Description of the Wireless Communications Library

TWL-SDK

Version 1.2.3

The content of this document is highly confidential and should be handled accordingly.

TWL-06-0005-001-D
Released: February 19, 2010

Table of Contents

Code

Tables

Figures

Revision History

Version	Revision Date	Description
1.2.3	2009/09/03	In section 3.2.1 Initialization and Shutdown Function Differences, added explanation of when the DSi wireless communications LED is illuminated.
1.2.2	2009/07/29	Corrected errors in the Receive buffer size in the table in section 3.11.2 Connection Process.
1.2.1	2008/10/01	Added information that had been omitted, stating that the 3.1 PR version of the SDK was changed to return `NULL` responses.
1.2.0	2008/09/16	Changed descriptions from NITRO-SDK to TWL-SDK. Added description related to TWL wireless.
1.1.8	2007/10/16	Added section 3.1.6 MP Communications with Wii. Added a note about channels to use. Changed the description about the V-Count variable range. Mentioned that the `WM_StartScan` function is not recommended. Removed a description of planned specification changes that were cancelled.
1.1.7	2007/02/20	Added a description of the conditions that cause the `WM_ERRCODE_OVER_MAX_ENTRY` error.
1.1.6	2006/02/20	Made additions to section 3.1.3 The Library's Internal States. Added section 3.3.4 GameInfo.
1.1.5	2006/01/13	Deleted old descriptions and cleaned up vague descriptions in anticipation of NITRO-SDK 3.0.
1.1.4	2005/12/20	Clearly stated that the operations relating to when communication are not possible in section 3.3.2 Select a Channel to Use. Added section 3.3.5 Precautions to Note When Ending Communications. Added section 3.4.5 Send and Receive Buffers for MP Communications. Added section 3.10 Taking Greater Control Over Communications and moved some items around. In the FAQ, added examples of how to determine communications parameters.
1.1.3	2005/12/06	Added to text relating to V-Blank synchronization and changed it to a separate section. Change in terminology: "Maximum number of bytes that can be sent" changed to "transmission capacity." Moved the section 3.4.4 Transmission Capacity. Added section 3.4.5 Dynamically Changing the Transmission Capacity. Added section 3.4.9 Timing Control Parameter of MP Communications.
1.1.2	2005/11/04	Updated Table of Contents.
1.1.1	2005/11/01	Indicated the addition of `WM_STATECODE_DISCONNECT_FROM_MYSELF` notification to each of the `WMStartParent`, `WMStartConnect`, and `WMSetPortCallback` functions. Indicated the addition of the `WM_STATECODE_PORT_INIT` notification to the `WMSetPortCallback` function and the addition of the `connectedAidBitmap` field to the `WMPortRecvCallback` structure.

Version	Revision Date	Description
1.1.0	2005/07/29	Indicated the addition of WM_STATECODE_INFORMATION notification to WMIndCallback function callback. Revised descriptions in each section due to change in the condition for a NULL response.
1.0.5	2005/07/12	Revised the return values of the WM_Initialize function in the list “Error Codes Returned from the Wireless Communications Library. In section 3.4.3 Operations When Communications Fail, added description of the MP notification that happens when there is a failure. Added descriptions of changes in section 3.11.12 Changes in MP Frame Sending Conditions.
1.0.4	2005/06/07	Added to each Key Sharing related description that we plan to phase it out. Changed the WM_StartKeySharing and WM_EndKeySharing functions in the list “Error Codes Returned from the Wireless Communications Library.” Added section 3.4.5. Limitations to the number MP Communications related to picture frames.
1.0.3	2005/03/29	Added warning about repeated calling to section 3.1.5 Asynchronous Function Callback and Asynchronous Notifications.
1.0.2	2005/03/22	Added note about CLASS1 state to section 3.1.3 The Library Internal State. Added note about CLASS1 state to section 3.10.2 Connection process.
1.0.1	2005/03/04	Made changes to the “List of Error Codes Returned from the Wireless Communications Library.” Changed the number of levels in the send queue from 64 to 32. Made changes to the items related to the WM_SetMPData function in the “Event Notifications Returned from the Wireless Communications Library”: changed the description related to the restBitmap field in the WMPortSendCallback structure, and added a description related to the sentBitmap field. Added items to “Important Notes for Recent Releases.”
1.0.0	2005/02/18	Initial version.

1 Wireless Communications Library Overview

1.1 Introduction

The TWL-SDK includes a set of functions for wireless communications. This document describes the basic features of the Wireless Communications library.

1.2 Basic Specifications for Wireless Communication Hardware

Table 1-1 shows the basic specifications of wireless hardware in the Nintendo DS system.

Table 1-1 Basic Specifications for Wireless Communication Hardware

Item	Description
Communication band	2.4-GHz band. May receive interference from microwave ovens or other wireless devices that use the 2.4-GHz band.
Communication standards	• IEEE 802.11 equivalent (Infrastructure mode) • Nintendo proprietary protocol (DS Wireless Play mode) • Nintendo proprietary protocol (DS Download Play mode)
Communication speed	1 Mbps or 2 Mbps
Communication range	10-30 m. This is highly variable and depends on the surrounding environment and position of the system.
Remarks	Not compatible with the Game Boy Advance Wireless Adapter.

In addition to the NTR wireless communications unit listed above, TWL has a TWL wireless communications unit. The TWL wireless communications unit is used only for Infrastructure Mode; it cannot be used for DS Wireless Play. This document deals only with the NTR wireless communications unit.

1.3 Configuration of the Wireless Communications Library

The TWL-SDK Wireless Communications library processing is performed by both ARM9 and ARM7. The ARM7 component controls the wireless communications hardware, and the ARM9 library transmits requests from the application to ARM7. When creating an application, ARM7 does not have to be considered. Caution is required for some cache-related processes, so follow instructions in the reference manual when data is exchanged with the Wireless Communications library.

Table 1-2 provides descriptions of the three main communication modes in the Wireless Communications library.

Table 1-2 Communication Modes of the Wireless Communications Library

Mode	Description
DS Wireless Play mode	Allows for wireless communications in which a single DS acts as the parent device, and up to 15 other DS devices act as child devices.
DS Single-Card Play mode	Also known as wireless multiboot. Allows for program and data download from a parent device to child devices that do not have Game Cards. The child device can then start up that program.
Infrastructure mode	Allows connection to the Internet through wireless access points that support the IEEE 802.11b/g standard.

This document focuses on DS Wireless Play. For more information about DS Single-Card Play, see *A Description of DS Single-Card Play* (`AboutMultiBoot.pdf`).

2 Glossary

Table 2-1 defines the terms used in this guide.

Table 2-1 Glossary for the Nintendo DS Wireless Communications Library

Term	Definition
AID	Association ID (the connection identifier). The parent's AID is always 0. Child devices are assigned AIDs from 1 to 15 when they connect. If the maximum number of children permitted to connect is set to *n*, assigned AIDs must be from 1 to *n*.
Beacon	A wireless signal, separate from an MP sequence, sent periodically by the parent. A child can receive the beacon even when it has not established a connection. A child making a new connection selects the parent based on the `GameInfo` in the beacon. Normally sent at intervals of several hundred ms.
Block transmission	Feature designed to transmit a chunk of data from the parent to multiple children at the same time. For more information, see “WBT*” in the *NITRO-SDK Function Reference Manual*.
BSS	Basic Service Set. Specifies a set that performs transactions for a single service. In DS Wireless Play, refers to the group that includes a parent and the children connected to it.
BSSID	Basic Service Set ID. This is an ID used to identify the BSS. For DS Wireless Play, the MAC address of the parent device is used as is for BSSID. This is used to designate a connection destination when a child device connects to the parent device.
Channel	A portion of the communication band. On the DS wireless communications hardware, 1 to 14 channels can be used, but the actual number of available channels is limited by regulations in each country. Also, adjacent channels can interfere with each other, so we recommend setting up channels with roughly five intervals between them.
Child device	The devices (1–15) that connect to a parent device in DS Wireless Play.
Continuous communication mode	The communication mode sustained using continuous MP sequences. (This contrasts with frame-synchronous communication mode.) It is effective for sending large amounts of data, but consumes lots of power and should not be used for long periods. It differs from frame-synchronous communications mode in that it starts MP sequences continuously, but from a control standpoint, the two modes are virtually identical. Thus, a single program can switch between the two modes.
Data sharing	Enables the sharing of data with a user-defined size in addition to key input.
Frame-synchronous communication mode	Communication mode that synchronizes MP communications with the picture frame. (This contrasts with continuous communications mode.) The number of times to communicate during each picture frame is specified. However, if signal conditions are poor and a resend is necessary, the transmission is resent without synchronizing with the frame (if possible).
Game frame	The period defining a unit of game processing. If a game is running at 30 frames per second, the game frame is 1/30 of second.
GameInfo	A data structure that indicates the type of game offered by a parent and contains the data needed to connect. Contains the GGID, TGID, maximum transmission capacity of the parent, and so on. It can also contain user-defined information. For example, with Single-Card play, the `GameInfo` contains the game name, icon data, and so on.

Term	Definition
GGID	Game Group ID. A unique four-byte ID assigned by Nintendo to each game title and series title. Used when connecting.
IEEE 802.11 Specification	A wireless communications standard defined by the IEEE. It defines a wireless communication method that permits speeds of up to 2 Mbps. (802.11b, which is in the same family of standards, allows a maximum speed of 11 Mbps and is a popular standard for wireless PC communications. 802.11b is backward compatible with 802.11.) The Nintendo DS system uses the 1–2 Mbps backward compatibility mode defined in the 802.11b standard.
Indication	A notification sent automatically from the wireless hardware to the application in response to receiving data or another event. Differentiated from a response to a request from the application.
Key response frame	The type of frame a child uses to respond to an MP frame from the parent.
Key sharing	Functionality common to key input data. Enables you to use wireless communications without worrying about the details. **We plan to discontinue key sharing and recommend that you use Data Sharing instead.**
MAC address	An ID number for the wireless communication hardware. Each Nintendo DS system has a different 6-byte MAC address.
MP communications	A generic term for communications that use *multi-poll (MP) sequences*. In some cases, it refers to the communication for a single MP sequence.
MP frame	The frame at the beginning of an MP sequence, in which the parent broadcasts to the child.
MP sequence	Multi-Poll sequence. Nintendo's proprietary extension of 802.11 enables wireless communications with a low latency time. For more information, see the chapter on DS Wireless Play.
MP_ACK frame	The frame broadcast by a parent to its children at the end of an MP sequence.
Null response frame	The frame that a child uses to respond to an MP frame sent by the parent. Sent when the response data cannot be sent within given time constraints.
Packet	A unit of communication that contains a header and footer. It contains a port number, a packet size, and when necessary, destination information, a sequential number, and so on. In actual communications, within a single MP sequence, a payload contains multiple packets.
Parent device	Device that controls all communications in DS Wireless Play.
Payload	Area in the MP and key response frames that carries data.
Picture frame	Time elapsed between one V-Blank interrupt signal and the next (1/60 of a second).
PollBitmap	A 16-bit bitmap in which each bit corresponds to the AID of a child device. In an MP frame, the bits of the children from which a response is desired are enabled. In an `MP_ACK` frame, the bits of the children from which the parent received no response are enabled.
Port	A concept used in the Wireless Communications library to realize multiple communication channels. If a transmission specifies a port that is an integer from 0 through 15, the destination calls a callback that corresponds to the number. Note that this port has a lower level of abstraction than ports used in TCP/IP.

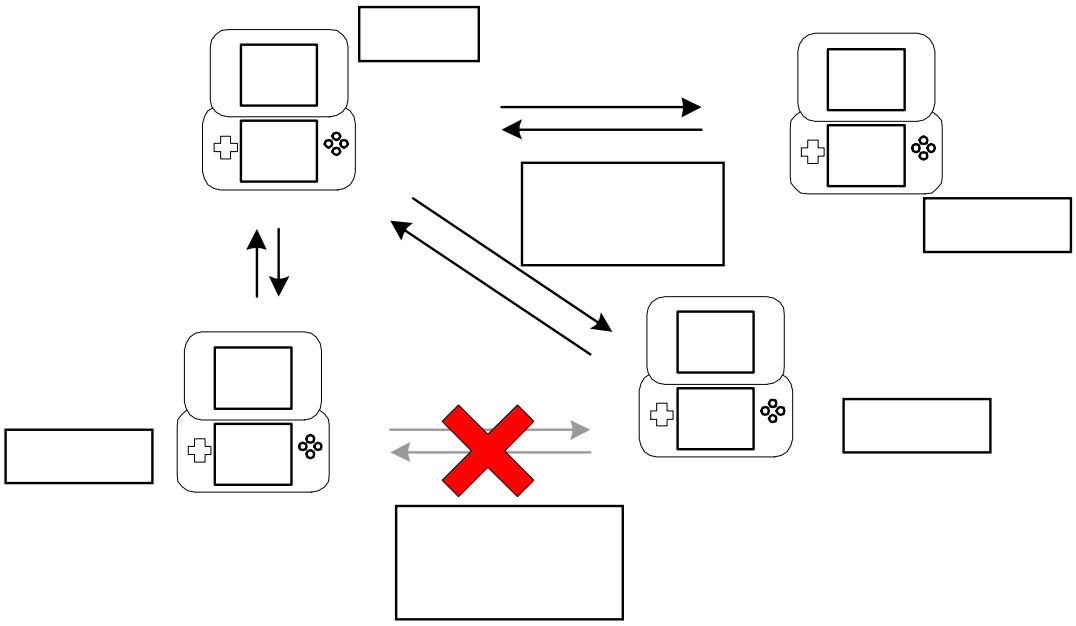
Term	Definition
Raw communication	A communication method that does not perform additional controls like those in sequential communications. Data may not reach its destination and the same data may be transmitted several times. If ports 0 through 7 are selected, raw communication is used.
Sequential communication	The upper layer of MP communications, which guarantees the integrity of communication. The Wireless Communications library uses sequential numbers to eliminate long packets and insure that packets reach their destination. If ports 8 through 15 are selected, sequential communication is used.
Session	The period of time from a single `WM_StartParent` to `WM_EndParent`.
SSID	Info used to screen children connecting to a parent. The child uses the GGID and TGID from `GameInfo` to generate an SSID. The parent connects only to children that have a matching GGID and TGID based on the SSID. The latter half of the user area that is not used for matching can be used for child-to-parent communications.
TGID	Temporary Group ID. A two-byte ID assigned when a new game or session is started. When the same DS continues to be used as the parent device, the TGID splits communications into new and old, because the BSSID and GGID are identical.

3 DS Wireless Play

3.1 Overview

3.1.1 Connection Configuration

Figure 3-1 DS Wireless Play Connection Configuration



In DS Wireless Play, the network is configured in a hub-and-spoke arrangement. Communication is limited to that between parent and children; children cannot communicate with each other. However, a parent can transmit data to multiple children at the same time.

3.1.2 DS Wireless Play Characteristics

- Low latency

When communications are being performed normally, the send data that is set in the beginning of the picture frame will be received by the communications partner application at the end of the picture frame.

- Data is transmitted at a specified time in one picture frame

Rather than visualizing that the data is sent at the timing desirable to the parent and child, consider that if the send data is set in advance with the `WM_SetMPDataToPort` function, the parent and child send data will be exchanged in fixed sizes when communication occurs.

- The more child devices there are, the smaller the data size that can be sent from each child device

  Because the maximum communication time that a single MP sequence can use is defined in the programming guidelines, the maximum size that can be sent decreases as more child devices are added.

  If there is only one child device, the maximum send size in the Wireless Communications library of 512 bytes can be sent on both the parent and child. If 15 child devices are connected, the parent device can send 256 bytes, and the child devices can only send 8 bytes each. For more information, see section 3.4.4 Transmission Capacity.

- The efficiency is better if broadcasting from the parent device

  A particular child device can be selected to send data with the `WM_SetMPDataToPort` function. As a characteristic of MP, however, even if there is a broadcast to multiple child devices, the appropriation time for a wireless channel will not change (except when data is resent or when a special communication mode is used).

- The efficiency is better for communications of a fixed volume

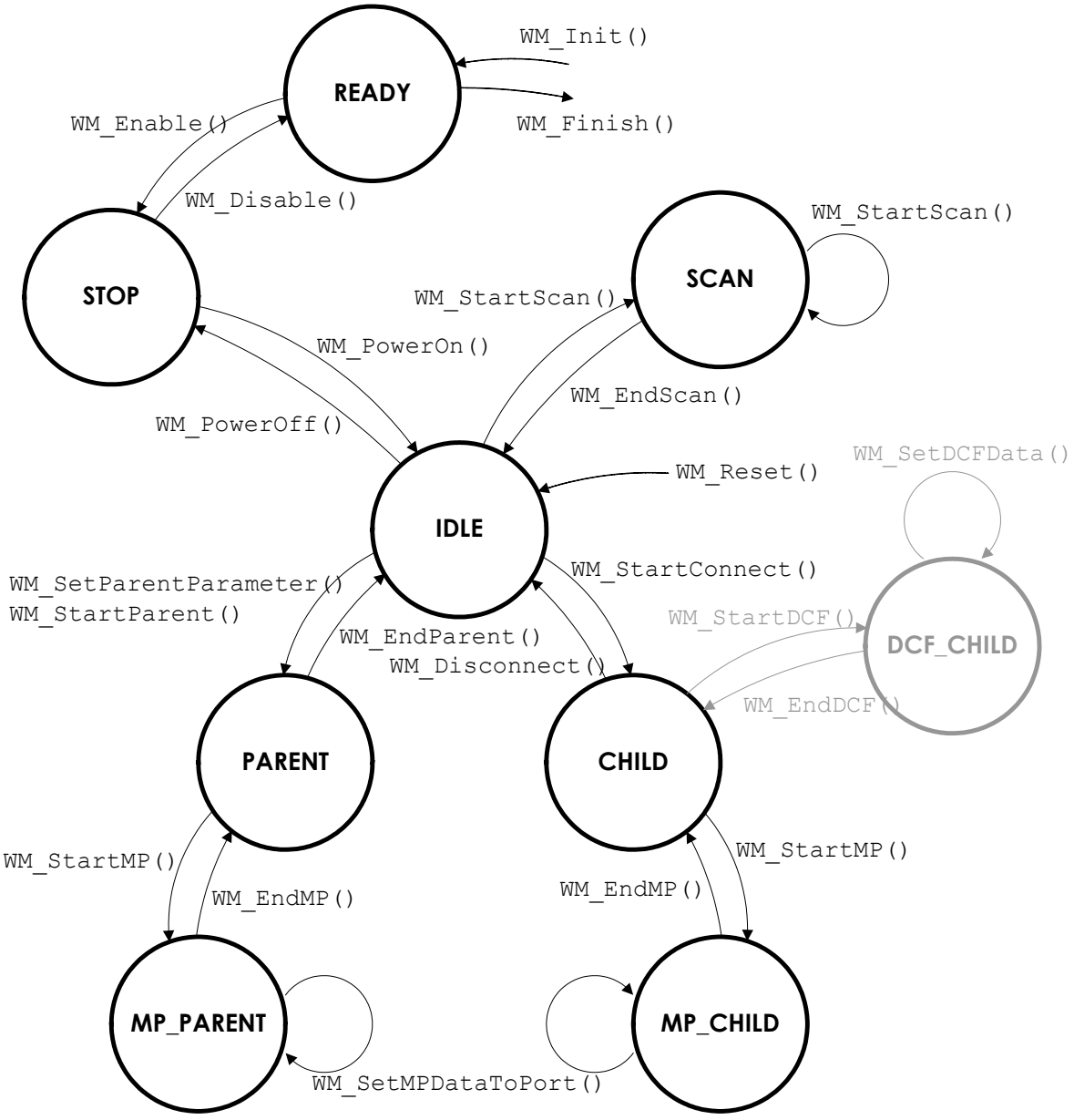
  Communications from a child device during a single MP sequence cycle always occupy a wireless channel for just the amount of time required for the child send size (the maximum size that a child can send on a single MP sequence). Therefore, thinking of this as a fixed-length communication allows for more efficient signal use. The initial value of the send size for a child device is set by the parent when the connection is established.

  Currently, no logic is implemented to increase the number of communications according to how full the send queue is. It is possible for the application to dynamically change the communication frequency, but we recommend that you use communication specifications that do not cause fluctuations in communication volume.

3.1.3 Internal States of the Wireless Communications Library

Figure 3-2 shows the internal states of the Wireless Communications library. The functions it can call depend on its state. Calling the `WM_Init` function after starting the Nintendo DS system makes it transition to a READY state. To determine the state in which a function can be called, see its description in the *Function Reference Manual*.

Figure 3-2 Internal States of the Wireless Communications Library



In Figure 3-2, the DCF_CHILD state cannot be used in DS Wireless Play.

Strictly speaking, during transitions between IDLE and CHILD and between IDLE and SCAN, the library temporarily exists in the CLASS1 state. However, WM asynchronous functions that have been state-transitioned are never in the CLASS1 state when they quit normally or when a callback is reported, so there is usually no need for concern about this state. However, the library will transition to the CLASS1 state when the `WM_StartConnect` function has failed during a specific stage of the connection process or when the child has disconnected from the parent during the connection.

The only function that can be executed from the CLASS1 state is `WM_Reset`. Therefore, when `WM_StartConnect` fails or when a child receives a disconnect notification, use `WM_Reset` to transition to the IDLE state before moving to the next operation.

3.1.4 Error Codes

With a few exceptions, the Wireless Communications library functions return the `WMErrCode` enumerated structure as its error code. Basically, when operations are normal, synchronous functions return `WM_ERRCODE_SUCCESS`, while asynchronous functions return `WM_ERRCODE_OPERATING`.

For more information, see `WMErrCode` in the *Function Reference Manual* or section 3.8 Error Codes Returned from the Wireless Communications Library.

3.1.5 Asynchronous Function Callback and Asynchronous Notifications

Because the Wireless Communications library sends instructions to the ARM7 driver, many of those instructions are asynchronous functions. These asynchronous functions take callback, a `WMCallbackFunc` type argument, and once the asynchronous process has ended, they call callback.

When an asynchronous function is called and its return value is `WM_ERRCODE_OPERATING`, the completion callback is always called.

Due to the nature of the communications, there are many asynchronously generated notifications. These notifications are sent as callback function calls. Table 3-1 shows the correspondence between the main notification type and its callback configuration function.

Table 3-1 Main Notification Types and Callback Configurations

Communications Type	Function that Configures Communications Callback Destination
Notification of a connection or a disconnection	`WM_StartParent, WM_StartConnect*`
MP sequence-related notification	`WM_StartMP*`
Reception to a port	`WM_SetPortCallback`
All other notifications	`WM_SetIndCallback`

The callback function types in the Wireless Communications library are defined as `WMCallbackFunc` types. The `WMCallbackFunc` type function takes the sole argument `WMCallback* arg`, but because some functions pass a structure unique to that function (for example, `WMPortRecvCallback`), you should use them as needed after casting them to that type. The type of callback argument returned by each function is described in `$TWLSDK_ROOT/man/en_US/wm/wm/WMCallbackFunc.html`.

There may be instances where a field-called state has been defined in the structures of the callback arguments for some functions. This state field is used to express notification types that cannot be expressed with the `errcode` field alone. For more information, see section 3.7 Event Notifications Returned from the Wireless Communications Library.

With a few exceptions, each asynchronous function in the Wireless Communications library registers its callback individually. Use caution, because if different callbacks are assigned to the same function at the same time, only the later assigned callback will be valid. This is not true for the `WM_SetMPData*` functions, which store callbacks separately each time they are called. They can be called repeatedly, setting a different callback each time without a problem.

Avoid making multiple calls to asynchronous functions that change the internal state of the Wireless Communications library. This can cause bugs that are difficult to reproduce.

3.1.6 MP Communications with Wii

Fundamentally, to communicate with Wii, a DS program can use the same implementation that it uses to communicate with another Nintendo DS system. However, there are specific limitations; for example, the Wii console can only be a parent device. For more information on communication limitations with Wii, see the MP Library Reference included in Revolution SDK Extensions (RevoEX).

3.2 Initializing the Wireless Communications Library

3.2.1 Initialization and Shutdown Function Differences

There are two procedures for initializing the Wireless Communications library: calling the three functions (`WM_Init`, `WM_Enable`, and `WM_PowerOn`) in order or calling the `WM_Initialize` function. Similarly, there are two procedures for the shutdown process: calling the three functions (`WM_PowerOff`, `WM_Disable`, and `WM_Finish`) in order or calling the `WM_End` function.

The `WM_Initialize` function performs the same process as calling the three functions (`WM_Init`, `WM_Enable`, and `WM_PowerOn)`, and the `WM_End` function performs the same process as calling the three functions (`WM_PowerOff`, `WM_Disable`, and `WM_Finish`).

Table 3-2 Functions Called by the Initialization and Shutdown Processes

Initialization Process	`WM_Initialize`	`WM_Init`	Allocates the buffer used by the Wireless Communications library.
		`WM_Enable`	Transitions the wireless communication hardware to a usable state. (The POWER LED will begin to blink irregularly [**Note:** Nintendo DS or Nintendo DS Lite].)
		`WM_PowerOn`	Starts providing power to the wireless communication hardware. (Power consumption will go up.)
Shutdown Process	`WM_End`	`WM_PowerOff`	Stops providing power to the wireless communication hardware. (Power consumption will go down.)
		`WM_Disable`	Transitions the wireless communication hardware to an unusable state. (Stops the irregular blinking of the POWER LED [**Note:** Nintendo DS or Nintendo DS Lite].)
		`WM_Finish`	Frees the buffer used by the Wireless Communications library.

Note: On the Nintendo DSi, the POWER LED does not blink irregularly; instead, the newly added wireless communications LED blinks for 2 seconds when wireless communications are initiated.

3.2.2 DS Wireless Communications ON State

The DS Wireless Communications ON state is defined as the period that begins when the `WM_Enable` function is called and ends when the `WM_Disable` function is called. Limitations, such as the need for user confirmation, apply when transitioning to the DS Wireless Communications ON state. For more information, see the *DS Programming Guidelines*.

3.2.3 Buffer for the Wireless Communications Library

During the period that begins when the `WM_Init` function is called and ends when the `WM_Finish` function is called, the Wireless Communications library holds the buffer that will be used inside the library. From the main memory, pass the `WM_SYSTEM_BUF_SIZE`-byte region aligned with the 32-byte boundary to the argument of the `WM_Init` function.

3.3 Connecting Parent and Child

3.3.1 Connection Process

Table 3-3 shows the process leading up to a connection.

Table 3-3 Connection Process

	Parent		Child
1.	Initialize the wireless communication hardware.	1.	Initialize the wireless communication hardware.
2.	Set the parent's GGID, TGID, and other data for communications.		
3.	Measure the degree of congestion on each of the wireless channels and select a channel to use.		
4.	Send the beacon on the specified channel. The beacon's `GameInfo` contains the GGID, TGID, and a flag indicating that it is available for connection.	2.	Scan the beacons on all channels that the application can possibly use and obtain the parent device `GameInfo`.
		3.	Based on the data in `GameInfo`, list the parent devices for the user and prompt for a selection.
		4.	Generate an SSID using the GGID and TGID contained in `GameInfo` and connect to the parent based on the BSSID (the parent's MAC address) and SSID.
5.	Compare the SSID of the incoming child with its own GGID and TGID, and OK the connection if there is a match.		
6.	Assign an AID to the child and complete the connection.	5.	Receive the assigned AID from the parent and complete the connection.

3.3.2 Select a Channel to Use

The 802.11 specifications define 14 channels, but the usable channels may be limited depending on regulations of the country or region in which they are used. Also, even if a channel can be used, neighboring channels may cause mutual interference, so try to use channels that are as far apart as possible.

The Nintendo DS system keeps its usable channels internally, and the `WM_GetAllowedChannel` function was prepared to present channels that are sufficiently spaced apart within those channels. In the application, a channel must be selected from the channels obtained with this function. The application has the responsibility to select from the presented channels the channel that has the lowest possible signal usage rate. The signal usage rate of a specific channel can be obtained with the `WM_MeasureChannel` function.

As of October 2007, the international unified specification states that the Nintendo DS system can use channels 1-13. However, this specification is subject to change, and in some situations, WM is forbidden from using any special channels internally. Avoid any programming that makes assumptions based on the currently permissible channels and rely on the results of the `WM_GetAllowedChannel` function instead.

Note: If the `WM_GetAllowedChannel` function returns 0, wireless communications are unavailable. Do not commence communications if this value is returned.

In addition, for some countries and regions, the usable channels are more restricted for a Wii console than for a Nintendo DS system. In this case, the parent device may be able to use only some of the channels obtained by the `WM_GetAllowedChannel` function on a Nintendo DS child device. Because the Nintendo DS system cannot tell the difference, implement scanning of the parent device in the same manner as used for DS-to-DS connections.

3.3.3 Beacon Information

While the parent device is in the `PARENT` or `MP_PARENT` state, it will transmit a signal known as a beacon at regular intervals (the `WMParentParam.beaconPeriod [ms]` interval). The child device that is trying to connect to a parent device will get this beacon with the `WM_StartScanEx` function. The child device will connect to that parent device by passing the included `WMBssDesc` structure as is to the argument of the `WM_StartConnect*` function.

`WMBssDesc` contains a variety of fields, but the three most important bits of information are the `WMBssDesc.bssid`, `WMBssDesc.gameInfo.ggid`, and TGID.

BSSID is an identifier for BSS. During DS Wireless Play, the parent device's MAC address is used as the BSSID. GGID and TGID are used to identify details of the services provided by the parent device. GGID is an ID assigned by Nintendo to each game or series (if communications are possible among the same series). The child device looks for the `WMBssDesc.gameInfo.ggid` of the scan results to confirm that it can connect to the parent device (a code for authorization must be described in the application). TGID, on the other hand, is assigned by the parent device in each new session so that connections for old sessions are not mistakenly made to new sessions.

3.3.4 GameInfo

Included in the beacon is the WMGameInfo structure, which stores the GameInfo information. GameInfo includes descriptions for GGID and TGID, as mentioned previously, as well as parent attributes such as the entry reception state and information about the transmission capacity of the parent and child. There is also a region called userGameInfo where the application is free to set data.

The WMGameInfo structure has a field called magicNumber and a field called ver. The magicNumber field is fixed to the WM_GAMEINFO_MAGIC_NUMBER (=0x0001) value. Check to make sure this field is correct before accessing any other GameInfo field. This verification process is done internally by the WM_IsValidGameInfo and WM_IsValidGameBeacon functions. The ver field indicates the GameInfo structure's version. As long as the magicNumber checks out OK, consider any GameInfo structure that has a meaningless value for the ver field to have the same functionality as the current version, because the structure is expanded to maintain backward compatibility.

The upper-limit size of the userGameInfo region is WM_SIZE_USER_GAMEINFO (=112) bytes. As for the possible applications of userGameInfo, one idea would be for the parent to use it to report information to a selected child about the current stage and participants. Before referencing userGameInfo, check the ggid field to determine if the information is on a known parent.

The initial value for the GameInfo sent by the parent on the beacon is specified using the WM_SetParentParam function. To change the GameInfo content on the beacon after the state becomes PARENT, use the WM_SetGameInfo function.

For more information about the fields, see the WMGameInfo structure reference.

3.3.5 Connection Operations

Inside the library, the BSSID and SSID are used when a child device connects to a parent device. As described in the previous section, BSSID is the parent device's MAC address. SSID is a total of 32 bytes; however, in DS Wireless Play the first 4 of these bytes are used to store GGID, and the next 2 bytes store TGID. A 2-byte reserved region is added to these and used as an 8-byte service identifier. To determine if a child device is an appropriate connection partner, the parent device compares its own GGID and TGID with the first 8 bytes of the SSID declared by the child device. If the child device is not an appropriate partner, it is automatically rejected during the initial steps of the connection process. Because the connection between the parent and child is maintained throughout this confirmation process, the number of connected clients may temporarily reach the parent device's limit. When this happens, a WM_ERRCODE_OVER_MAX_ENTRY error is returned to any child device that attempts to make new connection to that parent device.

Because the first 8 bytes of the SSID are automatically set in the WM_StartConnect function by the library based on the WMBssDesc.gameInfo.ggid and TGID, there is no need for the application to be aware of them. However, the latter 24 bytes in the SSIDs that are not used in the service identifier are released to the application and can set user data as arguments of the WM_StartConnect* function that is to be sent to the parent device. When the WM_StartParent function's callback function receives the WM_STATECODE_CONNECTED notification, the parent device receives the data that the child device sets as WMStartParentCallback.ssid in the latter 24 bytes of the SSID.

3.3.6 Precautions for Ending Communications

Either parent or child can perform disconnection. Try to avoid both of them disconnecting at the same time because, if this occurs, one of the two processes will fail and in some cases it may take a while for this result to be returned.

Note: There is a possibility that once the process for ending communications has been entered, some of the WM functions, depending on their states, may return errors. The end process may go into an infinite loop if it starts on the error "`abnormal end process due to communication error`". If the error occurs, try calling the `WM_Reset` function once and make sure that the error process does not cascade indefinitely.

3.4 MP Protocol Specifications

3.4.1 Communications Overview

Data can be sent once a connection has been established between a parent and child. Communications are performed in each picture frame in the order shown in Table 3-4.

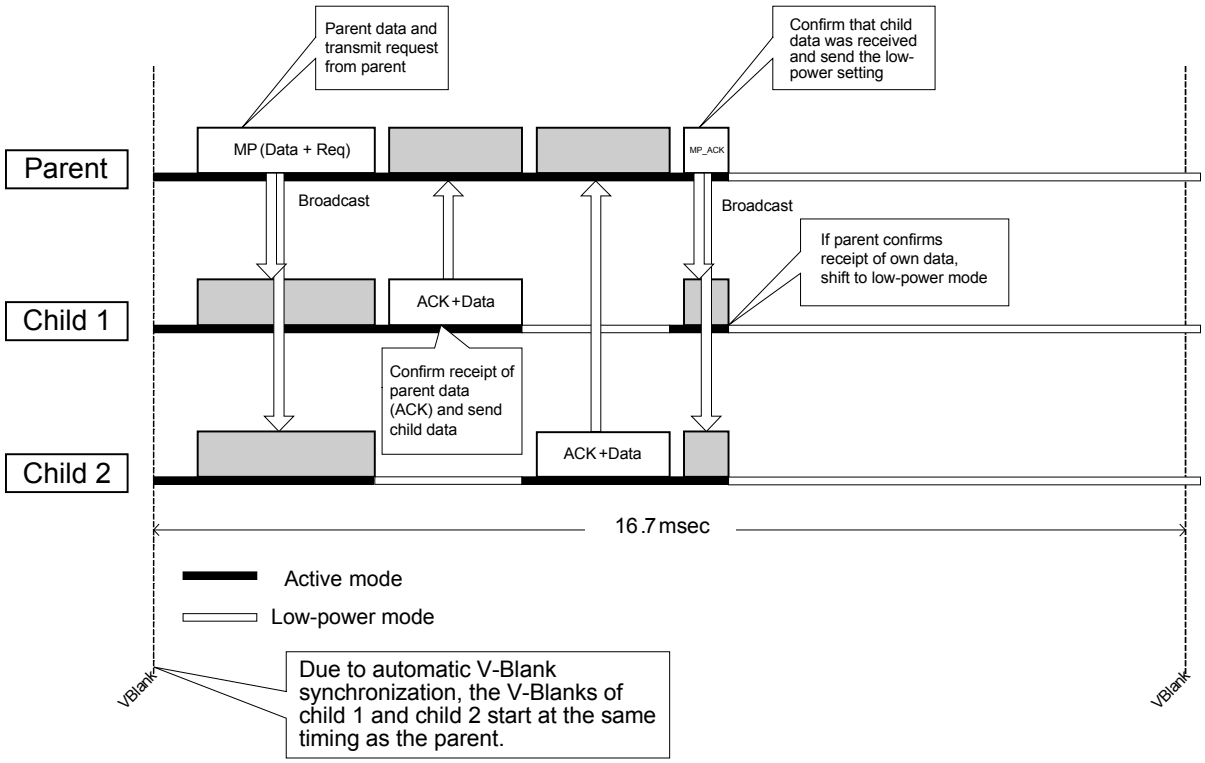
Table 3-4 Parent and Child Communication

Parent Device	Child Device
1. Set the send data with the `WM_SetMPDataToPort` function.	1. Set the send data with the `WM_SetMPDataToPort` function.
2. MP communications are automatically performed on the designated timing in each picture frame.	
3. The data reception notification from the child device arrives at the callback function designated to the `WM_SetPortCallback` function. 4. A notification that the send was a success is sent to the callback function designated with `WM_SetMPDataToPort` in step 1, above.	3. The data reception notification from the parent device arrives at the callback function designated to the `WM_SetPortCallback` function. 4. A notification that the send was a success is sent to the callback function designated with `WM_SetMPDataToPort` in step 1, above.

3.4.2 MP Communications Operations

Read this section as needed. You can use the Wireless Communications library even if you do not understand all of the MP protocol's details. Figure 3-3 shows the MP communications operations, which allow for DS Wireless Play mode.

Figure 3-3 MP Communications Operations



MP communications is a protocol where the parent completely controls the transmission timing.

1. The parent broadcasts an MP frame to all child devices.

   The MP frame includes not only the data to send from parent to child, but also the control data, such as `PollBitmap`, which indicates which children should respond, and `TXOP`, which determines how many bytes of transmission time are assigned to the children.

   Data in the MP frame determines the overall time distribution for that MP sequence.

2. Each child receives the MP frame, looks at `PollBitmap` and `TXOP`, and then sends the Key response frame to the parent after waiting for its turn to respond.

   The child's Key response contains confirmation that it has received data from the parent in addition to data it is sending to the parent. Because the Key response frame is sent by hardware automatically as the response to MP frame, the child needs to set the data in the Key response frame ahead of time. It is not possible to look at the data inside a received MP frame and then alter contents of the data that will be sent in the response during the same sequence.

   If the `TXOP` (the allowable transmission time) given by the parent is shorter than the set length of data, the child cannot send the data. If so, the child transmits a `NULL` response frame instead of the Key response frame. This occurs when the child's transmission capacity as specified by the parent is smaller than the transmission capacity as recognized by the child.

   The child will return a `NULL` response frame even if the send data is not set in the child when the MP frame is received from the parent.

3. The parent broadcasts an MP_ACK frame to all children to acknowledge receipt.

   This MP_ACK frame also has a field called PollBitmap that, in contrast to MP frames, indicates the children from which the parent failed to get either a Key response or a NULL response. To see whether the bit representing its own AID is enabled, each child checks PollBitmap in the received MP_ACK frame. If its AID bit is not enabled, it is guaranteed to send data to the parent successfully. If, however, its bit is enabled or if the child does not receive the MP_ACK frame within a set period of time, the transmission has failed.

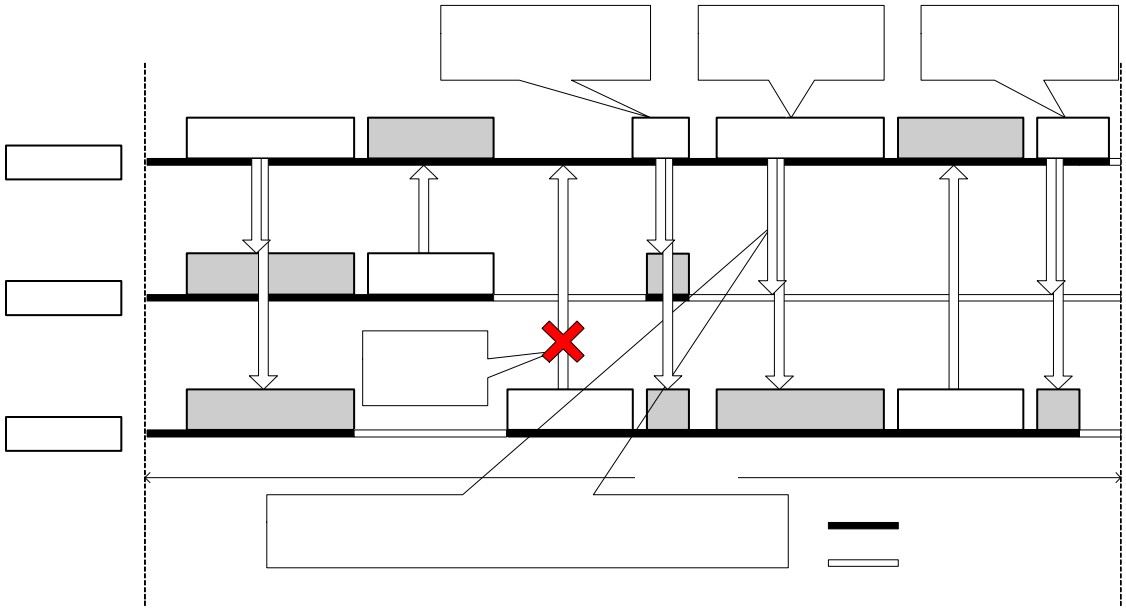
Because the wireless communication hardware consumes a lot of power when active, it enters low-power mode frequently during MP communications. This occurs automatically and normally, so the application can ignore the power mode.

Also, be aware of the child device designated with PollBitmap not being given the chance to respond. To secure a child-to-parent communication band, designate all connected child devices on WM as PollBitmap and perform an MP sequence. However, exceptions exist with resending (described in the next section) and with designating special communications.

3.4.3 Operations When Communications Fail

Figure 3-4 shows the flow of operations when the parent receives no acknowledgment from a child.

Figure 3-4 Operations Flow When Communications Fail



If reception fails, the MP_ACK frame will indicate the children from which no response has been received and then an MP sequence for a resend will be started. The resend MP sequence targets only the children from which a response was not received. The resend MP sequence sends packets that need

to be resent from the packets that just failed. Depending on the communication mode and the type of packet that was not received, the resend process may not be performed, and the application is notified that a transmission failure occurred. The resend will continue while communications have failed and there are still packets that need to be resent.

If only the `MP_ACK` frame communications fail, the parent device will be unable to learn about the failure, and the MP sequence for resending will not be performed. However, because the child device is unable to determine whether the send was a success, the failed packets will be resent in the next MP sequence.

The resend process differs depending on the type of communications packet. For more information, see section 3.5.3 Raw and Sequential Communications.

In addition, with the MP sequence for resending, the send destination is limited to the resend partner due to the `PollBitmap` setting. However, the rest of the process is the same as for a normal MP sequence, so receipt notifications are generated each time an MP frame is received.

The Port Receive callback (described later) only gets called when new data is received.

3.4.4 Transmission Capacity

In MP communications, the parent determines the transmission capacities for itself and for the child; these capacities are specified as the default values at the start of communications. To be specific, the default values for transmission capacity are set by the parent's parameter setting `WM_SetParentParameter` function. In the `WMParentParam` structure passed to the `WM_SetParentParameter` function, the `parentMaxSize` field indicates the default send capacity value for sending data from parent to child, and the `childMaxSize` field indicates the default send capacity value for sending data from child to parent. However, at the time of connection, the child initializes its own transmission capacity setting to the `childMaxSize` value found in the parent's beacon.

The transmission capacity value must meet these three requirements.

1. It must be a multiple of 2.
2. The maximum capacity cannot exceed 512 bytes for parent or child.
3. The time duration of one MP communication, as calculated from the parent and child transmission capacities, must not exceed 5600 microseconds. In other words, the following expression must be satisfied.

   *96 + (24 + 4 + [parent's transmission capacity] + 6 + 4) * 4 + (10 + 96 + (24 + [child's transmission capacity] + 4 + 4) * 4 + 6) * [number of children] + 10 + 96 + (24 + 4 + 4) * 4 <= 5600*

   ⇔

   *[parent's transmission capacity] + ([child's transmission capacity] + 60) * [number of children] < 1280*

(For more information, see section 6.3.3 Data Size of One MP Communication [Recommended] in the *Nintendo DS Programming Guidelines*.)

If WMParentParam.KS_Flag is set to TRUE, the number of transmission bytes for Key Sharing will be added internally, so the actual value that can be set for the transmission capacity is smaller than what is calculated by the above expression. In your calculation, include the bytes that will get secured internally for Key Sharing, this is [36 bytes + 6 bytes] (header and footer) for the parent and [6 bytes + 4 bytes] (header and footer) for the child.

A script for calculating restrictions on communications time can be accessed at $TWLSDK_ROOT/man/ja_JP/wm/wm/wm_time_calc.html.

The above expression is used to determine the maximum time required by communications. The amount of time taken by each MP sequence will actually be shorter, depending on the parent send data size. However, on the child side, the amount of time needed is based on the child send volume size. This happens because when an MP sequence starts, the parent sets timing on the information it has. A summary is shown below.

Amount of time needed to send parent data	Related to the time for the data size that the parent sent in the sequence. Not influenced by the parent send volume.
Amount of time needed to send child data	Related to the size that the parent configures as the child send volume. Not influenced by the size sent by the child.

3.4.5 Send and Receive Buffers for MP Communications

The Send and Receive buffers for MP communications are passed to the WM_StartMP function when MP communications begin. The sizes of these two buffers depend on the parent and child transmission capacities and the maximum number of connected children.

There are two ways to calculate sizes of the two buffers. One way is to call the WM_GetMPSendBufferSize or WM_GetMPReceiveBufferSize function in the PARENT or CHILD state. The other way is to make static calculations by passing the values for transmission capacity and the maximum number of connected children to the function macros shown in Table 3-5.

Based on the parent information being used in the current connection, the WM_GetMPSendBufferSize and WM_GetMPReceiveBufferSize functions dynamically calculate the sizes required of the Send and Receive buffers for MP communications. For the parent information, the value that was set by the WM_SetParentParameter function prior to the start of communications is referenced. For the child, the information in the beacon obtained from the parent during connection is used. Note that the value obtained by the child with these functions is a value obtained from an external source. If memory is to be secured based on this value, you must verify that the memory size you plan to secure is in the proper range or that the memory has been secured successfully. For more information on these functions and function macros, see the *NITRO-SDK Function Reference Manual*.

The Send and Receive buffers for MP communications that get passed to the WM_StartMP function must have a 32-byte alignment.

Table 3-5 Function Macros and Their Related Capacities

		Function Macros for Static Calculations	Related Communications Parameters		
			Parent's Transmission Capacity	Child's Transmission Capacity	Max. Number of Connected Children
Parent	Send Buffer size	WM_SIZE_MP_PARENT_SEND_BUFFER	O		
	Receive Buffer size	WM_SIZE_MP_PARENT_RECEIVE_BUFFER		O	O
Child	Send Buffer size	WM_SIZE_MP_CHILD_SEND_BUFFER		O	
	Receive Buffer size	WM_SIZE_MP_CHILD_RECEIVE_BUFFER	O		

3.4.6 V-Blank Synchronization

When MP communications start, the Wireless Communications library automatically synchronizes V-Blanks between the parent and child. While the timing of V-Blanks is being adjusted, the period between V-Blanks is longer than 16.7 ms. Each frame is prolonged by as much as 0.5 ms. At this time the V-Alarm count value varies between 202 and 210 counts, so do not use a V-Alarm with a count value in this range during communications.

The timing adjustment of V-Blank synchronization is mainly performed right after the connection is established, but it can occur at any time during communications.

3.4.7 Frame Synchronous Communications Mode and Continuous Communications Mode

Depending on the timing that starts the MP sequence, the parent device may be operating either in frame-synchronous communications mode or in continuous communications mode.

Frame-synchronous communications mode starts the MP sequence on a specific V-Count for each picture frame. After the MP sequence starts, the power-saving mode wait state is entered after the set number of MP sequences has continuously started.

Here, the number of MP sequences in the frame-synchronous mode is counted as the number of times an ACK was received from child devices. This count is done to secure a communications band for communications from the child devices. Even when there is no send data from the parent device, the MP sequence continues to be started so that the child devices can perform a prescribed number of sends. Also, if there is a failure in receiving the response frame from a child device, communications will be performed with the number of communications for resends tacked on to that prescribed number. If at this time there are too many communications and it goes into the next picture frame, the counter value for the remaining number of communications will increase cumulatively. However, if the counter exceeds a fixed value, it will not advance any further.

Continuous communications mode is a communications mode in which the next MP sequence starts immediately after the last MP sequence ends. This mode blocks transmissions and other instances

where large volumes of data are sent at once. This mode consumes relatively large amounts of power because there is little opportunity to enter into power-saving mode.

The reason there is a time limit of 5600 microseconds on individual MP sequences is to keep operations stable even when multiple parents and children reside on the same channel. When numerous MP sequences run during the same picture frame, the exclusive time on the wireless channel lengthens, destabilizing operations with multiple parents and children on the channel. We recommend keeping the frequency of MP sequences to one per picture frame and working to minimize the frequency to the bare minimum for communications in your applications.

3.4.8 Restrictions on the Number of MP Communications per Picture Frame

Whether frame-synchronous communication or continuous communication is set, the number of MP communications that may occur in one picture frame is limited. Use the `WM_SetMPParameter` function to set the upper limit. The default value is six.

The MP frequency in frame-synchronous communications mode is set to the number of communications to succeed in each picture frame. However, a limit on the number of communications is a limit for the total number of communications, including those that failed. This limit is set because if few children are connected, and the size of the parent send data temporarily becomes small, one MP communication would become as short as a few hundred microseconds, and the frequency of MP communications could increase more than expected by the application.

This restriction feature is how the `fixFreqMode` argument of the `WM_StartMPEx` function gets realized. The `fixFreqMode` argument can be set to place an upper limit on the number of communications. When the argument is set to `TRUE`, the upper limit is set to the same value as the MP frequency value.

3.4.9 Lifetime

If a communications partner suddenly disappears and communications do not take place for a fixed amount of time, the wireless communications driver will automatically disconnect from that partner according to the current lifetime setting. The two lifetimes for DS Wireless Play, `CAM` and MP communications, can both be configured with the `WM_SetLifeTime` function.

The `CAM` lifetime, which is normally set to 4 seconds, is a value that determines the length of wait before disconnecting if there is no wireless parent-to-child or child-to-parent device communications frame.

MP communications lifetime, which is normally set to 4 seconds, is a value that determines the length of wait before disconnecting if there is no `Key` response frame sent from the child device to parent device or no MP frame sent from the parent device to the child device.

For the `CAM` lifetime only, the child ARM7 will not disconnect properly if it freezes. Because this depends on the freeze timing, the wireless communication hardware will automatically return a `NULL` response frame in response to an MP frame. The MP communications lifetime was created to avoid this problem.

Even if a connection is established using WM_StartParent or WM_StartConnect, communications will not begin between the parent and child until the parent device calls the WM_StartMP function; thus, the lifetime may run out. However, because the child device will not respond until the WM_StartMP function is called, the MP communications lifetime may run out. Make sure to call WM_StartMP immediately after communications begin.

It is possible to disable automatic disconnection according to the lifetime, but it should always be used with the standard values because it is required in certain situations (such as when the power of a communications partner is suddenly turned off during communications).

3.5 Port Communications

3.5.1 About Port Communications

Due to the multiplexing of communication pathways in MP communications, the concept of the port has been introduced in the Wireless Communications library. Both parents and children have 16 virtual ports. By designating a port number and sending data, you can sort the processes on the receiving end.

3.5.2 Port Receive Callback

After wireless communications have been initialized, the receiving end configures the reception callback function to the port number being used with the WM_SetPortCallback function. After that, once the send data set by the send side with the WM_SetMPDataToPort* function arrives via MP communication, the receive callbacks that correspond to the port number are called on the receive side.

If there is a new connection or if a communications partner is disconnected, this is communicated to the receive callbacks of all ports.

For more information about notifications, see the entry for the WM_SetPortCallback function in section 3.7 Event Notifications Returned from the Wireless Communications Library.

3.5.3 Raw and Sequential Communications

There are two types of port communications, sorted by the port to use. Ports 0-7 perform raw communications, and ports 8-15 perform sequential communications.

There are practically no communication controls applied to raw communications. Sometimes data will not arrive at the communications partner, or the same data will arrive several times. On the other hand, sequential communication performs guaranteed and non-repetitive communication; it checks for duplication at the Wireless Communication library level by attaching a sequence number to each packet and by using a low-level resend process.

If communication by raw communications fails, resends will be attempted up to the number specified in the defaultRetryCount argument of the WM_StartMPEx function. If WM_StartMP is used to start MP communications, no resends will be attempted. Sequential communications will resend until successful.

When communications are successful or when communications fail even after the specified number of resends, the send-complete callback, which is specified when calling the WM_SetMPDataToPort*

function, is called. Do not overwrite the memory region where the send data specified with the WM_SetMPDataToPort* function is located until the send-complete callback arrives.

Unlike the relationship between TCP and UDP, latency and throughput for raw and sequential communications are similar. Select between them according to whether resends are required. With sequential communications, resends to a child device may become a bottleneck if that child device has a poor signal.

3.5.4 Priority and Send Queue

Port communications include the concept of four levels of priority: levels 0 to 3. The send data set by the WM_SetMPDataToPort* function is processed with a FIFO (First In, First Out) send queue, but there are four send queues with differing priorities. As long as a queue with a high priority is not empty, data will never be sent from a lower priority queue. Communications that are more likely to be performed in real time, such as Data Sharing, are set to priority 1; communications that are less likely to be performed in real time, such as block transfer, are set to priority 3.

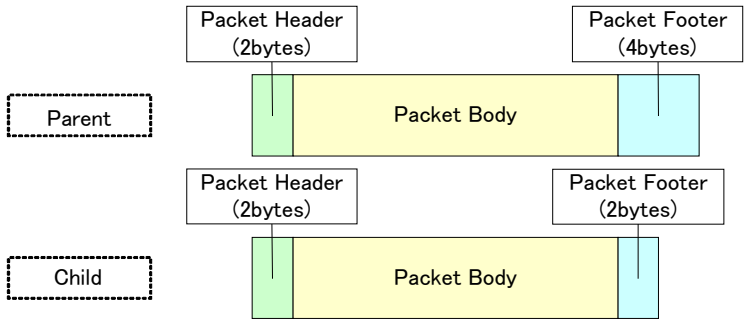
With raw communications, the priority can be changed and the data set while the same port number is specified. With sequential communications, however, inconsistencies in the order control can be caused by the sequence number if the data is set while the priority is changed. Specifically, if higher priority data is set later, the send will still be performed in the order of priority, but lower priority items that were skipped may not be properly sent (in the current implementation, skipped data is sometimes discarded).

If the WM_SetMPDataToPort* function is called when the send queue is full, WM_ERRCODE_SEND_QUEUE_FULL will be returned to the callback, and the function will fail. Up to 32 send packets of differing priorities can be placed in the queue. However, when performing controls that wait for the send-complete callback for the WM_SetMPDataToPort* function before setting the next data, only one level of the send queue is used, so it is unlikely that the queue will overflow with a normal use of this method. Data Sharing also uses a maximum of two levels.

3.5.5 Packet Headers and Footers

To make port communications work, a data structure for communications known as a packet is used in the Wireless Communications library layer.

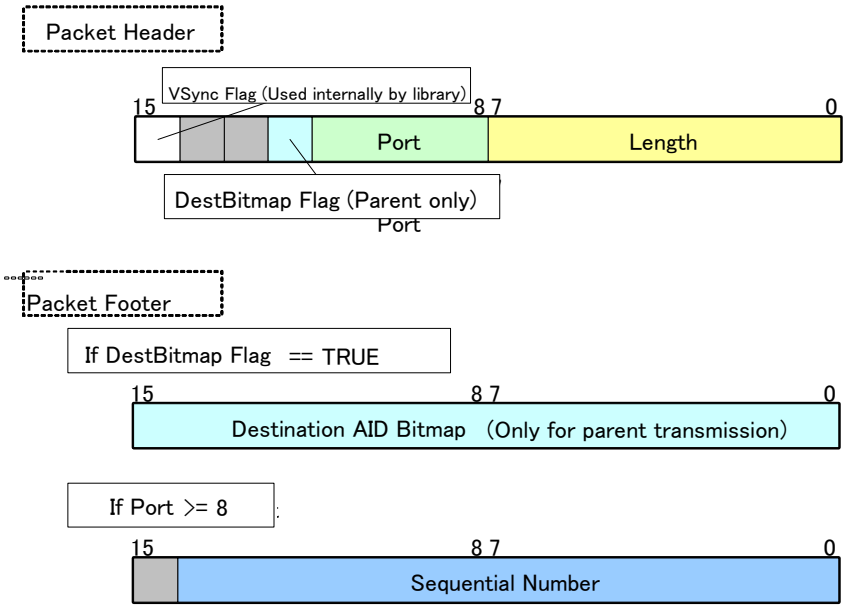
Figure 3-5 Parent and Child Packet Size Differences



A single packet consists of a 2-byte header, data to be sent, and a footer (a maximum of 4 bytes for a parent and 2 bytes for a child).

The bit assignments for headers and footers are shown in Figure 3-6. With the Wireless Communications library, there is no need for concern about the structure shown.

Figure 3-6 Bit Assignments for Headers and Footers



The header contains data length (in 2-byte units), port number, and control flags. The footer contains a bitmap of the destination children as well as sequential numbers. If the data length is 0, it is treated as 512 bytes.

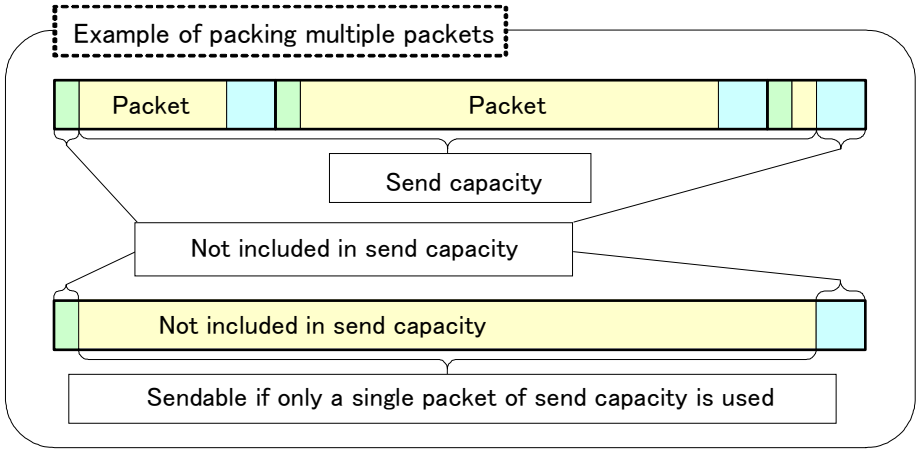
Packets sent from parent to children normally contain a destination bitmap. However, if the header's `DestBitmap` Flag is set to 0, the parent is broadcasting to all children, and the footer does not contain a destination bitmap.

The sequential number is used to control sequential communication. If the highest-order bit of the header's four-bit port number is enabled (that is, the port number is 8 or higher), a sequential number is added.

3.5.6 Packing Multiple Packets

In MP communications, only a method for transmitting data payload has been defined, but with this method, small amounts of data cannot be transmitted efficiently. Therefore, in port communications, multiple packets are packed as much as the maximum transmission capacity will allow, and then sent.

Figure 3-7 Packing Multiple Packets



Note that sizes of the header and footer portions of one packet are added internally, based on the value that has been set for the transmission capacity. The transmission capacity should be viewed as the maximum number of bytes available for user data.

Accordingly, when sending multiple packets, in addition to the actually transmitted data, the size of in-between headers and footers uses more space. For each packet, the size for each packet addition is up to 6 bytes for a parent transmission and up to 4 bytes for a child transmission.

When sending multiple packets at the same time, use this formula to determine the number of bytes.

[number of bytes used] = [total user data size to pack] + [added headers and footers] x ([number of packets to pack]-1)

[Added headers and footers] = 6 bytes for parent device or 4 bytes for child device

To simplify this document, consistent numeric values are used for the bytes added to headers and footers that assume all packets are sent using sequential communications. However, the raw communications footer is 2 bytes smaller; thus, in raw communications, you can send each packet with 2 bytes fewer than what is calculated in the above expression.

3.6 Data Sharing

3.6.1 Data Sharing

For games that rely on real-time communications, you can periodically share the same data (such as positional and movement information) with all participants. A Data Sharing library is available for this. We are planning to discontinue Key Sharing and the current implementation that uses Data Sharing internally.

Both Data Sharing and Key Sharing are the libraries that operate on ARM9 and use only the public Wireless Communications library functions.

3.6.2 How to Use

Code 3-1 Data Sharing Sample Code

```
#define DS_SIZE          8 // Share each 8 bytes
#define DS_MAX           8 // Max of 7 child devices + parent device
#define DS_BITMAP        0x00ff // aidBitmap for 8 devices

WMDataSharingInfo dsInfo; // This structure is about 2 KB; be careful where to allocate it
u16 sendData[DS_SIZE/sizeof(u16)]; // Send data
WMDataSet receiveData;    // Receive data
BOOL fUpdate;


... //  Initialize wireless communications and perform WM_StartMP()

WM_StartDataSharing( &dsInfo, DS_PORT, DS_BITMAP, DS_SIZE, TRUE );

//  Main loop
while ( TRUE )
{
    OS_WaitIrq(TRUE, OS_IE_VBLANK); // V-blank wait

    ... // Create sendData from PAD input and so on

    if ( WM_StepDataSharing( &dsInfo, sendData, &receiveData )
           == WM_ERRCODE_SUCCESS )
    {
        int i;
        for ( i=0; i<DS_MAX; i++ )
        {
            u16* p = WM_GetSharedDataAddress( &dsInfo, &receiveData, i );
            if ( p != NULL )
            {
                ... // Use p to configure the input from AID i
            }
        }
        fUpdate = TRUE;
    }
    else
    {
        fUpdate = FALSE;
    }

    ... // Execute the render process with the current internal state

    if ( fUpdate )
    {
        ...... // Update the game state based on the input
    }
}
```

First, immediately after MP communications have been started with the WM_StartMP function, call the WM_StartDataSharing function to initialize Data Sharing. Data can then be shared on the parent and child simply by calling the WM_StepDataSharing function at the start of each game frame.

If the WM_StepDataSharing function returns WM_ERRCODE_SUCCESS, this indicates that all participants in the Data Sharing are able to share data, so use that shared data to start a new game frame. The shared data can be obtained from the WM_StepDataSharing function as WMDataSet-type data. Use the WM_GetSharedDataAddress function to get data from this data set that was made by individual Nintendo DS systems.

On the other hand, if WM_ERRCODE_NO_DATASET is returned, this indicates that one of the communications partners is experiencing performance problems, so delay the game frame update and wait one picture frame.

For more information about this method, see the Data Sharing model demo in $TWLSDK_ROOT/build/demos/wm/dataShare-Model and *Wireless Communications Tutorial* (WmTutorial.pdf).

3.6.3 Single and Double Modes

The two operational modes in Data Sharing are Single and Double. To designate them, use the doubleMode argument of the WM_StartDataSharing function.

- Single Mode

  If the game frame is 30 fps or if the game frame is 60 fps but the frequency of the MP sequence is twice or more per picture frame, single mode can be used. It gets the data that was set with the previous WM_StepDataSharing function. When Data Sharing starts, a single empty data set that does not contain any AID data will be loaded.

- Double Mode

  Double mode is used when the game frame is 60 fps and the frequency of MP sequence is once per picture frame. It takes in the data that is set with the second WM_StepDataSharing function. When Data Sharing starts, two empty data sets that do not contain any AID data will be loaded.

One of the characteristics of MP communications is that two MP sequences are needed to collect data from a child and then return that data to the child device. Therefore, if there is one MP sequence in one picture frame, to call the WM_StepDataSharing function at a frequency of 60 fps (in other words, 1 game frame = 1 picture frame), a single buffer must be placed in the interval. This is Double Mode.

For the diagram, see section 3.6.7 General Information About Internal Operations. Essentially, the difference between Single Mode and Double Mode is the initial preparation of some empty data sets for loading.

3.6.4 Communications Data Size

The send data size used by Data Sharing is calculated as follows.

[parent device data size] = [shared data size] x [number of devices sharing data

(including parent device)] + 4

[child device data size] = [shared data size]

As a limitation on the library, the parent data size must be 512 bytes or less. This means that *[Shared data size] * [Number of shared devices]* must be less than or equal to 508. Also, the shared data size must be an even number. For example, if there are 5 child devices, the shared data size is up to 84 bytes.

*(84*6 + 4 = 508 <= 512, 86*6 + 4 = 520 > 512)*

When the number of children is 6 or more, the 5600-μs limitation for the required communication time, as explained in section 3.4.4 Transmission Capacity, determines the maximum size of shared data. For example, if there are 15 child devices, the shared data size is up to 12 bytes.

*((12*16 + 4) + (12+60)*15 = 1276 < 1280, (14*16 + 4) + (14+60)*15 = 1338 >= 1280)*

Table 3-6 contains a list of the maximum size of shared data for each number of child devices.

Table 3-6 Maximum Shared Data Size for Each Child Device

Number of Child Devices	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Maximum shared size due to restriction of the parent's data size <= 512 bytes.	**254**	**168**	**126**	**100**	**84**	72	62	56	50	46	42	38	36	32	30
Maximum shared size due to restriction of the required time for communication <= 5600 μs	-	-	-	-	-	**70**	**56**	**46**	**38**	**32**	**26**	**22**	**18**	**14**	**12**

When using this at the same time as a normal `WM_SetMPData*` function, multiple packets are packed.

Note: When calculating the respective maximum sizes for each parent and child device, the header and footer portion of the packet (6 bytes for parent, 4 bytes for child) must be added.

3.6.5 Precautions Related to Function Call Order

You must attempt to call the `WM_StartDataSharing` function immediately after calling the completion callback of `WM_StartMP`, and you must attempt to call the `WM_EndDataSharing` function immediately before the `WM_EndMP` function. This is a current limitation for Data Sharing.

To delay the start of Data Sharing, try not calling the `WM_StepDataSharing` function. No alarms or timers are used inside Data Sharing; its processes are driven by library function calls and send/receive callbacks. Even after the `WM_StartDataSharing` function is executed, as long as the `WM_StepDataSharing` function is not called, extra processes and communications are not carried out.

However, because nothing like a timer is being used, there are limits to the timing that calls the WM_StepDataSharing function. To perform stable Data Sharing, the WM_StepDataSharing function must be called at the earliest timing possible after a V-Blank interrupt. This way, the send data can be set up to the timing (in V-Count terms, child device 240 / parent device 260) that will carry out the preparation of the next MPS sequence on ARM7.

3.6.6 Precautions for Operating at 30 FPS or Less

With an application that has a game frame of 30 fps, the WM_StepDataSharing function is called once every two frames, but if WM_ERRCODE_NO_DATASET is returned, the next call must be performed in the very next frame.

Code 3-2 Shared Data Process Required When Operating at 30 FPS or Less

```
01: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
02: ----
03: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
04: ----
05: WM_StepDataSharing() == WM_ERRCODE_NO_DATASET
06: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
07: ----
08: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
09: ----
```

Perform the process in Code 3-2 when there is a failure and make sure that a one-frame interval is not placed between the 5th and 6th frames. Otherwise, if the parent and child are off by one frame, a fix will not be possible.

When at 30 fps and calling in the manner described above, the parent/child game frame timing discrepancy can be fixed, but at 20 fps and below, the timing cannot be completely brought into line. This is one of the current limitations for Data Sharing. However, even at 20 fps and below, the consistency of shared data is maintained. So even if the timing is a bit off, it is possible to share only the data and Data Sharing can be used.

3.6.7 General Information About Internal Operations

Figure 3-8 and Figure 3-9 illustrate the internal operations of Data Sharing. The WM_StepDataSharing function is noted as StepDS in these diagrams.

Figure 3-8 Data Sharing (Single Mode)

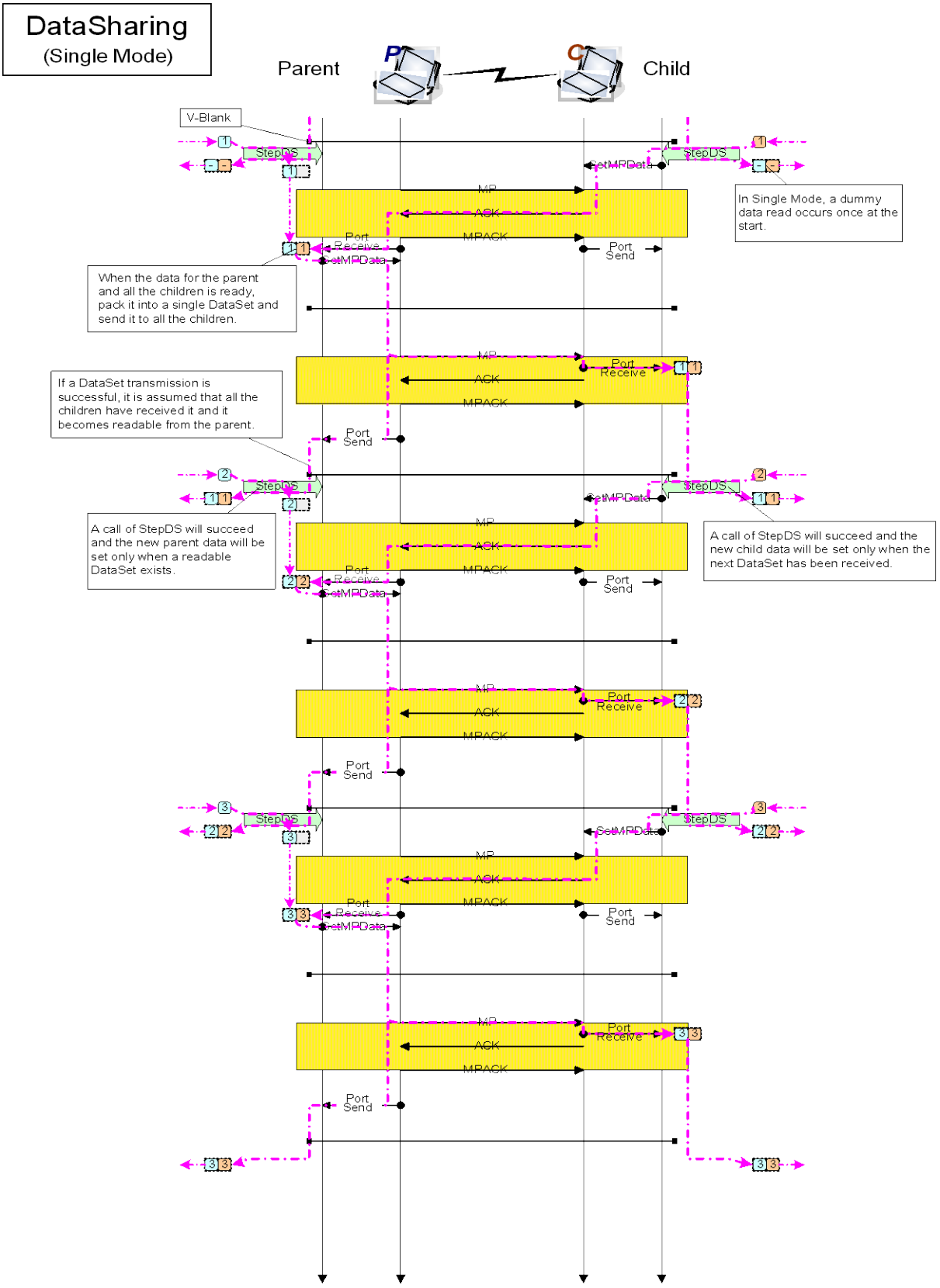
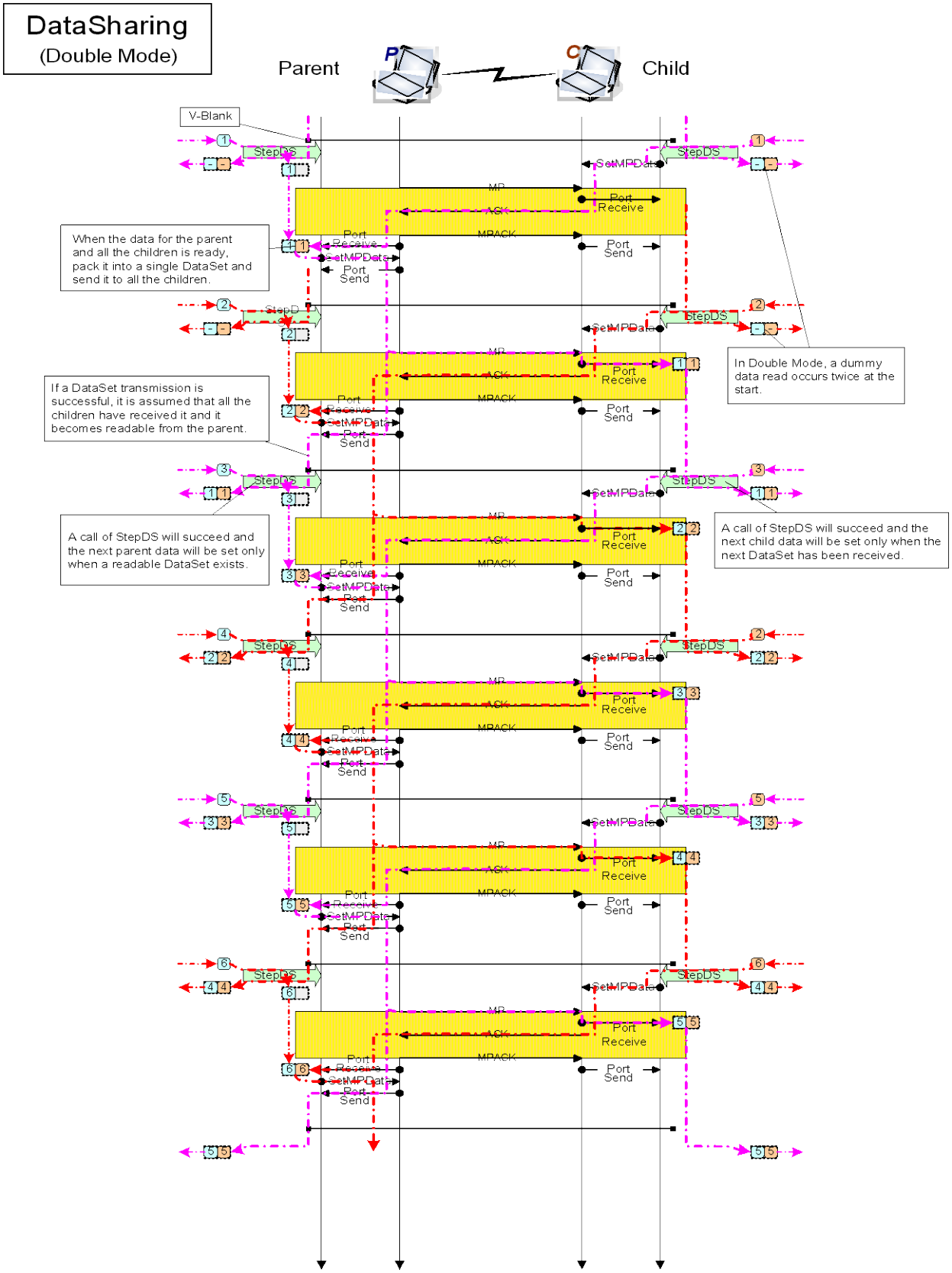


Figure 3-9 Data Sharing (Double Mode)



3.7 Event Notifications Returned from the Wireless Communications Library

For some asynchronous function callbacks, an event notification call from the Wireless Communications library is issued, in addition to the "operation complete" notification corresponding to the call. The trigger for the callback is stored as the value of the `WMStateCode` enumerated type in the state field of the `WM*Callback` structure of the callback argument.

Table 3-7 defines the notifications associated with various functions and the type of `WMStateCodes` for the notifications. With the exception of the `WM_SetMPData*` function, asynchronous functions have an internal table of callback functions by function, so do not assign a different callback function to the same function each time it is called.

Table 3-7 Wireless Communications Functions and Their Notifications

Function	WMStateCodes
`WM_StartParent`	• `WM_STATECODE_PARENT_START`: Asynchronous notification that the function call is complete. • `WM_STATECODE_BEACON_SENT`: The beacon is sent. No special processing is required. • `WM_STATECODE_CONNECTED`: Child device connected to the parent. At connection time, the following data is sent through callback. – AID of the child connected to `WMStartParentCallback.aid`. – MAC address of the child in `WMStartParentCallback.macAddress`. – User region (the second 24 bytes) of the SSID declared by the child in `WMStartParentCallback.ssid`. • `WM_STATECODE_DISCONNECTED`: A child disconnected from the parent. `WMStartParentCallback.aid` and `WMStartParentCallback.macAddress` behave as in `WM_STATECODE_CONNECTED`. • `WM_STATECODE_DISCONNECTED_FROM_MYSELF`: This notification is used when a WM function is called within an application and a parent disconnects its own child. The same values that are used with `WM_STATECODE_DISCONNECTED` are used in notifications.
`WM_StartConnect` `WM_StartConnectEx`	• `WM_STATECODE_CONNECT_START`: Asynchronous notification that the function call is complete. If no more entries are being received because the parent's entry flag is set to `FALSE`, or the device has reached its maximum number of connections, `WM_ERRCODE_NO_ENTRY` or `WM_ERRCODE_OVER_MAX_ENTRY` may be returned in `errcode`. Even if `WM_ERRCODE_SUCCESS` is returned in this state, it does not necessarily mean that the connection is complete. The completion of a connection is notified with `WM_STATECODE_CONNECTED`. Caution is also needed for the `WM_ERRCODE_OVER_MAX_ENTRY` returned when the parent device exceeds its maximum number of connections, because the notification is issued after a single `WM_ERRCODE_SUCCESS` is returned. `WM_ERRCODE_OVER_MAX_ENTRY` may be returned temporarily by a parent device when a connection is attempted by another client whose GGID or TGID does not match that of the parent. In this event, it is recommended to perform a retry. Specifically, this error occurs frequently when a child device in Download Play mode attempts a connection immediately after rebooting its program and another IPL child is attempting a download connection based on an old parent beacon.

Function	WMStateCodes
	• WM_STATECODE_BEACON_LOST: A connected parent device beacon fails to be received for a fixed amount of time. There is a high possibility that the signal has degraded and the V-blank period is damaged, but there is no further processing required. • WM_STATECODE_CONNECTED: Connection was made with the parent device. At connection time, the following data is sent through callback: – AID of the child connected to WMStartConnectCallback.aid • WM_STATECODE_DISCONNECTED: A parent disconnected from a child. WMStartConnectCallback.aid behaves as in WM_STATECODE_CONNECTED. • WM_STATECODE_DISCONNECTED_FROM_MYSELF: Used when a WM function is called in an application and a parent disconnected its own child. The same values that are used with WM_STATECODE_DISCONNECTED are used in notifications.
WM_StartMP WM_StartMPEx	• WM_STATECODE_MP_START: Asynchronous notification that the function call is complete. • WM_STATECODE_MPEND_IND: Parent device sends out the MP_ACK frame and successive MP sequences are finished. Normally, there is no specific need to perform this process. This notifies the pointer to the WMMpRecvHeader structure that stores the contents of the frame received from the child in WMStartMPCallback.recvBuf. To receive the data, we recommend using the port-receive callback. However, because the recvBuf field is defined as a pointer to a WMMpRecvBuf type, you must forcibly recast the field type. • WM_STATECODE_MP_IND: Child received the MP frame from parent. Notifies the pointer to the WMMpRecvBuf structure that stores the contents of the frame received from the parent in WMStartMPCallback.recvBuf. To receive the data, we recommend using the port-receive callback. If the port was not assigned with the PollBitmap of the MP frame, the errcode is WM_ERRCODE_INVALID_POLLBITMAP. Because this occurs most often when multiple child devices are connected, it should not be handled as an unrecoverable error. Also, if counting the header information and nothing was included in the received MP frame, WM_ERRCODE_NO_DATA is notified as the errcode. Normally, as long as the WM library is operating, this cannot occur. • WM_STATECODE_MPACK_IND: The child received the MP_ACK frame from the parent device. Normally, there is no particular need to perform this process. If not primarily self-designated with the PollBitmap of the MP frame that corresponds to this MP_ACK, the errcode is WM_ERRCODE_INVALID_POLLBITMAP. Because this occurs most often when multiple child devices are connected, it should not be handled as an unrecoverable error. Otherwise, if the parent is not notified with the PollBitmap field of the MPACK frame that the Key(Null) response frame was not received, the errcode is WM_ERRCODE_SEND_FAILED. Even if a given time passes after receiving the MP frame, if the MPACK frame could not be received, this indication occurs, the errcode is WM_ERRCODE_TIMEOUT, and there is a notification.

Function	WMStateCodes
WM_SetIndCallback	• WM_STATECODE_FIFO_ERROR: This WMStateCode is sent to the ARM7 when the execution control queue overflows due to a process overload on the ARM7. Treat as a non-recoverable fatal error. • WM_STATECODE_INFORMATION: Notification about some type of internal event. The notification is stored in WMIndCallback.reason. WM_INFOCODE_FATAL_ERROR, which is sent as notification when a fatal error occurs and the ignoreFatalError argument of the WM_StartMPEx function is set to TRUE, is defined as the value placed in reason. • WM_STATECODE_BEACON_RECV: Beacon from the connected parent is received. Normally, there is no need to perform this process. If WMIndCallback.state was this value, by recasting the type in WMBeaconRecvIndCallback, the GameInfo can be obtained from WMBeaconRecvIndCallback.gameInfoLength, WMBeaconRecvIndCallback.gameInfo, and so on. • WM_STATECODE_DISASSOCIATE: Used for debugging. Normally, you can ignore this constant. • WM_STATECODE_REASSOCIATE: Used for debugging. Normally, you can ignore this constant. • WM_STATECODE_AUTHENTICATE: Used for debugging. Normally, you can ignore this constant.
WM_SetPortCallback	• WM_STATECODE_PORT_INIT: This is called with the interrupts disabled while WM_SetPortCallback is called. This notification stores the AID bitmap for the partner currently connected to WMPortRecvCallback.connectedAidBitmap. If the connection has not started yet, 0 is stored in connectedAidBitmap. In addition, void* arg, passed to an argument to WMSetPortCallback, is passed to *.arg. • WM_STATECODE_PORT_RECV: Data is received from the communication partner. The following data is sent through callback. – AID of the child connected to WMStartConnectCallback.aid – AID of the send source in WMPortRecvCallback.aid – Pointer to the receive data in WMPortRecvCallback.data – Size of the receive data in WMPortRecvCallback.length – The void* arg given to the argument of WMSetPortCallback in *.arg • WM_STATECODE_CONNECTED: Immediately after notification in the callbacks of the WMStartParent and WMStartConnect* functions that the connection was established, similar notifications are sent to the receive callbacks of every port. Whether it is a parent or child, WMPortRecvCallback.aid always takes the AID of the connection partner at that time (the child device is fixed at 0, and the parent takes the AID of the connected child). Its own AID is stored in *.myAid. Also, the MAC address and user region SSID (for the parent device) of the respective communication partners are set to *.macAddress and *.ssid. • WM_STATECODE_DISCONNECTED: Immediately after notification in the callbacks of the WMStartParent and WMStartConnect functions that the connection was terminated due to an external cause, similar notifications are sent to the receive callbacks of every port. The same notes apply to AID as to WM_STATECODE_CONNECTED. Also, the MAC address of the disconnected partner is stored in *.macAddress. • WM_STATECODE_DISCONNECTED_FROM_MYSELF: This notification is used when a WM function is called in an application and a parent disconnects its own child. The same values used with WM_STATECODE_DISCONNECTED are used in notifications.

Function	WMStateCodes
WM_SetMPData WM_SetMPDataToPort WM_SetMPDataToPortEx	• WM_STATECODE_PORT_SEND: Only one kind of WMStateCode is notified as the completion callback of an asynchronous function, but because it is an important notification in the communication controls, it is described separately here. In WMPortSendCallback.errcode, the following data is sent through callback. – WM_ERRCODE_SUCCESS if the send succeeds. – WM_ERRCODE_SEND_FAILED if the send fails. – WM_ERRCODE_SEND_QUEUE_FULL if the send queue is full. With sequential communications, WM_ERRCODE_SEND_FAILED will not be returned except when communications have been terminated. Bitmap of the AID of the partner that must retry is stored in *.restBitmap. AID bitmap of the communications partner for which the send was a success is stored in *.sentBitmap. The send destinations that are not connected or that become disconnected during a send are not included in *.restBitmap or *.sentBitmap. The condition for WM_ERRCODE_SUCCESS to return to *.errcode is that *.restBitmap is 0. In other words, communications are successfully sent to all designated send destinations which are still connected. To confirm that everything was sent to the designated send destination, re-check *.sentBitmap (except when the partner has called the WM_EndMP function). While it is guaranteed that the send is a success for communications partners that are included in *.sentBitmap, there is no such guarantee for communications partners that are not included there. Exactly one callback will be called each time the WM_SetMPData* function is called. At this time, during the interval between the call to the function to the call to callback, do not overwrite the memory region for the send data. It is also possible to get the address of the set send data with WMPortSendCallback.data. The argument of the WM_SetMPDataToPortEx function is passed to *.arg.

3.8 Error Codes Returned from the Wireless Communications Library

3.8.1 Return Values of Functions That Return a WMErrCode Type

In Table 3-8, the rows contain functions, and the columns contain their return values. They are abbreviated by omitting the WM_ERRCODE_ prefix from the WMErrCode enumerated values.

Table 3-8 Return Values of Functions that Return a WMErrCode Type

Function Name	SUCCESS	FAILED	OPERATING	ILLEGAL_STATE	WM_DISABLE	NO_DATASET	INVALID_PARAM	NO_CHILD	FIFO_ERROR
WM_Initialize			○	○	○		○		○
WM_Init	○			○	○		○		
WM_Enable			○	○					○
WM_PowerOn			○	○					○
WM_End			○	○					○
WM_PowerOff			○	○					○

Function Name	SUCCESS	FAILED	OPERATING	ILLEGAL_STATE	WM_DISABLE	NO_DATASET	INVALID_PARAM	NO_CHILD	FIFO_ERROR
WM_Disable			○	○					○
WM_Finish	○			○					
WM_Reset			○	○					○
WM_StartMP*			○	○			○		○
WM_SetMPParameter			○	○					○
WM_SetMPData*			○	○			○	○[1]	○
WM_EndMP			○	○					○
WM_SetParentParameter			○	○			○		○
WM_StartParent			○	○					○
WM_EndParent			○	○					○
WM_StartScan*			○	○			○		○
WM_EndScan			○	○					○
WM_StartConnect*			○	○			○		○
WM_Disconnect			○	○			○	○[1]	○
WM_DisconnectChildren			○	○				○[1]	○
WM_SetIndCallback	○			○					
WM_SetPortCallback	○			○					
WM_StartDataSharing	○	○		○			○		
WM_EndDataSharing	○			○			○		
WM_StepDataSharing	○	○		○		○[1]	○		
WM_SetGameInfo			○	○			○		○
WM_SetBeaconIndication			○	○			○		○
WM_SetLifeTime			○	○					○
WM_MeasureChannel			○	○					○
WM_InitWirelessCounter			○	○					○
WM_GetWirelessCounter			○	○					○
WM_SetEntry			○	○					○
WM_StartKeySharing	○	○		○			○		
WM_EndKeySharing	○			○			○		
WM_GetKeySet	○	○		○		○[1]	○		
WM_ReadStatus	○			○			○		
WM_StartDCF			○	○			○		○
WM_SetDCFData			○	○			○		○
WM_EndDCF			○	○					○
WM_SetWEPKey			○	○			○		○
WM_SetWEPKeyEx			○	○			○		○

1: This error code is generated based on a variety of conditions, even if the application process is appropriate. Communications will continue as normal, so this is not treated as a fatal error.

3.8.2 errcode Values Returned to Callback Functions

The rows in Table 3-9 contain the functions and values of the state field of the `WM*Callback` structure returned to their callbacks. The columns contain the values of the `WM*Callback` structure's `errcode` field. They are abbreviated by omitting the `WM_STATECODE_` and `WM_ERRCODE_` prefixes. Δ indicates that the `WM*Callback.state` values are sometimes indefinite.

Table 3-9 errcode Values Returned to the Callback Function

Function Name	WM*Callback.state	SUCCESS	FAILED	OPERATING	ILLEGAL_STATE	WM_DISABLE	NO_DATASET	INVALID_PARAM	NO_CHILD	FIFO_ERROR	TIMEOUT	SEND_QUEUE_FULL	NO_ENTRY	OVER_MAX_ENTRY	INVALID_POLLBITMAP	NO_DATA	SEND_FAILED	FLASH_ERROR
WM_Initialize		○	○							○								
WM_Enable		○								○								
WM_PowerOn		○	○		○					○								
WM_End		○	○		○					○								
WM_PowerOff		○	○		○					○								
WM_Disable		○			○					○								
WM_Reset		○	○							○								
WM_StartMP*	MP_START	○			○			○		Δ								
	MPEND_IND1	○																
	MP_IND[1]	○													○[2]	○[3]		
	MPACK_IND[1]	○									○[2]				○[2]		○[2]	
WM_SetMPParameter		○			○			○		○								
WM_SetMPData*	PORT_SEND	○			○			○		Δ		○[2]					○[2]	
WM_EndMP		○	○		○					○								
WM_SetParentParameter		○	○					○		○								
WM_StartParent	PARENT_START	○	Δ		○			○		Δ								
	BEACON_SENT$_1$	○																
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																
WM_EndParent		○	○		○					○								
WM_StartScan*	PARENT_NOT_FOUND	○	Δ		○			○		Δ								
	PARENT_FOUND	○																
WM_EndScan		○	○		○					○								
WM_StartConnect*	CONNECT_START	○	Δ		○			○		Δ			○	○[4]				
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																
	BEACON_LOST$_1$	○																

Function Name	WM*Callback.state	SUCCESS	FAILED	OPERATING	ILLEGAL_STATE	WM_DISABLE	NO_DATASET	INVALID_PARAM	NO_CHILD	FIFO_ERROR	TIMEOUT	SEND_QUEUE_FULL	NO_ENTRY	OVER_MAX_ENTRY	INVALID_POLLBITMAP	NO_DATA	SEND_FAILED	FLASH_ERROR
WM_Disconnect		○	○		○					○								
WM_DisconnectChildren		○	○		○					○								
WM_SetGameInfo		○	○							○								
WM_SetBeaconIndication		○	○							○								
WM_SetLifeTime		○	○							○								
WM_MeasureChannel		○	○		○					○								
WM_InitWirelessCounter		○	○							○								
WM_GetWirelessCounter		○	○							○								
WM_SetEntry		○	○							○								
WM_StartDCF	DCF_START	○								Δ								
	DCF_IND	○																
WM_SetDCFData		○	○							○								
WM_EndDCF		○								○								
WM_SetWEPKey		○								○								
WM_SetWEPKeyEx		○	○							○								
WM_SetIndCallback	FIFO_ERROR									○								
	INFORMATION	○																
	BEACON_RECV[1]	○																
	DISASSOCIATE[1]	○																
	REASSOCIATE[1]	○																
	AUTHENTICATE[1]	○																
	UNKNOWN																	○
WM_SetPortCallback	PORT_RECVINIT	○																
	PORT_RECV	○																
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																

1: It is OK if processing is not normally performed on this state notification.

2: This errcode is generated based on a variety of conditions, even if the application process is appropriate.

3: Communications will continue as normal, so this is not treated as a fatal error.

4: This is an errcode that should not be generated as long as the library is operating normally. After the `WM_ERRCODE_SUCCESS` notification arrives, there may be a notification for this error again.

3.9 Precautions for Using the Wireless Communications Library

This section describes precautions for using the Wireless Communications library.

3.9.1 Load Due to the Use of Wireless Communications

Because the wireless communications driver in the current SDK is 100 KB or larger, the wireless communication driver codes cannot be loaded into the ARM7 working memory and are stored in main memory. Therefore, when performing wireless communications, ARM7 frequently accesses main memory. With continuous access from ARM9 to main memory, as happens with rendering, this results in large overhead (normally, ARM7 has a higher access priority to main memory than ARM9).

Conversely, if ARM9 is given priority to main memory access (for example, when HDMA is used), execution of the wireless communication driver may be adversely affected because ARM9 frequently accesses main memory. In particular, if multipurpose DMA accesses main memory, execution of ARM7 program is likely to be delayed for a long time. If the wireless communication drivers are operated when ARM9 has the higher access priority to main memory, try not to use the multipurpose DMA.

One effective method is allocating VRAM-C or VRAM-D for use by ARM7 and storing the wireless communication driver there to reduce the amount of time during which ARM7 uses the main memory bus. This method decreases the time for ARM7 to appropriate the main memory bus. For more information, see the WVR library reference or the *ichneumon* component of *Component Description* (`AboutComponents.pdf`).

3.9.2 Callback

The callback is called inside the PXI interrupt handler. Functions that cannot be called when interrupts are forbidden cannot be used. Also, try not to call any long-term processes. If another ARM9 interrupt is delayed, there are times when the ARM7 wireless communication driver waits for the ARM9 callback to finish. This wait time negatively impacts the wireless communications process.

3.9.3 Cache Process

Forced cache storage is performed in some functions to pass data to ARM7. We recommend that 32-byte aligned data be passed to the relevant function and that the data region be allocated as a multiple of 32 bytes. If this is not done, the surrounding memory regions are also forcibly stored together in the cache and unforeseen operations might occur.

On the other hand, two kinds of data are passed from the library to application: the data passed after the cache is invalidated and the data that requires the cache to be invalidated by the application.

Table 3-10 Memory Cache Processes

Description	Memory Regions
The given memory region is that which is stored in the internal cache of the library.	The `WM_SetParentParameter` function argument `pparaBuf` or `pparaBuf->userGameInfo`, `WM_StartConnect*`, `WM_SetMPData*`, `WM_StartDCF`, `WM_SetDCFData`, `WM_SetWEPKey*`.
The given memory region is that which is not stored in the cache. (It is stored after being copied to an internal buffer)	`WM_SetGameInfo`, `WM_StartScan*`, `WM_SetMPParameter`. Other functions that pass only small memory regions.
The region that takes in the data is that which is passed after the cache is invalidated inside the library.	Data fields of the port reception callback set with the `WM_SetPortCallback` function, `WM_ReadStatus`.
The region that takes in the data is that which is passed without the cache being invalidated inside the library.	The region designated with the `param->scanBuf` argument of the `WM_StartScan*` function. Passed after `WMStartScan*Callback` is invalidated.

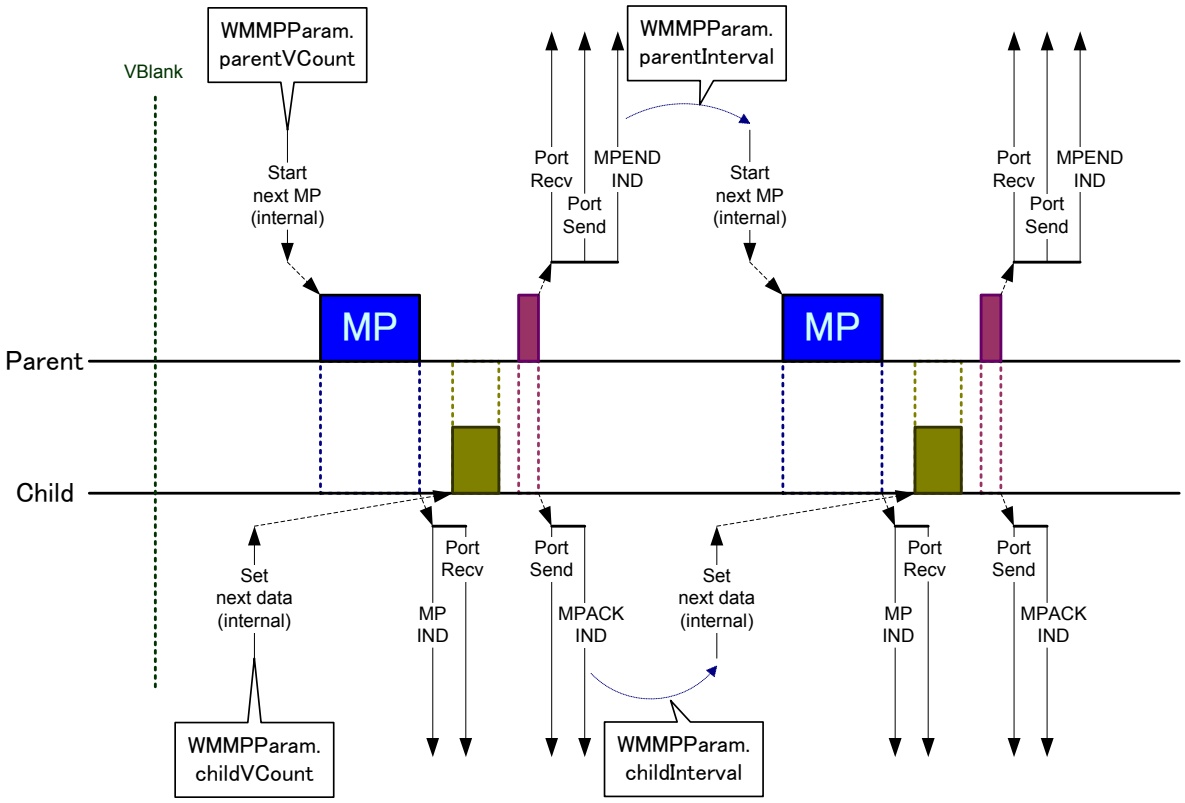
3.10 Taking Greater Control over Communications

This section describes higher-use wireless communications and ways to fine-tune performance.

3.10.1 Overview of the Timing Control Parameter of MP Communications

Figure 3-10 shows how a number of parameters can be used to control the timing of MP communications. Configure each of these parameters by setting their values in specific fields of the `WMMPParam` structure and calling the `WM_SetMPParentParameter` function.

Figure 3-10 Parameters That Can Be Used to Control MP Communications Timing



3.10.2 parentVCount, childVCount

In frame-synchronous communications mode, `WMMPParam.parentVCount` defines the V-Count value for internally starting the first MP sequence in each frame. The default value is 260. Similarly, `WMMPParam.childVCount` defines the V-Alarm value for internally setting the first Response data in each frame on the child side. The default value is 240. These two values can also be set by the `WM_SetMPParameter` function's wrapper function, `WM_SetMPTiming`.

By changing the values of `parentVCount` and `childVCount`, you can adjust the timing of MP sequence occurrences in frame-synchronous communications mode. Because the wireless driver on the ARM7 side frequently accesses main memory around the time of the MP sequence, applications on the ARM9 side will often stall when accessing main memory. You can sometimes lessen the impact of this stalling by adjusting the MP sequence to a time when the application on the ARM9 side is not heavily accessing main memory. The value of `parentVCount` is irrelevant to the timing of communications when the communications environment is poor or when communications are carried out in continuous communications mode. Also note that if the ARM7's internal processing is delayed, the MP sequence will actually occur later than the value set in `parentVCount`.

3.10.3 parentInterval, childInterval

In frame-synchronous communications mode, `WMMPParam.parentInterval` defines the time interval (in microseconds) that passes between the end of one MP sequence and the internal start of the next MP sequence by the parent. It affects the second and all subsequent MP sequences that occur. Similarly, `WMMPParam.childInterval` defines the interval between the end of one MP sequence and the internal setting of Response data to prepare for the next MP sequence by the child. The default value is 1000 microseconds for `parentInterval` and 0 microseconds for `childInterval`. Both of these values can also be set by the `WM_SetMPParameter` function's wrapper function, `WM_SetMPInterval`.

Note: If the ARM7's internal processing is delayed, the actual transmission interval between MP sequences during continuous communications will be longer than the value set in `parentInterval`.

In continuous communications, the Send data that will be transmitted in the next MP sequence is determined at the end of the interval period for both the parent and child and is based on the Send data set in the Send queue. In current implementation, the interval period does not begin until the Port Receive and Port Send callbacks of the previous MP sequence have ended on the ARM9 side. Because of this specification, data that gets set in the Send queue by the `WM_SetMPDataToPort` function during these callbacks will be set in time for transmission in the next MP sequence.

The default interval period is longer for the parent than for the child because this ensures that the child's Response data is set in time for the next MP sequence. Setting the parent and child to the same interval period will result in numerous communication errors when the child has a weighty callback process. This is because the wireless communication driver on the child's side waits for the ARM9's processing to end, so the next MP sequence can come before the child has had time to set the Response data. The child MP frame will stop responding until the response data is set, so the result will be a communication failure during that interval. An example of this can be observed in the *wbt-fs demo*, where the callback process on the child side occasionally takes around 700 microseconds, and communications fail with high frequency if the parent and child are set to have the same interval period.

If you know that the processing load inside the callbacks for parent and child will always be the same, you can shorten `parentInterval` and raise the throughput of MP communications during continuous communications. Conversely, if the child's Port Send and Port Receive callback processes take longer than 1000 microseconds, you will need to set `parentInterval` longer than the default value. Of course, you do not really want to have a process like a callback that takes longer than 1000 microseconds inside the interrupt handler. If this situation arises, you will need to rethink your design.

For heavy Send and Receive processes, limit your process requests from inside the Port Send and Receive callbacks to a thread for communications and immediately exit the callbacks. You can perform continuous communications without wasting time by using this thread to configure `parentInterval` and `childInterval` for the worst-case longest duration it should take to set the next Send data. Because the Send queue has 32 steps, an alternative strategy is a design where multiple sets of Send data in the Send queue are always buffered.

3.10.4 Changing Transmission Capacity Dynamically

Communications normally proceed using the transmission capacities that were set by the parent in `parentMaxSize` and `childMaxSize` using the `WM_SetParentParameter` function. However, the parent can use the `WM_SetMPParentSize` and `WM_SetMPChildSize` functions as required to reconfigure the transmission capacities of the parent and child.

Note: These values cannot exceed the initial values set by the `WM_SetParentParameter` function. Also, the child's transmission capacity gets updated to the value set by the parent every time the parent sends an MP sequence. As a result, even though the child can set its own transmission capacity using the `WM_SetMPChildSize` function, that value can only be used when Send data is prepared for the MP sequence that follows immediately.

Table 3-11 Changing Parent and Child Transmission Capacity

	Initial Value	Maximum Value That Can Be Set	How to Reset Value on Parent Side	How to Reset Value on Child Side
Parent transmission capacity	`parentMaxSize` value in the parent's beacon	Same as left	`WM_SetMPParentSize`	(Only has meaning on the parent side)
Child transmission capacity	`childMaxSize` value in the parent's beacon	Same as left	`WM_SetMPChildSize`	`WM_SetMPChildSize` This gets overwritten by parent's setting when MP frame is received

Be careful about clashes between the child transmission capacity set for the child and the child transmission capacity configured in the parent for the child. Communications can proceed without trouble if the child's transmission capacity is set smaller than the setting configured in the parent for the child. However, if the child transmission capacity is set larger than this configuration, the parent will not give sufficient time for the sending child data and the data will not return from the child to parent. For more information, see the description in section 3.4.2 MP Communications Operations. In the example MP sequence, the child transmission capacity on the child side gets updated by the MP sequence, so the subsequent resending of data proceeds without communication problems between parent and child. Unnecessary communications can be avoided through cooperation between parent and child and through use of the `WM_SetMPChildSize` function to simultaneously update the child transmission capacity.

Changing transmission capacity would be meaningful in two cases.

- When you want to cut unnecessary child transmission capacity.

According to the MP Communications specifications, if the child transmission capacity was set to 32 bytes, each communication would take *[32 bytes] x [number of children]* amount of time. This occurs even when the child is sending only 2 bytes of data every time. By designing the application to reduce the child transmission capacity according to the communications mode, you can reduce the overall time needed for communications, which will make communications more stable.

The parent transmission capacity does not need to be controlled because the time spent for sending data in communications on the parent side only takes as long as is necessary for the amount of Send data.

- When you want to maximize transmission capacity based on the number of connected children.

Assume the parent transmission capacity is 512 bytes and you want to connect a maximum of five children. When five children are connected to the parent, the child transmission capacity calculates out to 92 bytes because the limitation on communications time is 5600 microseconds. If the transmission capacity is fixed, it does not change even when only one child is connected. But if transmission capacity is set dynamically, it can be maximized depending on the number of connected children. With this setup, the transmission capacity calculates out to 512 bytes for one child, 322 bytes for two children, 194 bytes for three, and so on. However, the TWL-SDK does not have an upper-level protocol that can make use of this kind of dynamically set transmission capacity.

If your goal of changing transmission capacity is consistent with case 1, you can use these functions relatively safely. However, if your goal is more in line with case 2, you should be cautious about many points, and it is normally recommended to avoid this method. Here are some of the many points to heed for case 2.

Normally, when the `WM_StartMP` function executes, a pre-check is carried out on the transmission capacity and the buffer size used for the maximum number of children. You must disable this pre-check by using the `WM_SetMPParameter` function to set `WMMPParam.ignoreSizePrecheckMode` to `TRUE`. This results in two things: suppression of the warning related to the 5600-microsecond restriction on needed communications time and prevention of errors by pre-calculating the receive buffer size.

Taking case 2 as an example again, assume that `WMParentParam` sets both the parent and child transmission capacities to 512 bytes and the maximum number of connected children to five. If `ignoreSizePrecheckMode` is set to FALSE, when the `WM_StartMP` function executes, the pre-check will display a warning for debug output because the communications time totals 13970 microseconds (exceeding the 5600-microsecond limitation). Also, the maximal receive buffer size on the parent is 1408 bytes when having two children whose child send capacity is 322 bytes when actually changing the child send capacity within the 5600 microsecond limitation. However, when the `WM_StartMP` function runs the pre-check, it calculates that 5312 bytes are required (that is, the 512 bytes of child transmission capacity x 5 children). Thus, when a Receive buffer of 1408 bytes is passed to the `WM_StartMP` function, it generates the `WM_ERRCODE_INVALID_PARAM` error. To avoid this error and to fit the data into the smallest required buffer size, set `ignoreSizePrecheckMode` to `TRUE`.

When the pre-check is disabled, the parent will enter the `MP_PARENT` state even when the Receive buffer size appears too small. If a check for the Receive buffer size fails at execution, the MP sequence will not execute. If the transmission capacity and other parameters are not corrected to appropriate values, the MP lifetime will expire after a certain period and the connection will be dropped.

If more children are added and the transmission capacity is immediately adjusted to keep the Receive buffer size sufficient, communications can proceed without problems. However, if the child's Receive buffer is too small, communications will not operate normally nor be able to proceed. Because of this,

you need to prepare a Receive buffer for the child that can accommodate the maximum value for the parent transmission capacity as defined by `parentMaxSize` in the beacon.

Unlike the parent's Receive buffer, the child's Receive buffer is not affected by the number of connected children. Thus, even if you prepare this maximum-size buffer, it will not always affect the amount of memory used. With the pre-check disabled, the library will not check the 5600-microsecond limitation on communications time. Therefore, you will need to be very careful when setting parameters with your application.

As these points suggest, setting parameter values with `ignoreSizePrecheckMode` must be done carefully because mistakes will cause unusual behavior during execution. If any of these points are unclear, refrain from using `ignoreSizePrecheckMode`.

3.10.5 Controlling PollBitmap

The parent may want responses from all or specific children; `PollBitmap` in the MP frame indicates the children from which the parent wants a response. By controlling `PollBitmap` from the beginning, you can cut down on the overall communications time by restricting responses to specific children. However, a child not specified by `PollBitmap` cannot send out a Key Response frame, so note that the child will not get an opportunity to send data to the parent. For this reason, in normal MP communications the `PollBitmap` is always sent with the bit standing for all connected children at all times other than for retransmissions; this way, a window of opportunity exists for communications from each child.

To provide fine control over `PollBitmap`, the Wireless Communications library has prepared the `minPollBmpMode` and `singlePacketMode` operation flags in the `WM_StartMPEx` and `WM_SetMPParameter` functions. However, to use these operating modes, the complex restrictions below must be cleared. If you do not have thorough understanding of the wireless communications protocol, do not enable the flags under normal circumstances.

When `minPollBitmapMode` is enabled, specify as `PollBitmap` the logical OR of the send destination for the packet(s) that the parent is attempting to send in a particular sequence. In this case, use the flag along with `singlePacketMode` to prevent mistaken attempts to communicate with more partners than it is designed for. If a large send volume is specified under the assumption that the simultaneous communications partner is limited, and if the set value of `PollBitmap` is greater than intended, the parent receive buffer can overflow. If this happens, an insufficient buffer is detected when the MP sequence starts, and the sending will stop. After such a stop, it is possible to recover by reducing the send volume and avoiding limitations on the receive buffer, but it is difficult to determine any causal factors from the application side.

To use `minPollBmpMode`, you must perform communication once every 60 seconds with ports 8 through 15 on every child. This way, the sequence numbers used in Sequential communication do not cycle through. To avoid mistaken attempts to communicate with more partners than designed, use `minPollBmpMode` with `singlePacketMode`.

3.11 FAQ

Some of the common questions asked by Wireless Communications library users are shown below in question-and-answer format.

3.11.1 Initialization Process

Q: A valid value is not returned for the `WM_GetAllowedChannel` function.

A: The `WM_GetAllowedChannel` function does not return a valid value until after the `WM_Init` function is called. If it was called before the initialization, it returns `0x8000`, which indicates an error.

3.11.2 Connection Process

Q: How do I determine the values for transmission capacities, Send and Receive Buffer sizes, and all other communication parameters?

A: Table 3-12 lists procedures for determining typical parameter values.

Table 3-12 Procedures for Determining Parameter Values

Typical Determinations	Example
Determine the maximum number of connected children.	Assuming that 3 children are connected to the parent, set `WMParentParam.maxEntry` to 3.
If using data sharing, determine the number of shared bytes.	Set 16 bytes for data sharing.
To calculate the data size used by parent and child for data sharing, use the expression described in section 3.6.4 Communications Data Size.	The data size used by the parent for data sharing is 16 x (3+1) +4 = 68 bytes. The data size for each child is 16 bytes.
Determine the number of packets and the size for communications in situations other than data sharing. (When making these determinations, be aware that by increasing the maximum size of data that can be sent simultaneously from the children, the transmission time will always be consumed to this maximum value, even when there is only a small amount of data to send.)	To do a block transfer using WBT, have the parent use 128 bytes and each child use 14 bytes. For event notifications from the parent, use 32 bytes in an independent Sequential communication.
Count up the number of packets that can be sent at the same time, and use the expression in section 3.5.6 Packing Multiple Packets to calculate the number of bytes needed for the parent's transmission capacity.	For the parent, data sharing is 68 bytes, WBT is 128 bytes, and the independent communication for event notification is 32 bytes. So the total is 128 + 68 + 32 + 6 x 2 = 240 bytes. For the child, the total is 16 + 14 + 4 x 1 = 34 bytes. WBT is normally used on ports 4-7 for RAW communications, so WBT is 2 bytes smaller, or 238 bytes for the parent and 32 bytes for the child.

Typical Determinations	Example
Use the value calculated above for the parent's and child's transmission capacity. Verify that you have not exceeded the 512-byte limitation for transmission capacity.	`WMParentParam.parentMaxSize` is set to 240, and `childMaxSize` is set to 34. Neither value exceeds 512 bytes, so it is OK.
Use the expression in section 3.4.4 Transmission Capacity to calculate the required time for one MP sequence based on the parent and child transmission capacities and the maximum number of connected children. Check whether the result exceeds the 5600-microsecond limit; if this limit is exceeded, redesign the data sizes so that calculation result falls within the limit.	The calculation is: 96+(24+4+240+6+4)*4+(10+96+(24+34+4+4)*4+6)*3+10+96+(24+4+4)*4 The result is 2570 microseconds. Because this is below the 5600-microsecond limit, there is no problem. (This expression is easy to calculate if you use the "Wireless Communications Time Calculation Sheet" in the "Figures, Tables and Information" part of the *Function Reference Manual*.)
Based on the maximum number of connected children, and the parent and child transmission capacities, calculate the sizes of the Send and Receive buffers needed for MP communications.	For the parent, the size of the Receive buffer passed to the `WM_StartMP` function is `WM_SIZE_MP_PARENT_RECEIVE_BUFFER(34, 3, FALSE)`, and the Send buffer size is `WM_SIZE_MP_PARENT_SEND_BUFFER(240, FALSE)`. For the child, the Receive buffer size is `WM_SIZE_MP_CHILD_RECEIVE_BUFFER(240, FALSE)`, and the Send buffer size is `WM_SIZE_MP_CHILD_SEND_BUFFER(34, FALSE)`.
Determine the frequency of MP communications.	The data volumes sent in block transfer are not very large, so there should not be a problem with always performing MP communications at a frequency of once per picture frame. Set the `mpFreq` parameter passed to the function `WM_StartMP` to 1.
Determine the operations mode for data sharing, taking into consideration the frequency of MP communications and what the game frame fps will be.	The MP communications frequency is set to 1, and you want the game frame to move at a rate of 60 fps, so set `doubleMode` passed to the `WM_StartDataSharing` function to `TRUE`.

Q: I don’t know the value to set to `WMParentParam.tgid`.

A: Ideally, it should be a different value every time, even when the power is restored. An easy and convenient way to do this is to generate a pseudo-random number by combining return values of the `OS_GetVBlankCount` and `GX_GetVCount` functions. Also, by using the value for seconds or minutes on RTC, it is possible to guarantee this value to be different for some time even after power is restored. If a child reconnects to the parent multiple times, a secure connection may be achieved by sending some bits of TGID to the phase information of the parent to prevent a child from connecting to the parent with a different phase.

Q: When creating a list of parents from the scan result, a parent is sometimes difficult to find.

A: If all parents have the same beacon intervals and the beacon send timing happens immediately after the other parent’s beacon, that parent may be difficult to find. Also, processing overhead and the parent’s beacon interval matching the child’s scan interval can have an effect.

As a preventive measure, it is possible to achieve an overall resolution of this sort of problem by first using the WM_StartScanEx function, which can get multiple parent devices at one time. Use of the WM_StartScan function is no longer recommended.

You can also try to mix random numbers into the parent beacon interval and child scan interval.

The WM_GetDispersionBeaconPeriod and WM_GetDispersionScanPeriod functions were prepared for this purpose. Each of these functions returns random values that are about 200 ms and 30 ms, respectively. By setting a value of WM_GetDispersionBeaconPeriod to WMParentParam.beaconPeriod, the frequency of getting the same beacon intervals on the parents can be reduced. Set the beacon interval only once when starting the parent device. Changing the beacon interval dynamically affects the child device connection.

In the same way, variation in the child device scan timing can be achieved by resetting the maxChannelTime parameter to the WM_GetDispersionScanPeriod return value each time a child device calls the WM_StartScan or WM_StartScanEx function.

Q: The connection process with the WM_StartConnect* function is not stable.

A: Make sure that you did not forget to call WM_Reset when retrying a failed connection attempt. If the connection process has already made progress before failing, the internal state may be CLASS1, so WM_Reset must be used to restore the internal state to IDLE before WM_StartConnect is called again.

Also, to prevent unintentional connection to the child device after the parent stops accepting entries, you can call the WM_SetEntry function to disable the entry flag when the parent device stops accepting child device entries. To determine if the parent is accepting entries, the child device can check the WM_ATTR_FLAG_ENTRY bit of gameInfo.gameNameCount_attribute in the beacon before attempting to connect.

Q: When I end the communication once and reconnect to the same parent, the process sometimes fails.

A: When trying to reconnect after scanning, the timing of the child device reconnection process is too fast, and there are cases where an old beacon from the pre-shutdown parent device gets picked up. Before connecting, use the beacon information to check if the parent device has started a new connection. To figure out the parent device state from the beacon information, use a method such as including the parent device phase information in userGameInfo or checking for changes in TGID. After starting up in DS Download Play, the child device re-scans the parent device and makes a connection by checking the WM_ATTR_FLAG_MB of gameInfo attribute. This enables you to determine if the parent device is still in the DS Download Play mode.

If not rescanning, update TGID by the predetermined rule when reconnecting. Use some of the bits in TGID for the phase information of the parent and rewrite that section of WMParentParam and WMBssDesc on parent and child to reconnect. When doing so, the child cannot be reconnected if the parent accidentally changed the channel with the connection immediately before.

Q: My application has a parent device for DS Wireless Play rather than a parent device for DS Download Play. However, when I start it up, the `mb_child_simple.srl` that started up on another DS responds ("GameInfo Receiving…" keeps appearing).

A: Check if the `multiBootFlag` field of the `WMParentParam` structure specified by `WM_SetParentParam` is set to other than 0. To wait for the DS Download Play child device, make sure that `multiBootFlag` is not enabled on anything other than a parent that is sending out a beacon for DS Download Play.

3.11.3 General MP Communications

Q: How do I send out data with the shortest delay for MP communication?

A: With the frame synchronization communication mode, the first MP sequence start process is performed when the V-Count is 260 lines. When a child receives MP frame from the parent, the transmission data should already be set, so the transmission data setting process starts a little earlier at 240 lines. Therefore, to reduce the latency as much as possible, call the `WM_SetMPDataToPort*` function just before 260 lines for the parent and 240 lines for a child. However, immediate sending is not guaranteed due to possible unpredictable delays between the time the library function is called from ARM9 and from ARM7 wireless communication driver processing. In addition, if there is other data in send queue, that data is sent first.

Values of 260 and 240 lines can be reconfigured using the `WM_SetMPTiming` function.

In continuous communications mode and in frame synchronization mode, the parent starts the next MP sequence and the child sets the next group of Response data after an interval following the previous MP sequence. By calling the `WM_SetMPDataToPort*` function during this waiting period, you can get the Response data set in time for the next MP sequence.

Note: Depending on the state of the ARM7 Wireless Communications library, the Response data may not get set in time. The waiting period can be configured using the `WM_SetMPInterval` function.

Q: I received unpredictable results when continuously calling the `WM_SetMPDataToPort` function.

A: Did `WM_ERRCODE_FIFO_ERROR` get returned by the function? If there is an overflow in the FIFO used for sending commands from ARM9 to ARM7, this error will be returned. Try reducing the number and frequency of the calls so that the ARM7 processing can catch up.

Q: When I try to send large amounts of small data packets, the communications state degrades and things do not work well.

A: Is `WM_ERRCODE_FIFO_ERROR` being returned to some callback? When large amounts of small data packets are sent, the processing capacity of the child-side ARM7 is sometimes exceeded because the communication state degrades and remaining communications accumulate.

If `WM_ERRCODE_FIFO_ERROR` is returned to the callback in this way, there will be too many processes and the ARM7-side FIFO for internal processing will overflow. The communication state cannot generally be recovered from here, so try to immediately transition to the communications error screen to reset the communications. If heavy processing is being performed inside communication-related callback on the side of a child device, this problem will occur more frequently because ARM7 is waiting

for those processes to finish. Also, in comparison, there are times when the child device processes overflow if the parent device processing is too light. There are a variety of measures for avoiding this problem, such as reducing the processing load inside the child device-side callback, avoiding sending large amounts of small data packets, and using the WM_SetMPInterval function to increase the minimal send interval for the parent.

3.11.4 Data Sharing

Q: The WM_StepDataSharing function frequently returns WM_ERRCODE_NO_DATASET.

A: There are a few possibilities. If either the parent or child is always successful, the Step may fail because the device that continues to succeed experiences a performance slow down and the other device is waiting for the performance slow down. If WM_StepDataSharing is set to be called at every frame, and it always fails every other frame, check if doubleMode of the WM_StartDataSharing function is set to TRUE. If the WM_StepDataSharing function is set to be called every other frame and it fails on a regular basis, there may be a problem with the retry process if WM_ERRCODE_NO_DATASET was returned. Check to see if the next WM_StepDataSharing function is called in the frame immediately after the failure.

If it fails with parent and child randomly and at about the same frequency, the WM_StepDataSharing function may have been called using bad timing. Make sure to call the function at the earliest possible time immediately after V-blank. The WM_StepDataSharing function calls the WM_SetMPDataToPort function internally, but to perform data sharing with the least MP communication frequencies, it requires data to be on every MP sequence. Therefore, as explained in the previous item, data sharing may not be stable because of the communication timing if it is not 260 lines with the parent and 240 lines with a child. This is the same with Key Sharing because it performs data sharing internally.

Q: The code does not work properly when pausing with the WM_EndDataSharing function and restarting with the WM_StartDataSharing function.

A: The WM_EndDataSharing function is designed to be called as a series of processes for ending communication, and it may cause a problem if the termination during MP communications and restarting were performed in a row. If you want to interrupt Data Sharing, set a flag in the shared data in advance; once interrupt timing is determined on the parent and child, you can simply stop calling the WM_StepDataSharing function. Unless the WM_StepDataSharing function is called, excess processing time and communications related to Data Sharing will not be generated. Be aware that when restarting, the last data set before the interruption will still get through.

In the future, the API for pausing will be provided.

Q: Can the shared data size be changed by using the same port?

A: This feature is not currently available. Call the WM_StartDataSharing and WM_EndDataSharing functions once, for starting and ending communication, respectively, and use one port for data sharing of the same setting during the communication. Instead, the same process is achieved by performing two sets of data sharing with different shared sizes at different ports, and switching them. Precautions for switching are as shown above.

3.11.5 Others

Q: Sometimes the communication stops for unknown reasons.

A: There may be various causes such as the destruction of memory in the application. Check the following.

- Are you using a process that takes a while in the callback? Callback is in the interrupt handler so it may be in interrupt-prohibited state, and the wireless communication driver of ARM7 may be waiting for the callback to complete. It would cause negative effects in some areas, so avoid using processes that take some milliseconds.
- Are there multiple levels of nesting of function callbacks within a callback? Are you calling a function such as `OS_Printf` that consumes many stacks in a deeper level of nesting? Make sure to reduce the consumption of stacks because the IRQ stack used while the callback is executing is not very big. If it freezes during debug output, the situation may be improved by using the `OS_TPrintf` function instead of `OS_Printf`.
- Has the calling of the `WM_StartDataSharing` function been separated from the calling of the `WM_StartMP*` function in the child? These child functions must be called consecutively because of the current restrictions on the implementation.

Q: Is there anything that requires special attention when debugging the wireless communication portion?

A: First, make sure to allocate enough time for debugging wireless communication. It may seem to be working properly, but a problem that occurs once every few dozen times is very common.

For debugging, change to fixed channels by temporarily disabling the automatic channel selection feature and start multiple groups of parent and children on the same channel. By repeating this test, you may have a better chance of recreating the problem.

3.12 Important Notes for Recent Releases

Several important changes to the Wireless Communications library are explained below. Note that some changes are not obvious with compilation.

3.12.1 Changes in MP Frame Send Conditions (NITRO-SDK 2.2 PR and Later)

Previously, a parent in the `MP_PARENT` state sent MP frames regardless of the child's connection status. This has been changed so that nothing gets sent if a child is not connected. This eliminates occurrences of `MPEND` notifications when the number of connected children is 0. When using port send and receive callbacks, there is no change in behavior. MP frames may be sent immediately after a child is disconnected, even though the number of connected children becomes 0.

A restriction was added so that no more than 6 MP frames will be sent in one picture frame. Normally, this restriction will not be an issue during meaningful communication. For more information, see section 3.4.8 Restrictions on the Number of MP Communications per Picture Frame.

3.12.2 Addition of Notification to WM_SetIndCallback Function Callback (NITRO-SDK 3.0PR2 and Later)

WM_STATECODE_INFORMATION is now returned to the WM_SetIndCallback function callback. Its purpose is to provide notification of internal events. This type of event can be determined from WMIndCallback.reason, which is passed to the callback as an argument.

WM_INFOCODE_FATAL_ERROR is defined as the value placed in WMIndCallback.reason. This indicates that a fatal error occurred with the ignoreFatalError argument of the WM_StartMPEx function set to TRUE. In general, ignoreFatalError is set to FALSE, so there is no such notification.

3.12.3 Change in Null Response Condition (NITRO-SDK 3.0PR2 and Later)

Previously, if processing by the ARM7 for a child device in the MP_CHILD state was not fast enough, a null response was issued when the MP frame was received. However, no response is returned now. If no response is returned, no MP receive notification is generated internally by the child device. Although this somewhat decreases transmission efficiency, it may alleviate overloading of the child device.

In addition, because return of no response can be guaranteed if the child device is not in the MP_CHILD state, there are no longer problems that result from gaps between the calls to the WM_StartConnect and WM_StartMP functions. Be cautious, however, as much time between the calls will cause a disconnection due to lifetime expiration.

3.12.4 Addition of WM_STATECODE_DISCONNECT_FROM_MYSELF (NITRO-SDK 3.0RC and Later)

Until now, specifications did not include the generation of a disconnect notification when terminating one's own connection by explicitly calling the WM_DisconnectChildren, WM_Reset, WM_EndParent, or WM_Disconnect function. This was changed by adding WM_STATECODE_DISCONNECTED_FROM_MYSELF to WMStateCode so that such notifications can be made.

WM_STATECODE_DISCONNECTED_FROM_MYSELF has the same callback structure as WM_STATECODE_DISCONNECTED and is used for notifications to the callback of the WM_StartParent, WM_StartConnect, and WM_SetPortCallback functions.

This change increases the state codes that may be used to fill the state field of WMStartParentCallback, WMStartConnectCallback, or WMPortRecvCallback. Thus, care must be taken in a case where a program has been coded so that its execution is halted when anything other than an existing WM_STATE_CODE_* is received.

In addition, data sharing will no longer stop even when a child is explicitly disconnected from a parent using this notification.

3.12.5 Addition of WM_STATECODE_PORT_INIT (NITRO-SDK 3.0RC and Later)

WM_STATECODE_PORT_INIT was added to WMStateCode and specifications were changed so that a port-receive callback is called if this state code is in effect when WM_SetPortCallback is called. It is

designed to be used for initialization processing that uses the myAid and connectedAidBitmap fields of WMPortRecvCallback.

Note: When the WM_SetPortCallback function is called before connection, the value of 0 is stored in both the connectedAidBitmap and myAid fields.

To maintain consistency among connection notifications, do not perform too much processing, as calls made under WM_STATECODE_PORT_INIT are made while interrupts are disabled.

3.12.6 Changed Conditions for Issuing a NULL Response (NITRO-SDK 3.1 PR and Later)

NITRO-SDK 3.0 PR 2 was changed so that child devices in the MP_CHILD state would not return a NULL response when receiving an MP frame if ARM7 processing did not complete in time. However, unstable behavior would occur on rare occasions in some environments, so this change was cancelled and specifications reverted to returning a NULL response.

Specifications and processing for parent devices were changed to match this: If a child returns a NULL response, it will be treated as if the response failed. As a result, a child can only send successfully from the MP_CHILD state, so there are no issues with an interval between calls to the WM_StartConnect and WM_StartMP functions.

Functionality from the WM_SetInterval function was added as well, so there will be no problem with overloaded child processing even if a NULL response is returned.