is rebooted. During reboot, the connection with the parent device is completely terminated.

In this section, the sample program multiboot-Model is used to describe how Single-Card Play children are determined and how to obtain connection information used during Single-Card Play.

### 4.2.1 Single-Card Play Child Determination

The child uses the MB_IsMultiBootChild function to determine whether it was started using the Single-Card Play feature.

**Code 4-28 Check Whether Child Booted by Single-Card Play**

```
// Check to see if self is child that started from Single-Card Play.
if ( ! MB_IsMultiBootChild() )
{
    OS_Panic("not found Multiboot child flag!\n");
}
```

### 4.2.2 Getting Connection Information During Single-Card Play

The connection information used during Single-Card Play can be obtained using the MB_ReadMultiBootParentBssDesc function. If direct connection to the parent is to be made using the WMBssDesc obtained, the key-sharing flag and other settings must be the same as set for the parent at the time the information was obtained.

**Code 4-29 Obtain Connection Information–Parent and Child Must Match**

```
MB_ReadMultiBootParentBssDesc( &gMBParentBssDesc,
                WH_PARENT_MAX_SIZE,  // Parent max transfer size
                WH_CHILD_MAX_SIZE,   // Child max transfer size
                0,                   // Key sharing
                0 );                 // Continuous transfer mode flag
```

### 4.2.3 Starting the Child Application

Data is shared with the parent as a wireless-communication child.

First, perform the initialization processes required for data sharing. These are the same as those carried out for the parent: Use the GInitDataShare function to configure the initial settings of the buffer to be used for data-sharing communications, and use the WH_Initialize function to initialize the WM library and wireless communications.

**Code 4-30 Initialize Data Sharing, the WM Library, and Wireless Communications**

```
GInitDataShare();

//*******************************
// Initialize wireless communications
(void)WH_Initialize();
//*******************************
```

Next, try connecting to the parent with retries in the main loop (**1**). Once wireless communications begin, the state moves to `WH_SYSSTATE_DATASHARING` (**2**) and data sharing begins.

**Code 4-31 Connect Child to Parent, Change State, and Share Data**

```
// Main loop
for ( gFrame = 0 ; TRUE ; gFrame ++ )
{
    // Divide process based on communication state
    switch( WH_GetSystemState() )
    {
    case WH_SYSSTATE_CONNECT_FAIL:
        {
            // If WM_StartConnect() has failed, then the WM internal
            // state is illegal, so you need to reset  WM to the IDLE
            // state using M_Reset.
            WH_Reset();
        }
        break;
    case WH_SYSSTATE_IDLE:
        {
            static retry = 0;
            enum {
                MAX_RETRY = 5
            };

            if ( retry < MAX_RETRY )
            {
                ModeConnect();                                  1
                retry++;
                break;
            }
            // Display ERROR if cannot connect to parent in MAX_RETRY
        }
    case WH_SYSSTATE_ERROR:
        ModeError();
        break;
    case WH_SYSSTATE_BUSY:
    case WH_SYSSTATE_SCANNING:
        ModeWorking();
        break;
    case WH_SYSSTATE_CONNECTED:
    case WH_SYSSTATE_KEYSHARING:
    case WH_SYSSTATE_DATASHARING:                               2
        {
            BgPutString( 8 , 1 , 0x2 , "Child mode" );
            GStepDataShare( gFrame );
            GMain();
        }
        break;
    }
}
```

The connection to the parent is made using the `WH_ChildConnect` function inside the `ModeConnect` function. The arguments to this function include `WH_CONNECTMODE_DS_CHILD` (used to indicate data sharing) and `gMBParentBssDesc` (wireless communication connection information used by the Single-Card Play feature).

When reconnecting after the download, if the application has some special information that needs to be received from the parent, the parent will notify the child of that need via its game information beacon, and the child can rescan to get that information. If this is not necessary, then you can perform the reconnection by simply using `gMBParentBssDesc`. The `ModeConnect` function stores the codes for both parent and child, telling them apart using the `USE_DIRECT_CONNECT` switch, so select whichever one of these methods best suits the application at hand. (The default method is a simple reconnection.)

**Code 4-32 Child Connection Details**

```
static void ModeConnect( void )
{
#define USE_DIRECT_CONNECT

    // If directly connecting to parent without scanning again.
#ifdef USE_DIRECT_CONNECT

    //*******************************
    (void)WH_ChildConnect( WH_CONNECTMODE_DS_CHILD, &gMBParentBssDesc, TRUE);
    // WH_ChildConnect(WH_CONNECTMODE_MP_CHILD, &gMBParentBssDesc, TRUE);
    // WH_ChildConnect(WH_CONNECTMODE_KS_CHILD, &gMBParentBssDesc, TRUE);
    //*******************************
#else
    WH_SetGgid(gMBParentBssDesc.gameInfo.ggid);
    // If executing a rescan for the parent.
    //*******************************
    (void)WH_ChildConnectAuto(WH_CONNECTMODE_DS_CHILD, gMBParentBssDesc.bssid,
                          gMBParentBssDesc.channel);
    // WH_ChildConnect(WH_CONNECTMODE_MP_CHILD, &gMBParentBssDesc, TRUE);
    // WH_ChildConnect(WH_CONNECTMODE_KS_CHILD, &gMBParentBssDesc, TRUE);
    //*******************************
#endif
}
```

# 5 The cloneboot Sample Program

The `cloneboot` sample program uses the features described in 3 The Clone Boot Feature to act as a Single-Card Play parent, delivering copies of its own programs to child devices and data sharing with download children.

This cloneboot sample program shows the procedure for how to the unify the existing programs for both the parent and child from the multiboot-Model sample program to create a program that supports the clone boot feature. For details on the multiboot-Model sample, see 4 The Sample Program (Multiboot-Model).

This chapter describes the following changes to the program structure:

1. Unification of the program source directories
2. Changes to the ROM specification file
3. Changes to makefile
4. Additions to the build procedure for attaching authentication codes

The chapter also describes the following changes made to the program source:

1. Changes to main entry names
2. Addition of new entries
3. Specification of a parent-only region
4. Revision of the binary registration process

## 5.1 Changes to the Program Structure

The following sections describe the changes that must be made to a program in order for it to be able to support the clone boot feature.

### 5.1.1 Unification of the Program Source Directories

In the multiboot-Model sample, the child program is first created and then the parent program is created with that child program included as a separate file, so the overall structure is composed of two separate build projects. Programs that support clone boot can be unified into a single project because the parent is determined at the time of execution.

Here, the `src` and `include` directories, and all contents, included inside the `parent`, `child`, and `common` directories are moved to the project's root directory. At this time, the `main.c` files that exist in both the parent program and the child program get renamed to `parent.c` and `child.c`. (A new `main.c` is created in a later procedure.)

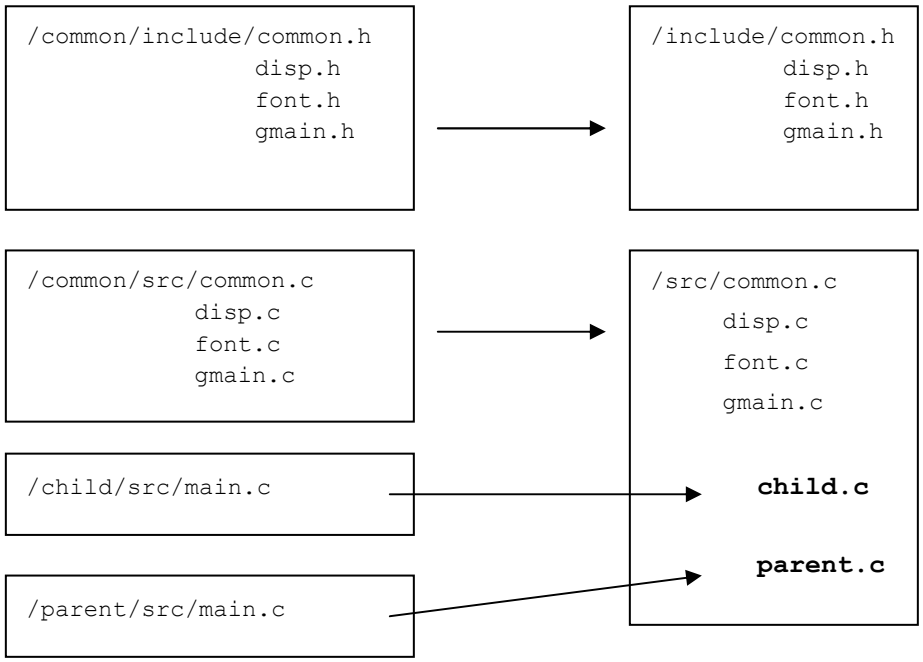



**Fig. 5-1 Unifying the source directories**

### 5.1.2 Changes to the ROM Specification File

The child program included in the file system in the multiboot-Model sample is no longer present in the program that supports clone boot, so delete the following lines from the `main.rsf` file:

```
# Delete this specification
#     HostRoot        $(MAKEROM_ROMROOT)
#     Root            /
#     File            $(MAKEROM_ROMFILES)
```

## 5.1.3 Changes to the Makefile

In order to unify the parent program and the child program and change them into a program that supports the clone boot feature, a number of changes and additions must be made to the parent program's makefile. These are described in the sections below. The makefile used in the build for the child program is no longer necessary.

### 5.1.3.1 Correcting Directory and Source Specifications

Steps are taken so the changes to the directory structure that were made in 5.1.1 Unifying the Program Source Directories are correctly reflected in makefile. Also, the main source for both parent and child with changed filenames get added to the project.

```
# The child program's build is no longer necessary, so delete the sub-build
specifications.
# SUBDIRS             =       child
...

# Specify references to the new, unified directory.
SRCDIR        = ./src
INCDIR        = ./include
...

# Add the two main.c files with changed filenames (parent.c and child.c) to the
build source.
SRCS          =       main.c      \
                      common.c    \
                      disp.c      \
                      font.c      \
                      gmain.c
```

### 5.1.3.2 Specifying an LCF Template File for Clone Boot

To create a program that supports clone boot, you must secure a parent-only region, as described in 3.2.1 Placing Data in ROM. There is an LCF template file that has ROM placement configured for clone boot. You need to explicitly specify this template:

```
$NitroSDK/include/nitro/specfiles/ARM9-TS-cloneboot-C.lcf.template
```

```
# Specify the link configuration template for clone boot.
LCFILE_TEMPLATE   = $(NITRO_SPECDIR)/ARM9-TS-cloneboot-C.lcf.template
```

#### 5.1.3.3 Additions to Build Procedure to Attach Authentication Code

Programs that support the clone boot feature have a procedure for getting the authentication code that differs from the usual procedure for Single-Card Play programs described in 3.2.2 Authentication Code Attachment.

For programs that support the clone boot feature, use the `emuchild` tool to create a binary for getting the signature code. The procedure for doing this is as follows:

```
# For retail-version applications, specify the distributed libsyscal.a and the
corresponding libsyscall_child.bin
LIBSYSCALL           = ./ etc / libsyscall.a
LIBSYSCALL_CHILD     = ./ etc / libsyscall_child.bin

# Since already built, this is the procedure for creating the transfer-use
binary with the emuchild tool.
# The created bin / sign.srl gets sent to the server that creates the
authentication code.
presign:
    $(EMUCHILD)    \
       bin / $(NITRO_BUILDTYPE) / $(TARGET_BIN)    \
       $(LIBSYSCALL_CHILD)     \
       bin / sign.srl

# The procedure for including the obtained authentication code in the binary is
the same as normal for clone boot.
# Here, the binary main_with_sign.srl is created with the authentication code as
bin / sign.sgn.
postsign:
    $(ATTACHSIGN)     \
       bin / $(NITRO_BUILDTYPE) / $(TARGET_BIN)     \
       bin / sign.sgn    \
       main_with_sign.srl
```

This notation is added for the sake of convenience of the task. If you enter the notation directly on the command line, you do not need to add it to the makefile.

## 5.1.4 Changes to the Program Source

A number of corrections must be made to the program source in line with those changes to the overall program structure made in the last section..

### 5.1.4.1 Change Main Entry Names

Since the original pair of `main.c` files (`parent.c` and `child.c`) both include the `NitroMain` function, which is a main entry, their names must be changed appropriately.

```
child.c:

// Change name to be main entry for child.
// void NitroMain( void )
void ChildMain( void )
{
   ...


parent.c:

// Change name to be main entry for parent.
// void NitroMain( void )
void ParentMain( void )
{
   ...
```

Also, be sure to add the function prototype declarations to `common.h` using the changed names.

```
common.h
// Originally the parent's NitroMain function.
void ParentMain( void );
// Originally the child's NitroMain function.
void ChildMain( void );
```

#### 5.1.4.2 Add New Main Entries

Add a new `NitroMain` function for calling the parent main entry and child main entry whose names have been changed. In programs that support the clone boot feature, the `main.c` file is created as outlined below so that processes called for the parent and processes called for the child can be separated based on the value returned by the `MB_IsMultiBootChild` function.

```
main.c

#include <nitro.h>
#include "common.h"

void NitroMain( void )
{
    if( ! MB_IsMultiBootChild() )
    {
        ParentMain();
    }
    else
    {
        ChildMain();
    }
    /* The process does not reach this point */
}
```

In the example used here, the goal is to move from the multiboot-Model as easily as possible. Processes that are the same for the parent and the child can be shared. However, you always need to be careful that a card has not been plugged into the child device.

#### 5.1.4.3 Specify a Parent-Only Region

Part of the clone boot program code must be included in the parent-only ROM region, described in section 3.2.1 Placing Data in ROM.

Since the parent-only part of the card is a secure region, like a ROM header, etc., it cannot be read again from the parent after booting. For this reason, be careful not to reinitialize changeable data that has been placed in this region (such as `.bss` section and `.daya` section data) when performing a software reset using the `OS_ResetSystem` function.

When using the `OS_ResetSystem` function, only the following C language items can be used as data in the parent-only region:

- Constants.
- Functions that do not have any internal static variables.
- Global variables accompanied by an explicit dynamic initialization process. (In C++ , an object accompanied by a constructor.)

In addition, content to be included in the parent-only region should not only be "essential to the parent" but must also "not be used at all by the child."

There is no simple standard that can be applied at this point because the ability to judge these two criteria depends on the overall design of the application regarding how one identifies the main version of the software versus versions distributed for Single-Card Play. However, as a general rule, it is both easy and effective to include state transitions to states where only the main part of the program can be played in this parent-only region.

In the `cloneboot` sample, all functions that are included in `parent.c` are specified for placement in the parent-only region. This region is specified using the NITRO-SDK include files `parent_begin.h` and `parent_end.h`, as described below.

```
parent.c

...

//============================================================
//   Function definitions
//============================================================

// The parent-only region .parent section definitions start from here.
// Only functions that do not include static variables exist below this point.
#include <nitro/parent_begin.h>
void ParentMain( void )
{
...

}
// The parent-only region .parent section definitions end here.
#include <nitro/parent_end.h>
// End of file.
```

Here, `parent.c` includes all Single-Card Play parent processes. The conditions for placement in the parent-only region are satisfied: the content is essential to the parent and never used by the child.

Two representative examples of the many types of content that should not be specified for the parent-only region are given below for your reference.

Since changing code so these functions are not called invalidates them, there is no reason to place functions which do not need to be called in the parent-only region from a security standpoint.

```
/* Function gets placed in parent-only region (for debug output only) */
void  no_use( void )
{
  OS_Printf( "called!\n" );
}
...
void NitroMain( void )
{
  ...
  /* If parent, this gets called.  (No trouble if it is not called) */
  if( !MB_IsMultiBootChild( ) )  no_use( );
}
```

Next, one must absolutely avoid unintentionally placing a function that is used by both the parent and the child in the parent-only region. Here, the distinction between the "main part of the program" and a "delivered program" affects the quality of the game.

```
/* Function gets placed in parent-only region.  (A screen presentation process
that the child is not expected to use) */
void  draw_special_effect_1000( void )
{
   ... /* Screen presentation process */
}

/* Game process shared by parent and child */
void  UpdateGameFrame( void )
{
  /* Unexpectedly, function gets called by both parent and child under certain
conditions */
  if( score >= 1000 ) draw_special_effect_1000( );
}
```

#### 5.1.4.3 Correct the Binary Registration Process

The process that registers binaries for the Multiboot library also needs to be changed in order to allow cloneboot. This procedure is describeed in section 3.2.3 Clone Boot Binary Registration.

```
parent.c
...
const MBGameRegistry mbGameList =
{
    // If the MBP_Start function gives NULL for the path name, the process is
    // treated as a clone boot.
    // To read details about the function's internal processes, see
    // $NitroSDK/build/demos/wireless_shared/mbp/mbp.c.
    NULL,
    (u16*)L"DataShareDemo",     // Game name
    (u16*)L"DataSharing demo(cloneboot)",   // Description of game content
...
 }
```