ワイヤレス通信ライブラリ解説

Ver 1.1.8

任天堂株式会社発行

目次

改訂履歴

版	改訂日	改 訂 内 容	担当者
1.1.8	2007-10-16	・「3.1.6 WiiとのMP通信」を追加 ・使用チャンネルに関する記述の追記 ・V カウントの変動範囲に関する記述を変更 ・WM_StartScan 関数が非推奨であるという記述の追加 ・キャンセルされた仕様変更予定に関する記述の削除	清木
1.1.7	2007-02-20	・WM_ERRCODE_OVER_MAX_ENTRY の発生条件に関する説明の追記	清木
1.1.6	2006-02-20	・「3.1.3 ライブラリの内部状態」に追記 ・「3.3.4 GameInfo」を追加	清木
1.1.5	2006-01-13	・NITRO-SDK3.0 のリリースに向け、古い記述の削除と分かりにくい記述の整理	清木
1.1.4	2005-12-20	・「3.3.2 使用チャンネルの選択」に通信不能時の動作に関する言及を明記 ・「3.3.5 通信終了時の注意」を追加 ・「3.4.5 MP 通信用送受信バッファ」を追加 ・「3.10 より高度な通信制御」節を追加し、関連する項目を移動 ・FAQ に通信パラメータの決定手順の例を追加	清木
1.1.3	2005-12-06	・V ブランク同期に関する記述を追記し、独立した節へ変更 ・用語の変更:「最大送信バイト数」→「送信容量」 ・「3.4.4 送信容量」節の移動 ・「3.4.5 送信容量の動的変更」の追加 ・「3.4.9 MP 通信のタイミング制御パラメータ」の追加	清木
1.1.2	2005-11-04	・目次の更新	清木
1.1.1	2005-11-01	・WMStartParent，WMStartConnect，WMSetPortCallback の各関数のコールバックに WM_STATECODE_DISCONNECT_FROM_MYSELF の通知が追加されたことを追記 ・WMSetPortCallback 関数のコールバックに WM_STATECODE_PORT_INIT の通知が追加されたことと、WMPortRecvCallback 構造体に connectedAidBitmap フィールドが追加されたことを追記	清木
1.1.0	2005-07-29	・WMIndCallback 関数のコールバックに WM_STATECODE_INFORMATION の通知が追加されたことを追記 ・Null 応答の発生条件を変更したことによる各所の記述の修正	清木
1.0.5	2005-07-12	・「ワイヤレス通信ライブラリから返されるエラーコード一覧」で WM_Initialize 関数の返り値を修正 ・「3.4.3 通信失敗時の動作」に失敗時の MP の通知についての記述を追加 ・「3.11.12 MP フレーム送信条件の変更」に変更点の記述を追加	清木
1.0.4	2005-06-07	・Key Sharing に関する各記述に今後廃止予定であることを追記 ・「ワイヤレス通信ライブラリから返されるエラーコード一覧」で WM_StartKeySharing 関数と WM_EndKeySharing 関数を変更 ・「3.4.5 ピクチャーフレームあたりの MP 通信回数制限」を追加	清木
1.0.3	2005-03-29	「3.1.5 非同期関数のコールバックと非同期通知」に連続呼出に関する注意を追加	清木
1.0.2	2005-03-22	「3.1.3 ワイヤレスライブラリの内部状態」に CLASS1 ステートに関する記述を追加 「3.10.2 接続状態」に CLASS1 ステートに関する記述を追加	清木
1.0.1	2005-03-04	・「ワイヤレス通信ライブラリから返されるエラーコード一覧」を変更 ・送信キューの段数を 64 から 32 に変更 ・「ワイヤレス通信ライブラリから通知されるイベント一覧」の WM_SetMPData 関数に	清木

		関する項目において、WMPortSendCallbak 構造体の restBitmap フィールドに関する説明の変更、及び sentBitmap フィールドに関する説明の追加 ・「過去のリリースからの注意すべき変更点」へ項目の追加	
1.0.0	2005-02-18	初版	清木

1 ワイヤレス通信ライブラリの概要

1.1 はじめに

NITRO-SDK ではワイヤレス通信のための一連の関数が用意されています。このドキュメントでは、ワイヤレス通信ライブラリの基本的な使用方法について解説します。

1.2 ワイヤレス通信ハードウェアの基本仕様

ニンテンドーDS （以下 DS）に搭載されているワイヤレス通信ハードウェアの基本仕様は以下のとおりです。

使用通信帯	2.4GHz 帯 （電子レンジや他の 2.4GHz 帯を利用している無線機器の影響を受けます）
通信規格	IEEE802.11 相当（インフラストラクチャーモード） 任天堂独自プロトコル（DS ワイヤレスプレイモード） 任天堂独自プロトコル（DS ダウンロードプレイモード）
通信速度	1Mbps または 2Mbps
通信到達距離	10～30m（周囲の環境や本体の向きにより大きく変動することがあります）
備考	ゲームボーイアドバンス専用ワイヤレスアダプタとの通信はできません。

1.3 ワイヤレス通信ライブラリの構成

NITRO-SDK のワイヤレス通信に関するライブラリは ARM9 と ARM7 の双方に分かれて存在しています。ARM7 側コンポーネントがワイヤレス通信ハードウェアの制御を担当し、ARM9 側ライブラリはアプリケーションからの要求を ARM7 側に伝える役割を担っています。アプリケーションを作成する過程では基本的には ARM7 側のコンポーネントを考慮する必要はありません。しかし、キャッシュ関係の処理で気をつけなければならない部分もありますので、ワイヤレス通信ライブラリとデータをやり取りする際には、リファレンスマニュアルの指示に注意してください。

ワイヤレス通信ライブラリが提供する通信モードは大きく分けて以下の3つとなります。

DS ワイヤレスプレイモード 1 台の DS を親機とし、最大 15 台までの DS を子機としてワイヤレス通信で接続して通信を行うモード。

DS ダウンロードプレイモード 別名ワイヤレスマルチブート。カードを持たない子機を、親機からプログラムとデータをワイヤレス通信でダウンロードして起動させるモード。

インフラストラクチャーモード IEEE802.11b/g 規格に対応した無線アクセスポイントを通じて、インターネットに接続して通信を行うモード。

このドキュメントでは DS ワイヤレスプレイモードを中心に解説します。DS ダウンロードプレイモードについての詳細は「DS ダウンロードプレイ解説 (AboutMultiBoot.pdf)」を参照してください。

2 用語集

ワイヤレス通信ライブラリで用いられる用語について説明します。

親機 DS ワイヤレスプレイモードにおいて、全ての通信の管理を行う DS。

子機 DS ワイヤレスプレイモードにおいて、親機に接続して通信を行う残りの DS。

AID Association ID（接続識別子）の略。親機は 0 で固定。子機は接続時に 1～15 までの番号が割り振られる。最大接続子機数を n に設定した場合、必ず 1～n の範囲で割り振られる。

MP シーケンス Multi-Poll シーケンス。802.11 上の任天堂の独自拡張プロトコルで、低レイテンシのワイヤレス通信を実現する。詳細は「3.4 **MPプロトコル仕様**」を参照。

MP 通信 MP シーケンスによる通信の総称。
場合により、MP シーケンス 1 回分の通信を指す。

MP フレーム MP シーケンスの最初に、親機が子機にブロードキャストするフレームのこと。

Key 応答フレーム MP フレームを受信した子機が親機に応答するフレームの一種。

Null 応答フレーム MP フレームを受信した子機が親機に応答するフレームの一種。応答データのセットが間に合わなかった場合に送信される。

MP_ACK フレーム MP シーケンスの最後に、親機が子機にブロードキャストするフレームのこと。

ペイロード MP フレームおよび Key 応答フレームの中にある、データを送信するためのエリアのこと。

PollBitmap 16 ビットの各ビットを子機の AID に対応させたビットマップ。MP フレームにおいては返答してほしい子機に対応するビットが立ち、MP_ACK フレームにおいては親機が受信できなかった子機のビットが立っている。

indication ワイヤレス通信ハードウェアがデータ受信などのイベントをきっかけに自律的にアプリケーションに行う通知のこと。アプリケーションからの要求に対する応答とは区別される。

ピクチャーフレーム V ブランク割り込みから次の V ブランク割り込みまでの 1/60 秒。

ゲームフレーム ゲーム処理の単位となる期間。秒間 30 フレームのゲームであれば 1/30 秒。

フレーム同期通信モード ピクチャーフレームに同期して MP 通信を行う通信モードのこと。連続通信モードと対比して用いられる。1 ピクチャーフレームに n 回通信する、という指定を行う。ただし、電波の状態が悪く、再送が発生している場合はフレームに同期せずに送信できる限り再送を行う。

連続通信モード MP シーケンスを連続で続ける通信モード。フレーム同期通信モードと対比して用いられる。大きなデータを通信する際に用いると効果的だが、電力消費が大きく常用には向かない。MP シーケンスが連続して駆動される点が異なるのみで、全体的な制御はフレーム同期通信モードとほとんど変わらないため、通常は同じプログラムで通信モ

	ードのみ切り替えることが可能である。
port	ワイヤレス通信ライブラリが通信路の多重化のために用いる概念。0～15 の整数で指定して送信すると、受信側で番号に対応するコールバックが呼び出される。TCP/IP における port よりもずっと低水準な抽象化であることに注意。
パケット	データにヘッダとフッタがついた、通信の一単位。port 番号・パケットのサイズなどの他に、必要に応じて送信先情報・シーケンス番号などの情報も含まれる。実際の通信では 1 回の MP シーケンスでペイロード中に複数のパケットがパックされて送信される。
Sequential 通信	通信の確実性を保証する MP 通信の上位レイヤ。ワイヤレス通信ライブラリがシーケンス番号を用いて冗長なパケットの除去と到達保証を行う。8～15 番の port を使用するとこのモードで通信が行われる。
Raw 通信	Sequential 通信と比して、何も追加の制御を行わない通信方法。データが到達しない可能性と、同じデータが複数回届く可能性がある。0～7 番の port を使用するとこのモードで通信が行われる。
Key Sharing	キー入力データの共有機能。できるだけ意識せずにワイヤレス通信を利用できるよう用意されている。今後、廃止予定である。代わりに Data Sharing の使用を推奨。
Data Sharing	キー入力だけではなく、ユーザ定義サイズのデータを共有する機能。
Block 転送	ひとまとまりのデータを親機から複数子機へ一度にまとめて送信することに特化した機能。詳細は WBT_* のリファレンスを参照。
MAC アドレス	ワイヤレス通信ハードウェアの識別番号。DS 一台ごとに異なる。サイズは 6 バイト。
セッション	1 回の WM_StartParent から WM_EndParent までの期間。
beacon	MP シーケンスとは別に、定期的に親機が発している無線信号。子機は接続を確立しなくても beacon を受信することができる。新規に接続する子機は、beacon に含まれている GameInfo を元に親機を選択する。通常は数百 ms 間隔で発信される。
BSS	Basic Service Set の略。一つのサービスをやり取りする集合を指す。DS ワイヤレスプレイモードの場合は、親機とそれに接続している子機のグループのこと。
BSSID	Basic Service Set ID。BSS を識別するための ID で、DS ワイヤレスプレイモードの場合は親機の MAC アドレスをそのまま BSSID として用いる。子機が親機と接続する時に接続先を指定するために使用。
GameInfo	親機が提供するゲームの種類や接続に必要な情報を収めたデータ構造体。GGID, TGID, 親子の最大送信容量などを含み、その他にユーザ定義の情報を含むことができる。DS ダウンロードプレイでは、ゲーム名・アイコン情報なども GameInfo に含まれる。
GGID	Game Group ID。ゲームタイトルやシリーズタイトルごとに異なる 4 バイトの ID。任天堂から割り当てられる。接続時に使用する。
TGID	Temporary Group ID。2 バイトの ID で、新しい対戦プレイを始めるなどして新規のセ

ッションを開始するごとに新規の値を設定する。同じ DS が続けて親機になる場合、BSSID も GGID も同一になってしまうため、TGID で新旧の通信を区別する。

SSID 親機が接続してきた子機を受け入れる際に比較する情報。子機は GameInfo から得た GGID と TGID を用いて SSID を構築し、親機は接続してきた SSID を見て自分と GGID と TGID が一致した子機のみを受け入れる。マッチングに用いられない後半のユーザ領域は、子機から親機への接続時の通信として使用可能。

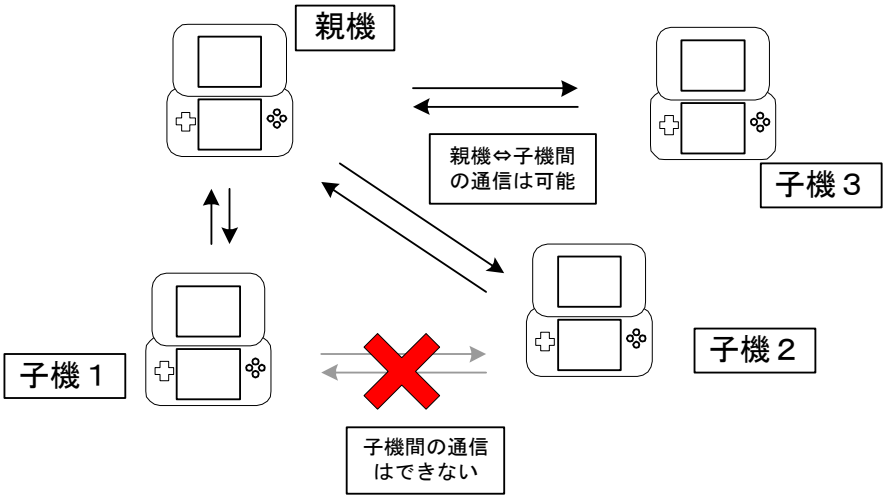
チャンネル 周波数帯を分けたもの。DS のワイヤレス通信ハードウェアでは 1～14ch まで使用可能であるが、各国の法規制により実際に使用可能なチャンネルは制限される。また、隣接するチャンネルは相互に干渉し合うため、実際には 5 つ程度間隔をあけて使用したほうがよい。

IEEE 802.11 規格 IEEE によるワイヤレス通信の規格の一つ。最大 2Mbps でのワイヤレス通信の方法を規定する。同じファミリーで最大通信速度 11Mbps の IEEE 802.11b が PC 用ワイヤレス通信では一番普及しているが、IEEE 802.11 はこの IEEE 802.11b と下位互換性を持つ。

3 DS ワイヤレスプレイ

3.1 全般

3.1.1 接続形態



DS ワイヤレスプレイにおいて、ネットワーク構成はスター型になります。常に親機⇔子機の通信に限定され、子機間の通信はできません。その代わりに、親機から複数の子機に対して同時にデータを送信することが可能です。

3.1.2 DS ワイヤレスプレイの特徴

□ **低レイテンシである**

通信が正常に行われた場合、ピクチャーフレームの頭で送信用にセットしたデータは、ピクチャーフレームの終わりには通信相手のアプリケーションが受け取っています。

□ **1 ピクチャーフレームの中の特定のタイミングでデータが転送される**

親子が好きなタイミングでデータを送信するというイメージでなく、WM_SetMPDataToPort 関数で送信データをセットしておくと、通信が発生したタイミングで親子の送信データが一定サイズ分だけ交換される、というイメージで捕らえてください。

□ **子機数が多くなるほど、子機からの送信可能なサイズが少なくなる**

プログラミングガイドラインで 1 回のMPシーケンスで使用可能な最大の通信時間が定められているため、子機数が多くなるほど子機の最大送信サイズは少なくなります。

子機が 1 台のみの場合は、親子共にワイヤレス通信ライブラリの最大送信サイズである 512 バイトを送信できますが、子機が 15 台繋がった場合は、親機が 256 バイト送ろうと思うと、子機は各 8 バイトしか送信できません。

詳細は、「3.4.4 送信容量」を参照してください。

□ **親機からはブロードキャストしたほうが効率がよい**

WM_SetMPDataToPort 関数では特定の子機を選んでデータを送信することもできますが、MP 通信の性格上、複数の子機にブロードキャストしても無線チャンネルの占有時間は変わりません。(特殊な通信モードを使用している場合や、再送が発生した場合は除きます)

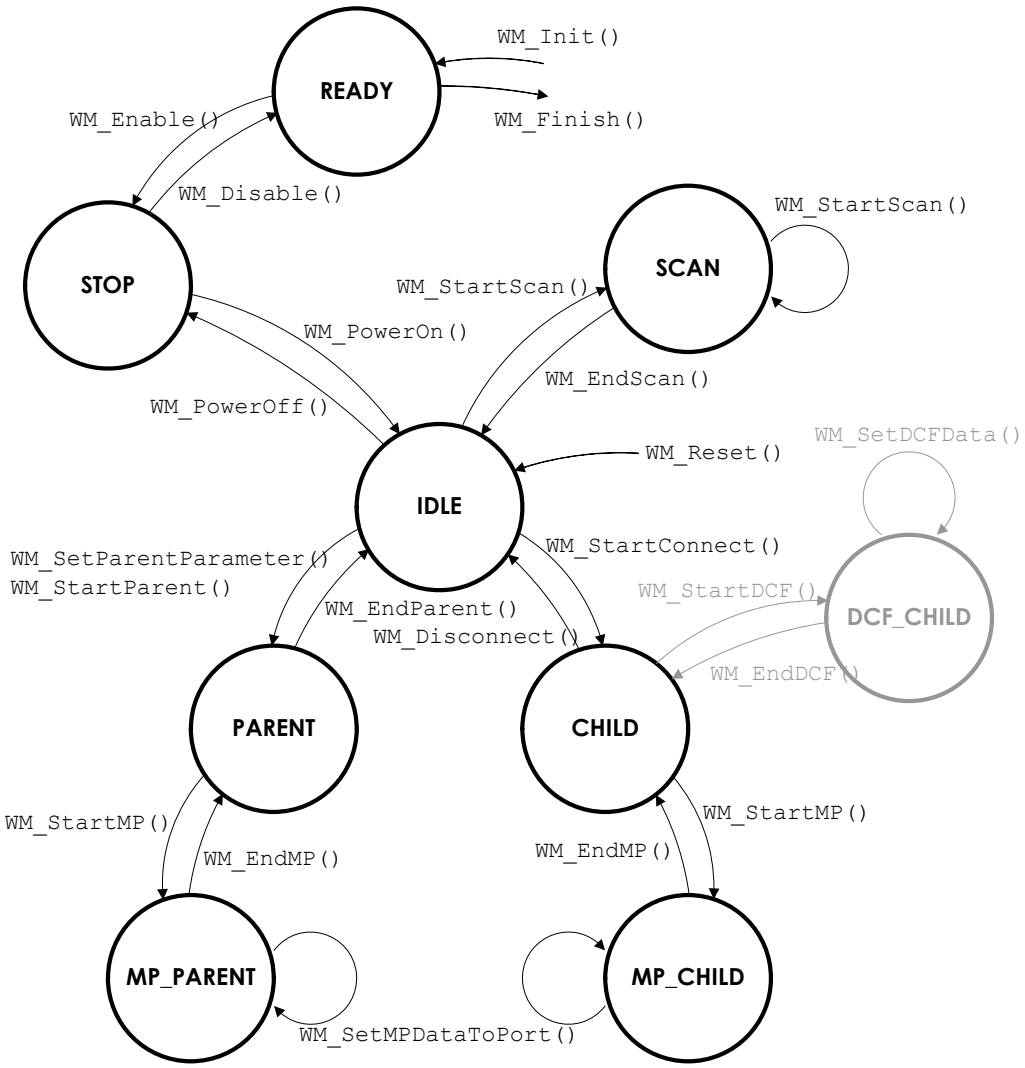
□ **通信量一定のほうが効率がよい**

子機からの通信に関しては、1 回の MP シーケンスで常に子機送信容量(子機が 1 回の MP 通信で送信可能な最大サイズ)分の時間だけ無線チャンネルを占有しますので、固定長通信と考えたほうが効率よく電波を利用

できます。なお、子機送信容量の初期値は接続時に親機が指定します。
現在、送信キューの溜まり具合に適応して通信回数を増やすようなロジックは実装されていません。アプリケーションが通信頻度を動的に変更することは可能ですが、通信量があまり変動しないような通信設計をすることをお勧めします。

3.1.3 ライブラリの内部状態

ワイヤレス通信ライブラリは下図のような内部状態を持っており、状態によって呼び出すことのできる関数に制限があります。DS の起動後に WM_Init 関数を呼び出すことにより、READY ステートに遷移します。各関数がどのステートで呼び出しが可能なのかはリファレンスの各関数の項目を参照してください。



なお、上図の DCF_CHILD ステートには、DS ワイヤレスプレイにおいては遷移できません。

また、厳密には IDLE ステートと CHILD ステートの間と、IDLE ステートと SCAN ステートの間には、CLASS1 というステートが存在しています。IDLE ステートから CHILD または SCAN ステートに移行する途中に、一時的にこのステートになりますが、ステートを移行させた WM の非同期関数が正常に完了し、コールバック通知が来た時点では必ず CLASS1 ステート以外の状態になっていますので、通常は意識する必要はありません。ただし、WM_StartConnect 関数で接続する途中の特定の段階で失敗した場合や、接続中に子機が親機から切断された場合などに、CLASS1 ステートに遷移したままになります。CLASS1 ステートで実行可能な関数は WM_Reset 関数だけですので、WM_StartConnect 関数に失敗した場合や、子機が切断通知を受け取った場合は、次の操作に移る前に WM_Reset 関数で IDLE ステートに遷移するようにしてください。

3.1.4 エラーコード

ワイヤレス通信ライブラリの関数はいくつかの例外を除き、エラーコードとして WMErrCode 列挙体を返します。

基本的には、正常に動作しているときには同期関数は WM_ERRCODE_SUCCESS を、非同期関数は WM_ERRCODE_OPERATING を返します。

詳細はリファレンスの「WMErrCode」、及びこのドキュメントの「3.8 ワイヤレス通信ライブラリから返されるエラーコード一覧」を参照してください。

3.1.5 非同期関数のコールバックと非同期通知

ワイヤレス通信ライブラリは ARM7 側のドライバに指示を送らなければならない関係上、多くが非同期関数となっています。非同期関数は WMCallbackFunc 型の callback という引数をとり、非同期処理が終了するとその callback を呼び出します。

非同期関数を呼び出し、その返り値が WM_ERRCODE_OPERATING であった場合は、必ず完了コールバックが呼び出されます。

また、通信という性質上、非同期で発生する通知も数多くあります。これらはコールバック関数の呼び出しとして通知されます。主な通知の種類とコールバックの設定関数の対応は以下のとおりです。

通知の種類	通知先のコールバックを設定する関数
接続・切断などの通知	WM_StartParent, WM_StartConnect*
MP シーケンスに関する通知	WM_StartMP*
port への受信	WM_SetPortCallback
その他の通知	WM_SetIndCallback

ワイヤレス通信ライブラリにおけるコールバック関数の型は WMCallbackFunc 型として定義されています。WMCallbackFunc 型の関数は、唯一の引数 WMCallback* arg を取りますが、いくつかの関数では、その関数独自の構造体(例:WMPortRecvCallback)を渡しますので、必要に応じてその型へキャストしてから使用してください。各関数が返すコールバック引数の型は、$NITROSDK_ROOT¥man¥ja_JP¥wm¥wm¥WMCallbackFunc.html に記されています。

いくつかの関数のコールバック引数の構造体には、stateというフィールドが定義されている場合があります。このstateフィールドは、errcodeフィールドだけでは表しきれない通知の種類を表現するために用いられます。詳細は「3.7 ワイヤレス通信ライブラリから通知されるイベント一覧」を参照してください。

ワイヤレス通信ライブラリの非同期関数では、一部の例外を除いて、関数ごとにコールバックを登録します。同一の関数に対して同時に別のコールバックを設定すると、あとで設定したものが有効になりますので注意してください。例外はWM_SetMPData*関数で、これに関しては呼出し毎にコールバックを別に記憶していますので、毎回異なるコールバックを設定して連続で呼び出しても問題はありません。

また、ワイヤレス通信ライブラリの内部状態を遷移させる非同期関数を多重で呼び出すことは避けるようにしてください。再現の難しい不具合の原因になります。

3.1.6 Wii との MP 通信

DS は Wii と MP 通信を行うことができます。DS 側のプログラムは基本的には DS 同士の MP 通信と同様の実装で Wii と通信することが可能です。ただし、Wii 側は親機にしかなれないなどの固有の制限があります。Wii との通信に関する制限の詳細は RevolutionSDK Extensions (RevoEX) に含まれる MP ライブラリのリファレンスを参照してください。

3.2 ワイヤレス通信ライブラリの初期化

3.2.1 各初期化・終了関数の相違

ワイヤレス通信ライブラリの初期化には、WM_Init, WM_Enable, WM_PowerOn の 3 関数を順番に呼ぶ方法と、WM_Initialize 関数だけを呼び出す方法の 2 通りの手順があります。終了処理も同様に、WM_PowerOff, WM_Disable, WM_Finish の 3 関数を順番に呼ぶ方法と、WM_End 関数だけを呼び出す方法があります。

WM_Initialize 関数は WM_Init, WM_Enable, WM_PowerOn の 3 つの関数を順番に呼び出すことと同等の処理を行い、WM_End 関数は WM_PowerOff, WM_Disable, WM_Finish の 3 関数を順番に呼び出すことと同等の処理を行います。

初期化処理	WM_Initialize	WM_Init	ワイヤレス通信ライブラリが用いるバッファの割り当て
		WM_Enable	ワイヤレス通信ハードウェアの使用可能状態への移行 （電源ランプの変速点滅が開始する）
		WM_PowerOn	ワイヤレス通信ハードウェアへの電力供給の開始 （電力消費が上がる）
終了処理	WM_End	WM_PowerOff	ワイヤレス通信ハードウェアへの電力供給の停止 （電力消費が下がる）
		WM_Disable	ワイヤレス通信ハードウェアの使用禁止状態への移行 （電源ランプの変速点滅が停止する）
		WM_Finish	ワイヤレス通信ライブラリが用いるバッファの開放

3.2.2 DS ワイヤレス通信 ON 状態

DS ワイヤレス通信 ON 状態は WM_Enable 関数を呼び出してから WM_Disable 関数を呼び出すまでの間として定義されています。DS ワイヤレス通信 ON 状態に移行するにはユーザへの確認を必要とするなどの制限があります。詳しくは DS プログラミングガイドラインを参照してください。

3.2.3 ワイヤレス通信ライブラリ用バッファ

ワイヤレス通信ライブラリは、WM_Init 関数が呼び出されてから WM_Finish 関数が呼ばれるまでの間、ライブラリ内部で使用するバッファを保持します。メインメモリ上から、32 バイト境界に合った WM_SYSTEM_BUF_SIZE バイトの領域を WM_Init 関数の引数に渡してください。

3.3 親子の接続

3.3.1 接続までの流れ

接続までの大まかな流れは以下の通りです。

親機	子機
1. ワイヤレス通信ハードウェアを初期化。	1. ワイヤレス通信ハードウェアを初期化。
2. 接続情報として GGID, TGID などを親機情報に設定する。	
3. 各無線チャンネルの混雑度を計測して、使用するチャンネルを選ぶ。	
4. 指定したチャンネルに beacon を発信する。beacon 中の GameInfo には GGID, TGID と、接続受付中のフラグが含まれている。	2. アプリケーションが使用する可能性のある全てのチャンネルで beacon を scan し、親機の GameInfo を得る。
	3. GameInfo に含まれている情報を元に、ユーザに親機一覧を提示し、接続先を選択させる。
	4. GameInfo 中の GGID, TGID から SSID を構築し、BSSID(親機の MAC アドレス)と SSID を元に親機に接続する。
5. 接続してきた子機の SSID と自分の GGID, TGID を比較し、合致すれば受け入れる。	
6. 子機に AID を割り振り、接続完了。	5. 親機から割り振られた AID を受け取り、接続完了。

3.3.2 使用チャンネルの選択

802.11 の規格上は 1～14ch までが存在しますが、国や地域によって法規制により使用できるチャンネルに制限がかかります。また、使用可能であっても、隣接するチャンネルは相互に干渉するため、使用するチャンネルはできるだけ離す必要があります。

DS は使用可能なチャンネルを内部で保持しており、その中で十分に離れたチャンネル群を提示する関数 WM_GetAllowedChannel 関数が用意されています。アプリケーションではこの関数で得られたチャンネルの中から使用するチャンネルを選択しなければなりません。提示された中のどのチャンネルを選択するかは各アプリケーションに任せられていますが、できるだけ電波使用率の低いチャンネルを選択するようにしてください。なお、特定のチャンネルの電波使用率は WM_MeasureChannel 関数で取得可能です。

2007 年 10 月現在、DS では全世界統一仕様で 1～13ch を使用します。しかし、この仕様は今後変更になる可能性がありますし、WM が内部的に特定のチャンネルを使用禁止にする場合がありますので、現在の許可チャンネルを前提としてプログラミングをせず、必ず WM_GetAllowedChannel 関数の結果を使用してください。特に WM_GetAllowedChannel 関数が 0 を返した場合は、ワイヤレス通信が使用不能であることを示しますので、通信を開始しないようにしてください。

また、親機が Wii である場合、親機側の使用可能チャンネルが DS の使用可能チャンネルよりも制限されている国や地域があります。その場合、子機の DS 側で WM_GetAllowedChannel 関数により得られるチャンネル群のうち、一部のチャンネルしか親機が使用できない場合があります。そういった場合でも、DS 側では判別できませんので、DS 同士の接続と同様に親機をスキャンするという実装を行ってください。

3.3.3 beacon 情報

親機は PARENT か MP_PARENT ステートである間、定期的(WMParentParam.beaconPeriod[ms]間隔)に beacon と呼ばれる信号を発信しています。親機に接続したい子機は WM_StartScanEx 関数で beacon を取得し、これに含まれる WMBssDesc 構造体をそのまま WM_StartConnect*関数の引数に渡すことで親機に接続することが可能になります。
WMBssDesc には様々なフィールドがありますが、子機が親機に接続する際に重要となるのはそのうち WMBssDesc.bssid と WMBssDesc.gameInfo.ggid および tgid の 3 つの情報です。

BSSID は BSS を識別する ID であり、DS ワイヤレスプレイでは親機の MAC アドレスがそのまま用いられます。残りの GGID, TGID は親機が提供するサービスの内容を識別するために用いられます。GGID はゲームごと、ないしはシリーズごと(同シリーズ間で通信が可能な場合)に任天堂から割り当てられる ID です。子機は、scan 結果の WMBssDesc.gameInfo.ggid を調べ、自分が接続できる親機かを確認することができます(アプリケーション側で確認用コードを記述する必要があります)。一方、TGID は新しいセッションごとに親機が新しい値を割り振るもので、古いセッション向けの接続が誤って繋がらないようにするためのものです。

3.3.4 GameInfo

beacon の中には GameInfo 情報を格納する WMGameInfo 構造体が含まれています。前述のとおり、GameInfo には GGID と TGID が記述されており、他にはエントリー受付状態などの親機の属性や、親機と子機の送信容量の情報が書かれています。また、userGameInfo と呼ばれる領域もあり、ここにはアプリケーションが任意のデータを設定できます。

WMGameInfo 構造体には magicNumber フィールドと ver フィールドが存在しています。magicNumber フィールドは WM_GAMEINFO_MAGIC_NUMBER (=0x0001) で固定されています。GameInfo のほかのフィールドにアクセスする前に、magicNumber フィールドが正しいかどうかを確認してください。WM_IsValidGameInfo 関数と WM_IsValidGameBeacon 関数は内部的にこの確認を行います。また、ver フィールドは GameInfo の構造体バージョンを表します。ただし、GameInfo 構造体は、magicNumber が等しい限り下位互換を保つように拡張されますので、ver フィールドが未知の値であった場合は現在のバージョンと同等の機能を持つものとして扱うようにしてください。

userGameInfo 領域のサイズの上限は WM_SIZE_USER_GAMEINFO (=112) バイトです。userGameInfo の使用用途としては、親機が現在のステージ情報や参加者情報を接続先選択中の子機に通知することなどを想定しています。なお、userGameInfo を参照する前には、必ず ggid フィールドを確認して既知の親機かどうかを判定してください。

親機が beacon に乗せる GameInfo の初期値は WM_SetParentParam 関数で指定します。また、PARENT 状態になった後で beacon に乗せる GameInfo の内容を変える場合は、WM_SetGameInfo 関数を使用します。

フィールドの詳細は WMGameInfo 構造体のリファレンスを参照してください。

3.3.5 接続時の動作

ライブラリ内部では、子機が親機に接続する際には BSSID と SSID を用います。BSSID は前節で述べた親機の MAC アドレスです。SSID は全部で 32 バイトですが、DS ワイヤレスプレイモードでは、そのうちの最初の 4 バイトを GGID、続く 2 バイトを TGID の格納に使用し、これらに 2 バイトの予約領域を加えた 8 バイトをサービスの識別に用います。親機は接続してきた子機が申告する SSID の最初の 8 バイトを自分の GGID や TGID と比較して、その子機が正しい相手かを確認します。この時に合致しない子機は接続処理の初期段階で自動的に拒否されます。なお、この確認中には接続が張られている扱いになりますので、一時的に接続子機数が上限に達し、新規に接続しようとした子機に WM_ERRCODE_OVER_MAX_ENTRY のエラーが返ることがあります。

SSID の前半 8 バイトは WM_StartConnect*関数内部で WMBssDesc.gameInfo.ggid, tgid の内容を元に、ライブラリが自動的にセットしますので、アプリケーションプログラマは意識する必要はありません。一方、SSID のうち、サービ

スの識別に用いられない後半 24 バイトはアプリケーションに開放されており、WM_StartConnect*関数の引数 ssid として、自由なユーザデータをセットして親機に送ることが可能です。子機が SSID の後半 24 バイトにセットしたデータを、親機は WM_StartParent 関数のコールバック関数へ WM_STATECODE_CONNECTED の通知を受けた際に、WMStartParentCallback.ssid として受け取ります。

3.3.6 通信終了時の注意

切断は親子のどちらから行うことも可能ですが、親子が同時に切断処理を行うことはできるだけ避けるようにしてください。同時に切断を行った場合はどちらかの切断処理は失敗し、場合によっては結果が返ってくるまで時間がかかることがあります。

また、通信終了処理に入ると WM の各関数は状態によってエラーを返すようになります。しかし、このエラーを元に通信エラーによる異常終了の処理を開始してしまうと終了処理の無限ループにはまる恐れがあります。エラーが発生した場合は WM_Reset 関数を一度だけ呼び出すようにし、エラー処理が無限に連鎖しないように注意してください。

3.4 MP プロトコル仕様

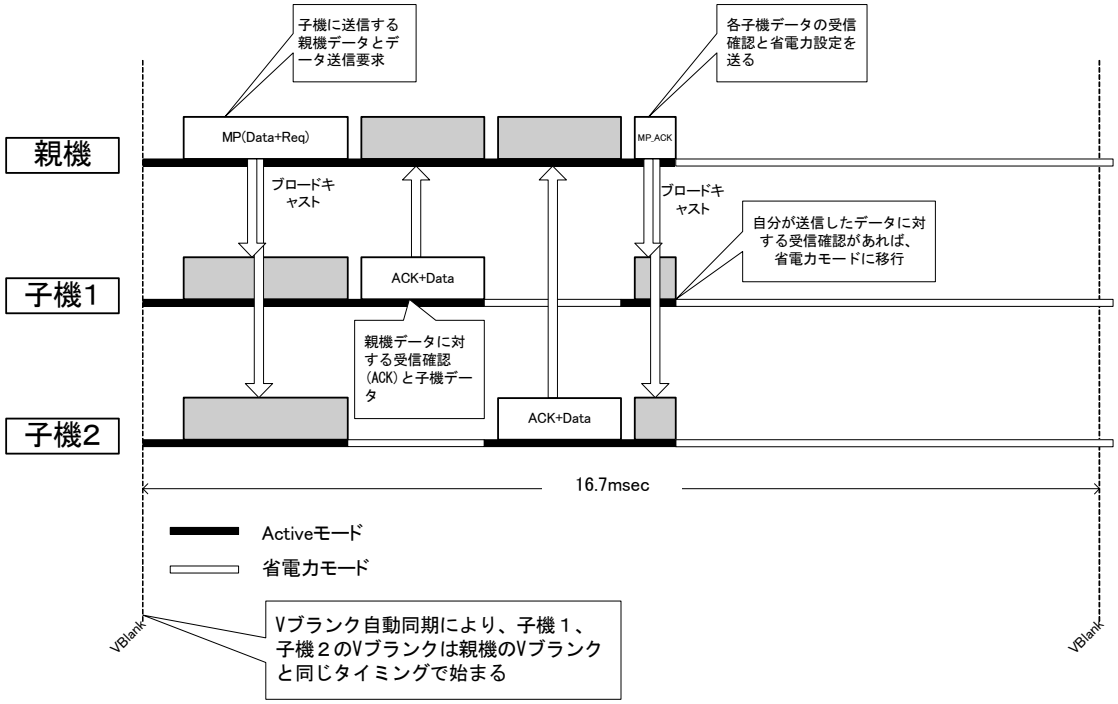
3.4.1 通信の概要

親子で接続が確立したら、実際にデータを送受信できます。毎ピクチャーフレームで以下のような手順で通信が行われます。

親機	子機
1. WM_SetMPDataToPort 関数で送信データをセットする。	1. WM_SetMPDataToPort 関数で送信データをセットする。
2. 毎ピクチャーフレーム中の特定のタイミングで自動的に MP 通信が行われる。	
3. WM_SetPortCallback 関数に指定したコールバック関数に子機からのデータの受信通知が来る。	3. WM_SetPortCallback 関数に指定したコールバック関数に親機からのデータの受信通知が来る。
4. 1. の WM_SetMPDataToPort 関数で指定したコールバック関数に送信成功通知が来る。	4. 1. の WM_SetMPDataToPort 関数で指定したコールバック関数に送信成功通知が来る。

3.4.2 MP 通信の動作

この節は必要に応じてお読みください。MP プロトコルの詳細を理解しなくてもワイヤレス通信ライブラリは使用できます。DS ワイヤレスプレイモードを実現する MP 通信の動作は下図のようになります。



MP 通信の特徴は、親機が送信タイミングを完全に制御するプロトコルであることです。

1. 親機はまず全ての子機に対し、MP フレームをブロードキャストします。
   MP フレームは親機から子機へ送信するデータの他に、同時にどの子機が応答して欲しいか(PollBitmap)と子機に何バイト分の送信時間を与えるか(TXOP)といった制御データを含みます。
   MP フレーム中のこれらの情報で、その回の MP シーケンス全体の時間配分が確定します。
2. MP フレームを受け取った子機は、PollBitmap と TXOP を見て、自分の返信タイミングを待ち、Key 応答フレームを親機へ送信します。

Key 応答は親機からのデータの受信確認と、子機から親機へ送信するデータを含みます。Key 応答フレームは MP フレームへの応答としてハードウェアによって自動的に送出されるため、子機は Key 応答フレームに乗せるデータを前もってセットしておく必要があります。そのため、受信した MP フレーム中のデータを見てから、同じシーケンス内で返信するデータの内容を変更することはできません。
なお、もしも親から与えられた送信可能時間(TXOP)が、セットされている送信データの長さよりも短かった場合、データを送信できませんので Key 応答フレームの代わりに、Null 応答フレームを送出します。これは親機が設定している子機送信容量が、子機の把握している子機送信容量よりも少なかった場合に発生します。
また、親機から MP フレームを受け取ったタイミングで、まだ子機側の送信データがセットされていなかった場合は、子機は何も返信しません。

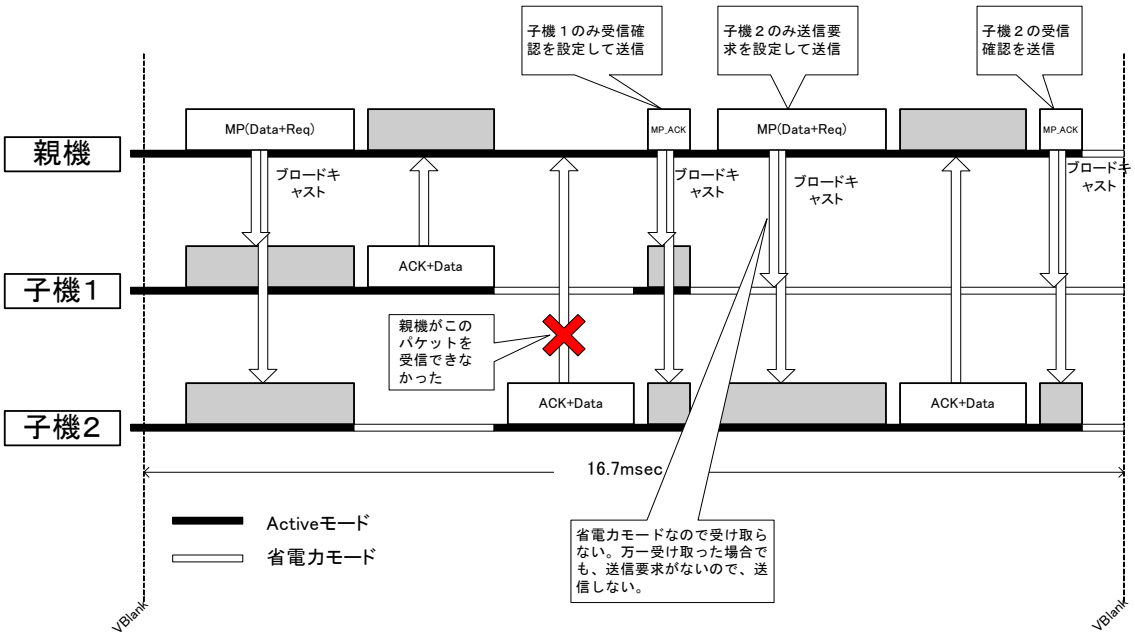
3. 最後に、親機は受信確認として MP_ACK フレームを全ての子機にブロードキャストします。
MP_ACK フレームにも PollBitmap と呼ばれるフィールドがありますが、こちらは MP フレームとは異なり、親機が Key 応答も Null 応答も受信できなかった子機を示す情報が入っています。各子機は受信した MP_ACK フレーム中の PollBitmap を見て、自分の AID にあたる部分のビットが立っていないことを確認します。ここで、ビットが立っていなければ子機から親機への送信に成功したことが保証されます。ビットが立っているか、そもそも MP_ACK フレームを一定時間以内に受信できなかった場合は送信失敗とみなします。

ワイヤレス通信ハードウェアがアクティブになると大きく電力を消費しますので、MP 通信中にはこまめにワイヤレス通信ハードウェアが省電力モードに移行します。これらの制御は自動的に行われますので、基本的にはアプリケーション側で気にする必要はありません。

また、PollBitmap で指定されなかった子機には返答の機会が与えられないことに注意してください。子機から親機への通信帯域を確保するため、WM では基本的には PollBitmap として接続中の子機全てを指定して MP シーケンスを行います。ただし、次項で述べる再送時と、特殊な通信モード設定がされていた場合は除きます。

3.4.3 通信失敗時の動作

親機が受信に失敗した場合の動作の例を下図に挙げます。



受信に失敗した場合は、MP_ACK フレームにて受信失敗した子機を通知し、続けて再送用の MP シーケンスを開始します。再送用の MP シーケンスでは、受信に失敗した子機とだけ通信を行うことに注意してください。

再送用の MP シーケンスでは、直前に失敗したパケットのうち再送が必要なものを送信します。通信モードや送信に失

敗したパケットの種類によっては、再送処理は行われず、アプリケーションに送信失敗が通知されます。通信に失敗し、かつ再送が必要なパケットが残っている間は、再送は続けられます。

なお、MP_ACK フレームのみが通信失敗した場合、親機側はその失敗を知ることができないので再送用の MP シーケンスは行いません。しかし、子機側は送信が成功したかどうかを判断することができないため、再送が必要なパケットについては次の MP シーケンスで再び送信します。

再送処理は通信パケットの種類によって異なりますので、「3.5.3 Raw通信とSequential通信」も参照してください。

なお、再送用の MP シーケンスは、PollBitmap の設定によって送信先が再送する相手だけに限定されています。しかし、あとは通常の MP シーケンスと同様の処理が行われますので、MP の各フレームの受信通知はフレームを受信するたびに発生します。ただし、後述する port 受信コールバックは新しいデータを受信したときだけ呼ばれます。

3.4.4 送信容量

MP 通信においては、親機が親子それぞれの送信容量を決定し、そのデフォルト値を通信開始時に指定します。具体的には、親機のパラメータを設定する WM_SetParentParameter 関数で送信容量のデフォルト値を設定します。WM_SetParentParameter 関数へ与える WMParentParam 構造体の parentMaxSize フィールドが親機から子機への送信容量のデフォルト値を、childMaxSize フィールドが子機から親機への送信容量のデフォルト値を表します。一方、子機は接続時に親機の beacon に乗っている childMaxSize の値で自分の送信容量の設定を初期化します。

送信容量が満たさないといけない制限は 3 つあります。

1. 2 バイトの倍数であること。
2. 親子のいずれの送信容量も 512 バイトを超えないこと。
3. （プログラミングガイドライン「6.3.3 1 回の MP 通信のデータサイズ【ランク B】」）
   親子の送信容量から計算される、1 回の MP シーケンスにかかる時間が 5600μ秒を超えないこと。
   すなわち、以下の式を満たすこと:

$$96+(24+4+[\text{親機送信容量}]+6+4)*4+(10+96+(24+[\text{子機送信容量}]+4+4)*4+6)*[\text{子機台数}]+10+96+(24+4+4)*4 <= 5600$$

$$\Leftrightarrow$$

$$[\text{親機送信容量}]+([\text{子機送信容量}]+60)*[\text{子機台数}] < 1280$$

なお、WMParentParam.KS_Flag を TRUE に設定した場合、内部的に Key Sharing 用の送信バイト数を追加するため、上式よりも送信容量として設定できる値は少なくなります。内部的に、親機で 36 バイト＋6 バイト（ヘッダ・フッタ）、子機で 6 バイト＋4 バイト（ヘッダ・フッタ）を確保しますので、その分も計算に入れるようにしてください。

通信時間の制限に関しては、$NITROSDK_ROOT¥man¥ja_JP¥wm¥wm¥wm_time_calc.html に計算用のスクリプトがありますので、ご活用ください。

なお、上記式は通信の所要時間の最悪値を求めるためのものであり、実際に各回の MP シーケンスにかかる時間は、親機の送信データサイズによって短くなります。ただし、子機側に関しては常に子機送信容量だけの時間を占有します。これは、MP シークエンスの開始時に、親機がその時点で自分が知っている情報のみでタイミング制御を行うためです。まとめると以下のようになります。

親機のデータ送信の所要時間	その回に親機が送信したデータサイズ分の時間と関係し、親機送信容量には影響されない。
子機のデータ送信の所要時間	親機が子機送信容量として設定したサイズと関係し、子機が送信したサイズには影響されない。

3.4.5 MP 通信用送受信バッファ

WM_StartMP 関数で MP 通信を開始する際に、MP 通信で使用する送受信バッファを渡します。送受信バッファサイズは親子の送信容量と最大接続子機数によって変化します。

送受信バッファのサイズを計算する方法は 2 通りあります。PARENT ステートまたは CHILD ステートの状態で WM_GetMPSendBufferSize, WM_GetMPReceiveBufferSize の各関数を呼び出す方法と、下の表にまとめた関数マクロに送信容量と最大接続子機数を与えることで静的に計算する方法です。

WM_GetMPSendBufferSize, WM_GetMPReceiveBufferSize の各関数は、現在の接続で使用されている親機情報から MP 通信で必要な送受信バッファサイズを動的に計算します。親機情報は、親機であれば通信開始前に WM_SetParentParameter 関数で設定した値が参照され、子機は接続時に親機から取得した beacon 内の情報が使用されます。注意点として、子機側でこれらの関数から得られる値は外部から与えられた値ですので、この値を元にメモリを確保する場合は、確保しようとしているメモリサイズが妥当な範囲であるかを確認するか、メモリ確保に成功したかどうかを確認するかのどちらかを必ず行うようにしてください。

各関数と関数マクロの詳細はリファレンスを参照してください。なお、WM_StartMP 関数に渡す MP 通信用送受信バッファは 32 バイト境界に合っている必要があります。

		静的計算用関数マクロ	関係する通信パラメータ		
			親機送信容量	子機送信容量	最大接続子機数
親機	送信バッファサイズ	WM_SIZE_MP_PARENT_SEND_BUFFER	○		
	受信バッファサイズ	WM_SIZE_MP_PARENT_RECEIVE_BUFFER		○	○
子機	送信バッファサイズ	WM_SIZE_MP_CHILD_SEND_BUFFER		○	
	受信バッファサイズ	WM_SIZE_MP_CHILD_RECEIVE_BUFFER	○		

3.4.6 V ブランク同期

MP 通信を開始すると、ワイヤレス通信ライブラリは自動的に親子の間で V ブランクの同期をとります。V ブランクのタイミング調整中は V ブランク間の周期が 16.7ms より長くなることに注意してください。1 フレームにつき最大で約 0.5ms 延びます。また、この間は V アラームカウント値が 202 から 210 カウントの間で変動することがありますので、通信中はこの範囲のカウント値での V アラームを使用しないようにしてください。

V ブランク同期のタイミング調整は主に接続開始直後に行われますが、通信中も随時発生します。

3.4.7 フレーム同期通信モードと連続通信モード

親機が MP シーケンスを起動するタイミングによって、フレーム同期通信モードと連続通信モードに分かれます。

フレーム同期通信モードは、毎ピクチャーフレームの特定の V カウントで MP シーケンスを起動する通信モードです。MP シーケンスを起動後、設定した回数の MP シーケンスを連続して起動したのちに、省電力モードでの待機状態に入ります。

ここで、フレーム同期モードにおける MP シーケンスの回数は全子機からの ACK を受け取れた回数でカウントします。これは子機からの通信帯域を確保するためです。親機から送信するデータが無かった場合も、子機が所定の回数の送信ができるように MP シーケンスを起動し続けます。また、子機からの応答フレームの受信に失敗した場合には、設定した回数に再送用の通信回数を上乗せて通信を行います。この時、通信回数が多すぎて次のピクチャーフレームにかかってしまった場合は残り通信回数のカウンタ値が累積されていきますが、このカウンタは一定値以上になるとそれ以上は累積されません。

一方、連続通信モードは、1 回の MP シーケンスが終わったら直ちに次の MP シーケンスを起動する通信モードです。ブロック転送など、大量のデータを一度に送信したい場合に使用されます。連続通信モードでは省電力モードに落ちる機会が少なくなり、電力消費的に大きく不利であることに注意してください。

なお、1 回の MP シーケンスの 5600μ秒という時間制限は、同じチャンネルに複数の親子が同居している場合でも安定して動作できるようにするためという理由もあります。1 ピクチャーフレームに複数回の MP シーケンスを行うと無線チ

ャンネルの専有時間が長くなりますので、複数の親子が存在した場合の動作が不安定になります。MP シーケンスの頻度の基本は 1 ピクチャーフレームに 1 回であることを理解した上で、アプリケーションで必要な最低限の頻度で通信を行うことを推奨します。

3.4.8 ピクチャーフレームあたりの MP 通信回数制限

前節のフレーム同期通信モードと連続通信モードの設定に関係なく、1 ピクチャーフレーム内での MP 通信回数には上限が設定されています。上限の回数は WM_SetMPParameter 関数で設定可能であり、デフォルト値は 6 回です。

フレーム同期通信モードで設定する MP 頻度では 1 ピクチャーフレームあたりに成功させたい通信の回数を設定しますが、この通信回数制限では失敗した通信も含めた MP 通信の総回数を制限します。

この制限が設定されている理由は、子機の接続台数が少なく、かつ親機側の送信データサイズが一時的に小さくなった場合に、1 回の MP 通信の時間が数百 us と短くなることがあり、これによってアプリケーションで想定していないほど高い頻度で MP 通信が行われる可能性があるからです。

WM_StartMPEx 関数の引数 fixFreqMode による通信回数の上限設定は、この制限機能により実現しています。fixFreqMode が真の場合は、上限数は MP 頻度に等しい値に設定されます。

3.4.9 ライフタイム

通信相手が突然居なくなって一定時間無通信が続いた場合、現在のライフタイムの設定に従い、ワイヤレス通信ドライバがその通信相手への接続を自動的に切断します。DS ワイヤレスプレイに関係するライフタイムには CAM ライフタイムと MP 通信ライフタイムの 2 種類があり、WM_SetLifeTime 関数で設定することが可能です。

CAM ライフタイムは、親機であれば子機、子機であれば親機からのワイヤレス通信のフレームがどのくらいの間来なければ切断するかを設定する値です。標準では 4 秒に設定されています。

また、MP 通信ライフタイムは、親機であれば子機からの Key 応答フレームが、子機であれば親機からの MP フレームがどのくらいの間来なければ切断するかを設定する値です。標準では 4 秒に設定されています。

CAM ライフタイムだけでは、子機側の ARM7 がフリーズしている場合にうまく切断されません。フリーズしたタイミングによってはワイヤレス通信ハードウェアが自動的に親機の MP フレームに反応して Null 応答フレームを返してしまうことがあるためです。この問題を回避するために MP 通信ライフタイムが設けられています。

なお、WM_StartParent 関数や WM_StartConnect 関数を用いて接続を確立しても、親機が WM_StartMP 関数を呼び出すまでは親子間で通信が始まりませんのでライフタイム切れになる恐れがあります。一方、子機は WM_StartMP 関数を呼び出すまでは返信しませんので、やはりライフタイム切れになる恐れがあります。WM_StartMP 関数は接続を開始したら直ちに呼び出すようにしてください。

ライフタイムによる自動切断を無効にすることも可能ですが、通信中に突然通信相手の電源が切られてしまった場合などの対応に必要な処理ですので、通常は標準値で使用してください。

3.5 port 通信

3.5.1 port 通信とは

MP 通信上での通信路の多重化のために、ワイヤレス通信ライブラリでは port という概念を導入しています。親子共に 16 個の仮想的な port を持ち、その番号を指定してデータを送信することによって、受信側の処理を振り分けることができます。

3.5.2 port 受信コールバック

受信側は、ワイヤレス通信の初期化後に WM_SetPortCallback 関数によって、使用する番号の port への受信コールバック関数を設定します。その後、送信側が WM_SetMPDataToPort*関数によってセットした送信データが MP 通信によって届けられると、受信側では port 番号に応じた受信コールバックが呼び出されます。

なお、新しい接続があった場合や、通信相手が切断した場合には、全ての port の受信コールバックに通知されます。

通知の詳細は「3.7 ワイヤレス通信ライブラリから通知されるイベント一覧」のWM_SetPortCallback関数の項を参照してください。

3.5.3 Raw 通信と Sequential 通信

port 通信には 2 タイプあり、使用する port 番号によって分かれます。0～7 番 port では Raw 通信が行われ、8～15 番 port を使うと Sequential 通信が行われます。

Raw 通信ではほとんど通信制御を行いません。データが相手に届かない場合と、同じデータが複数回届いてしまう場合がありえます。一方、Sequential 通信では、各パケットにシーケンス番号を付けて通信を行うことによって、ワイヤレス通信ライブラリのレベルで重複確認を行っており、低レベルでの再送処理と組み合わせて、確実かつ重複なく通信を行います。

通信が失敗した場合、Raw 通信では WM_StartMPEx 関数の引数の defaultRetryCount で指定した回数だけ再送を試みます。WM_StartMP で MP 通信を開始した場合は 0（再送無し）を指定したものとみなされます。Sequential 通信では、再送は成功するまで無限に行われます。

通信に成功するか、指定回数まで再送しても失敗すると、WM_SetMPDataToPort*関数の呼び出し時に指定した送信完了コールバックが呼び出されます。なお、送信完了コールバックが来るまで WM_SetMPDataToPort*関数で指定した送信データのあるメモリ領域は書き換えないでください。

TCP と UDP の関係とは異なり、Sequential 通信と Raw 通信はレイテンシもスループットもあまり変わりません。再送処理の必要の有無に応じて選択してください。電波状態の悪い子機が居た場合に、Sequential 通信ではその子機宛の再送処理がボトルネックとなりますので注意が必要です。

3.5.4 優先度と送信キュー

port 通信には 0～3 の 4 段階の優先度の概念があります。WM_SetMPDataToPort*関数で送信予約されたデータは FIFO (First in First out)の送信キューで処理されますが、送信キューは優先度別に 4 本存在し、高い優先度のキューが空にならない限りは、低い優先度のキューからデータが送信されることはありません。Data Sharing などのリアルタイム性の高い通信は優先度1に設定され、ブロック転送などリアルタイム性の低い通信は優先度 3 に設定されています。

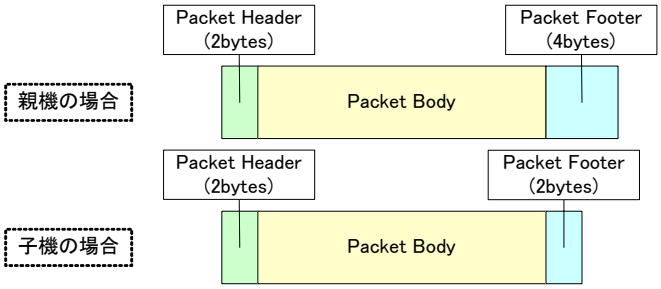
なお、Raw 通信の場合は同じ port 番号を指定しながら優先度を変えてデータをセットすることが可能ですが、Sequential 通信で優先順位を変えながらデータをセットすると、シーケンス番号による順序制御と不整合を起こすことがあります。具体的には、あとから高い優先順位のデータをセットすると、優先順位順に送信は行われますが、その時

に追い越された低優先順位のデータが正常に送信されない場合があります。(現在の実装では、追い越されたデータが捨てられる場合があります)

送信キューが一杯の状態で WM_SetMPDataToPort*関数を呼ぶと、WM_ERRCODE_SEND_QUEUE_FULL がコールバックに返り、失敗します。全優先度を合わせて 32 の送信パケットをキューに入れておくことができます。ただし、WM_SetMPDataToPort*関数の送信完了コールバックを待って次のデータをセットするような制御を行っている場合は送信キューを 1 段しか使用しませんので、このような通常の使用法で溢れることは考えにくいでしょう。Data Sharing も最大で 2 段までしか使用しません。

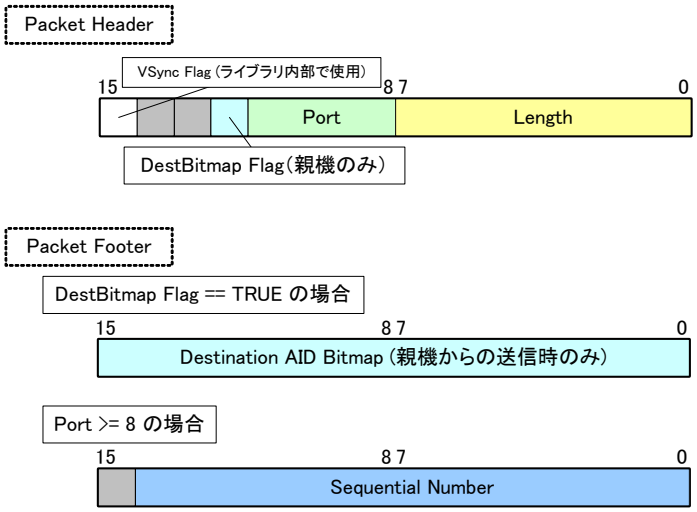
3.5.5 パケットのヘッダ・フッタ

port 通信を実現するため、ワイヤレス通信ライブラリのレイヤにおいてパケットという通信用のデータ構造を使用しています。



1 つのパケットは 2 バイトのヘッダと、親機で最大 4 バイト、子機で最大 2 バイトのフッタが送信したいデータの前後を挟む形で構成されます。

ヘッダとフッタのビット割り振りは以下のようになっています。なお、ワイヤレス通信ライブラリを使用する際には、以下の構造は特に意識する必要はありません。



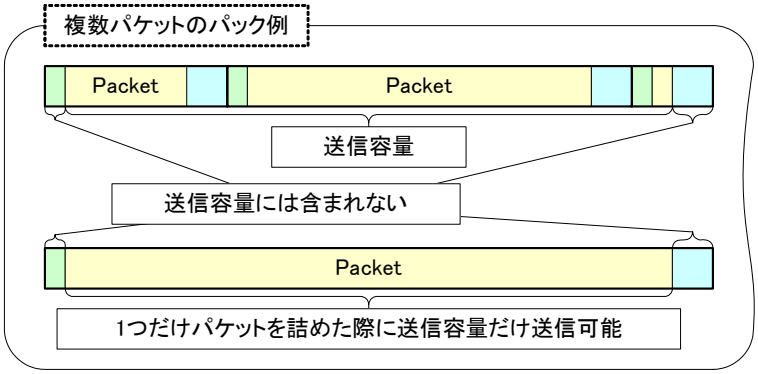
ヘッダはデータ長(2 バイト単位)と port 番号、および制御用のフラグを含みます。またフッタは送信先子機番号のビットマップとシーケンス番号を含みます。なお、データ長が 0 の場合は特別に 512 バイトとみなします。

親機から子機への通信時は基本的に送信先ビットマップを含みますが、ヘッダ中の DestBitmap Flag が 0 の場合は全子機へのブロードキャストを示し、フッタには送信先ビットマップは付加されません。

また、シーケンス番号は Sequential 通信の制御のために用いられる情報です。ヘッダに記されている 4 ビットの port 番号の最上位ビットが立っている場合(port 番号が 8 以上)に付加されます。

3.5.6 複数パケットのパック

MP 通信では、データの塊(ペイロード)を転送する方法のみを定めていますが、これでは少量のデータを効率よく転送できません。そのため、port 通信では、送信容量の制限が許す限り、複数のパケットをパックして送信します。



送信容量の設定値に対し、内部的に 1 パケット分のヘッダとフッタ分のサイズが追加されることに注意してください。送信容量は「その設定で最大に送信可能なユーザデータのバイト数」という意味付けをされています。

よって、複数パケットを送信する場合は、実際に送信するデータに加えて、間に挟まるヘッダ・フッタのサイズが余計に消費されます。このサイズは、1 パケット追加で詰めるたびに、親機の場合最大 6 バイト、子機の場合最大 4 バイトです。

複数のパケットを同時に送信する場合の使用バイト数は以下の式のようになります。

[使用バイト数] ＝ [パックするユーザデータサイズの合計]＋[追加のヘッダ・フッタ]×([パックするパケットの個数]−1)

[追加のヘッダ・フッタ] ＝ 6 バイト（親機の場合） or 4 バイト（子機の場合）

なお、このドキュメント内では簡単のために、一貫して追加のヘッダ・フッタのバイト数には全てのパケットが Sequential 通信であると仮定した場合の数値を使用しています。しかし、実際には Raw 通信はフッタが 2 バイト少ないため、上記の式に比べて Raw 通信のパケット 1 つにつき 2 バイト少ない送信容量でも通信が可能です。

3.6 Data Sharing

3.6.1 Data Sharing とは

リアルタイム性が高い通信ゲームにおいては、定期的に同じデータ(位置情報・行動情報等)を全ての参加者で確実に共有したいケースがしばしばあります。このような状況で利用可能なものとして、Data Sharing ライブラリ一式が用意されています。また、キー入力を共有することに特化した Key Sharing ライブラリを利用することも可能です。Key Sharing は今後廃止予定であり、現在の実装では内部で Data Sharing を使用しています。

なお、Data Sharing 及び Key Sharing は、いずれも公開されているワイヤレス通信ライブラリ関数のみを使用した、純粋な ARM9 上のライブラリです。

3.6.2 使用法

```
#define DS_SIZE          8 // 各8バイトを共有
#define DS_MAX           8 // 最大で子機7台＋親機
#define DS_BITMAP        0x00ff // 8台分の aidBitmap

WMDataSharingInfo dsInfo; // 約2kバイトの構造体なので確保場所に注意
u16 sendData[DS_SIZE/sizeof(u16)]; // 送信データ
WMDataSet receiveData;    // 受信データ
BOOL fUpdate;


... // ワイヤレス通信を初期化し、WM_StartMP() を行う

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

// メインループ
while ( TRUE )
{
    OS_WaitIrq(TRUE, OS_IE_VBLANK); // Vブランク待ち

    ... // sendData を PAD 入力などから作成

    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 )
            {
                ... // p を使って AID i からの入力を設定
            }
        }
        fUpdate = TRUE;
    }
    else
    {
        fUpdate = FALSE;
    }

    ... // 現在の内部状態で描画処理を実行

    if ( fUpdate )
    {
        ..... // 入力によりゲーム状態を更新
    }
}
```

まず、WM_StartMP 関数で MP 通信を開始した直後に、WM_StartDataSharing 関数を呼び出して Data Sharing を初期化します。その後、各ゲームフレームの頭で WM_StepDataSharing 関数を呼び出すだけで親子でデータを共有できます。

WM_StepDataSharing 関数が WM_ERRCODE_SUCCESS を返した場合は、Data Sharing に参加している全員がデータを共有できたということですから、その共有データで新しいゲームフレームを開始します。共有データは WM_StepDataSharing 関数から WMDataSet 型のデータセットとして得ることができます。このデータセットから個々の DS がセットしたデータを得るには WM_GetSharedDataAddress 関数を用います。

一方、WM_ERRCODE_NO_DATASET を返した場合は、通信相手の誰かが処理落ちしているということですのでゲームフレームの更新を遅らせて 1 ピクチャーフレーム待ちます。

使用法の詳細は、`$NITROSDK_ROOT/build/demos/wm/dataShare-Model` のデータシェアモデルデモ及び、「ワイヤレス通信チュートリアル(`WmTutorial.pdf`)」を参照してください。

3.6.3 Single Mode と Double Mode

Data Sharing には、Single Mode と Double Mode という 2 つの動作モードがあり、WM_StartDataSharing 関数の doubleMode 引数で指定します。

- **Single Mode**
  ゲームフレームが 30fps、もしくは、ゲームフレームが 60fps だが MP シーケンスの頻度が 1 ピクチャーフレームに 2 回以上の場合に使用することができます。直前の WM_StepDataSharing 関数でセットしたデータを受け取ります。Data Sharing 開始時に、どの AID のデータも含まない空のデータセットが 1 つ読み出されます。

- **Double Mode**
  ゲームフレームが 60fps で、かつ MP シーケンスの頻度が 1 ピクチャーフレームに 1 回の場合に使用します。2 回前の WM_StepDataSharing 関数でセットしたデータを受け取ります。Data Sharing 開始時にどの AID のデータも含まない空のデータセットが 2 つ読み出されます。

MP 通信の性質上、子機からデータを集めて、また子機に返すには 2 回の MP シーケンスが必要です。そのため、MP シーケンスが 1 ピクチャーフレームに 1 回の場合、60fps(すなわち、1 ゲームフレーム＝1 ピクチャーフレーム)の頻度で WM_StepDataSharing 関数を呼び出せるようにするには、間に 1 つバッファを挟まなくてはなりません。このモードが Double Mode です。

動作のイメージ図は「3.6.7 内部動作概説」を参照してください。基本的にSingle ModeとDouble Modeの差は、最初にいくつ空読み用のデータセットを用意するかの違いです。

3.6.4 通信データサイズ

Data Sharing が使用する送信データサイズは以下のとおりです。

[親機データサイズ] ＝ [共有データサイズ]×[データを共有する台数(親機含む)]＋4

[子機データサイズ] ＝ [共有データサイズ]

また、ライブラリの制限として、親機データサイズは 512 バイト以下でなければなりません。つまり、[共有データサイズ]×[共有台数]<=508 を満たしてください。また共有データサイズは偶数である必要があります。そのため、例えば子機 5 台の場合は、共有データサイズは 84 バイトまでとなります。

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

子機台数が 6 台以上の場合は、「3.4.4 送信容量」で解説した通信所要時間の 5600μ秒制限が共有データサイズの上限を決めるようになります。例えば子機 15 台の場合は、共有データサイズは 12 バイトまでとなります。

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

参考までに、各子機台数における最大共有データサイズの一覧を以下に示します。

子機台数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
親機データサイズ<=512 バイト の制限による最大共有サイズ	254	168	126	100	84	72	62	56	50	46	42	38	36	32	30
通信所要時間<=5600μ秒 の制限による最大共有サイズ	-	-	-	-	-	70	56	46	38	32	26	22	18	14	12

なお、通常の WM_SetMPData*関数と同時に使用する場合は、複数パケットをパックすることになりますので、親機・子機それぞれの最大送信サイズを計算する場合はパケットのヘッダ・フッタの分だけ親機で 6 バイト、子機で 4 バイトが追加で必要になることに注意してください。

3.6.5 関数の呼び出し順に関する注意

WM_StartDataSharing 関数は必ず WM_StartMP 関数の完了コールバックが呼ばれた直後に呼び出すようにし、WM_EndDataSharing 関数は WM_EndMP 関数の直前に呼び出すようにしてください。これは、現在の Data Sharing における制限事項です。

Data Sharing の開始を遅らせたい場合は WM_StepDataSharing 関数をただ呼び出さないようにするだけで構いません。Data Sharing 内部ではアラームやタイマーなどは使用しておらず、ライブラリ関数の呼び出しと送受信コールバックで処理を駆動しています。そのため、WM_StartDataSharing 関数を行った後でも、WM_StepDataSharing 関数を呼び出さない限りは余分な処理や通信は行われません。

ただし、タイマーなどを使用しないために WM_StepDataSharing 関数を呼び出すタイミングに制限があります。安定した Data Sharing を行うためには、WM_StepDataSharing 関数は V ブランク割り込み後のできるだけ早いタイミングで呼び出す必要があります。これは ARM7 で次の MP シーケンスの準備が行われるタイミング（V カウントにして子機 240・親機 260）までに、送信データをセットできるようにするためです。

3.6.6 30fps 以下での使用時の注意

ゲームフレームが 30fps のアプリケーションでは、WM_StepDataSharing 関数の呼び出しが 2 フレームに一回となりますが、WM_ERRCODE_NO_DATASET が返ってきた場合に関しては、次の呼び出しは直後のフレームで行わねばなりません。

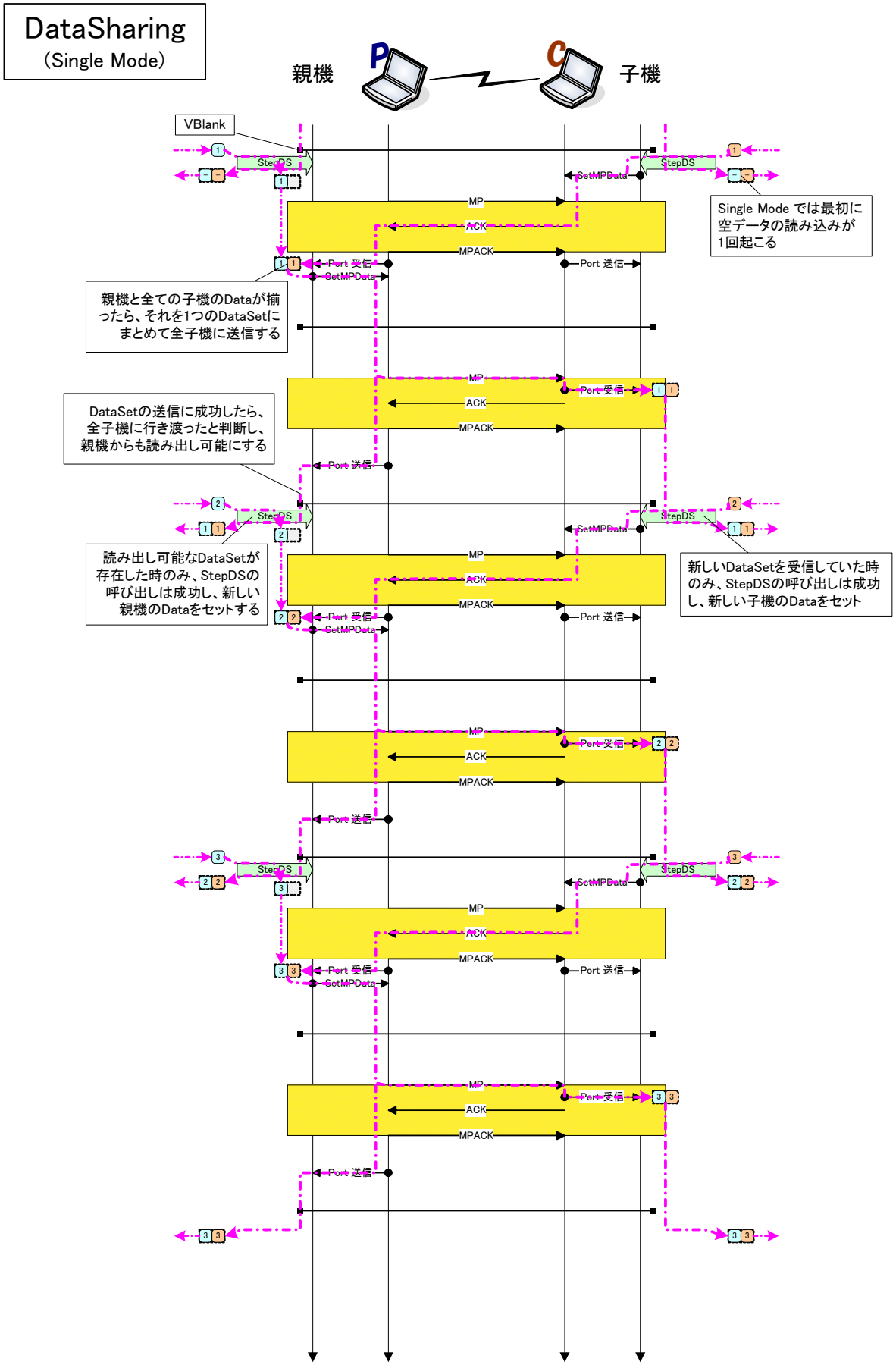
```
01: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
02: ----
03: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
04: ----
05: WM_StepDataSharing() == WM_ERRCODE_NO_DATASET
06: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
07: ----
08: WM_StepDataSharing() == WM_ERRCODE_SUCCESS
09: ----
```

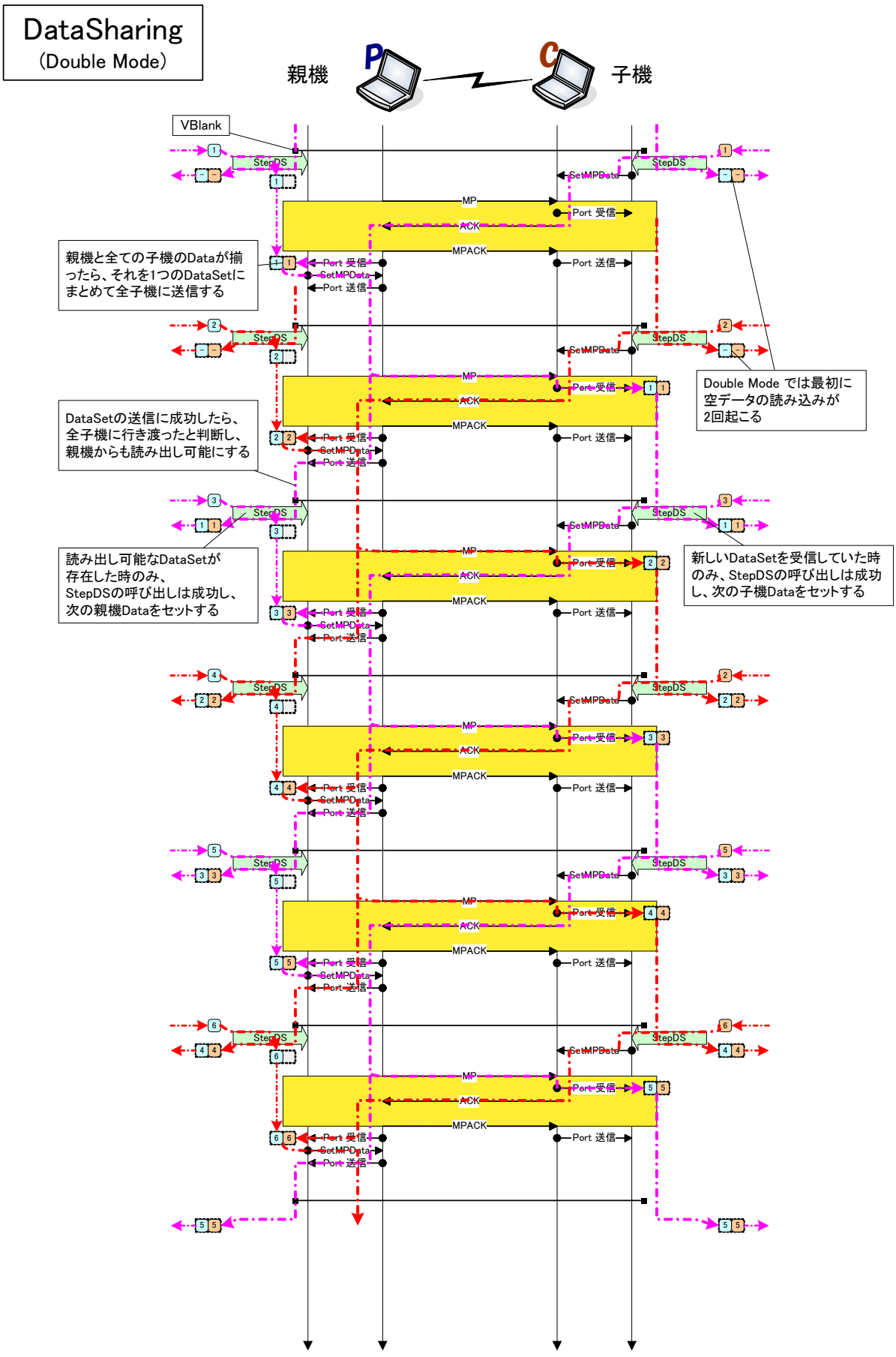
という形で失敗した時の処理を行うようにし、5 フレーム目と 6 フレーム目の間に 1 フレーム分の間を挟まないようにしてください。さもないと、親子で 1 フレームずれた場合に修正が効きません。

なお、30fps では上記のように呼び出すことにより、親子でのゲームフレームのタイミングのずれを修正することができますが、20fps 以下ではタイミングを完全に一致させることができません。これは、現在の Data Sharing における制限事項です。ただし、20fps 以下で使用した場合も、共有データの一貫性は保たれますので、タイミングが多少ずれてもデータだけ共有できればよい場合には Data Sharing を利用できます。

3.6.7 内部動作概説

Data Sharing の内部動作を下図に示します。図中では WM_StepDataSharing 関数を StepDS と表記します。




DataSharing
(Double Mode)
親機
P
C
子機
VBlank
StepDS
SetMPData
MP
Port 受信
ACK
MPACK
Port 送信
親機と全ての子機のDataが揃ったら、それを1つのDataSetにまとめて全子機に送信する
Double Mode では最初に空データの読み込みが2回起こる
DataSetの送信に成功したら、全子機に行き渡ったと判断し、親機からも読み出し可能にする
読み出し可能なDataSetが存在した時のみ、StepDSの呼び出しは成功し、次の親機Dataをセットする
新しいDataSetを受信していた時のみ、StepDSの呼び出しは成功し、次の子機Dataをセットする

3.7 ワイヤレス通信ライブラリから通知されるイベント一覧

一部の非同期関数のコールバックには、呼び出しに対応する操作完了通知の他に、ワイヤレス通信ライブラリからのイベント通知の呼び出しがかかります。コールバックが呼び出された要因はコールバックの引数の WM*Callback 構造体の state フィールドに WMStateCode 列挙型の値で格納されています。

以下にどの関数にどの WMStateCode でどういう意味の通知が来るのかを列挙します。なお、WM_SetMPData*関数を除いた非同期関数は、内部では関数単位でコールバック関数のテーブルを持っているため、同一の関数に対して呼び出しごとに違うコールバック関数を与えることは避けてください。

WM_StartParent	**WM_STATECODE_PARENT_START** 関数呼び出しに対する非同期の完了通知。 **WM_STATECODE_BEACON_SENT** ビーコンを送信するごとに通知。特に処理を行う必要は無い。 **WM_STATECODE_CONNECTED** 子機が接続してくるたびに通知。 この時、WMStartParentCallback.aid に接続してきた子機に割り振られた aid が、～.macAddress にその子機の MAC アドレスが、そして、～.ssid に子機が申告して来た SSID のユーザ領域（後半 24 バイト）が通知される。 **WM_STATECODE_DISCONNECTED** 子機との接続が切断された時に通知。 aid と macAddress に WM_STATECODE_CONNECTED の通知の際と同様の値が入る。 **WM_STATECODE_DISCONNECTED_FROM_MYSELF** アプリケーション内で WM の関数を呼び、自ら子機との接続を切断した時に通知。WM_STATECODE_DISCONNECTED の場合と同様の値が通知される。
WM_StartConnect **WM_StartConnectEx**	**WM_STATECODE_CONNECT_START** 関数呼び出しに対する非同期の完了通知。 親機のエントリーフラグが FALSE である場合や、最大接続数に達してしまった場合など、エントリーを受け付けなくなっていた場合には、errcode に WM_ERRCODE_NO_ENTRY, WM_ERRCODE_OVER_MAX_ENTRY が返る場合がある。この state で WM_ERRCODE_SUCCESS が返ったとしても、接続が完了したわけではないことに注意。接続の完了は WM_STATECODE_CONNECTED で通知される。また、親機が最大接続数をオーバーしていた場合に返る WM_ERRCODE_OVER_MAX_ENTRY は WM_ERRCODE_SUCCESS が 1 回返った後に発行されるので注意が必要。WM_ERRCODE_OVER_MAX_ENTRY は、親機に対して GGID や TGID が一致しない不適切な他の子機が接続してきていた瞬間に一時的に発生することがあるため、リトライ処理を行うことが推奨される。特に、ダウンロードプレイにおける子機プログラムのブート直後の接続で、親機の古いビーコンを元にダウンロードの接続に来る他の IPL 子機が存在した場合が、問題の発生しやすいケースである。 **WM_STATECODE_BEACON_LOST** 接続中の親機の beacon を一定回数分の時間、受信失敗した際に通知される。電波状態が悪くなっている可能性が高く、V ブランク同期が崩れる可能性があるが、特に処理を行う必要はない。 **WM_STATECODE_CONNECTED** 親機と接続を確立した際に通知。 この時、WMStartConnectCallback.aid に自分に割り振られた aid が通知される。 **WM_STATECODE_DISCONNECTED** 親機との接続が切断された時に通知。

	aid に WM_STATECODE_CONNECTED と同様の値が入る。 **WM_STATECODE_DISCONNECTED_FROM_MYSELF** アプリケーション内で WM の関数を呼び、自ら親機との接続を切断した時に通知。WM_STATECODE_DISCONNECTED の場合と同様の値が通知される。
WM_StartMP **WM_StartMPEx**	**WM_STATECODE_MP_START** 関数呼び出しに対する非同期の完了通知。 **WM_STATECODE_MPEND_IND** 親機のみ。MP_ACK フレームを送出し、一連の MP シーケンスを終了した時点で通知される。通常は特に処理を行う必要は無い。WMStartMPCallback.recvBuf に子機から受信したフレーム内容を保持した WMMpRecvHeader 構造体へのポインタが通知される。ただし、受信データの受け取りには port 受信コールバックを使用することが推奨されている。なお、recvBuf フィールドは WMMpRecvBuf 型へのポインタとして定義されているため、強制的なキャストが必要。 **WM_STATECODE_MP_IND** 子機のみ。親機からの MP フレームを受け取った時点で通知される。通常は特に処理を行う必要は無い。WMStartMPCallback.recvBuf に親機から受信したフレーム内容を保持した WMMpRecvBuf 構造体へのポインタが通知される。ただし、受信データの受け取りには port 受信コールバックを使用することが推奨されている。MP フレームの PollBitmap で自分が指定されていなかった場合は errcode が WM_ERRCODE_INVALID_POLLBITMAP となる。複数台の子機が接続された場合にはしばしば発生するため、回復不能なエラーとして扱ってはならない。また、受信した MP フレームにヘッダ情報も含めて何も含まれていなかった場合に WM_ERRCODE_NO_DATA が errcode として通知されるが、正常にライブラリが動作している限りは発生しえない。 **WM_STATECODE_MPACK_IND** 子機のみ。親機からの MP_ACK フレームを受け取った時点で通知される。通常は特に処理を行う必要は無い。この MP_ACK に対応する MP フレームの PollBitmap でそもそも自分が指定されていなかった場合は errcode が WM_ERRCODE_INVALID_POLLBITMAP となる。複数台の子機が接続された場合にはしばしば発生するため、回復不能なエラーとして扱ってはならない。そうでない場合で、親機が子機の Key(Null)応答フレームを受け取っていないことが MP_ACK フレームの PollBitmap フィールドで通知された場合は、errcode が WM_ERRCODE_SEND_FAILED となる。また、MP フレームを受信してから一定時間経っても MP_ACK フレームを受信できなかった場合もこの indication は発生し、WM_ERRCODE_TIMEOUT が errcode に入って通知される。
WM_SetIndCallback	**WM_STATECODE_FIFO_ERROR** ARM7 側の処理が過負荷になったなどの理由により、実行制御用のキューが溢れてしまった場合に、この WMStateCode で ARM9 側に通知される。回復不能な致命的エラーとして扱わねばならない。 **WM_STATECODE_INFORMATION** 内部で何か事象が発生した場合に通知される。WMIndCallback.reasonに通知内容が格納される。reasonに入る値としては、WM_StartMPEx関数のignoreFatalError引数をTRUEにしていた場合にfatal errorが発生すると通知されるWM_INFOCODE_FATAL_ERRORが定義されている。

	WM_STATECODE_BEACON_RECV 接続中の親機が発する beacon を受信するたびに通知される。通常は特に処理を行う必要は無い。WMIndCallback.state がこの値だった場合は、さらに WMBeaconRecvIndCallback にキャストすることで、～.gameInfoLength, ～.gameInfo などから GameInfo を得ることができる。 **WM_STATECODE_DISASSOCIATE** デバッグ用。普段は処理する必要はない。 **WM_STATECODE_REASSOCIATE** デバッグ用。普段は処理する必要はない。 **WM_STATECODE_AUTHENTICATE** デバッグ用。普段は処理する必要はない。
WM_SetPortCallback	**WM_STATECODE_PORT_INIT** WM_SetPortCallback の呼び出し中に割り込み禁止状態で呼び出される。WMPortRecvCallback.connectedAidBitmap に現在接続している相手の AID のビットマップが格納される。まだ接続を開始していない場合は connectedAidBitmap には 0 が格納される。また、～.arg に WMSetPortCallback の引数で与えた void* arg が渡される。 **WM_STATECODE_PORT_RECV** 通信相手からデータを受信した際に通知される。WMPortRecvCallback.aid に送信元の AID が、～.data に受信データへのポインタが、～.length に受信データのサイズが収められている。また、～.arg に WMSetPortCallback の引数で与えた void* arg が渡される。 **WM_STATECODE_CONNECTED** 接続の確立が WM_StartParent 関数, および WM_StartConnect*関数のコールバックに通知された直後に、全ての port の受信コールバックに対して同様に通知される。なお、親子問わず、WMPortRecvCallback.aid には常にその時接続してきた相手の AID が入ることに注意（子機は 0 で固定・親機は接続してきた子機に割り振られた AID）。自分の AID は～.myAid に格納されている。また、～.macAddress と～.ssid にそれぞれ接続相手の MAC アドレスと、SSID のユーザ領域（親機の場合）とがセットされている。 **WM_STATECODE_DISCONNECTED** 外部要因による通信の切断の発生が WM_StartParent 関数, および WM_StartConnect*関数のコールバックに通知された直後に、全ての port の受信コールバックに対して同様に通知される。AID に関して、WM_STATECODE_CONNECTED と同様の注意点がある。また、～.macAddress に切断した相手の MAC アドレスが格納される。 **WM_STATECODE_DISCONNECTED_FROM_MYSELF** アプリケーション内で WM の関数を呼び、自ら接続を切断した時に通知。WM_STATECODE_DISCONNECTED の場合と同様の値が通知される。
WM_SetMPData **WM_SetMPDataToPort** **WM_SetMPDataToPortEx**	**WM_STATECODE_PORT_SEND** 非同期関数の完了コールバックとして 1 種類の WMStateCode しか通知されないが、通信の制御上重要な通知であるため特にここに記述する。 WMPortSendCallback.errcode には、送信成功時の WM_ERRCODE_SUCCESS のほかに、通信失敗時に WM_ERRCODE_SEND_FAILED、送信キューが一杯だった場合に WM_ERRCODE_SEND_QUEUE_FULL がそれぞれ返る。なお、基本的に Sequential 通信では、通信切断時を除き WM_ERRCODE_SEND_FAILED が返ることはない。 ～.restBitmap にはリトライが必要な相手の AID のビットマップが格納される。ま

	た、～.sentBitmap には送信が成功した相手の AID のビットマップが格納される。接続されていない、もしくは送信中に切断されてしまった送信先は、～.restBitmap にも～.sentBitmap にも含まれない。～.errcode に WM_ERRCODE_SUCCESS が返る条件は、～.restBitmap が 0 になることであるが、これはつまり、送信先に指定され且つまだ接続中である全ての通信先に送信が成功した場合である。送信時に指定した送信先の全てに送れたかを確認したい場合は、～.sentBitmap を改めて確認すること。なお、～.sentBitmap に含まれている相手には送信できたことが保証される（相手が WM_EndMP 関数を呼び出した後だった場合を除く）が、含まれていない相手に送信されなかったことは保証されない。 1 回の WM_SetMPData* 関数の呼び出しに対して、ちょうど 1 回だけコールバックが呼び出される。この時、関数呼び出しからコールバックが呼び出されるまでの間は送信データのメモリ領域を上書きしてはならない。セットした送信データのアドレスは WMPortSendCallback.data で取得することも可能である。なお、WM_SetMPDataToPortEx 関数の引数 arg は、～.arg に渡される。

3.8 ワイヤレス通信ライブラリから返されるエラーコード一覧

3.8.1 WMErrCode 型を返す関数の返り値一覧

縦が関数、横が返り値です。WMErrCode の列挙値から接頭辞の WM_ERRCODE_を省略して記載しています。

関数名	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			○	○					○
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: アプリケーションの処理が適切でも、状況により
発生することのある返り値です。
通信は正常に続いていますので、
致命的エラーとして扱ってはいけません。

3.8.2 コールバック関数に返る errcode 値一覧

縦が関数とコールバックに帰る WM*Callback 構造体の state フィールドの値、横が WM*Callback 構造体の errcode フィールドの値です。接頭辞の WM_STATECODE_と WM_ERRCODE_は省略しています。
△は WM*Callback.state の値が不定値の場合があることを示します。

関数名	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	○			○			○		△								
	MPEND_IND$_1$	○																
	MP_IND$_1$	○													○$_2$	○$_3$		
	MPACK_IND$_1$	○									○$_2$				○$_2$		○$_2$	
WM_SetMPParameter		○			○			○		○								
WM_SetMPData*	PORT_SEND	○			○			○		△		○$_2$					○$_2$	
WM_EndMP		○	○		○					○								
WM_SetParentParameter		○	○					○		○								
WM_StartParent	PARENT_START	○	△		○			○		△								
	BEACON_SENT$_1$	○																
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																
WM_EndParent		○	○		○					○								
WM_StartScan*	PARENT_NOT_FOUND	○	△		○			○		△								
	PARENT_FOUND	○																
WM_EndScan		○	○		○					○								
WM_StartConnect*	CONNECT_START	○	△		○			○		△			○	○$_4$				
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																
	BEACON_LOST$_1$	○																
WM_Disconnect		○	○		○					○								
WM_DisconnectChildren		○	○		○					○								
WM_SetGameInfo		○	○							○								

		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_SetBeaconIndication		○	○							○								
WM_SetLifeTime		○	○							○								
WM_MeasureChannel		○	○		○					○								
WM_InitWirelessCounter		○	○							○								
WM_GetWirelessCounter		○	○							○								
WM_SetEntry		○	○							○								
WM_StartDCF	DCF_START	○								△								
	DCF_IND	○																
WM_SetDCFData		○	○							○								
WM_EndDCF		○								○								
WM_SetWEPKey		○								○								
WM_SetWEPKeyEx		○	○							○								
WM_SetIndCallback	FIFO_ERROR									○								
	INFORMATRION[1]	○																
	BEACON_RECV[1]	○																
	DISASSOCIATE[1]	○																
	REASSOCIATE[1]	○																
	AUTHENTICATE[1]	○																
	UNKNOWN																	○
WM_SetPortCallback	PORT_INIT	○																
	PORT_RECV	○																
	CONNECTED	○																
	DISCONNECTED	○																
	DISCONNECTED_FROM_MYSELF	○																

1: この state の通知に対しては通常は処理を行わなくても構いません。
2: アプリケーションの処理が適切でも、状況により発生することのある errcode です。
通信は正常に続いていますので、致命的エラーとして扱ってはいけません。
3: ライブラリが正常に動いている限りは発生しないはずの errcode です。
4: WM_ERRCODE_SUCCESS の通知が来た後で、改めてこのエラーが通知されることがあります。

3.9 ワイヤレス通信ライブラリ使用上の注意点

ここでは、ワイヤレス通信ライブラリの使用に際して注意が必要な事項に関して解説します。

3.9.1 ワイヤレス通信使用による負荷

現在の SDK では、ワイヤレス通信ドライバが 100k バイト以上のサイズになってしまっている関係上、ワイヤレス通信ドライバのコードが ARM7 用のワークメモリに乗り切らず、メインメモリに配置されています。そのため、ワイヤレス通信を行う際に ARM7 からメインメモリへの頻繁なアクセスが発生し、その影響で、描画処理などでの ARM9 からメインメモリへの連続アクセス時に大きなオーバーヘッドが生じる場合があります。(標準状態ではメインメモリへのアクセス優先度は ARM7 のほうが ARM9 よりも高くなっています)

また、メインメモリへのアクセス優先度を ARM9 優先に変更した場合(HDMA を使用する場合など)は、逆に ARM9 からメインメモリへの頻繁なアクセスが発生するとワイヤレス通信ドライバの実行に支障を来たす可能性があります。特に汎用 DMA にてメインメモリにアクセスした場合に、長期間 ARM7 のプログラム実行が遅延する可能性が高いと考えられます。メインメモリへのアクセス優先度を ARM9 優先に変更した状態でワイヤレス通信ドライバを動作させる場合には、できるだけ汎用 DMA を使用しないように注意して下さい。

効果的な対策として、VRAM-C または VRAM-D を ARM7 用に割り当ててワイヤレス通信ドライバを置くことにより、ARM7 によるメインメモリへのバスの占有時間を減らすことが可能です。詳細は WVR ライブラリのリファレンス、及び「コンポーネント解説(AboutComponents.pdf)」の ichneumon コンポーネントの解説を参照してください。

3.9.2 コールバック

コールバックは PXI 割り込みハンドラ内で呼び出されることに注意してください。割り込み禁止状態で呼び出せない関数は使用できません。また、長期間の処理を行わないようにしてください。ARM9 の他の割り込みが遅延される上に、ARM7 のワイヤレス通信ドライバが ARM9 のコールバックの終了を待っている場合があり、ワイヤレス通信の処理に悪影響が出ます。

3.9.3 キャッシュ処理

いくつかの関数では ARM7 にデータを渡すために、強制的なキャッシュのストアを行います。そのため、該当する関数に渡すデータは 32 バイト境界に合わせること、そしてデータの領域を 32 バイトの倍数で確保することを推奨します。そうしなかった場合は、前後のメモリ領域も一緒に強制的にキャッシュストアされてしまうため、場合によっては意図しない動作が起こる可能性があります。

一方、ワイヤレス通信ライブラリからアプリケーションへ渡されるデータに関しては、キャッシュの無効化をされてから渡されるものと、アプリケーション側でキャッシュの無効化をする必要があるものがあります。

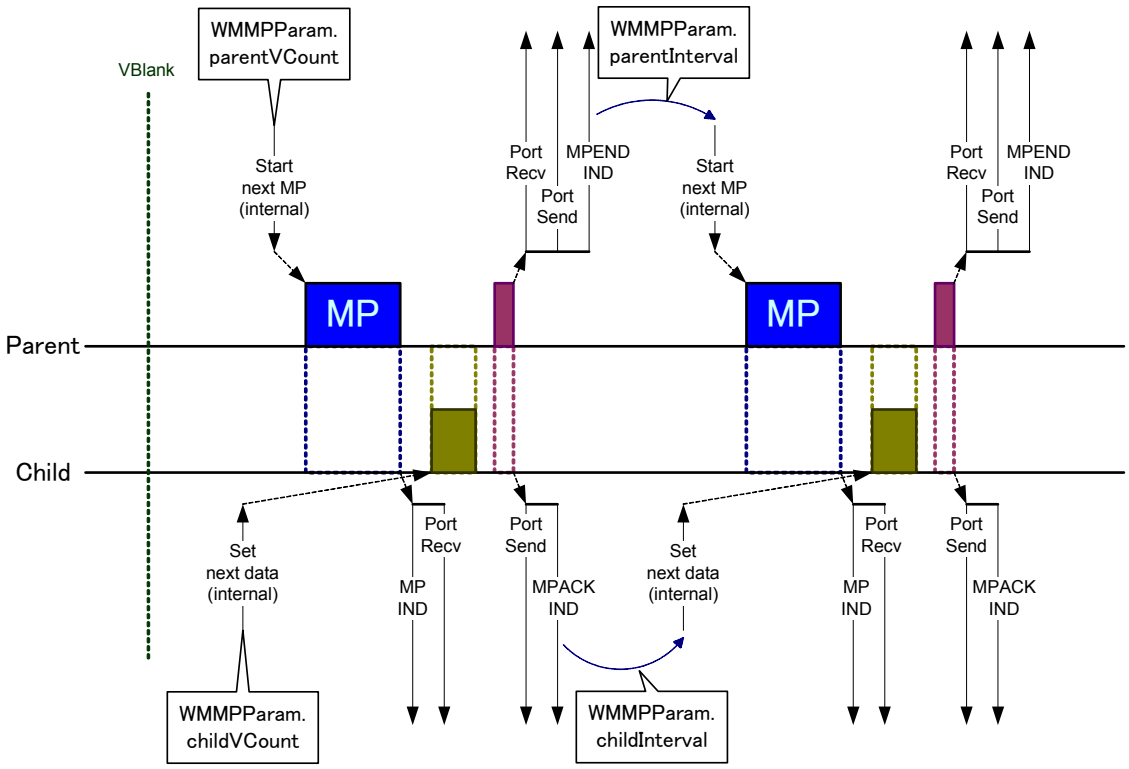
与えたメモリ領域がライブラリ内でキャッシュストアされるもの	WM_SetParentParameter 関数の引数 pparaBuf 及び pparaBuf->userGameInfo, WM_StartConnect*, WM_SetMPData*, WM_StartDCF, WM_SetDCFData, WM_SetWEPKey*
与えたメモリ領域がキャッシュストアされないもの (内部バッファに一度コピーしてからストアされます)	WM_SetGameInfo, WM_StartScan*, WM_SetMPParameter, 小さなメモリ領域しか渡さないその他の関数
データを受け取る領域がライブラリ内でキャッシュの無効化をされてから渡されるもの	WM_SetPortCallback 関数でセットした port 受信コールバックの data フィールド, WM_ReadStatus
データを受け取る領域がライブラリ内ではキャッシュの無効化をされずに渡されるもの	WM_StartScan*関数の引数 param->scanBuf で指定した領域 ※WMStartScan*Callback は無効化されてから渡される

3.10 より高度な通信制御

以下はワイヤレス通信を特に高度に用いたり、パフォーマンスチューニングを行うための情報です。

3.10.1 MP 通信のタイミング制御パラメータの概要

下図のように、いくつかのパラメータによって MP 通信の動作タイミングは制御されています。いずれも WMMPParam 構造体の所定のフィールドに値を設定して WM_SetMPParentParameter 関数を呼び出すことにより設定できます。



3.10.2 parentVCount, childVCount

WMMPParam.parentVCount は、フレーム同期通信モードにおいて、毎フレーム 1 回目の MP シーケンスを内部的に起動する V アラームカウント値で、デフォルト値は 260 です。WMMPParam.childVCount は、同様に子機側が毎フレーム 1 回目の返信データのセットを内部的に行う V アラームカウント値で、デフォルト値は 240 です。これらの値は WM_SetMPParameter 関数のラッパー関数である WM_SetMPTiming 関数でも設定することが可能です。

parentVCount, childVCount を変更することにより、フレーム同期通信モードにおける MP シーケンスの発生タイミングを調整することが可能です。MP シーケンスの前後では ARM7 側のワイヤレス通信ドライバがメインメモリに頻繁にアクセスを行うため、ARM9 側のアプリケーションによるメインメモリへのアクセスのストールが多発します。ARM9 側のメインメモリアクセスが少ないタイミングに MP シーケンスをずらすことによって、ストールの影響を押さえることができる場合があります。ただし、通信状態が悪い場合や連続通信モードでの通信中は parentVCount に関係ないタイミングで通信が発生します。なお、ARM7 の内部処理の遅延により、実際に MP シーケンスが発生するのは parentVCount より遅れることに注意してください。

3.10.3 parentInterval, childInterval

WMMPParam.parentInterval は、連続通信モードないしはフレーム同期通信モードでの 2 回目以降の MP シーケンスにおいて、親機が直前の MP シーケンスが終わってから次の MP シーケンスを内部的に起動するまでの時間をマイクロ秒単位で指定します。同様に WMMPParam.childInterval は、子機が直前の MP シーケンスが終わってから次の MP シーケンスに備えて返信データを内部的にセットするまでの時間をマイクロ秒単位で指定します。デフォル

ト値は parentInterval が 1000us、childInterval が 0us です。これらの値は WM_SetMPParameter 関数のラッパー関数である WM_SetMPInterval 関数で設定することも可能です。なお、ARM7 側の内部処理の遅延のため、実際の連続通信時の MP シーケンスの送信間隔は parentInterval より長くなることに注意してください。

連続した MP シーケンスにおいては、親子ともに interval 期間の終了時点で送信キューにセットされていた送信データを元に、次回の MP シーケンスの送信データを決定します。現在の実装では、直前の MP シーケンスの Port 受信コールバックと Port 送信コールバックが ARM9 側で終了するのを待ってから interval 期間に入ります。この仕様により、これらのコールバック中に WM_SetMPDataToPort 関数で送信キューに設定したデータは次回の MP シーケンスの送信に間に合います。

デフォルト値として親機のほうが長い間隔になっているのは、子機の返信データ設定が次の MP シーケンスに確実に間に合うようにするためです。親子の interval 値を等しくした場合、子機のコールバック処理が重いと通信エラーが多発するようになります。子機側のワイヤレス通信ドライバが ARM9 側の処理の終了を待っているため、返信データが設定される前に次回の MP シーケンスが来てしまうからです。返信データが設定されるまで子機は MP フレームに反応しなくなりますので、結果としてその間の通信が失敗することになります。例を挙げると、wbt-fs デモでは子機側のコールバック処理が場合により 700us 程度になることが分かっており、親子の interval 値が等しい場合は通信が高頻度で失敗します。

もしも、親子でコールバック内の処理の負荷が常に同等であることが分かっている場合は、parentInterval を縮めることにより、MP 通信の連続通信時のスループットを上げることが可能です。逆に、子機の Port 送受信コールバック処理が 1000us 以上かかる場合は、parentInterval をデフォルト値より長くしなければなりません。しかし、そもそも割り込みハンドラ内であるコールバックで 1000us を越えるような処理を行うことは望ましくありませんので、こういったケースでは設計を考え直す必要があります。

重い送受信処理を行う場合は、Port 送受信コールバック内からは通信用スレッドに処理要求を投げることだけを行い、直ちにコールバックから抜けるようにしてください。そのスレッドで次の送信データをセットするまでにかかる時間の最悪値を parentInterval, childInterval に設定することで、無駄なく連続通信することが可能になります。もしくは、送信キューは 32 段もありますので、常に複数の送信データを送信キュー内にバッファリングするような設計にすることも有効です。

3.10.4 送信容量の動的変更

通常は最初に親機が WM_SetParentParameter 関数で parentMaxSize, childMaxSize に設定した送信容量を用いて通信を行いますが、親機は必要に応じて WM_SetMPParentSize, WM_SetMPChildSize の各関数で親機と子機の送信容量を再設定することが可能です。これらの値は、最初に WM_SetParentParameter 関数で設定した値を越えることはできません。また、子機の送信容量は親機から MP シーケンスを行うたびに親機が設定した値に更新されます。このため、子機が WM_SetMPChildSize 関数で設定した子機送信容量は、直後の MP シーケンス用の返信データ準備でのみ利用されます。

	初期値	設定可能な最大値	親機側での再設定方法	子機側での再設定方法
親機送信容量	親機の beacon 中の parentMaxSize	同左	WM_SetMPParentSize()	（親機のみ意味を持つ）
子機送信容量	親機の beacon 中の childMaxSize	同左	WM_SetMPChildSize()	WM_SetMPChildSize() ただし、MP フレーム受信時に親機の設定で上書きされる

ここで注意しないといけないのは、子機が把握している子機送信容量の設定と、親機が設定している子機送信容量の設定が食い違っていた場合です。子機が把握している容量のほうが小さい場合は問題なく通信が行われますが、逆の場合は親機から子機データを送信するのに十分な時間が与えられないことになりますので、子機から親機にデータが返りません。動作に関しては「3.4.2 MP通信の動作」の解説を参照してください。ただし、このMPシーケンスで子機側の子機送信容量の設定が更新されますので、その後の再送で親子間の通信は問題なく継続します。親子で協調し、

WM_SetMPChildSize関数で子機送信容量を同時に更新することによりこの無駄な通信を避けることが可能です。

送信容量の変更が意味を持つのは、以下の 2 つのケースがあります。

1. **子機の不要な送信容量を絞りたい場合。**
   MP 通信の仕様上、たとえ子機が 2 バイトしか送信しないと分かっている場合でも、子機送信容量が 32 バイトと指定されていた場合は、毎回（32 バイト×子機台数）分の通信時間が消費されます。アプリケーションが把握している通信モードに応じて子機送信容量を減らすことによって、全体の通信所要時間が減り、結果として通信がより安定します。
   なお、親機側の送信は常に送信バイト数に見合った時間しか通信時間を消費しませんので、親機送信容量を制限することにあまり意味はありません。

2. **子機の接続台数によって送信容量を制限ぎりぎりまで使いたい場合。**
   親機送信容量 512 バイトで最大で 5 台の子機を繋げたいとします。ここで送信容量を固定にしてしまうと、最大の 5 台の子機が繋がった場合で通信時間の 5600us 制限を計算しないといけませんので、子機送信容量は 92 バイトとなります。送信容量は固定ですから、これは 1 台しか繋がっていないときでも変わりません。しかし、動的に送信容量を設定すれば子機接続数に応じて送信容量を最大化し、子機が 1 台のときは 512 バイトにし、2 台のときは 322 バイト、3 台で 194 バイト……、といったように設定することが可能になります。ただし、NITRO-SDK では動的に変化する送信容量を有効に活用できるような上位の通信プロトコルを用意していません。

上記の 1 の目的で使用する場合には、比較的安全に使用することができます。一方、上記の 2 の目的で使用する場合には、注意しないといけないポイントがいくつもありますので、通常は使用しないことをおすすめします。

2 の目的で使用する場合の注意事項をいくつか挙げておきます。まず、通常は WM_StartMP 関数の実行時に、最大子機接続数を用いた送信容量と送受信バッファサイズの事前チェックが入りますが、これを無効にしなければなりません。WM_SetMPParameter 関数で WMMPParam.ignoreSizePrecheckMode を TRUE に設定してください。

ignoreSizePrecheckMode を TRUE に設定した場合、2 つの効果があります。通信所要時間の 5600us 制限に関する警告の抑止と、受信バッファサイズの事前計算によるエラーの回避です。例えば、上の 2 で挙げた例では WMParentParam の親機送信容量・子機送信容量ともに 512 バイトで、最大子機接続数は 5 台と設定することになります。そうすると ignoreSizePrecheckMode が FALSE の場合、WM_StartMP 時の事前チェックではその設定でフルに繋がった場合の通信時間である 13970us が 5600us を越えているために警告がデバッグ出力に表示されます。また、親機の受信バッファサイズは、実際に子機送信容量を 5600us の制限の中で変化させていった際には子機 2 台で子機送信容量が 322 バイトだった場合の 1408 バイトが最大となります。しかし、WM_StartMP 関数での事前チェックでは 512 バイトの子機送信容量×5 台分の 5312 バイトが必要であると計算され、1408 バイトの受信バッファを WM_StartMP 関数に渡した場合、WM_ERRCODE_INVALID_PARAM でエラーになります。このエラーを回避し、必要最低限のバッファサイズで収めるためには ignoreSizePrecheckMode を TRUE にする必要があります。

ただし、事前チェックを無効にした場合、親機は見かけの受信バッファが足りなくても MP_PARENT ステートになれますが、実際の通信時に行われる子機送信容量と通信相手の子機数による受信バッファサイズの実行時チェックに引っかかった場合は MP シーケンスが実行されません。この場合、送信容量などを適正な値に修正しなければ一定時間後に MP のライフタイム切れで切断されます。逆に、子機接続数が増えた場合に直ちに受信バッファが足りるだけの適正な値に送信容量を修正することで、問題なく通信を継続させることは可能です。一方、子機の場合は受信バッファが足りなかった場合は通信が正常に行われず、通信を継続することもできません。そのため、子機では beacon に parentMaxSize として乗っている親機送信容量の最大値の分の受信バッファを必ず用意するようにしてください。親機とは異なり、子機の受信バッファは子機の接続台数には影響されませんので、常に最大で用意してもメモリ使用量にインパクトはありません。また、事前チェックを無効にしている場合は 5600us の通信時間制限はライブラリ側ではチェックされませんので、アプリケーション側で各パラメータを注意深く設定してください。

このように、ignoreSizePrecheckMode を使用する場合は、設定のミスが実行時におかしな挙動として現れますので、設定値には十分に注意を払うようにしてください。少しでも不明な点がある場合は無理に使用しないほうがよいでしょう。

3.10.5 PollBitmap の制御

親機がどの子機に応答して欲しいかを指示するのが、MP フレームに含まれる PollBitmap です。最初から PollBitmap を制御すれば、必要な子機だけ応答させることができ、全体の通信時間を短縮できます。しかし、PollBitmap で指定されなかった子機は、Key 応答フレームを送出することができず、親機にデータを送信する機会を与えられないことに注意が必要です。そのため、通常の MP 通信では、子機からの通信帯域を確保するために再送時以外は常に全ての接続子機に対して PollBitmap を立てて送信します。

ワイヤレス通信ライブラリでは、PollBitmap を細かく制御できるように、WM_StartMPEx 関数と WM_SetMPParameter 関数に minPollBmpMode と singlePacketMode という動作フラグを用意しています。しかし、この動作モードを正しく使用するためには、下記の複雑な制限事項をクリアする必要があります。ワイヤレス通信プロトコルに対する深い理解が必要ですので、通常は有効にしないようにしてください。

minPollBitmapMode が有効になっている場合、親機がその回で送信しようとしているパケットの送信先の論理和を取った値を PollBitmap として指定します。この際、間違って想定以上の相手と通信を試みてしまわないように、通常は singlePacketMode を併用するようにしてください。同時通信相手を限定していると仮定して大きい送信容量を設定しつつ、間違って想定以上の PollBitmap を立ててしまった結果、親機の受信バッファが溢れる通信条件になってしまった場合は、MP シーケンス開始時にバッファ不足を検出して送信がストップしまいます。これによる送信停止からは、送信容量を減らして受信バッファの制限を回避することで復帰することが可能ですが、アプリケーション側から原因を知ることは困難です。

また、minPollBmpMode を使用する場合には、Sequential 通信で使用しているシーケンス番号が子機の知らない間に回りきってしまわないように、全ての子機の 8～15 番の範囲で使用している port に関して、60 秒に 1 回は通信が行われる必要があります。

3.11 FAQ

ワイヤレス通信ライブラリの使用者からの相談事例について、代表的なものを Q&A 形式で解説します。

3.11.1 初期化処理

Q. WM_GetAllowedChannel 関数に有効な値が返ってきません。

A. WM_GetAllowedChannel 関数は WM_Init 関数を呼び出したあとでないと有効な値を返しません。初期化前に呼び出した場合はエラーを示す 0x8000 を返します。

3.11.2 接続処理

Q. 送信容量や送受信バッファサイズなどさまざまな通信パラメータがあってどう決定すればいいのかよく分かりません。

A. 以下に典型的なパラメータの決定手順を挙げます。

典型的な決定手順	例
最大接続子機数を決定する。	親機に子機を 3 台繋ぐことにしよう。 WMParentParam.maxEntry は 3 だ。
データシェアリングを使用するなら、共有するバイト数を決定する。	データシェアリングでは 16 バイトを共有しよう。
「3.6.4 通信データサイズ」の式でデータシェアリングで使用する親機と子機のデータサイズを計算する。	データシェアリングで使用する親機のデータサイズは 16×(3+1) + 4 で 68 バイト、子機は 16 バイトだ。
データシェアリング以外で通信するパケット数とサイズを決める。 (この際、子機から同時に送信される最大のデータ量を増やすと、データ量が少ない場合でも常にその最大値分の送信時間を消費してしまうことに注意して通信内容を決定する必要がある。)	WBT を使ったブロック転送を行うために、親機は 128 バイト、子機は 14 バイトを使用しよう。それとは別に、親機からのイベント通知のために独自の Sequential 通信で 32 バイトを使おう。
同時に通信される可能性のあるパケットを数え上げ、「3.5.6 複数パケットのパック」の式で親子の送信容量として必要なバイト数を計算する。	親機に関しては、データシェアリングが 68 バイト、WBT が 128 バイト、イベント通知用の独自通信が 32 バイトだから、128+68+32+6×2 で 240 バイト。子機は同様に 16+14+4×1 で 34 バイトだ。 (※ 正確には、通常は WBT は port 番号 4～7 の Raw 通信で使用するため、親子ともに必要な送信バイト数は 2 バイトずつ少なくなり、238 バイトと 32 バイトになります)
前項で計算した値をそれぞれ親機送信容量、子機送信容量として使用する。このとき、送信容量の制限である 512 バイトを超えていないことを確認する。	WMParentParam.parentMaxSize は 240、childMaxSize は 34 だ。 両者とも 512 バイトは超えていないので OK。
「3.4.4 送信容量」の式を用いて、親子の送信容量と最大接続子機数から計算できる 1 回のMPシーケンスの所要時間が 5600μ秒の制限を越えていないかを確認する。もしも越えていたら、収まるようになるまで上記のデータサイズの設計をやり直す。	96+(24+4+240+6+4)*4+(10+96+(24+34+4+4)*4+6)*3+10+96+(24+4+4)*4 を計算すると 2570μ秒となる。これは 5600μ秒の制限を満たしているので問題ない。 (この式は関数リファレンスの「図表・情報」にある「無線通信時間計算シート」を用いれば簡単に計算することができます)
最大接続子機数と親子の送信容量から、MP 通信で必要な送受信バッファサイズを計算する。	親機において WM_StartMP 関数に渡す受信バッファのサイズは WM_SIZE_MP_PARENT_RECEIVE_BUFFER(36, 3, FALSE) で、送信バッファサイズは WM_SIZE_MP_PARENT_SEND_BUFFER(240, FALSE)、同様に子機の受信バッファと送信バッファのサイズは WM_SIZE_MP_CHILD_RECEIVE_BUFFER(240, FALSE) と WM_SIZE_MP_CHILD_SEND_BUFFER(36, FALSE) だ。
MP 通信頻度を決定する。	ブロック転送ではそれほど大量のデータは送らないので、常に 1 ピクチャーフレームに 1 回の MP 通信頻度で問題はないと判断し、WM_StartMP 関数に渡す mpFreq パラメータは 1 にしよう。
MP 通信頻度と、ゲームフレームを何 fps にするかを考えて、データシェアリングの動作モードを決定する。	MP 通信頻度は 1 ではあるものの、60fps でゲームフレームを動かしたいので、WM_StartDataSharing 関数に渡す doubleMode は TRUE にしよう。

Q. WMParentParam.tgid に設定するべき値がわかりません。

A. 電源を再投入した際でも毎回異なる値であるのが理想です。手軽な方法としては OS_GetVBlankCount 関数や GX_GetVCount 関数の返り値を組み合わせることで擬似乱数を作成できます。さらに、RTC の秒や分の値を使うことによって、電源の再投入を行ってもしばらくの間は同じ値にならないことを保証することも可能です。また、子機が親機に何度か再接続するような実装の場合、TGID の何ビットかを親機のフェーズ情報に使って子機がフェーズの違う親機へ誤って接続しないようにチェックすると、より確実な接続を期待できます。

Q. scan 結果から親機一覧を作成していると、ときどき見つかりにくい親があります。

A. 全ての親機で beacon 間隔が同じの場合、beacon 送出タイミングがたまたま他の親が出す beacon の直後になってしまうと、その親機が発見されにくくなってしまう場合があります。また、処理のオーバーヘッドが作用して、親機の beacon 間隔が子機の scan 間隔の倍数になってしまった場合などに、やはり見つけにくくなることがあります。

対策として、まず、一度に複数の親機を取得できる WM_StartScanEx 関数の利用により、これらの問題を大幅に解消することが可能です。WM_StartScan 関数の使用は推奨されなくなりました。

また、親機の beacon 間隔と子機の scan 間隔に乱数を混ぜるようにしてください。
この目的で、WM_GetDispersionBeaconPeriod 関数と WM_GetDispersionScanPeriod 関数を用意しています。それぞれ 200ms と 30ms を中心とした乱数値を返す関数です。WMParentParam.beaconPeriod に WM_GetDispersionBeaconPeriod の値を設定することにより、親機の beacon 間隔が一致する頻度を減らすことができます。なお、beacon 間隔の設定は親機開始時に 1 回だけにしてください。動的に beacon 間隔を変更すると子機の接続に影響が出ます。

同様に、子機が WM_StartScan 関数や WM_StartScanEx 関数を呼び出す度にパラメータの maxChannelTime に WM_GetDispersionScanPeriod 関数の返り値を設定しなおすことで、子機の scan するタイミングにばらつきを持たせることができます。

Q. WM_StartConnect*関数での接続処理が安定しません。

A. 接続に失敗した場合の再試行時に WM_Reset 関数を呼び忘れていないか確認してください。途中まで接続シーケンスが進んでから失敗した場合、内部状態が CLASS1 ステートになっている場合があるため、WM_StartConnect* 関数を再び呼び出す前に WM_Reset 関数で内部状態を IDLE ステートに戻す必要があります。
また、親機が子機のエントリーを締め切るタイミングで WM_SetEntry 関数を呼び出してエントリーフラグを落とすことにより、締め切った後の意図しないタイミングでの子機の接続を防ぐことができます。子機側では実際に接続を試みる前に beacon 中の gameInfo. attribute の WM_ATTR_FLAG_ENTRY のビットを見て、親機がエントリー受付状態かどうかを確認してください。

Q. いったん通信を切ってから同じ親機へ再接続する場合に、ときどき失敗します。

A. scan してから再接続する場合、子機の再接続処理のタイミングが早すぎて、終了前の親機の古い beacon を拾って接続しに行ってしまうケースがあります。接続前に beacon の情報から親機が新しい接続を開始したかを確認するようにしてください。beacon 情報から親機の状態を知るには、userGameInfo に親機のフェーズ情報を含める方法、TGID の変化をチェックする方法などがあります。なお、DS ダウンロードプレイでの起動後に子機が親機を再 scan して接続する場合には gameInfo. attribute の WM_ATTR_FLAG_MB を確認することで親機がまだ DS ダウンロードプレイ用のモードにいるかを判定することも可能です。
また、再 scan をしない場合は、再接続にはあらかじめ取り決めておいたルールで TGID を更新してください。TGID のうち何ビットかを親機のフェーズ情報に使用して、親子で WMParentParam と WMBssDesc のその部分だけを書き換えて再接続を行うのが確実です。その際、親機が間違って直前の接続から channel を変えてしまうと子機が再接続できませんので注意が必要です。

Q. 私のアプリケーションは DS ダウンロードプレイ用の親機ではなく DS ワイヤレスプレイ用の親機なのですが、これを起動すると、別の DS で起動させた mb_child_simple.srl が反応します。
（「ゲームじょうほうじゅしんちゅう」または「GameInfo Receiving...」という表示が出続けます）

A. WM_SetParentParam 関数で指定した WMParentParam 構造体の multiBootFlag フィールドが 0 以外になっていないか確認してください。DS ダウンロードプレイ子機を待ち受けるために DS ダウンロードプレイ用の beacon を出している親機以外では multiBootFlag を立てないようにしてください。

3.11.3 MP 通信全般

Q. MP 通信で一番遅延が少なくデータを送信するにはどうすればよいですか。

A. フレーム同期通信モードの場合、フレーム最初の MP シーケンスの開始処理は親機の V カウントが 260 ラインのタイミングで行われます。また、子機は親機から MP フレームが来た時には返信データがセットされていなければならないため、少し早めの 240 ラインで返信データのセットの処理が開始されます。そのため、できるだけレイテンシを小さくしたい場合は親機で 260 ライン、子機で 240 ラインの少し前に WM_SetMPDataToPort*関数を呼び出してください。ただし、ARM9 でライブラリ関数を呼び出してから ARM7 のワイヤレス通信ドライバが処理するまでには予想不可能な遅延がありますし、送信キューに他のデータがあった場合はそちらから先に送信しますので、直ちに送信できるかは保証されません。なお、260 ラインと 240 ラインという数値は WM_SetMPTiming 関数で再設定することが可能です。
連続通信モードやフレーム同期モードでの 2 回目以降の MP シーケンスの場合は、直前の MP シーケンスが終わってから一定時間待ってから、親機で次の MP シーケンスが駆動され始めると共に、子機では次の返信データがセットされます。この間に WM_SetMPDataToPort*関数を呼び出すことにより、次の MP シーケンスにデータのセットを間に合わせることが可能です。ただし、ARM7 のワイヤレス通信ドライバの状態によっては間に合わない場合もあります。この待ち時間は WM_SetMPInterval 関数で設定することが可能です。

Q. WM_SetMPDataToPort 関数を連続して呼び出したところ、うまく動きません。

A. 関数の返り値として WM_ERRCODE_FIFO_ERROR が返ってきていませんか？ARM9 から ARM7 にコマンドを送るための FIFO が溢れるとこのエラーが返ってきます。ARM7 の処理が追いつくように、回数や頻度を減らすようにしてください。

Q. 細かいデータを大量に送っていたところ、通信状態を悪くするとうまく動かなくなります。

A. 何かのコールバックに WM_ERRCODE_FIFO_ERROR が返ってきていませんか？細かいデータを大量に送信している場合に、通信状態が悪くなって残り通信回数が蓄積されるなどして、連続して MP 通信が行われるようになると、子機側の ARM7 の処理能力を越えてしまうことがあります。
このようにコールバックに WM_ERRCODE_FIFO_ERROR が返る場合は、処理量がオーバーして ARM7 側の内部処理用の FIFO が溢れてしまっています。一般にはここから通信状態を回復させることはできませんので、ただちに通信エラー画面に遷移して通信をリセットするようにしてください。
子機側の通信関係のコールバック内で重い処理をしていると ARM7 がその完了を待つためにこの問題の発生率が上がります。また、相対的に親機が軽すぎる場合にも子機の処理が溢れることがあります。回避策としては、1.子機側のコールバック内の処理を軽くする。 2.あまり細かいパケットを大量に送信することは避ける。 3.WM_SetMPInterval 関数で親機側の最低送信間隔を空けるようにする。などが考えられます。

3.11.4 Data Sharing

Q. WM_StepDataSharing 関数が頻繁に WM_ERRCODE_NO_DATASET を返します。

A. いくつか可能性が考えられます。親子のどちらかは常に成功している場合は、その成功し続けているほうが処理落ちをしており、もう片方は相手の処理落ちを待っているために Step が失敗している可能性があります。また、毎フレーム WM_StepDataSharing 関数を呼び出す使い方をしている時に必ず 2 フレームに 1 回失敗する場合は、WM_StartDataSharing 関数の doubleMode を TRUE にしているか確認してください。2 フレームに 1 回 WM_StepDataSharing 関数を呼び出している際に定期的に失敗する場合は、WM_ERRCODE_NO_DATASET が返ってきた場合のリトライ処理がおかしい可能性があります。失敗した直後のフレームで次の WM_StepDataSharing 関数を呼び出すようにしているか確認してください。
親子共にランダムに同程度の頻度で失敗する場合は、WM_StepDataSharing 関数の呼び出しタイミングが悪い可

能性があります。できるだけ V ブランク割り込み直後の早いタイミングで WM_StepDataSharing 関数を呼び出すようにしてください。WM_StepDataSharing 関数は内部で WM_SetMPDataToPort 関数を呼び出しますが、最低の MP 通信頻度で Data Sharing を行うためには各 MP シーケンスに漏れなくデータが乗ることを要求します。そのため、前節の MP 通信の項目の解説にあるとおり、親機で 260 ライン、子機で 240 ラインのタイミングに間に合わない場合、通信タイミングの関係で Data Sharing が不安定になる場合があります。Key Sharing も内部で Data Sharing を行っていますので同様です。

Q. WM_EndDataSharing 関数で一時中断して、WM_StartDataSharing 関数で再開するというコードがうまく動きません。

A. WM_EndDataSharing 関数は通信終了の一連の処理で呼び出されることを前提としており、MP 通信中に終了と再開を続けて行うと不具合が出ることがあります。Data Sharing を中断したい場合は、あらかじめ共有データ内にフラグなどを設定して親子で中断するタイミングを取り決めた上で、WM_StepDataSharing 関数の呼び出しを止めるだけで構いません。WM_StepDataSharing 関数を呼び出さない限り、Data Sharing に関する余分な処理時間・通信が発生することはありません。なお、再開時には中断前の最後にセットしたデータが届くことに注意してください。

Q. 同じ port を使って、Data Sharing の共有データサイズの変更はできますか。

A. 現時点ではできません。WM_StartDataSharing 関数と WM_EndDataSharing 関数はそれぞれ、通信開始時・通信終了時に 1 回ずつ呼び出すようにして、通信中はひとつの port は同じ設定の Data Sharing で占有するようにしてください。
その代わりに、別の port で共有サイズの異なる 2 つの Data Sharing を行い、切り替えて使用することで同等の処理を実現できます。切り替え時の注意事項は前項に準じます。

3.11.5 その他

Q. 原因が分からないのですが、ときどき通信が止まります。

A. アプリケーション側でのメモリ破壊など、さまざまな原因が考えられますが、以下の点も確認してください。

- コールバック中で長期間の処理を行っていませんか？コールバックは割り込みハンドラ内ですので割り込み禁止状態となっている上に、ARM7 のワイヤレス通信ドライバがコールバックの終了を待っている場合もあります。各所に悪影響が出ますので、数 ms もかかるような処理を行うことは避けてください。
- コールバック中で関数呼び出しの深いネストが起こっていませんか？また、深いネストの中で OS_Printf 関数などのスタックを多く消費する関数を呼んでいませんか？コールバック実行中に使用される IRQ スタックはそれほど大きくありませんので、スタックの消費をできるだけ抑えるようにしてください。デバッグ出力をするとフリーズするような場合は OS_Printf 関数の代わりに OS_TPrintf 関数を使うと状況が改善されることがあります。
- 子機の WM_StartMP*関数から WM_StartDataSharing 関数の呼び出しが離れていませんか？現在の実装の制限により、子機側のこれらの関数は連続して呼び出す必要があります。

Q. ワイヤレス通信部分のデバッグ時に気をつけることはありますか。

A. まず、ワイヤレス通信のデバッグには十分な時間を確保するようにしてください。一見正常に動いた後も、数十回に 1 回発生するようなタイプの不具合が出てくるケースが非常に多いためです。
デバッグ時には、チャンネルの自動選択機能を一時的に無効にして固定チャンネルにした上で、複数グループの親子を同じチャンネルで同時に開始する、というテストを繰り返すと不具合の再現率が上がることが多いようです。

3.12 過去のリリースからの注意すべき変更点

ワイヤレス通信ライブラリに行われた変更のうち、コンパイルがそのまま通ってしまうなどして、変更点が分かりにくいものについて、以下に解説します。

3.12.1 MP フレーム送信条件の変更(NITRO-SDK 2.2PR 以降)

従来は、MP_PARENT ステートの親機は子機接続状態に関わらず MP フレームを送信していましたが、子機が接続されていない場合は送信しないように変更しました。これにより、子機接続台数が 0 の場合は MPEND の通知が来なくなります。port の送受信コールバックを利用している場合は挙動に変更はありません。なお、子機が切断された直後では、子機接続台数が 0 になっていても MP フレームを送信することはあります。

また、1 ピクチャーフレームに 6 回より多くの頻度でMPフレームを送出しないように制限が加わりました。意味のある通信を行っていれば通常はこの制限にかかることはありません。詳細は「3.4.8 ピクチャーフレームあたりのMP通信回数制限」を参照してください。

3.12.2 WM_SetIndCallback 関数のコールバックへの通知の追加(NITRO-SDK 3.0PR2 以降)

WM_SetIndCallback 関数のコールバックに、新たに WM_STATECODE_INFORMATION が返されるようになりました。これは、内部で発生した事象を通知するためのものです。コールバックに引数として渡される WMIndCallback.reason から事象の種類を知ることができます。

reason に入る値としては、WM_INFOCODE_FATAL_ERROR が定義されています。これは、WM_StartMPEx 関数で ignoreFatalError 引数を TRUE に設定していた場合に、fatal error が発生したことを示します。通常は ignoreFatalError は FALSE で運用しますので、通知されることはありません。

3.12.3 Null 応答発生条件の変更(NITRO-SDK 3.0PR2 以降)

従来は MP_CHILD ステートの子機において ARM7 側の処理が間に合わなかった場合、MP フレーム受信時に Null 応答が出ることがありましたが、これを応答を返さないように変更しました。なお、応答を返さない場合は、子機内で MP フレーム受信通知も発生しません。これにより通信効率が若干落ちますが、子機側の過負荷を軽減することが可能となります。

また、子機が MP_CHILD ステートにならないと応答を返さないことが保証されるようになりましたので、子機側の WM_StartConnect 関数と WM_StartMP 関数の呼び出しの間に多少の間があっても問題がなくなりました。ただし、あまり間が開きすぎるとライフタイム切れで切断されますので注意が必要です。

3.12.4 WM_STATECODE_DISCONNECT_FROM_MYSELF の追加 (NITRO-SDK 3.0RC 以降)

これまでは、WM_DisconnectChildren , WM_Reset, WM_EndParent, WM_Disconnect 関数などを明示的に呼び出して自ら接続を切断した場合は、切断通知が発生しない仕様でした。これを、新規に WMStateCode に WM_STATECODE_DISCONNECTED_FROM_MYSELF を追加し、これで切断通知するように変更しました。

WM_STATECODE_DISCONNECTED_FROM_MYSELF は WM_STATECODE_DISCONNECTED と同様のコールバック構造体の内容で WM_StartParent, WM_StartConnect, WM_SetPortCallback の各関数のコールバックに通知されます。

この変更により WMStartParentCallback, WMStartConnectCallback, WMPortRecvCallback の state フィールドに入る可能性のあるステートコードが増えていますので、既知の WM_STATECODE_* 以外を受け取った場合はプログラムの実行を停止する、などの処理を行っている場合は注意が必要です。

また、この通知を利用して DataSharing は親機から明示的に子機を切断しても停止しなくなりました。

3.12.5 WM_STATECODE_PORT_INIT の追加 (NITRO-SDK 3.0RC 以降)

WMStateCode に WM_STATECODE_PORT_INIT を追加し、WM_SetPortCallback 関数の呼び出し時にこのステートコードで port 受信コールバックを呼び出すように変更しました。 WMPortRecvCallback の myAid フィールドや connectedAidBitmap フィールドを使用した初期化処理を行うことを想定しています。

なお、接続前に WM_SetPortCallback を呼び出した場合は connectedAidBitmap フィールドも myAid フィールドも 0 が格納されています。

接続通知の一貫性を保つため、WM_STATECODE_PORT_INIT での呼び出しも割り込み禁止状態で行われますので、あまり多くの処理を行わないように注意してください。