DS ダウンロードプレイ解説

Ver 1.0.5

任天堂株式会社発行

目次

コード

図

改訂履歴

版	改訂日	改 訂 内 容	担当者
1.0.5	2007-09-27	2.1.1.4 記述追加（アイコン画像の作成方法に関する補足）	吉崎
1.0.4	2006-05-16	4.1 記述修正（サンプルコード例の記述修正）	奥畑
1.0.3	2005-03-06	2.3.3 記述修正（NITRO_COMPRESS 指定の指示に関する部分を削除） 2.3.4 記述追加（DS ダウンロードプレイの不具合に関連する補足）	北瀬
1.0.2	2005-08-08	2.1.1.1 用語表記訂正(WM_GetAllowedChannel 関数のスペルミス) 4.1.1 記述修正（本文で使用される参照番号をソースコードに反映）	吉崎
1.0.1	2005-03-11	1 記述修正（NITRO-SDK インストール先の表記形式を統一） 1.3 記述修正（次項目との重複箇所を削除） 1.4 項目名変更（「libsyscall.a の使用について」から） 記述修正（前項目に関連した補足） 1.5 記述修正（カードからの起動と同等である旨の明記） 2 用語表記訂正（AID） 2.1 関数名訂正（MB_StartParentFromIdle、MB_EndToIdle） 2.1.1.2 用語表記訂正（GGID、TGID） 2.1.1.3 記述修正（最大接続子機台数の詳説） 2.1.1.4 項目書式修正 記述修正（最大接続子機台数とプレイヤー数に関して補足） 記述訂正（ライブラリとサンプルモジュールの呼称に関して） 記述削除（セグメントデータに関する古い制限） 2.3.1 記述追加（複数の通信モードを判別する場合についての補足） 2.3.3 記述追加（ビルドスイッチに関する補足） 3.1 記述修正（図の修正、親機専用領域の補足） 3.2.1 記述修正（データ配置の理由を補足） 4 記述訂正（mb_parent.h を mbp に） 記述修正（サンプルコードを抜粋元の最新状態に反映） 4.1 記述追加（MB_StartParentFromIdle 関数使用時の変更手順を補足） 5 追加（サンプルプログラム(cloneboot)の解説）	吉崎
1.0.0	2004-10-29	初版	高野

1 はじめに

NITRO-SDK では DS ダウンロードプレイのための一連の API 群が用意されています。このドキュメントでは、DS ダウンロードプレイの基本的な用法について解説します。
以降、このドキュメントでの解説においては NITRO-SDK のインストール先を $NitroSDK と表記します。

1.1 概要

ニンテンドーDS(以下 DS)は、カードの挿さっている DS ダウンロードプレイ親機から、カードの挿さっていない DS ダウンロードプレイ子機に対してバイナリコードを転送して、子機をカードなしでブートさせる DS ダウンロードプレイという機能を持っています。

DS ダウンロードプレイは、開発者向け資料や SDK のソースファイルでは、「ワイヤレスマルチブート」とも呼ばれます。この機能を使用すると、親機から子機のメインメモリへ最大 2.5MB までのバイナリコードをダウンロードして子機をブートすることが可能です。

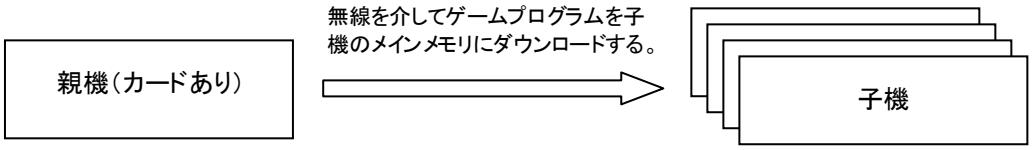

図 1-1 DS ダウンロードプレイ概念図

1.2 DS ダウンロードプレイ起動手順

DS ダウンロードプレイでゲームを起動するにあたって、プレイヤーは次の手順で実行を開始します。

(1) DS ダウンロードプレイ親機を起動する。

(2) 子機は DS の起動画面から「DS ダウンロードプレイ」を選択し、ダウンロードしたい親機のプログラムを選択する。

但し、IPL では不正なコードの起動防止のため、認証コードの付加されていないバイナリコードは実行されませんので、DS ダウンロードプレイから子機を起動する為には、送信するバイナリに認証コードを付加する必要があります。NITRO-SDK では開発効率を考慮し、認証コードのないバイナリコードを起動させるために mb_child を用意しています。NITRO-SDK に含まれる mb_child.srl を利用して以下の手順で実現してください。 デバッガ環境で実行する場合にも同様に mb_child を利用してください。

(1) NITRO-SDK に収録されている以下のいずれかのビルド済みプログラムを NITRO フラッシュカードに書き込む。(デバッガの場合はデバッガに mb_child.srl のバイナリコードをロードする。)
$NitroSDK/bin/ARM9-TS/Rom/mb_child.srl
$NitroSDK/bin/ARM9-TS/Rom/mb_child_simple.srl
$NitroSDK/bin/ARM9-TS/Release/mb_child_simple.srl
mb_child.srl および mb_child_simple.srl に関しては、関数リファレンスの [ビルド済みプログラム] の項

を別途ご参照ください。

(2) DS ダウンロードプレイ親機を起動する。

(3) DS ダウンロードプレイ子機として mb_child を起動し、ダウンロードしたい親機のプログラムを選択する。

実際に実機で起動させるために、子機へ送信するバイナリに認証コードを付加する方法については 1.3 章を参照してください。

1.3 認証コードの付加

DS ではワイヤレス経由で転送された不正なバイナリコードが実行されることを防止するために、子機で起動するバイナリコードには認証コードを付加する必要があります。

実機で認証コードのないバイナリコードを実行しようとした場合には、ブートの途中(画面に表示された Nintendo ロゴがフェードアウトする辺り)で停止しますのでご注意ください。

以下の手順で、転送するバイナリコードに認証コードを付加することができます。

(1) まず、子機へ転送するバイナリコードを生成します。

(2) 弊社に設置された認証サーバへこのバイナリコードを送信し、認証コードを取得します。

(3) $NitroSDK/tools/bin/attachsign.exe を使用して元のバイナリコードに認証コードを付加します。

(4) DS ダウンロードプレイ親機では、(3)で得られたバイナリコードをリンクして子機へ配信します。

以上の手順で、実機で動作するバイナリコードを子機へ配信することができます。

認証コードの取得に関する詳細は、弊社窓口までお問い合わせください。

1.4 システムコールライブラリおよび ROM ヘッダの使用について

製品版の ROM 作成時には、弊社より提供する製品用システムコールライブラリ（libsyscall.a）および ROM ヘッダ(rom_header_****.template.sbin）を使用することになっていますが、子機用バイナリに関しては親機とは異なり NITRO-SDK に付属しているシステムコールライブラリと ROM ヘッダを使用してビルドする必要があります。

1.5 転送可能バイナリコードサイズ

DS ダウンロードプレイにおけるバイナリコードのサイズはカードからの起動条件と同等の制限を受け、
転送可能な常駐コードは ARM9 が最大 2.5MB、ARM7 が最大 256kB までとなります。
なお、compstatic ツールによる圧縮時にもカードから起動した場合と同様に、展開後でなく圧縮時のサイズが対象となります。

上記制限を上回るサイズのバイナリコードを転送したい場合には、子機が DS ダウンロードプレイから一旦起動した後に、必要な追加分のバイナリコードを親機からダウンロードすることによって実現することができます。
ただし、実行コードそのものの転送に関してはセキュリティ上の理由による制限がありますので、ガイドラインの記述に従ってください。

1.6 カード・カートリッジバックアップ領域へのアクセス

DS ダウンロードプレイで起動した子機からカードやカートリッジスロットへ挿入されたバックアップ領域へアクセスする事も技術的には可能です。但し、その際にはいくつかの制約がありますので、実際の運用についてはガイドラインの記述に従っておこなってください。

2 DS ダウンロードプレイの動作

DS ダウンロードプレイ親機を作成するにあたって、必要な手順や接続シーケンスについて説明します。

DS ダウンロードプレイの処理は、NITRO-SDK に収録されている MB ライブラリを使用することで実装が可能です。
MB ライブラリでは内部で WM ライブラリを用いて動作しており、現状では他の WM の機能と同時に併用することはできません。

2.1 親機側処理の流れ

DS ダウンロードプレイ開始前の親機の準備について説明します。

親機は次の手順でバイナリ配信準備をおこないます。

1. 通信チャンネルの選択
2. 親機パラメータの設定
3. 親機の実行を開始
4. 子機バイナリ情報の登録
5. 子機からのリクエスト受付
6. バイナリ配信、ブート処理

4.の子機のバイナリ情報の登録を済ませた時点で、親機は自動的に情報配布を開始し、子機を受け付ける状態になります。

2.1.1 親機の準備

2.1.1.1 ワイヤレス通信チャンネルの選択

ワイヤレス通信時の通信チャンネルを決定する際には WM_GetAllowedChannel 関数で得られた使用可能チャンネルに対して WM_MeasureChannel 関数で電波使用状況を調査し、最も空いているチャンネルを選択する方法を推奨しています。

ただ、WM_MeasureChannnel 関数は WM ライブラリの状態が IDLE ステートでのみ実行可能なのに対して、MB ライブラリでは READY ステートから PARENT ステートまで MB_StartParent 関数内部で自動的に遷移してしまいますので、MB 起動後には実行できません。

このため、一度 WM_Initialize 関数で IDLE ステートになってから電波使用状況をチェックする必要があります。

通信チャンネルを決定した後は、WM_End 関数で一旦 WM ライブラリを終了してから DS ダウンロードプレイを開始する方法と、MB_StartParentFromIdle 関数を使用して IDLE ステートから DS ダウンロードプレイを開始する方法があります。

MB_StartParentFromIdle 関数を使用する場合には、WM_Initialize 関数を別で呼びだす分、MB_Init 関数に渡すワークバッファのサイズを WM_SYSTEM_BUF_SIZE バイトだけ小さく設定して構いません。
MB_StartParentFromIdle 関数と MB_EndToIdle 関数、MB_StartParent 関数と MB_End 関数は必ず対で呼ばれるようにしてください。

2.1.1.2 親機パラメータの設定

DS ダウンロードプレイ親機を開始する際には、通常のワイヤレス通信と同様に GGID, TGID を設定する必要があります。また、DS ダウンロードプレイではIPL子機画面に表示されるニックネームなど、以下の親機プレイヤー情報を設定する必要があります。

- プレイヤーのニックネーム

  UTF16-LE で最大 10 文字。 OS_GetOwnerInfo 関数で取得できるニックネームと同じ形式になっています。

- お気に入りカラー

  プレイヤーの好きな色を表す色セット番号です。OS_GetOwnerInfo 関数で取得できる favoriteColor と同じ色セットが使用されます。詳しくは OS_GetFavoriteColorTable 関数のリファレンスを参照してください。

- プレイヤー番号

  親機は常に 0 に固定です。

2.1.1.3 最大接続子機台数の設定

MB ライブラリはデフォルトで最大台数(親機 1 台、子機 15 台)を想定して WM ライブラリでワイヤレス駆動します。
そのため、プレイヤー数を 16 台より少なく設定された配信プログラムにおいては、本来可能であるはずの転送効率が得られなかったり、最大プレイヤー数を超える子機からの接続要求が発生するという状態になりえます。

親機から配信するプログラムの個数とそれらのうち最大のプレイヤー数があらかじめわかっている場合には、MB_SetParentCommParam 関数を使用することで接続可能な子機台数を設定することができます。接続してくる子機の AID の最大値はこの関数の maxChildren 引数で設定した値になります。また、sendSize 引数については、さきの maxChildren 引数とあわせてワイヤレス通信の既定時間の範囲内で自由な送信バッファサイズを設定できます。
このサイズは最小 MB_COMM_PARENT_SEND_MIN、最大 MB_COMM_PARENT_SEND_MAX バイトです。

2.1.1.4 子機バイナリ情報の登録

子機へ配信するバイナリの登録をおこなう際には、以下の情報を設定する必要があります。

- 配信バイナリコードのデータへのポインタ

  子機の起動時には子機バイナリコード全体のうち ROM スペックファイルで ARM9 常駐モジュール及び ARM7 常駐モジュールとして割り振られているコードのみが転送されます。子機の起動に必要とされるコードは MB_ReadSegment 関数でバイナリコードから抽出することができます。常駐モジュール(以下 Static セグメント)の設定方法については別途 makerom のリファレンスを参照してください。
  その他のバイナリデータは、ブート後に親機から子機へワイヤレス通信を使用して転送する必要があります。データ転送のプロトコルのひとつとして SDK には WBT ライブラリが用意されており、アプリケーションでの用途に応じてこれを採用することもできます。また、WBT を使用して子機自身のファイルシステムをワイヤレス経由で再構築するサンプルプログラムが、$NitroSDK/build/demos/wireless_shared/wfs 以下にモジュール形式で用意されています。

- ゲーム名

  UTF16-LE で最大 48 文字。 ただし IPL 表示時に1行の横幅 185 ドット以内におさまる文字列にする必要があります。

- ゲームの説明文

  UTF16-LE で最大 96 文字。 ただし IPL 表示時に2行の横幅 199 ドット以内におさまる文字列にする必要があります。

- IPLでのダウンロードゲーム表示用アイコンのパレットデータ、イメージデータ
  16 色のパレットデータと 32 ドット×32 ドットのイメージデータです。
  これは通常のプログラムにおけるバナーイメージと全く同じフォーマットであり、データの作成には $NitroSDK/tools/bin/ntexconv.exe を使用することができます。
  ntexconv.exe ツールの使用方法は $NitroSDK/man/ja_JP/tools/ntexconv.html をご参照ください。

- GGID
  ブート後の子機に対して通知するゲームグループ ID です。ここで設定された GGID は、子機がブート後に MB_GetMultiBootParentBssDesc 関数から取得できる構造体の ggid メンバに反映され、ブート後の再接続などに利用することができます。

- 最大プレイヤー数
  子機 IPL 画面に表示するこのゲームの親機を含む最大プレイヤー数を指定します。親機を含む総数であり最大子機台数ではありませんので、特に MB_SetParentCommParam 関数の maxChildren 引数との取り違えに注意してください（同じプレイヤー数の設定で両者の関数を呼び出す場合は、
  (最大プレイヤー数) = (maxChildren + 1) の関係になります)。
  また、この値は単に子機 IPL 画面への情報としてのみ使用され、実際に接続を受け付ける子機台数は MB_SetParentCommParam 関数で設定された maxChildren 引数の値以下であることにも注意してください。

MB ライブラリでは、MB_RegisterFile 関数での子機バイナリの登録は MB_StartParent 関数で親機の処理を開始した後におこなう必要がありますので注意してください。

MB ライブラリでは一台の親機で異なる複数の子機バイナリを最大 16 個まで登録することが可能です。この場合、子機の DS ダウンロードプレイメニュー画面では複数のゲームが配信されているように表示されます。

2.1.2 子機へのデータ配信と起動

バイナリコードの配信準備が完了すると、親機では子機からのリクエストを待ち、それぞれの子機に対して「エントリー」→「ダウンロード」→「ブート」の手順で処理をおこないます。

子機の状態は MB_CommSetParentStateCallback 関数で設定したコールバック関数に通知される他、MB_CommGetParentState（子機 AID） 関数で取得できます。

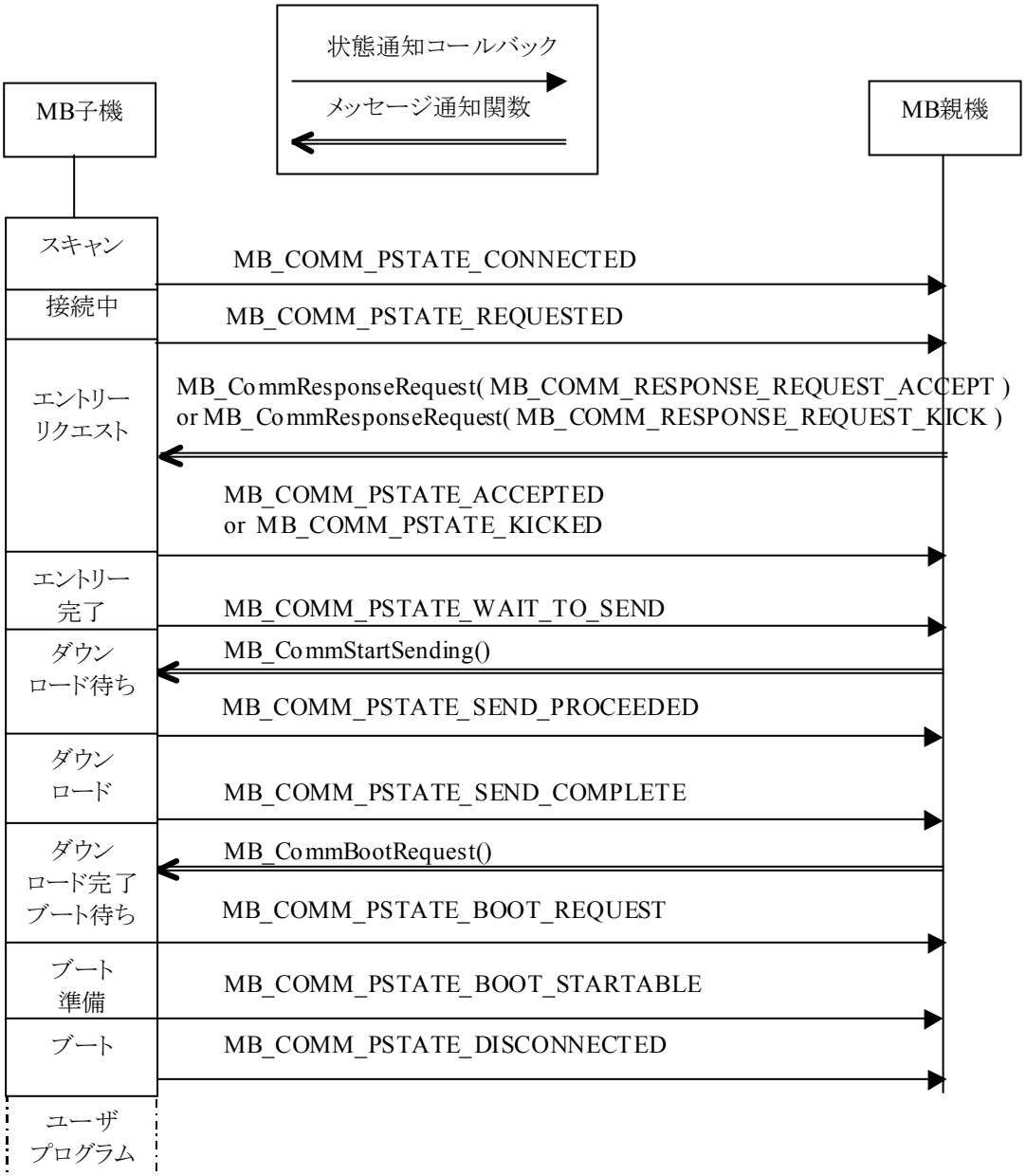

図 2-1 DS ダウンロードプレイ子機のデータ受信状態遷移と親機のリクエスト

子機の状態と親機からのリクエストの流れを図 2-1 に示します。

子機の状態が遷移する度に、親機ではコールバックが発生します。このコールバックによる状態変更通知または MB_CommGetParentState 関数で取得される状態を元に、それぞれの状態に対して親機側から適切なコマンドを発行してください。

親機と子機の接続シーケンスは次のような流れになります。

1. 接続

   IPL の DS ダウンロードプレイ子機プログラムが親機に対して接続すると、MB_COMM_PSTATE_CONNECTED 状態となります。このコールバックで子機の MAC アドレスを取得することができます。

2. エントリー

   親機に対して子機からエントリー要求があると、MB_COMM_PSTATE_REQUESTED 状態が通知されます。この時、子機は親機から MB_COMM_RESPONSE_REQUEST_ACCEPT メッセージまたは MB_COMM_RESPONSE_REQUEST_KICK メッセージが来るのを待ちます。
   MB_COMM_RESPONSE_REQUEST_ACCEPT が送られるとエントリー処理がおこなわれデータダウンロードの準備をおこないます。

3. ダウンロード

   子機側でデータダウンロードの準備が完了すると MB_COMM_PSTATE_WAIT_TO_SEND 状態が通知されます。この状態になってはじめて親機はデータの送信を開始することができます。MB_COMM_PSTATE_ACCEPTED 状態でデータ送信を開始してしまわないよう注意してください。
   データの送信が完了すると子機から MB_COMM_PSTATE_SEND_COMPLETE 状態が通知され、子機はブート要求があるまでこの状態のまま待機します。

4. ブート

   データ送信が完了した状態で親機から MB_CommBootRequest 関数でコマンドを発行すると子機はブート処理に入ります。ブートの際には MB_BOOT_STARTABLE 状態が通知された後一旦親機との通信は完全に切断されます。

2.2 親機との再接続

DS ダウンロードプレイではブート時に一旦通信が完全に切断されてしまうので、再び一から通信を確立する必要があります。再接続の際には以下の点に注意してください。

- 子機のブートタイミング

  現在 MB ライブラリでは MB 通信と他の WM 通信を同時にはおこなえませんので、親機は子機をブート後 MB_End 関数で一旦通信を終了する必要があります（MB_StartParentFromIdle 関数を使用して開始した場合には MB_EndToIdle 関数で IDLE ステートまで戻るだけで構いません）。DS ダウンロードプレイでのブート後に親機と子機が再接続して通信するためには、子機にブート要求を送信するタイミングを合わせるなどの対策が必要となります。

- 親機情報を利用して接続処理

  子機は MB_ReadMultiBootParentBssDesc 関数でブート前の親機情報を取得することができます。
  ここで取得できる WMBssDesc から直接親機と接続することも可能ですが、親機の GGID、TGID と子機が想定している GGID,TGID が異なると接続できません。また、最大送信サイズや KS,CS フラグが異なると接続後の通信が不安定になる場合がありますので、アプリケーション側で事前に取り決めておく必要があります。
  WMBssDesc の中にある MAC アドレス(bssid)を指定して親機を再スキャンすると、親子間での通信設定の食い違いを防ぐことができます。

- TGID の取り扱い

  子機が再接続の際に誤ってワイヤレス機能を再起動する前の親機に接続してしまったり、再起動後に無関係の IPL 子機から接続されたりすることを回避するために、親機のワイヤレス機能を再起動する際には親機 TGID

を変更することを推奨します。
ただし子機から再スキャンなしで接続する際には、親子間で TGID の同期を取っておかなければならないので、親子共に TGID を固定数だけインクリメントするなどの方法で親子の TGID の設定を合わせてください。

- 親機マルチブートフラグ

  WM_SetParentParameter 関数の引数に渡す親機情報の中にマルチブートフラグの情報がありますが、このフラグは通常は設定しないでください。DS ダウンロードプレイでのブート後に親機のワイヤレス機能を再起動して再接続する際にもマルチブートフラグを立てる必要はありません。

2.3 その他 注意事項

2.3.1 複数の通信モードを持つアプリケーション

マルチカードでの通信対戦モードと、1カードでの DS ダウンロードプレイモードの両方のモードを有しているなど、複数の通信モードを持つアプリケーションでは異なる通信モードから親機が見えてしまうと困る場合があると思います。

一般的に、子機側で複数の通信モードを判別する場合は、親機側で設定する userGameInfo に識別情報を入れておき、スキャン時にそれを参照するという方法を使用します。ただし、MB ライブラリでは、userGameInfo は使用できませんので、MB ライブラリ使用中かどうかを判定する場合は、WMBssDesc.gameInfo.gameNameCount_attribute の WM_ATTR_FLAG_MB フラグを参照してください。

上記以外にも、複数個の GGID を取得して判別するという方法もあります。

2.3.2 IRQ スタックについて

ワイヤレス通信時のコールバック関数はすべて IRQ モードで動作しますので注意してください。

コールバック内でスタックを大きく消費するような処理をおこなう場合には、lcf ファイルで IRQ スタックサイズを少し大きめに設定しておいた方が安全です。特にデバッグ時に OS_Printf 関数は大量のスタックを消費するので、コールバック内ではできるだけ軽量版の OS_TPrintf 関数を使用するようにしてください。

2.3.3 DS ダウンロードプレイ子機プログラムのオーバーレイについて

DS ダウンロードプレイ子機プログラムがオーバーレイ機能を使用する場合、自身のバイナリに含まれるオーバーレイテーブルおよび個々のオーバーレイセグメントを親機から別途受信する必要があります。
このとき、受信データの正当性を保証するために以下の点を正しく守る必要があります。

- ビルドスイッチ NITRO_DIGEST の指定
  DS ダウンロードプレイ子機プログラムのビルドには、NITRO_DIGEST のビルドスイッチを指定する必要があります。これは、オーバーレイテーブルおよび個々のオーバーレイセグメントが確かに自身のものと一致するための正当性判定を NITRO-SDK が正しく行うために必要です。このビルドスイッチが指定されていないプログラムでオーバーレイを使用した場合、実行時に強制停止します。
  このビルドスイッチの指定は compstatic.exe ツールに -a オプションを与えて呼び出すことと等価です。
  なお、このビルドスイッチはアプリケーションにのみ必要なもので、SDK のビルドでは無視されます。

- FS ライブラリ関数の使用
  上記ビルドスイッチの指定とあわせて、NITRO-SDK がこの正当性判定を確実に実行することを保証するため、オーバーレイの操作には必ず以下の FS ライブラリ関数を使用する必要があります。
  [常に使用]
  ・FS_AttachOverlayTable 関数
  [ロード処理を同期的に行う場合のみ使用]
  ・FS_LoadOverlay 関数
  [ロード処理を非同期的に行う場合のみ使用]
  ・FS_LoadOverlayInfo 関数
  ・FS_LoadOverlayImage 関数または FS_LoadOverlayImageAsync 関数
  ・FS_StartOverlay 関数

2.3.4 DS ダウンロードプレイの不具合について

IPL 上の DS ダウンロードプレイ機能は、いくつかの不具合が存在します。詳しい症状、対策について以下に示します。

2.3.4.1 DSダウンロードプレイの不具合（1）

症状

DS ダウンロードプレイの子機 A のダウンロードが完了し、ブート処理を行っている最中に、DS ダウンロードプレイの子機 B がダウンロードを開始すると、子機 B がフリーズする場合があります。（発生頻度：低）

対策

ゲームアプリ側で以下の処理を行うことで、問題の発生頻度を下げることができます。（完全な対策ではありません）

1. NITRO-SDK2.0RC2 以降の SDK を導入する。
2. ダウンロードが許可されていない期間にダウンロード要求を送ってきた子機をキックする際には、関数 MB_DisconnectChild を使用して、子機側から接続を解除させる。

上記対策を実装済みのデモプログラムが $NitroSDK/build/demos/mb/multiboot-Model に存在します。

2.3.4.2 DS ダウンロードプレイの不具合（2）

症状

DS ダウンロードプレイの子機のダウンロードが完了し、親機が子機にブート処理を送信している最中に、子機の接続がキャンセルされると、子機がフリーズする場合があります。（発生頻度：低）

対策

この不具合に対するゲームアプリ側での有効な対策はありません。

2.3.4.3 DS ダウンロードプレイの不具合（3）

症状

DS ダウンロードプレイの子機のダウンロードリストに、親機 A と親機 B の2つのゲームのバナーが表示されている時に、子機側で以下の手順を踏むと、選択していない親機のゲームがダウンロードされます。（発生頻度：毎回）

1. 親機 A を選択する。
2. 「このソフトをダウンロードしますか？」という画面まで進む。
3. 親機 A の電源を切る。
4. 1 分以上経過した後でダウンロードを開始する。
5. 親機 B のゲームがダウンロードされる。

対策

この不具合に対するゲームアプリ側での有効な対策はありません。

子機の画面はダウンロードの最終確認状態になると更新されなくなりますが、内部では親機リストが更新される為、画面表示と実際に選択されている親機が一致しなくなることが不具合の原因です。
なお、親機の電源が切られても、その親機情報は一定の間（1 分）リスト上に残るため、その間は不具合は発生しません。（親機が存在しないため、ダウンロードそのものは失敗します）

2.3.4.4 DS ダウンロードプレイの不具合（4）

症状

DS ダウンロードプレイの子機のダウンロードリストに、親機 A と親機 B の 2 つのゲームのバナーが表示された後で、タイムアウトにより 2 つのバナーが同時にリセットされると、再リストアップされた際にカーソルが消えてしまう場合がありま

す。この状態のままバナーを選択すると、ゲームタイトル情報のアイコン等が化け、ダウンロードも失敗してしまいます。（発生頻度：低）

対策

この不具合に対するゲームアプリ側での有効な対策はありません。

2.3.4.5 DS ダウンロードプレイの不具合（5）

症状

DS ダウンロードプレイの子機がダウンロードの最終確認画面（「このソフトをダウンロードしますか？」のメッセージが表示される画面）になってから、実際にダウンロードが行われている間、通信メンバーの人数及び名前の表示が更新されません。（発生頻度：毎回）

対策

ゲームアプリ側での有効な対策はありません。

DS 起動メニューの仕様と考えてください。

2.3.4.6 DS ダウンロードプレイの不具合（6）

症状

DS ダウンロードプレイの子機がゲームを選択している時に、親機側で以下の手順を踏むと、選択していない親機のゲームがダウンロードされます。（発生頻度：毎回）

1. 親機 A を選択する
2. 「このソフトをダウンロードしますか？」という画面まで進む
3. 親機 A は一度 DS ダウンロードプレイを終了し、再度、GGID のみ前回と同じだが内容の異なるゲームを配信する。
4. 3～4秒経過した後で子機がダウンロードを開始する。
5. 親機 A で新たに配信している方のゲームがダウンロードされる。

対策

この不具合に対しては、ゲームアプリが MB_RegisterFile 関数で登録する個々のゲームごとに異なる GGID を MBGameRegistry 構造体へ設定しておくことによって発生を回避することができます。

この問題は、DS ダウンロードプレイの不具合（3）と類似した原因により発生します。
子機側のダウンロード決定までにゲーム情報リストは内部で自動的に更新され続けていますが、受信したビーコンが既知のゲーム情報と一致するかどうかの判定は GGID と MAC アドレスで行われており、かつ、ゲーム情報が一致しつつ TGID が更新された場合には情報を再取得するよう実装されているため、同一親機が配信する GGID の同じゲームなら情報がすり替わってしまいます。
GGID が異なる場合には別個のゲームとしてリストに追加されるためこの問題は発生しません。

3 クローンブート機能

SDK では、親機の Static セグメントをそのまま子機へ送信して DS ダウンロードプレイ子機をブートするクローンブート機能が用意されています。 この章ではクローンブートの手順について説明します。

3.1 クローンブートについて

クローンブートを使用すると、子機には親機と同じ Static セグメントが配信されます。
親機及びブートされた子機は、MB_IsMultiBootChild 関数から自分が DS ダウンロードプレイ子機であるかどうかを取得して処理を分岐させることになります。

Static セグメントに含まれないデータは、ブート後に親機と再接続して WBT ライブラリ等を使用して取得する必要があります。

※ただし、「3.2 クローンブートの手順」で後述するようにStaticセグメントの一部は親機専用の領域になります。

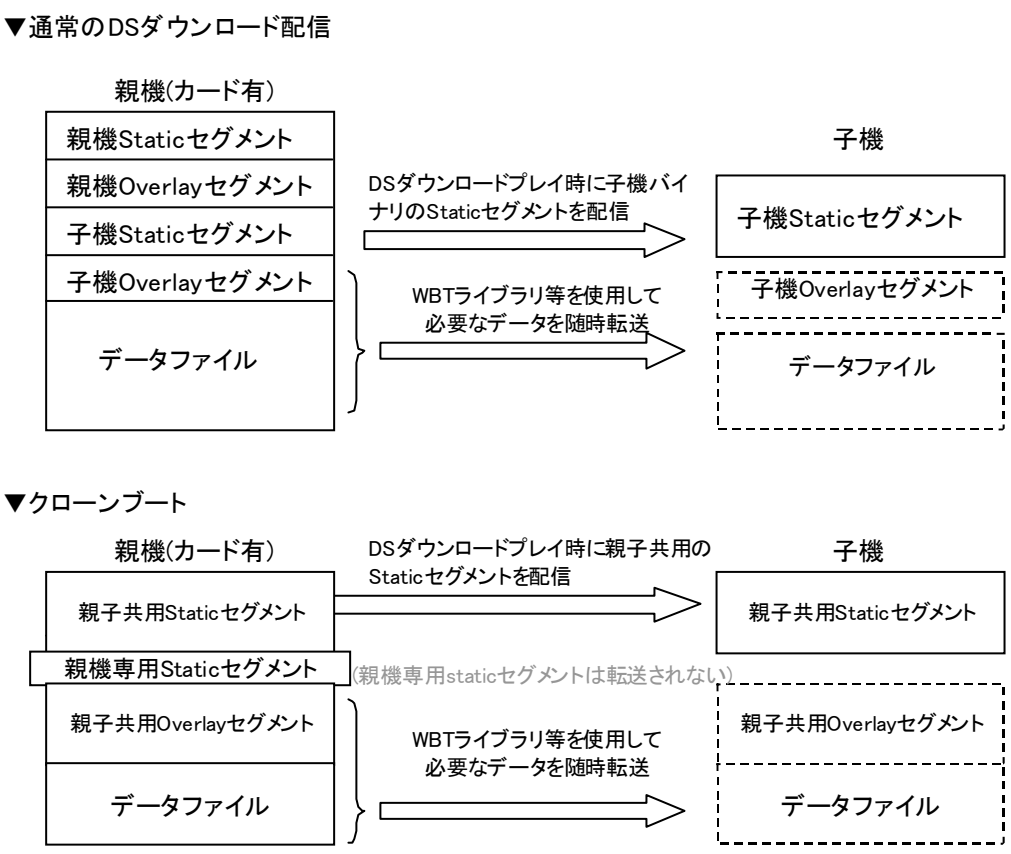

図 3-1 クローンブート

3.2 クローンブートの手順

クローンブートをおこなう為の手順を以下に示します。

3.2.1 ROM のデータ配置

クローンブート機能に対応したプログラムは、カードから起動する場合と同様に DS ダウンロードプレイ子機をブートさせることができます。そのため、配信されたデータからプログラム本体が完全に再現されることを避ける目的でマルチブートライブラリではデータ配置によるセキュリティ手段を提供しています。

クローンブート対応プログラムではカードのセキュア領域にあたる 0x5000－0x6FFFに配置されたデータが親機専用データとして扱われ、DSダウンロードプレイの配信データから除外されます。製品プログラムの再現と複製を防ぐセキュリティ確保のため、この領域には親機で必ず使用するが子機では使用しないデータを配置するようにしてください。この領域配置を設定する方法およびデータの配置方法については、「5 サンプルプログラム(cloneboot)の解説」で後述します。

カードのセキュア領域の詳細についてはプログラミングマニュアルを参照ください。

3.2.2 認証コードの付加

通常の DS ダウンロードプレイでは DS 実機で動作させる場合に、子機用バイナリに認証コードを付加する必要がありますが、クローンブートの場合にも同様に認証コードを付加する必要があります。

クローンブートの認証の為には、まず製品版の親機で使用する libsyscall.a の他にクローン子機用の libsyscall に対応するバイナリファイル（以下 libsyscall_c.bin と表記） を弊社より入手していただく必要があります。

ビルドして出来上がった srl ファイルに対して$NitroSDK/tools/bin/emuchild.exe を使用すると、DS ダウンロードプレイに必要な Static セグメントのみ切り出され子機用の libsyscall_c.bin が付加された状態の署名用バイナリファイルが出力されます。(以下 srl' と表記)。このファイルで通常の DS ダウンロードプレイ認証と同じ手順で署名手続きを行い、得られた認証コードを元の srl ファイルにアタッチしてください。

この際、ROM 作成時に RomFootPadding を指定してパディングをおこなっている場合には attachsign で適切な位置に署名が挿入されますので、署名を入れるだけの容量的な余裕がある場合には srl ファイルのサイズ増加は発生しません。

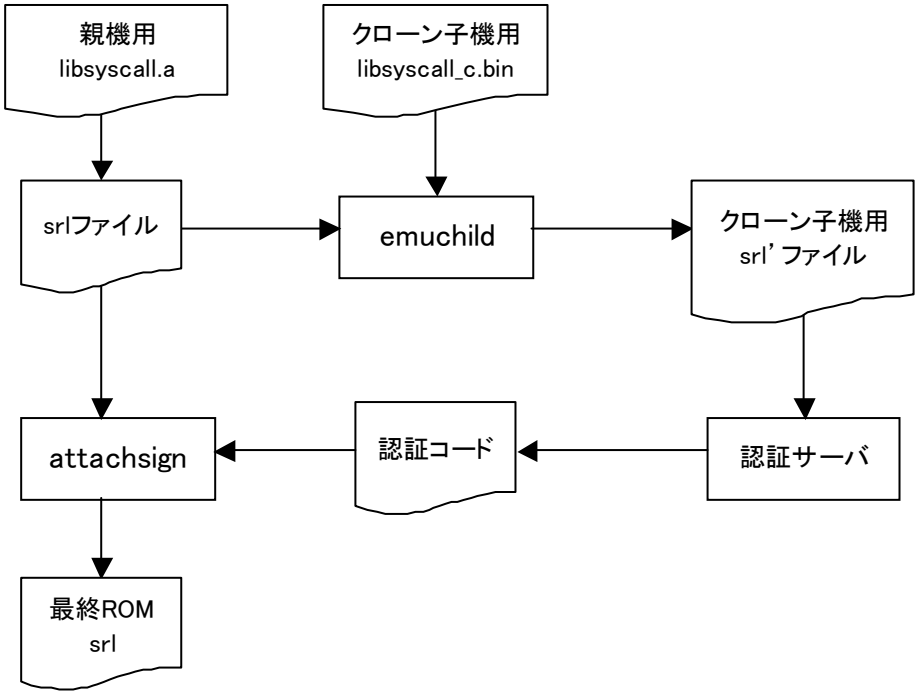

図 3-2 クローンブートバイナリ認証手順

3.2.3 クローンブートバイナリの登録

MB ライブラリで用意されている MB_GetSegmentLength 関数, MB_ReadSegment 関数では、引数に渡す子機バイナリファイルポインタとして NULL を渡すとクローンブートとして動作するようになっています。
その他は通常の DS ダウンロードプレイと全く同様の処理となります。

```
// クローンブート用データのセグメントサイズを取得
bufferSize = MB_GetSegmentLength( NULL );
if ( bufferSize == 0 )
{
    return FALSE;
}
// メモリを確保
sFilebuf = OS_Alloc( bufferSize );
if ( sFilebuf == NULL )
{
    return FALSE;
}
// セグメント情報を抽出
if ( ! MB_ReadSegment( NULL, sFilebuf, bufferSize ) )
{
    OS_Free( sFilebuf );
    return FALSE;
}
// ダウンロードプログラムの登録
if ( ! MB_RegisterFile( gameInfo, sFilebuf ) )
{
    OS_Free( sFilebuf );
    return FALSE;
}
```

表 1 クローンブートバイナリの登録例

4 サンプルプログラム（multiboot-Model）の解説

multiboot-Model サンプルでは、DS ダウンロードプレイ機能を利用して親機から子機用のプログラムを配信し、配信したプログラムで親機と子機間のデータシェアリング通信を行います。

この章では、親機について以下の項目を説明します。

1. DSダウンロードプレイ機能のための前準備
2. 親機の初期化
3. 親機の動作開始
4. 子機からの接続待ち受け
5. 子機用プログラムの配信
6. 子機の再起動
7. 親機アプリケーションの開始
8. 親機の状態

子機については以下の項目を説明します。

1. DSダウンロードプレイ子機判定処理
2. DSダウンロードプレイ時の接続情報取得
3. 子機アプリケーションの開始

サンプルプログラムでは、DS ダウンロードプレイの親機にとって必要とされる MB ライブラリ関連の一連の処理を $NitroSDK/build/demos/wireless_shared/mbp 以下にモジュール形式でまとめています。実際に DS ダウンロードプレイ機能を利用したプログラムを作成する際にご利用ください。ご利用の際にはワイヤレスマネージャのラッパーモジュールである wh.h が必要になりますので注意してください。（wh.h については別紙「ワイヤレス通信チュートリアル」をご覧ください。）

4.1 DS ダウンロードプレイ親機

DS ダウンロードプレイ機能の親機として必要な処理をサンプルプログラムの処理の流れに沿って説明します。

4.1.1 DS ダウンロードプレイ機能のための前準備

「2.1.1.1ワイヤレス通信チャンネルの選択」にありますように、DSダウンロードプレイ機能を利用するにあたって、マルチブート(MB)ライブラリの初期化処理の前に、空いている通信チャンネルの検索を行う必要があります。

以下に、通信チャンネル検索処理部分のプログラムを引用します(実際のサンプルプログラムから説明に不要なコメントなどは省略しています)。

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

        switch (WH_GetSystemState())
        {
            //---------------------------------------
            // 初期化完了
        case WH_SYSSTATE_IDLE:                                 ②
            (void)WH_StartMeasureChannel();
            break;
            //---------------------------------------
            // チャンネル検索完了
        case WH_SYSSTATE_MEASURECHANNEL:                       ③
            {
                sChannel = WH_GetMeasureChannel();
                (void)WH_End();
            }
            break;
            //---------------------------------------
            // WM終了
        case WH_SYSSTATE_STOP:                                 ④
            /* WM_Endが完了したらマルチブート処理へ */
            return;
            //---------------------------------------
            // ビジー中
        case WH_SYSSTATE_BUSY:
            break;
            //---------------------------------------
            // エラー発生
        case WH_SYSSTATE_ERROR:
            (void)WH_Reset();
            break;
            //---------------------------------------
        default:
            OS_Panic("Illegal State¥n");
        }
        OS_WaitVBlankIntr();          // Vブランク割込終了待ち
    }}
```

まず、①で WH_Initialize 関数を使用してワイヤレス通信機能の初期化を行います。WH_Initialize 関数はワイヤレス通信に必要な送受信バッファの確保と初期化、ワイヤレス通信ハードウェアを初期化したあとで、ワイヤレスマネージャ(WM)ライブラリの状態を IDLE 状態へ遷移させます。

WM ライブラリが IDLE 状態になった時点(②の状態)で、WM_MeasureChannel 関数を使用してチャンネルごとの電波使用率を調査することが可能になります。サンプルプログラムでは WH_StartMeasureChannel 関数を呼び出して、最も電波使用率の低いチャンネルを検索しています。

通信チャンネルの検索が完了した時点(③の状態)で、通信チャンネルの検索結果を WH_GetMeasureChannel 関数を使用して取得しています。通信チャンネルの検索が完了し、通信チャンネルの取得ができましたので WH_End 関数を使用して WM ライブラリの終了処理を呼び出します。MB ライブラリと WM ライブラリは同時に使用することができませんので、WM ライブラリをここで一旦終了させなければなりません。

WM ライブラリが終了処理を終えた時点(④の状態)で、通信チャンネル検索処理から抜け出し、DS ダウンロードプレイ処理へと移行します。

以降の手順で MB_StartParentFromIdle 関数を使用する場合には IDLE ステートまで遷移するだけでよいので、前述のプログラムを変更して以下のように③の状態で処理を抜けることになります。

```
        //----------------------------------------
        // チャンネル検索完了
        case WH_SYSSTATE_MEASURECHANNEL:              ③
            /* IDLE 状態を維持したままマルチブート処理へ */
            return;
        //----------------------------------------
        // WM終了
        ...
```

4.1.2 DS ダウンロードプレイ機能

取得した通信チャンネルを使用して、DS ダウンロードプレイ機能の初期化、子機の受け付け、ダウンロード配信、子機の再起動を行います。

4.1.2.1 親機の初期化

親機の初期化にはダウンロード配信するプログラムの情報やアイコン情報、GGID が登録されているマルチブートゲーム登録情報と通信チャンネル検索処理で取得した通信チャンネル、TGID を使用します。

想定しない子機から接続されるのを防ぐために、TGID は親機が起動される度に違う値を設定することを推奨します。

以下に、親機の初期化処理部分のプログラムを引用します(実際のサンプルプログラムから説明に不要なコメントなどは省略しています)。

```
static BOOL ConnectMain( u16 tgid )
{
    MBP_Init( mbGameList.ggid, tgid );              ①

    while ( TRUE )
    {
        ～ 省略 ～
    }
}
```

サンプルプログラムでは、親機の初期化と情報の設定を MBP_Init 関数で行っています(①)。MBP_Init 関数では子機の画面に表示する親機プレイヤー情報の設定と MB ライブラリの初期化を行っています。

```
void MBP_Init( u32 ggid, u16 tgid )
{
    /* 子機画面へ表示する親機情報設定用 */
    MBUserInfo      myUser;

    OSOwnerInfo info;

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

    myUser.playerNo = 0;        // 親機は0番                     ③

    // ステータス情報を初期化
    mbpState    = (const MBPState) { MBP_STATE_STOP, 0, 0, 0, 0, 0, 0 };

    /* MB 親機制御を開始します. */
    // MBワーク領域確保。
    sCWork = OS_Alloc( MB_SYSTEM_BUF_SIZE );                     ④

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

    MBP_ChangeState( MBP_STATE_IDLE );
}
```

MBP_Init関数では、親機プレイヤー情報にIPLオーナー情報から取得したプレイヤーのニックネームと好きな色を設定しています。 設定内容の詳細については「2.1.1.2親機パラメータの設定」を参照してください。

そして、MB ライブラリが使用するワーク領域を確保(④)し、MB_Init 関数を使用して MB ライブラリを初期化します。

⑤では MB ライブラリが通知する親機状態変化のコールバック関数を設定しています。このコールバック関数の中で、通知された親機状態に対する処理を行います。

以降の手順でMB_StartParentFromIdle関数を使用する場合には、「4.1.1 親機の初期化」で説明したようにIDLEステートが保持されていますので、前述のプログラムを変更して以下のように確保サイズを小さく出来ます。
(大きすぎるバッファがそのまま与えられても、特に問題はありません)

```
    // MBワーク領域確保。
    sCWork = OS_Alloc( MB_SYSTEM_BUF_SIZE - WM_SYSTEM_BUF_SIZE );      ④
    ...
```

4.1.2.2 親機の動作開始

MB_Init 関数によって MB ライブラリを初期化した後は、DS ダウンロードプレイ親機としての動作を開始し、ワイヤレスダウンロードに使用するファイルの登録を行います。

以下に、親機の動作開始処理部分のプログラムを引用します（実際のサンプルプログラムから説明に不要なコメントなどは省略しています）。

```
static BOOL ConnectMain( u16 tgid )
{
    ～ 省略 ～

    while ( TRUE )
    {
        switch ( MBP_GetState() )
        {
        //-----------------------------------------
        // アイドル状態
        case MBP_STATE_IDLE :                          ①
            {
                MBP_Start( &mbGameList, sChannel );
            }
            break;

    ～ 省略 ～

        }
    }
}
```

MBP_Init 関数の処理が完了（①の状態）すると、MBP_Start 関数を使用して DS ダウンロードプレイ機能を開始し、親機が子機からの接続を受け付けるようにしたのち、ワイヤレスダウンロードするプログラムの情報を登録しています。

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

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

    /* ------------------------------------------------------------------ *
     * MB_StartParent()コール時に初期化されてしまいます。
     * MB_RegisterFile()は必ずMB_StartParent()の後で登録してください。
     * ------------------------------------------------------------------ */

    /* ダウンロードプログラムファイル情報を登録します． */          ④
    if ( ! MBP_RegistFile( gameInfo ) )
    {
        OS_Panic("Illegal multiboot gameInfo¥n");
    }
}
```

通信チャンネルを引数にして MB_StartParent 関数を呼び出し(③)、DS ダウンロードプレイ親機としての動作を開始させます。

MB_StartParent 関数を呼び出した時にダウンロードプログラム情報が初期化されてしまいますので、必ず MB_StartParent 関数を呼び出した後に MB_RegisterFile 関数を使用してダウンロードプログラム情報の登録を行なう必要があります。

サンプルプログラムでは MBP_RegistFile 関数を呼び出して DS ダウンロードプレイで送信するバイナリコードをメモリにロードし、ダウンロードプログラム情報を登録しています。(④)

このサンプルプログラムで使用しているダウンロードプログラム情報は以下のように設定されています。

```
/* このデモがダウンロードさせるプログラム情報 */
const MBGameRegistry mbGameList =
{
    "/child.srl",               // 子機バイナリコード
    (u16*)L"DataShareDemo",     // ゲーム名
    (u16*)L"DataSharing demo", // ゲーム内容説明
    "/data/icon.char",          // アイコンキャラクタデータ
    "/data/icon.plt",           // アイコンパレットデータ
    WH_GGID,                    // GGID
    MBP_CHILD_MAX + 1,          // 最大プレイ人数
};
```

もしMB_StartParentFromIdle関数を使用する場合には、「4.1.1 親機の初期化」や「4.1.2 親機の動作開始」で説明した変更とあわせて、③の部分を以下のように変更します。

```
    MBP_ChangeState( MBP_STATE_ENTRY );
    if ( MB_StartParentFromIdle( channel ) != MB_SUCCESS )        ③
    {
    ...
```

次に MBP_RegistFile 関数での処理の流れに沿って、ダウンロードプログラム情報の登録処理を説明します。

まず、登録するダウンロードファイルを読み込むために、ファイルシステムを使用してファイルをオープンします。(⑤)
なおMBP_RegistFile関数はクローンブート機能(詳細は後述)にも対応しており、与えられたファイルパス名が NULL であればクローンブートを指定されたものとして振舞うようになっています。

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

    /*
     * この関数の仕様として, romFilePathp が NULL であれば
     * クローンブートとして動作します.
     * そうでなければ指定されたファイルを子機プログラムとして扱います.
     */
    if (!gameInfo->romFilePathp)
    {
        p_file = NULL;
    }
    else
    {
        /*
         * プログラムファイルは FS_ReadFile() で読み出せる必要があります.
         * 通常はプログラムを CARD-ROM 内にファイルとして保持するはずなので
         * 問題はありませんが, もし特殊なマルチブートシステムを想定する場合,
         * FSArchive で独自アーカイブを構築して対処してください.
         */
        FS_InitFile(&file);
        if (!FS_OpenFile(&file, gameInfo->romFilePathp))
        {
            /* ファイルが開けない, */
            OS_Warning("Cannot Register file¥n");
            return FALSE;
        }
        p_file = &file;
    }
    ～ 省略 ～
}
```

次に、MB_GetSegmentLength 関数を使用してセグメント情報のサイズを取得(⑥)し、セグメント情報を読み込むためのメモリを確保(⑦)します。

サンプルプログラムでは、セグメント情報を 1 ファイルのみ保持するようにしていますので、複数のダウンロードファイルを登録する場合にはセグメント情報を複数保持するような処理に変更する必要があります。

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

    ～ 省略 ～

    /*
     * セグメント情報のサイズを取得.
     * 正当なダウンロードプログラムでない場合,
     * このサイズに 0 が返ってきます.
     */
    bufferSize = MB_GetSegmentLength( p_file );          ⑥
    if ( bufferSize == 0 )
    {
        OS_Warning( "specified file may be invalid format.¥"%s¥"¥n",
                    gameInfo->romFilePathp );
    }
    else
    {
        /*
         * ダウンロードプログラムのセグメント情報を読み込むメモリを確保.
         * ファイルのレジストに成功した場合はMB_End()が呼ばれるまで
         * この領域が使用されます。
         * このメモリはサイズさえ充分であれば 静的に用意されていても構いません.
         */
        sFilebuf = (u8*)OS_Alloc( bufferSize );                  ⑦
        if ( sFilebuf == NULL )
        {
            /* セグメント情報を格納するバッファの確保失敗 */
            OS_Warning("can't allocate Segment buffer size.¥n");
        }
        else
        {
    ～ 省略 ～

}
```

最後に、セグメント情報を MB_ReadSegment 関数を使用してファイルから読み込み(⑧)、セグメント情報を MB_RegisterFile 関数を使用(⑨)して登録します。

ダウンロードファイルの登録が終了した時点で、オープンしていたダウンロードファイルは不要になりますのでクローズします。(⑩)

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

    ～ 省略 ～

            /*
            * セグメント情報をファイルから抽出.
             * 抽出したセグメント情報は，ダウンロードプログラム配信中
             * メインメモリ上に常駐させておく必要があります.
             */
            if ( ! MB_ReadSegment( p_file, sFilebuf, bufferSize ) )       ⑧
            {
                /*
                 * 不正なファイルによりセグメント抽出に失敗
                 * サイズ取得が成功したにも関わらずここで抽出処理が失敗するのは,
                 * ファイルハンドルに何か変更を与えた場合などです.
                 * (ファイルを閉じた，位置をシークした，...)
                 */
                OS_Warning(" Can't Read Segment¥n" );
            }
            else
            {
                /*
                 * 抽出したセグメント情報と MBGameRegistry で
                 * ダウンロードプログラムを登録します.
                 */
                if ( ! MB_RegisterFile( gameInfo, sFilebuf ) )          ⑨
                {
                    /* 不正なプログラム情報により登録失敗 */
                    OS_Warning(" Illegal program info¥n");
                }
                else
                {
                    /* 処理は正しく完了しました */
                    ret = TRUE;
                }
            }
            if (!ret)
                OS_Free(sFilebuf);
        }
    }
    /* クローンブートでなければファイルをクローズ */
    if (p_file == &file)
    {
        (void)FS_CloseFile(&file);
    }
    return ret;
}
```

以上で DS ダウンロードプレイの親機としての動作が開始され、登録したダウンロードファイルが子機へのダウンロード配信に使用されるようになります。

4.1.2.3 子機からの接続待ち受け

DS ダウンロードプレイの親機としての動作を開始した後は、子機からの接続要求に対する処理を行います。

子機からの接続要求は「4.1.2.1 親機の初期化」において、MB_CommSetParentStateCallback関数で設定したコールバック関数に通知されます。接続要求に限らず子機からのさまざまな通知がこのコールバック関数に通知されますので、それぞれの通知に対して適切な処理が必要です。

サンプルプログラムでは、子機からの接続要求を待ち受けている状態を「MBP_STATE_ENTRY(接続要求受付)」として定義しています。

また、親機の状態を示す MBP_GetState 関数の返り値が MBP_STATE_ENTRY でない場合には、子機からの接続要求を拒否しています。

子機からの接続要求は、親機に子機が接続された状態を示す「MB_COMM_PSTATE_CONNECTED」と、DS ダウンロードプレイの子機としてのエントリー要求を示す「MB_COMM_PSTATE_REQUESTED」の 2 つの状態からなります。

サンプルプログラムでは、子機から MB_COMM_PSTATE_CONNECTED が通知されたときに子機の接続管理情報(mbpState.connectChildBmp)を更新しています。

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------------
    // 子機からの接続がきた瞬間の通知
    case MB_COMM_PSTATE_CONNECTED:
        {
            // 親機がエントリー受付状態以外の場合の接続は受け付けない
            if ( MBP_GetState() != MBP_STATE_ENTRY )
            {
                break;
            }

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

コールバック関数に MB_COMM_PSTATE_REQUESTED が通知されたときには、エントリー要求を受け付ける(②)のか拒否する(①)のかを決定します。

サンプルプログラムでは親機の状態によるエントリー拒否以外には MBP_AcceptChild 関数ですべてのエントリー要求を受け付け、子機のエントリー要求管理情報(mbpState.requestChildBmp)を更新し、MB_CommGetChildUser 関数を使用して子機のプレイヤー情報を取得しています。

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------------
    // 子機からエントリーリクエストが来た瞬間の通知
    case MB_COMM_PSTATE_REQUESTED:
        {
            const MBUserInfo* userInfo;

            // 親機がエントリー受付状態でない場合にエントリー要求をしてきた
            // 子機は確認なしにキックする。
            if ( MBP_GetState() != MBP_STATE_ENTRY )
            {
                MBP_KickChild( child_aid );                                    ①
                break;
            }

            // 子機のエントリーを受け付ける
            mbpState.requestChildBmp |= 1 << child_aid;

            MBP_AcceptChild( child_aid );                                      ②

            // MB_COMM_PSTATE_CONNECTEDのタイミングではまだUserInfoがセットされていないので
            // MB_CommGetChildUserはREQUESTED以降の状態で呼ばないと意味がありません。
            userInfo = MB_CommGetChildUser( child_aid );
            if ( userInfo != NULL )
            {
                MI_CpuCopy8( userInfo, &childInfo[ child_aid - 1 ].user, sizeof(
MBUserInfo ) );
            }
            MBP_Printf("playerNo = %d¥n", userInfo->playerNo );
        }
        break;
    }
}
```

子機からの接続要求を受け付ける場合(①)は MB_COMM_RESPONSE_REQUEST_ACCEPT(接続許可応答)を、接続要求を拒否する場合(②)は MB_COMM_RESPONSE_REQUEST_KICK(接続拒否応答)を、それぞれ子機に MB_CommResponseRequest 関数を使用して通知します。

サンプルプログラムでは子機への通知とともに、子機の接続管理情報を更新しています。

```
void MBP_AcceptChild( u16 child_aid )                                    ①
{
    if ( ! MB_CommResponseRequest( child_aid, MB_COMM_RESPONSE_REQUEST_ACCEPT )
)
    {
        // リクエストに失敗した場合その子機を切断する。
        MBP_DisconnectChild( child_aid );
        return;
    }

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

void MBP_KickChild( u16 child_aid )                                      ②
{
    if ( ! MB_CommResponseRequest( child_aid, MB_COMM_RESPONSE_REQUEST_KICK ) )
    {
        // リクエストに失敗した場合その子機を切断する。
        MBP_DisconnectChild( child_aid );
        return;
    }

    {
        OSIntrMode enabled = OS_DisableInterrupts();

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

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

親機から MB_COMM_RESPONSE_REQUEST_KICK を通知された子機は親機との接続を切断します。

まず、親機には子機が接続拒否応答を受け取ったことを示す MB_COMM_PSTATE_KICKED がコールバック関数に通知されます。この通知は接続拒否応答の通知に対する確認通知です。

親機から MB_COMM_RESPONSE_REQUEST_ACCEPT を通知された子機はダウンロード配信を受信することができる状態へと遷移します。

まず、親機には子機が接続許可応答を受け取ったことを示す MB_COMM_PSTATE_REQ_ACCEPTED がコールバック関数に通知されます。この通知は接続許可応答の通知に対する確認通知です。

続いて、子機がダウンロード配信を受信可能な状態へと遷移したことを示す MB_COMM_PSTATE_WAIT_TO_SEND がコールバック関数に通知されます。この通知を受け取るまでに、その子機に対してデータ転送を開始しても正常に実行されません。

サンプルプログラムではMB_COMM_PSTATE_KICKEDとMB_COMM_PSTATE_ACCEPTEDが通知された場合には何もしませんが、MB_COMM_PSTATE_WAIT_TO_SENDが通知された場合には子機の接続管理情報を更新し、親機の状態によっては子機に対してダウンロード配信を開始しています。(MBP_StartDownload関数については「4.1.2.4 子機用プログラムの配信」で説明します)

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // 子機へのACCEPTに対するACKの通知
    case MB_COMM_PSTATE_REQ_ACCEPTED:
        // 特に処理は必要ない。
        break;
    //---------------------------------------------------------
    // 子機をキックした時の通知
    case MB_COMM_PSTATE_KICKED:
        // 特に処理は必要ない。
        break;
    //---------------------------------------------------------
    // 子機からのダウンロード要求を受けた時の通知
    case MB_COMM_PSTATE_WAIT_TO_SEND:
        {
            // 子機の状態をエントリー済み=ダウンロード待ちの状態へ変更
            // 割り込み中の処理なので特に割り込み禁止設定にせずに変更
            mbpState.requestChildBmp  &= ~( 1 << child_aid );
            mbpState.entryChildBmp    |= 1 << child_aid;

            // メインルーチンからMBP_StartDownload()がコールされるとデータ送信を開始する
            // すでにデータ送信状態に入ってしまっている場合はこの子機へもデータ送信を開始する。
            if ( MBP_GetState() == MBP_STATE_DATASENDING )
            {
                MBP_StartDownload( child_aid );
            }
        }
        break;
    }
}
```

接続処理中の待ち受け部分(①)では、ダウンロード配信を受信可能状態となった子機が存在した場合(③)には子機に対してダウンロード配信を開始する(④)ことができます。逆に、DS ダウンロードプレイ機能をキャンセルして終了する(②)こともできます。

```
static BOOL ConnectMain( u16 tgid )
{
while ( TRUE )
    {
        switch ( MBP_GetState() )
        {
        //---------------------------------------
        // 子機からのエントリー受付中
        case MBP_STATE_ENTRY :                                                    ①
            {
                BgSetMessage( PLTT_WHITE, " Now Accepting          ");

                if ( IS_PAD_TRIGGER( PAD_BUTTON_B ) )
                {
                // Bボタンでマルチブートキャンセル
                    MBP_Cancel();                                                 ②
                    break;
                }

                // エントリー中の子機が一台でも存在すれば開始可能とする
                if (MBP_GetChildBmp(MBP_BMPTYPE_ENTRY) ||
                    MBP_GetChildBmp(MBP_BMPTYPE_DOWNLOADING) ||
                    MBP_GetChildBmp(MBP_BMPTYPE_BOOTABLE))                        ③
                {
                    BgSetMessage( PLTT_WHITE, " Push START Button to start   ");

                    if ( IS_PAD_TRIGGER( PAD_BUTTON_START ) )
                    {
                        // ダウンロード開始
                        MBP_StartDownloadAll();                                   ④
                    }
                }
            }
            break;
        }
    }
}
```

4.1.2.4 子機用プログラムの配信

親機は MB_COMM_PSTATE_WAIT_TO_SEND が通知された時点で、通知してきた子機へのダウンロード配信を開始することができます。ダウンロード配信の開始には MB_CommStartSending 関数または MB_CommStartSendingAll 関数を使用します。MB_CommStartSendingAll 関数を使用する場合には接続中の子機がすべてダウンロード配信を受信可能になってから使用してください。
MB_COMM_PSTATE_WAIT_TO_SEND 状態になっていない子機に対しては配信が開始されませんので、。MB_CommStartSendingAll 関数実行後に遅れて MB_COMM_PSTATE_WAIT_TO_SEND 通知を受けた場合は、個別にダウンロード配信を開始してください。

サンプルプログラムでは MBP_StartDownload 関数の中で MB_CommStartSending 関数を使用しています。また、ダウンロード配信を開始した子機の接続状態も更新しています。

```
void MBP_StartDownload( u16 child_aid )
{
    if ( ! MB_CommStartSending( child_aid ) )
    {
        // リクエストに失敗した場合その子機を切断する。
        MBP_DisconnectChild( child_aid );
        return;
    }

    {
        OSIntrMode enabled = OS_DisableInterrupts();

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

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

MBP_StartDownloadAll 関数では、親機の状態を MBP_STATE_DATASENDING（ダウンロード配信中）へと遷移させて接続要求の受け付けを終了（①）し、子機の接続状態を調べてダウンロード配信を受信することができる子機に対しては MBP_StartDownload 関数を使用してダウンロード配信を開始します（④）。接続要求を受け付けたがダウンロード配信を受信することができる状態になっていない子機に対しては、後で受信可能な状態になった時点でダウンロード配信を開始します（②）。

その他の状態にある子機に対しては切断処理を行います。（③）

```
void MBP_StartDownloadAll( void )
{
    u16 i;

    // エントリー受付終了
    MBP_ChangeState( MBP_STATE_DATASENDING );                          ①

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

        if ( mbpState.requestChildBmp & (1 << i) )                     ②
        {
            // 現在エントリー中の子機は後で準備ができてMB_COMM_PSTATE_WAIT_TO_SEND通知
            // を受けた時に処理をする
            continue;
        }

        // エントリー中でない子機は切断する
        if ( ! ( mbpState.entryChildBmp & (1 << i) ) )                 ③
        {
            MBP_DisconnectChild( i );
            continue;
        }

        // エントリー中の子機はダウンロード開始
        MBP_StartDownload( i );                                        ④
    }
}
```

MB_CommStartSending 関数は内部で、子機に対して MB_COMM_RESPONSE_REQUEST_DOWNLOAD（配信開始応答）を通知しています。子機が配信開始応答通知を受け取り、ダウンロード配信が開始されると親機には確認通知として MB_COMM_PSTATE_SEND_PROCEED がコールバック関数に通知されます。そして、子機へのダウンロード配信が完了した時点でコールバック関数には MB_COMM_PSTATE_SEND_COMPLETE が通知されます。

サンプルプログラムでは MB_COMM_PSTATE_SEND_PROCEED が通知された場合には何もしませんが、MB_COMM_PSTATE_SEND_COMPLETE が通知された場合には子機の接続管理情報を更新しています。

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //------------------------------------------------------------
    // 子機へバイナリ送信を開始する時の通知
    case MB_COMM_PSTATE_SEND_PROCEED:
        // None.
        break;
    //------------------------------------------------------------
    // 子機へのバイナリ送信が終了した時の通知
    case MB_COMM_PSTATE_SEND_COMPLETE:
        {
            // 割り込み中の処理なので特に割り込み禁止設定にせずに変更
            mbpState.downloadChildBmp &= ~( 1 << child_aid );
            mbpState.bootableChildBmp |= 1 << child_aid;
        }
        break;
    }
}
```

4.1.2.5 子機の再起動

子機へのダウンロード配信が完了した時点で、子機の再起動を行うことができます。子機の再起動を行うことができるかどうかは MB_CommIsBootable 関数を使用して確認します。

サンプルプログラムでは MBP_IsBootableAll 関数を使用して接続されている子機がすべて再起動を行うことができる状態であるかどうかを確認しています。

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

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

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

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

すべての子機がダウンロードを完了していれば、子機に対して再起動要求を送ります。

```
static BOOL ConnectMain( u16 tgid )
{
    ～ 省略 ～

    while ( TRUE )
    {
        //---------------------------------------
        // プログラム配信処理
        case MBP_STATE_DATASENDING :
            {

                // 全員がダウンロード完了しているならばスタート可能.
                if ( MBP_IsBootableAll() )
                {
                    // ブート開始
                    MBP_StartRebootAll();
                }
            }
            break;

    ～ 省略 ～
        }
    }
}
```

子機への再起動要求は MB_CommBootRequest 関数または MB_CommBootRequestAll 関数を使用して行います。MB_CommBootRequestAll 関数を使用する場合は接続されているすべての子機がダウンロードを完了していることを確認してから使用してください。

サンプルプログラムでは MBP_StartRebootAll 関数を使用して子機の再起動要求を行っています。MBP_StartRebootAll 関数内では子機の接続状態を確認して MB_CommBootRequest 関数を使用しています。そして、親機の状態を MBP_STATE_REBOOTING（子機再起動待ち）へと遷移させています。

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

    for ( i = 1; i < 16; i++ )
    {
        if ( ! ( mbpState.bootableChildBmp & (1 << i) ) )
        {
            continue;
        }
        if ( ! MB_CommBootRequest( i ) )
        {
            // リクエストに失敗した場合その子機を切断する。
            MBP_DisconnectChild( i );
            continue;
        }
        sentChild |= ( 1 << i );
    }

    // 接続子機が0になったらエラー終了
    if ( sentChild == 0)
    {
        MBP_ChangeState( MBP_STATE_ERROR );
        return;
    }

    //子機の再起動待ち状態へ遷移します。
    MBP_ChangeState( MBP_STATE_REBOOTING );
}
```

MB_CommBootRequest 関数は内部で、子機に対して MB_COMM_RESPONSE_REQUEST_BOOT（再起動要求）を通知しています。子機が再起動要求通知を受け取り、再起動処理が完了するとコールバック関数に MB_COMM_PSTATE_BOOT_STARTABLE が通知されます。

そして、子機が再起動処理を完了した時点で親機との間のワイヤレス通信が切断されているため、コールバック関数には MB_COMM_PSTATE_DISCONNECTED が通知されます。

サンプルプログラムでは MB_COMM_PSTATE_BOOT_STARTABLE が通知された場合には子機の接続管理情報の更新を行い、すべての子機が再起動を完了した時点で MB_End 関数を使用して DS ダウンロードプレイ機能を終了させています。

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------
    // 子機に対してブートが正しく完了した時の通知
    case MB_COMM_PSTATE_BOOT_STARTABLE:
        {
            // 割り込み中の処理なので特に割り込み禁止設定にせずに変更
            mbpState.bootableChildBmp &= ~( 1 << child_aid );
            mbpState.rebootChildBmp |= 1 << child_aid;

            // 全子機がブート完了した場合には親機も再接続処理に入る
            if ( mbpState.connectChildBmp == mbpState.rebootChildBmp )
            {
                MBP_Printf("call MB_End()¥n");
                MB_End();
            }
        }
        break;
    //---------------------------------------------------------
    // 子機が接続を切った時の通知
    case MB_COMM_PSTATE_DISCONNECTED:
        {
            // 子機のリブート以外の条件で切断された場合はエントリーを削除します。
            if ( MBP_GetChildState( child_aid ) != MBP_CHILDSTATE_REBOOT )
            {
                MBP_DisconnectChildFromBmp( child_aid );
            }
        }
        break;
    }
}
```

これまでの手順で MB_StartParentFromIdle 関数を使用するように変更した場合は、終了にあたって MB_End 関数のかわりに MB_EndToIdle 関数を呼び出すよう、以下のように変更します。

```
            // 全子機がブート完了した場合には親機も再接続処理に入る
            if ( mbpState.connectChildBmp == mbpState.rebootChildBmp )
            {
                MBP_Printf("call MB_EndToIdle()¥n");
                MB_EndToIdle();
            }
            ...
```

DS ダウンロードプレイ機能を終了させるとコールバック関数には MB_COMM_PSTATE_END が通知されます。

サンプルプログラムでは親機の状態を MBP_STATE_COMPLETE(処理完了)へと遷移させ、ダウンロード配信に使用していたワーク領域を開放しています。

```
static void ParentStateCallback( u16 child_aid, u32 status, void* arg )
{
    switch ( status )
    {
    //---------------------------------------------------------------
    // マルチブート終了時の通知
    case MB_COMM_PSTATE_END:
        {
            if ( MBP_GetState() == MBP_STATE_REBOOTING )
            // リブート処理が完了、MBを終了して子機と再接続を行なう
            {
                MBP_ChangeState( MBP_STATE_COMPLETE );
            }
            else
            // シャットダウン完了、STOP状態へ戻す
            {
                MBP_ChangeState( MBP_STATE_STOP );
            }
            // ゲーム配信用に使用していたバッファを解放
            // MB_COMM_PSTATE_ENDのコールバックが帰ってきた時点で
            // ワークは解放されているのでFreeしてしまってよい。
            if ( sFilebuf )
            {
                OS_Free( sFilebuf );
                sFilebuf = NULL;
            }
            if ( sCWork )
            {
                OS_Free( sCWork );
                sCWork = NULL;
            }
            /* MB_Endを呼びworkを解放することで登録情報も同時にクリアされるため *
              * MB_UnregisterFileは省略することができます。                    *
              */
        }
        break;
    }
}
```

最後に、再起動中(①)から DS ダウンロードプレイ機能終了(②)部分を引用します。

```
static BOOL ConnectMain( u16 tgid )
{
    ～ 省略 ～

    while ( TRUE )
    {
        //---------------------------------------
        // リブート処理
        case MBP_STATE_REBOOTING:                                    ①
            {
                BgSetMessage( PLTT_WHITE, " Rebooting now             ");
            }
            break;

        //---------------------------------------
        // 再接続処理
        case MBP_STATE_COMPLETE :                                    ②
            {
                // 全員無事に接続完了したらマルチブート処理は終了し
                // 通常の親機としてワイヤレス通信を再起動する。
                BgSetMessage( PLTT_WHITE, " Reconnecting now          ");

                OS_WaitVBlankIntr();
                return TRUE;
            }
            break;

    ～ 省略 ～
        }
    }
}
```

4.1.3 親機アプリケーションの開始

multiboot-Model サンプルでは DS ダウンロードプレイ機能を使用した子機へのダウンロード配信と子機の再起動の次に、ワイヤレス通信の親機としてダウンロード配信した子機用プログラムとのデータシェアリングを行います。

DS ダウンロードプレイ機能を終了した時点で子機とのワイヤレス通信の接続は切断されてしまうため、再度子機との間でワイヤレス通信の接続を確立しなければなりません。

サンプルのデータシェアリングでは DS ダウンロードプレイ機能で使用していた接続情報を用いています。

まず、最初にデータシェアリングに必要な初期化処理を行います。

GInitDataShare 関数でデータシェアリング通信に使用するバッファの初期設定を行います。

WH_Initialize 関数を使用して WM ライブラリを初期化し、ワイヤレス通信の初期化を行います。
これまでの手順で MB_StartParentFromIdle 関数を使用するよう変更している場合、ここでは IDLE ステートが保持されており WH_Initialize 関数もすでに呼ばれた状態になっていますので、呼び出す必要はありません。

```
// データシェアリング通信用にバッファを設定
GInitDataShare();
// MB_StartParent / MB_End を使用した場合にはここでワイヤレス通信の初期化
(void)WH_Initialize();
```

次に、ワイヤレス通信を開始すると DS ダウンロードプレイ機能で子機用プログラムを配信した子機以外の端末が接続を要求してくることがありますので、WH_SetJudgeAcceptFunc 関数を使用して接続許可を判定する関数を設定します。

```
// 接続子機の判定用関数を設定
WH_SetJudgeAcceptFunc( JudgeConnectableChild );
```

判定用関数 JudgeConnectableChild 関数は以下のようになっています。

①で接続してきた端末の MAC アドレスから DS ダウンロードプレイ時のプレイヤー番号(AID)を取得できた場合には接続を許可しています。

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

    /*  cb->aid の子機のマルチブート時のAIDをMACアドレスから検索します */
    playerNo = MBP_GetPlayerNo( cb->macAddress );                    ①

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

    if ( playerNo == 0 )
    {
        return FALSE;
    }

    sChildInfo[ playerNo ] = MBP_GetChildInfo( playerNo );
    return TRUE;
}
```

最後に、親機としてワイヤレス通信を開始させ、データシェアリングを行います。

WH_Initialize 関数が完了した時点での状態は WH_SYSSTATE_IDLE(①)ですので、WH_ParentConnect 関数を使用してワイヤレス通信を開始します。その際に引数として、データシェアリングモードを示す「WH_CONNECTMODE_DS_PARENT」と DS ダウンロードプレイ機能で使用していた TGID と通信チャンネルを使用しています。

ワイヤレス通信が開始されると、状態は WH_SYSSTATE_DATASHARING(②)へと遷移し、データシェアリングが開始されます。

```
    /* メインルーチン */
    for ( gFrame = 0 ; TRUE ; gFrame++ )
    {
        OS_WaitVBlankIntr();

        ReadKey();

        BgClear();

        switch ( WH_GetSystemState() )
        {
        case WH_SYSSTATE_IDLE :                                    ①
            /* -----------------
             * 子機側で再スキャンなしに同じ親機に再接続させたい場合には
             * 子機側とtgid及びchannelを合わせる必要があります。
             * このデモでは、マルチブート時と同じchannelとマルチブート時のtgid+1を
             * 親子ともに使用することで、再スキャンなしでも接続できるようにしています。
             *
             * MACアドレスを指定して再スキャンさせる場合には同じtgid, channelでなくても
             * 問題ありません。
             * ---------------- */
            (void)WH_ParentConnect(WH_CONNECTMODE_DS_PARENT, (u16)(tgid + 1),
sChannel);
            break;

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

4.1.4 親機の状態

MBP_GetState 関数を使用して取得することのできる親機の状態一覧です。

MBP_GetState 関数の返り値	親機の状態
MBP_STATE_STOP	MBP_Cancel 関数などから MB_End 関数が呼び出され、DS ダウンロードプレイ機能が停止した状態です。
MBP_STATE_IDLE	MBP_Init 関数が完了し、MBP_Start 関数を呼び出して親機としての動作を開始することができます。
MBP_STATE_ENTRY	MBP_Start 関数が完了し、子機からの接続を待ち受けている状態です。この状態のときにのみ、子機からの接続を受け付けることができます。
MBP_STATE_DATASENDING	MBP_StartDownloadAll 関数を呼び出し、接続されている子機にダウンロード配信を開始している状態です。
MBP_STATE_REBOOTING	MBP_StartRebootAll 関数を呼び出し、接続されていた子機を再起動させている状態です。
MBP_STATE_COMPLETE	接続されていた子機のすべてが再起動要求を受け付け、MB_End 関数によって DS ダウンロードプレイ機能を終了した状態です。
MBP_STATE_CANCEL	MBP_Cancel 関数を呼び出した直後の状態です。
MBP_STATE_ERROR	何らかのエラーが発生した状態です。

表 4-1 親機状態の一覧

4.2 DS ダウンロードプレイ子機

DS ダウンロードプレイ子機としてのユーザプログラムは、親機から DS ダウンロードプレイのデータが転送されて再起動した後の状態から開始されます。この時親機との接続は完全に切断されています。

multiboot-Model サンプルでは、DS ダウンロードプレイ子機の判定処理と、DS ダウンロードプレイ時に使用していた接続情報の取得方法を説明します。

4.2.1 DS ダウンロードプレイ子機判定処理

DS ダウンロードプレイ機能を使用して起動されたかどうかを判定するには、MB_IsMultiBootChild 関数を使用します。

```
// 自分がマルチブートから起動した子機であるかどうかをチェックします。
if ( ! MB_IsMultiBootChild() )
{
    OS_Panic("not found Multiboot child flag!¥n");
}
```

4.2.2 DS ダウンロードプレイ時の接続情報取得

DS ダウンロードプレイ時の接続情報は MB_ReadMultiBootParentBssDesc 関数を使用して取得することができます。取得した WMBssDesc を使用して直接親機との接続を行う場合には、取得の際にキーシェアリングフラグなどの設定を親機の設定と同じにしなければなりません。

```
MB_ReadMultiBootParentBssDesc( &gMBParentBssDesc,
                               WH_PARENT_MAX_SIZE,  // 親機最大送信サイズ
                               WH_CHILD_MAX_SIZE,   // 子機最大送信サイズ
                               0,                  // キーシェアリングフラグ
                               0 );                 // 連続転送モードフラグ
```

4.2.3 子機アプリケーションの開始

ワイヤレス通信の子機として親機とのデータシェアリングを行います。

まず、最初にデータシェアリングに必要な初期化処理を行います。

親機での処理と同様に、GInitDataShare 関数でデータシェアリング通信に使用するバッファの初期設定を行い、WH_Initialize 関数を使用して WM ライブラリを初期化し、ワイヤレス通信の初期化を行います。

```
GInitDataShare();

//*******************************
// ワイヤレス通信初期化
(void)WH_Initialize();
//*******************************
```

次に、メインループ内でリトライをしながら親機との接続を試みます。(①)

ワイヤレス通信が開始されると、状態は WH_SYSSTATE_DATASHARING(②)へと遷移し、データシェアリングが開始されます。

```
// メインループ
for ( gFrame = 0 ; TRUE ; gFrame ++ )
{
    // 通信状態により処理を振り分け
    switch( WH_GetSystemState() )
    {
    case WH_SYSSTATE_CONNECT_FAIL:
        {
            // WM_StartConnect()に失敗した場合にはWM内部のステートが不正になっている為
            // 一度WM_ResetでIDLEステートにリセットする必要があります。
            WH_Reset();
        }
        break;
    case WH_SYSSTATE_IDLE:
        {
            static retry = 0;
            enum {
                MAX_RETRY = 5
            };

            if ( retry < MAX_RETRY )
            {
                ModeConnect();                                    ①
                retry++;
                break;
            }
            // MAX_RETRYで親機に接続できなければERROR表示
        }
    case WH_SYSSTATE_ERROR:
        ModeError();
        break;
    case WH_SYSSTATE_BUSY:
    case WH_SYSSTATE_SCANNING:
        ModeWorking();
        break;
    case WH_SYSSTATE_CONNECTED:
    case WH_SYSSTATE_KEYSHARING:
    case WH_SYSSTATE_DATASHARING:                                 ②
        {
            BgPutString( 8 , 1 , 0x2 , "Child mode" );
            GStepDataShare( gFrame );
            GMain();
        }
        break;
    }
}
```

親機との接続は ModeConnect 関数内で WH_ChildConnect 関数を使用して行われています。その際に引数として、データシェアリングモードを示す「WH_CONNECTMODE_DS_CHILD」と DS ダウンロードプレイ機能で使用していたワイヤレス通信接続情報「gMBParentBssDesc」を使用しています。

ダウンロード後の再接続に際して何らかのアプリケーション固有情報を親機から受信する必要がある場合には、
親機からビーコンのゲーム情報として子機へ通知し子機はそれを再スキャンすることで実現することが出来ます。
そうでない場合は単に gMBParentBssDesc を使用して再接続するだけで問題ありません。
ModeConnect 関数では USE_DIRECT_CONNECT スイッチで切り分けた両者のコードを収録していますので、
用途に応じていずれかを選択ください。(デフォルトでは単に再接続する方式を採用しています)

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

    // 親機の再スキャンなしに直接接続する場合。
#ifdef USE_DIRECT_CONNECT
    //*******************************
    (void)WH_ChildConnect(WH_CONNECTMODE_DS_CHILD, &gMBParentBssDesc);
    // WH_ChildConnect(WH_CONNECTMODE_MP_CHILD, &gMBParentBssDesc, TRUE);
    // WH_ChildConnect(WH_CONNECTMODE_KS_CHILD, &gMBParentBssDesc, TRUE);
    //*******************************
#else
    WH_SetGgid(gMBParentBssDesc.gameInfo.ggid);
    // 親機の再スキャンを実行する場合。
    //*******************************
    (void)WH_ChildConnectAuto(WH_CONNECTMODE_DS_CHILD,
gMBParentBssDesc.bssid,
                              gMBParentBssDesc.channel);
    // WH_ChildConnect(WH_CONNECTMODE_MP_CHILD, &gMBParentBssDesc, TRUE);
    // WH_ChildConnect(WH_CONNECTMODE_KS_CHILD, &gMBParentBssDesc, TRUE);
    //*******************************
#endif
}
```

5 サンプルプログラム（cloneboot）の解説

cloneboot サンプルでは、「3. クローンブート機能」で示したクローンブート機能を使用し、DS ダウンロードプレイ親機として自身のプログラムを配信し、ダウンロード子機とデータシェアリング通信を行います。

このサンプルは、「4. サンプルプログラム（multiboot-Model）の解説」で説明された multiboot-Model サンプルの親機・子機両方の既存プログラムを統合するという形式でクローンブート機能対応プログラムの作成手順を示します。multiboot-Model サンプル自体の一連の詳細手順については前章を参照ください。

この章では、プログラム構成の変更について以下の項目を説明します。

1. プログラムソースディレクトリの統合
2. ROM スペックファイルの変更
3. makefile の変更
4. 認証コード付加用のビルド手順を追加

また、プログラムソースの変更について以下の項目を説明します。

1. メインエントリ名の変更
2. 新しいメインエントリの追加
3. 親機専用領域の指定
4. バイナリ登録処理の修正

5.1 プログラム構成の変更

以下に、クローンブート機能対応のプログラムにとって必要な変更作業の手順と内容を説明します。

5.1.1 プログラムソースディレクトリの統合

multiboot-Model サンプルでは先に子機プログラムを生成してから親機プログラムがそれをファイルとして含むために2個のビルドプロジェクトから構成されていました。クローンブート対応プログラムでは親子の区別を実行時に行うため、単一のプロジェクトへ統合することが出来ます。

ここでは、parent, child, common の各ディレクトリに含まれる src および include ディレクトリとその内容をすべてプロジェクトのルートディレクトリへ移動します。その際に、親機プログラムと子機プログラムにそれぞれ存在する main.c については、parent.c, child.c と名前を変更します。(新しい main.c は後の手順で作成します)

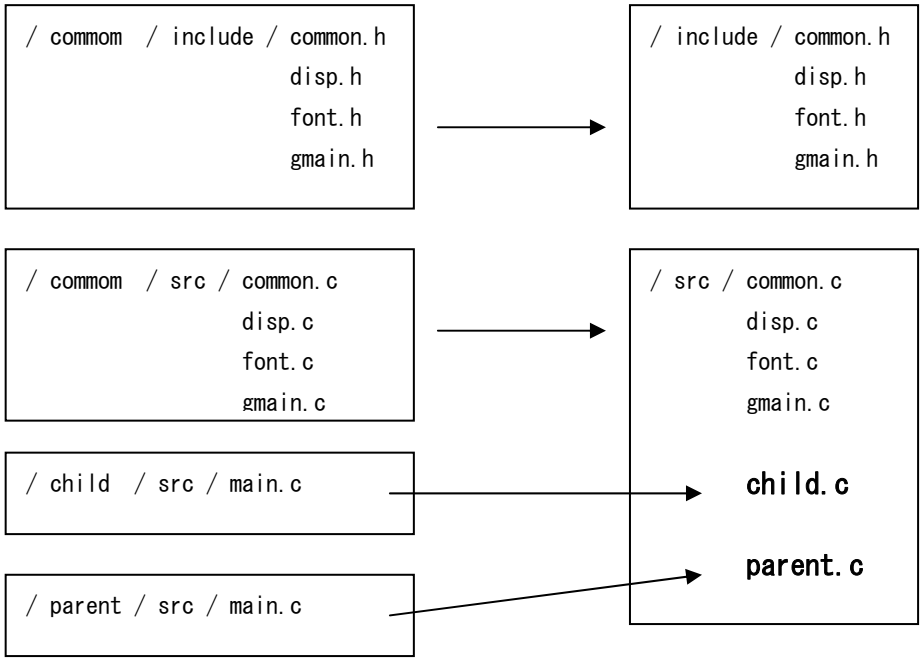

図 5-1 ソースディレクトリ統合

5.1.2 ROM スペックファイルの変更

multiboot-Model サンプルでファイルシステムに含めていた子機プログラムはクローンブート機能対応によって無くなりますので、main.rsf ファイル内での以下の記述を削除します。

```
# この指定を削除する.
#       HostRoot $(MAKEROM_ROMROOT)
#       Root            /
#       File            $(MAKEROM_ROMFILES)
```

5.1.3 makefile の変更

親機プログラムと子機プログラムを統合してクローンブート機能対応プログラムへ変更するため、
親機プログラムの makefile を主体にして以下に示すいくつかの変更と追加を行います。
なお、子機プログラムのビルドに使用していた makefile は不要になります。

5.1.3.1 ディレクトリとソース指定の修正

すでに「5.1.1 プログラムソースディレクトリの統合」で行ったディレクトリ構成の変更を正しくmakefileに反映します。
また、ファイル名を変更した親機・子機双方のメインソースをプロジェクトに追加します。

```
# 子機プログラムのビルドは不要となったのでサブビルド指定を削除する.
# SUBDIRS                 =       child
...

# 統合した新しいディレクトリを参照指定する.
SRCDIR          = ./src
INCDIR          = ./include
...

# ファイル名を変更した 2 つの main.c (parent.c および child.c)をビルドソースに追加する
SRCS            =       main.c      ¥
                        common.c    ¥
                        disp.c      ¥
                        font.c      ¥
                        gmain.c
SRCS            +=      parent.c    child.c
```

5.1.3.2 クローンブート用 LCF テンプレートファイルの指定

クローンブート機能対応プログラムは、「3.2.1 ROMのデータ配置」で述べたように親機専用領域を確保する必要があります。クローンブート用にROM配置を設定されたLCFテンプレートファイルが
$NitroSDK/include/nitro/specfiles/ARM9-TS-cloneboot-C.lcf.template
にあり、これを明示的に指定する必要があります。

```
# クローンブート用リンク設定テンプレートを指定する.
LCFILE_TEMPLATE     = $(NITRO_SPECDIR)/ARM9-TS-cloneboot-C.lcf.template
```

5.1.3.3 認証コード付加用のビルド手順を追加

クローンブート機能対応プログラムは、「3.2.2 認証コードの付加」 で述べたように認証コードの取得に関して通常のDSダウンロードプレイプログラムとは異なる手順を含みます。
emuchildツールを使用して署名コード取得用のバイナリを生成する手順は以下の通りです。

```
# 製品版アプリケーションのために弊社サポートより
# 配布させていただいている libsyscal.a と 対応バイナリ libsyscall_child.bin を指定
LIBSYSCALL                  = ./ etc / libsyscall.a
LIBSYSCALL_CHILD            = ./ etc / libsyscall_child.bin

# すでにビルド済みの状態から、 emuchild ツールで送信用バイナリを生成する手順.
# 生成された bin / sign.srl を、認証コード生成サーバへ送信する.
presign:
    $(EMUCHILD)     ¥
        bin / $(NITRO_BUILDTYPE) / $(TARGET_BIN)     ¥
        $(LIBSYSCALL_CHILD)     ¥
        bin / sign.srl

# 得られた認証コードをバイナリに含める手続きはクローンブートも通常と同様.
# ここでは認証コードを bin / sign.sgn としてバイナリ main_with_sign.srl を生成する.
postsign:
    $(ATTACHSIGN)     ¥
        bin / $(NITRO_BUILDTYPE) / $(TARGET_BIN)     ¥
        bin / sign.sgn     ¥
        main_with_sign.srl
```

なお、この記述は作業の利便性のために追加するものであり、コマンドラインから直接入力するのであれば特に makefile 内に追加する必要はありません。

5.1.4 プログラムソースの変更

前節で変更されたプログラム全体の構成に合わせて、いくつかのプログラムソースにも修正を施す必要があります。

5.1.4.1 メインエントリ名の変更

元は 2 個の main.c であった parent.c および child.c には、それぞれにメインエントリである NitroMain 関数を含んでいますので、これらの名前を適宜変更します。

```
child.c:

// 子機用メインエントリとして名前変更.
// void NitroMain( void )
void ChildMain( void )
{
    ...
```

```
parent.c:

// 親機用メインエントリとして名前変更.
// void NitroMain( void )
void ParentMain( void )
{
    ...
```

また、変更した名前での関数プロトタイプ宣言を common.h に追加しておきます。

```
common.h

// 元々は親機の NitroMain 関数.
void ParentMain( void );

// 元々は子機の NitroMain 関数.
void ChildMain( void );
```

5.1.4.2 新しいメインエントリの追加

名前を変更した親機メインエントリと子機メインエントリを呼び出す新しい NitroMain 関数を追加します。
クローンブート機能対応プログラムでは、 MB_IsMultiBootChild 関数の返り値から判断して親機用と子機用の処理を呼び分けるよう、おおよそ以下のように main.c を作成します。

main.c

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

void NitroMain( void )
{
    if( ! MB_IsMultiBootChild() )
    {
        ParentMain();
    }
    else
    {
        ChildMain();
    }
    /* 処理はここまで到達しない */
}
```

今回の例では、multiboot-Model からなるべく簡潔に移行することを目的としていますが、親子で同様の処理があるならば任意に共通化してかまいません。ただし、子機にはカードが存在しないという点を常に注意する必要があります。

5.1.4.3 親機専用領域の指定

「3.2.1 ROMのデータ配置」で説明した親機専用領域へ、クローンブートプログラムのコードの一部を含ませる必要があります。

親機専用領域とされているカード部分はセキュア領域なため、ROM ヘッダ等と同様、起動後には親機自身からも再び読み込むことができません。そのため、この領域に配置された変更可能データ（.bss セクションや.data セクション)は OS_ResetSystem 関数によるソフトウェアリセットなどの際に再初期化されないことに注意してください。

OS_ResetSystem 関数を使用する場合、C 言語においては次のいずれかの条件を満たすもののみが親機専用領域データとして使用可能です。

・定数

・内部に static 変数を持たない関数

・明示的な動的初期化処理をともなうグローバル変数（C++においてはコンストラクタをともなうオブジェクト)

また、含むべき内容は「親機にとって必須」でありながら「子機は一切使用しない」ものである必要があります。
この判断はプログラム本編と DS ダウンロードプレイによる配信版との間にどのような差別化を図るかというアプリケーション設計に全面的に依存するため一概に基準をもうけることができませんが、主な指針としてはプログラム本編のみプレイ可能とする機能への状態遷移をそのまま親機専用領域に含める方法が容易かつ有効です。

以上を踏まえ、今回のサンプルでは parent.c 内にある全ての関数を親機専用領域に含めるよう指定します。
領域の指定は、NITRO-SDK のインクルードファイル parent_begin.h および parent_end.h を使用して
以下のように記述します。

parent.c

```
...

//============================================================
//     関数定義
//============================================================

// ここから親機専用領域 .parent セクションの定義範囲を開始.
// 以下は、static 変数を内部に持たない関数しか存在しない.
#include <nitro/parent_begin.h>
void ParentMain( void )
{
...

}
// 親機専用領域 .parent セクションの定義範囲を終了.
#include <nitro/parent_end.h>
// ファイル終端.
```

ここで parent.c に含まれているものは「DS ダウンロードプレイ親機処理」の全てであり、親機専用領域に求められる「親機にとって必須」でありながら「子機は一切使用しない」という要件を適切に満たしています。

逆に親機専用領域の指定として好ましくないいくつかの代表的な例を、参考として以下に示します。

まず、呼び出さなくてもよい関数を配置しても、これを呼び出さないようコードを変更されれば容易に無効化されるので、セキュリティとしてあまり意味を成しません。

```
/*  親機専用領域に配置する関数. (デバッグ出力するのみ) */
void    no_use( void )
{
   OS_Printf( "called!¥n" );
}
...
void NitroMain( void )
{
   ...
   /*  親機ならこれを呼び出す. (呼び出さなくとも動作に何ら支障なし) */
   if( !MB_IsMultiBootChild( ) )    no_use( );
}
```

次に、親子ともに使用する関数を意図せず含んでしまうという事態も、必ず避ける必要があります。
これは「本編」と「配信プログラム」の差別化をゲーム内の品質面で行う場合に発生しえます。

```
/*  親機専用領域に配置する関数. (子機側では使用しないはずの画面演出) */
void    draw_special_effect_1000( void )
{
    ... /*  画面演出処理  */
}

/*  親子共通のゲーム処理*/
void    UpdateGameFrame( void )
{
   /*  想定と異なり、親子とも条件次第で呼び出される  */
   if( score >= 1000 ) draw_special_effect_1000( );
}
```

5.1.4.4 バイナリ登録処理の修正

マルチブートライブラリへバイナリを登録する処理をクローンブート用に変更する必要があります。
この手順は「3.2.3 クローンブートバイナリの登録」で述べたとおりです。

```
parent.c

...

const MBGameRegistry mbGameList =
{
    // MBP_Start 関数はパス名に NULL を与えるとクローンブートとして処理する.
    // 関数内部での具体的な処理については
    // $NitroSDK/build/demos/wireless_shared/mbp/mbp.c を参照.
    NULL,
    (u16*)L"DataShareDemo",        // ゲーム名
    (u16*)L"DataSharing demo(cloneboot)",    // ゲーム内容説明
...
}
```