3DS

3DS プログラミングマニュアル

システム編

2016-06-24

Version 1.6

目次

コード

表

図

1. はじめに

本ドキュメントは、3DS のアプリケーション開発者を対象に、各機能の概要、利用する関数、プログラミング手順などについて説明したものです。

「2. システム」では、ハードウェアブロック図および各機能の概要を説明しています。まず、ここでシステムの全体像を把握してください。

「3. メモリ、ニンテンドー3DS 専用ゲームカード」では、アプリケーションでアクセス可能なメモリ領域とカードアプリが記録されるニンテンドー3DS 専用ゲームカード(3DS カード)について説明しています。メモリ領域の種類やアクセス時の注意点、3DS カードのメモリマップなどをここで知ることができます。

「4. ソフトウェア構成」では、ソフトウェア構成について説明しています。ハードウェアとアプリケーション種別による動作の違いや、どのようなライブラリが SDK で用意されているかなどをここで知ることができます。

「5. アプリケーションの初期化と状態遷移のハンドリング」では、アプリケーションの実行に必要な初期化処理とスリープなどの状態遷移への対処について説明しています。OS、ファイルシステム、グラフィックスといった、アプリケーションの根幹部分で使用するライブラリの初期化手順と状態遷移のハンドリングをここで理解してください。

「6. 入力装置からの入力」では、キー入力、タッチパネル、加速度センサー、ジャイロセンサー、マイク、カメラなど、本体に搭載されている入力デバイスをアプリケーションで利用するための方法を説明しています。デバイスからの入力をアプリケーションで利用したい場合に参照してください。

「7. ファイルシステム」では、メディア上に存在するファイルやディレクトリにアクセスするための方法を説明しています。3DS カード内のファイルやバックアップメモリ、SD カード上のファイルを読み込んだり、書き込んだりしたい場合に参照してください。

「8. 時間」では、チック、タイマ、アラーム、RTC といった時間に関連する機能について説明しています。経過時間の計測や RTC を利用したい場合に参照してください。

「9. スレッド」では、スレッドや複数スレッドの同期を取るために利用することのできるクラスについて説明しています。スレッドの作成方法や複数スレッドでリソースを共有する方法などを知ることができます。

「10. サウンド」では、サウンドの再生に必要なライブラリの利用方法について説明しています。アプリケーションでサウンドを再生したい場合に参照してください。

「11. 本体設定」では、本体に設定されている情報へのアクセスに必要なライブラリの利用方法について説明しています。アプリケーションでユーザーの名前やサウンド出力モードなどの情報を取得したい場合に参照してください。

「12. アプレット」では、アプリケーションからアプレットの利用方法について説明しています。3DS で提供されているアプレットを利用したい場合に参照してください。

「13. 補助ライブラリ」では、3DS シリーズ特有の機能を使用するための、補助的なライブラリの利用方法について説明しています。歩数計の情報や内蔵されているリソースにアクセスしたい場合に参照してください。

「14. 本体間赤外線通信」では、赤外線通信モジュールを使用したライブラリの利用方法について説明しています。赤外線による本体間通信を利用したい場合に参照してください。

「15. TWL モードと TWL 本体の動作の違い」では、3DS で対応している TWL のエミュレーションモードと TWL 本体での動作の違いを説明しています。3DS 上でも動作するニンテンドーDS シリーズのアプリケーションを開発する際の注意点として参照してください。

補足: グラフィックス機能については、別冊の「3DS プログラミングマニュアル – グラフィックス基本編」および「3DS プログラミングマニュアル – グラフィックス応用編」を参照してください。

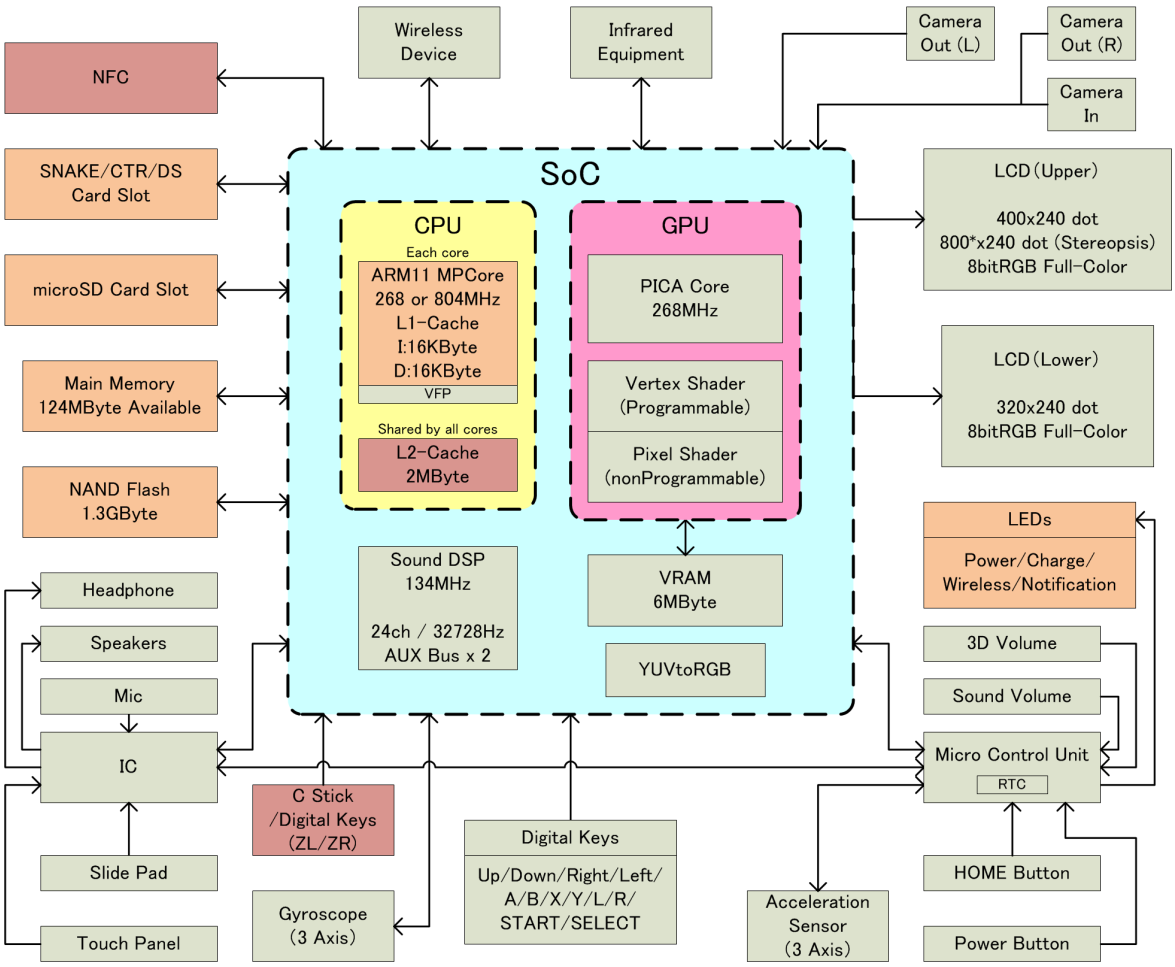
2. システム

この章では、SNAKE および CTR のシステムブロック全体図と各ハードウェアブロックの情報を記載しています。

2.1. システムブロック全体図(SNAKE)

SNAKE のシステム全体図を 図 2-1 に示します。

図 2-1. システムブロック全体図(SNAKE)



＊立体視時の 800 dot は左目用と右目用の合計です。

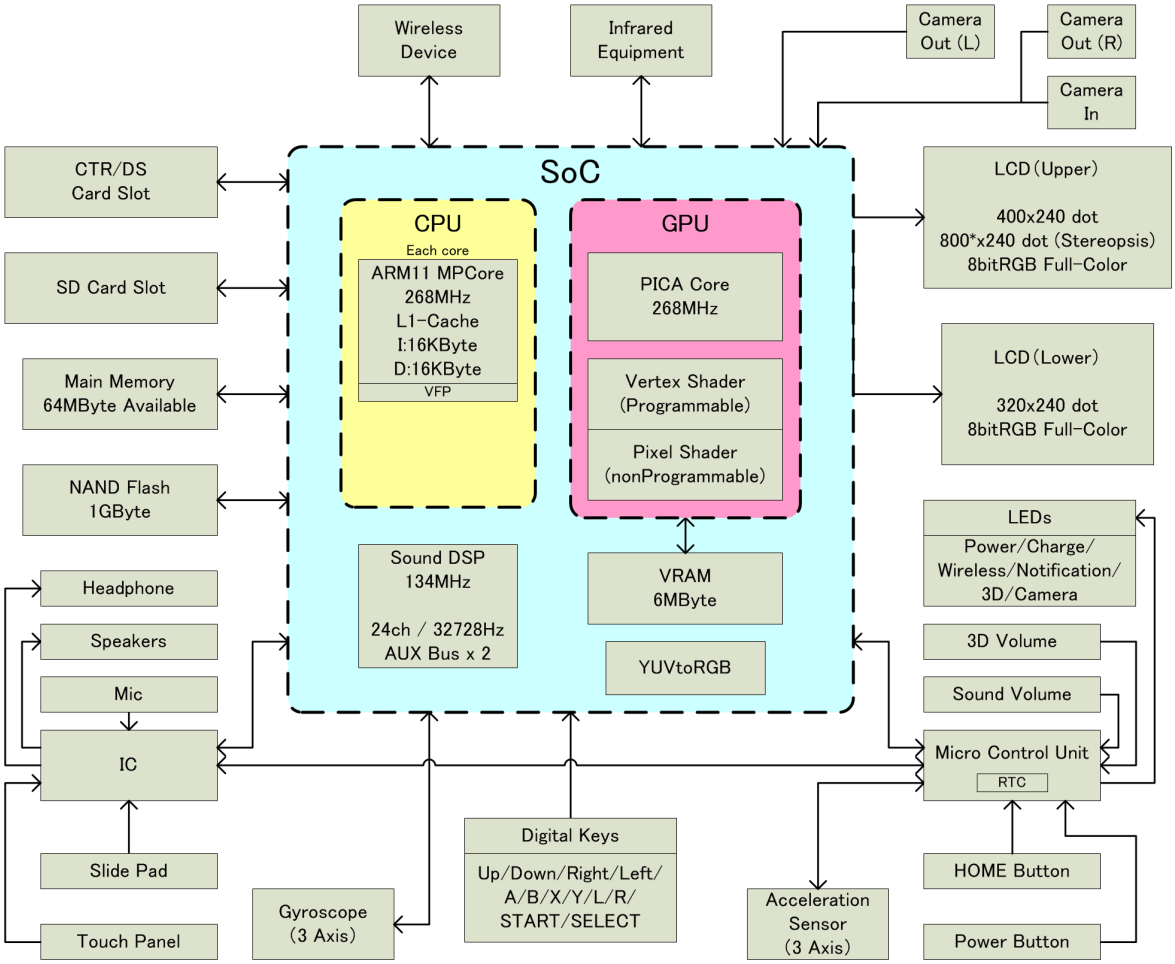
図中で背景色が赤いものは SNAKE で新たに追加された仕様です。背景色が橙色のものは CTR から変更された仕様です。

補足: 本体のバリエーションにより、液晶サイズなどの仕様が異なります。仕様の違いについては、「3DS オーバービュー」の「コンセプト」を参照してください。

2.2. システムブロック全体図(CTR)

CTR のシステム全体図を図 2-2 に示します。

図 2-2. システムブロック全体図(CTR)



＊立体視時の 800 dot は左目用と右目用の合計です。

補足: FTR(ニンテンドー2DS)は、スピーカーの数や上画面の仕様などが CTR とは異なります。仕様の違いについては、「3DS オーバービュー」の「コンセプト」を参照してください。

なお、アプリケーション開発者が、CTR と FTR の違いを意識する必要はありません。

2.3. SoC

SoC は CPU や GPU、DSP、VRAM といった主要な回路を 1 つにまとめた混載チップです。各部の仕様は以下のようになっています。

2.3.1. CPU

複数の ARM11 MPCore とそれぞれに浮動小数点演算用のコプロセッサ（VFP: Vector Floating Point）を搭載しています。

	SNAKE	CTR
プロセッサコア	ARM11 MPCore	
動作周波数	268 MHz or 804 MHz	268 MHz
L1 キャッシュ	4way セットアソシエイティブ（コアごとに搭載）	
	コア 0 ～ コア 1: 命令用 16 KByte、データ用 16 KByte コア 2 ～ コア 3: 命令用 32 KByte、データ用 32 KByte	コア 0 ～ コア 1: 命令用 16 KByte、データ用 16 KByte
L2 キャッシュ	16way セットアソシエイティブ （全コア共通。命令・データ共通 2MByte）	なし
エンディアン	リトルエンディアン	

CTR は 2 つの CPU コア（コア 0 ～ コア 1）、SNAKE は 4 つの CPU コア（コア 0 ～ コア 3）で構成されます。アプリケーションはコア 0 を独占して使用することができ、残りの CPU コアはシステムが使用します。アプリケーションが独占している CPU コアを「アプリコア」、システムが使用している CPU コアを「システムコア」と呼びます。各 CPU コアは以下の用途で使用されます。

CPU コア	使用用途
コア 0	アプリケーション用
コア 1	システム用 （ただしアプリケーションで 30 %まで利用可能）
コア 2	未使用 （システム予約）
コア 3	システム用 （3Dブレ防止機能の制御）

補足： システムコアの一部をアプリケーションから使用することができますが、バックグラウンドで動作しているプロセスに影響が出るなどのデメリットもあります。

2.3.2. GPU

ディジタルメディアプロフェッショナル（DMP）社開発のグラフィックスコアを搭載しています。

フレームバッファを介して処理するため、DS のときのような BG や OBJ の概念はありません。

グラフィックスコア	PICA グラフィックスコア
動作周波数	268 MHz

アーキテクチャ	フレームバッファアーキテクチャ
シェーダ	頂点処理（プログラマブル）、ピクセル処理（非プログラマブル）
グラフィックス機能	OpenGL ES 1.1 ベースの固定パイプライン + 独自拡張（一部は OpenGL ES 2.0 相当） よく使用される 3D グラフィックス処理をハードウェアで搭載

2.3.3. VRAM

GPU からのみアクセス可能な 3 MByte のビデオメモリを 2 つ搭載しています。2 つのメモリでパフォーマンスに違いはありません。

主に、カラーバッファやデプスバッファ（Z バッファ）、ステンシルバッファの配置や、頻繁に使用されるテクスチャや頂点バッファの配置に利用することになります。

VRAM はメインメモリから独立したメモリ領域で、CPU から直接アクセスすることはできません。

2.3.4. サウンドDSP

サウンド処理用の DSP を搭載しています。TWL に搭載されていた DSP とは異なり、ほかの処理を行わせることはできません。

チャンネル数	24 ch
サンプリングレート	約 32728 Hz（正確には Fs = 16756991 * 16 / (32 * 256) = 32728.49805... Hz）
パラメータ更新間隔	約 4.889 ms（正確には T = 160 / Fs = 4.888705... ms）
サンプルフォーマット	DSP ADPCM / PCM8 / PCM16
リサンプリング	ポリフェイズフィルタ / 線形補間/なし
AUXバス数	2

2.4. メインメモリ

アプリケーションから使用可能な領域として、以下のメモリ空間が用意されています。

	SNAKE	**CTR**
メインメモリ	124 MByte （開発機は 178 MByte）	64 MByte （開発機は 96 MByte）

メインメモリにテクスチャと頂点バッファを配置し、GPU から直接参照することもできます。

メモリマップなどの詳細な情報については「3.1. メモリ」を参照してください。

2.5. 動作モード

SNAKE の動作モードには、CTR と同等の動作となる「標準モード」と、SNAKE 独自の動作となる「拡張モード」の 2 つのモードが存在することになります。

以下に標準モードと拡張モードとの性能の差異を示します。

表 2-1. 標準モードと拡張モードとの性能の差異

	標準モード	拡張モード
CPU の動作周波数	268 MHz	804 MHz(三倍速)
CPU の L2 キャッシュ	無効	有効(2 MByte)
メインメモリのサイズ	64 MByte	124 MByte

2.6. LCD

2 つの LCD が搭載されています。

		SNAKE	CTR(SPR)	FTR
上画面	**画面サイズ**	3.88 インチ LL は 4.88 インチ	3.5 インチ SPR は 4.88 インチ	3.5 インチ
	解像度 (立体視表示時)	400 × 240 ピクセル 800 × 240 ピクセル(右目用と左目用の 2 ピクセルを通常時の 1 ピクセルの大きさで表示)		
	色の階調	8 bit RGB で約 1677 万色		
	液晶タイプ	半透過型(グレア加工)		
	バックライト	アクティブバックライト搭載(省エネモード設定の ON/OFF で制御) HOME メニュー表示中に輝度調整が可能		
	立体視表示	モード設定とアプリケーションでの対応により、裸眼での立体視が可能になります		なし
下画面	**画面サイズ**	3.33 インチ LL は 4.18 インチ	3.0 インチ SPR は 4.18 インチ	3.0 インチ
	解像度	320 × 240 ピクセル		
	色の階調	8 bit RGB で約 1677 万色		
	液晶タイプ	半透過型(ノングレア加工)		
	バックライト	アクティブバックライト搭載(省エネモード設定の ON/OFF で制御) HOME メニュー表示中に輝度調整が可能		
	タッチパネル	抵抗膜方式(マルチタッチなし。液晶のドット単位で入力座標を取得することができます)		

注意: 本体を縦に持った(液晶画面の長辺が縦方向に来る状態で見た)ときは立体視表示にはなりません。

補足: 立体視表示については「3DS プログラミングマニュアル - グラフィックス応用編」を参照してください。

2.7. 記憶装置

2.7.1. ゲームカードスロット

ゲームカードスロットには、以下のカードを挿入することができます。

	3DS 専用	DS 専用 DSi 対応 DSi 専用
SNAKE	○	○
CTR	○	○

補足： AGB カートリッジスロットはありません。

2.7.1.1. ニンテンドー3DS 専用ゲームカード

TWL/NITRO カードに比べて高速なアクセスが可能で、セキュリティも新しくなっています。

CARD1 と CARD2 の 2 種類のカードが用意されています。CARD1 の ROM 容量は 4 GByte(32 Gbit)まで対応しています。CARD1 に搭載可能なバックアップメモリの容量は 128 KByte(1 Mbit)または 512 KByte(4 Mbit)です。CARD2 は、ROM とバックアップメモリの容量の合計で 2 GByte(16 Gbit)まで対応可能です。

ROM およびバックアップメモリは、その容量のうちの一部をシステムが使用します。詳細については、「3.2. ニンテンドー3DS 専用ゲームカード」および「7.3. セーブデータ」を参照してください。

2.7.2. 本体 NAND メモリ

本体には、以下の容量の NAND メモリが内蔵されています。本体 NAND メモリは、本体内蔵アプリなどのデータの保存に利用します。

	SNAKE	CTR
容量	1.3 GByte	1 GByte

2.7.3. microSD/SD カードスロット

以下のメディアに対応した microSD/SD カードスロットが搭載されています。

専用のライブラリ(SoundDB、ImageDB など)で、一部のファイルへのアクセス方法を提供しています。SD カードから直接アプリケーション(DSi ウェア以外のダウンロードアプリ)を起動することができます。

	SNAKE	CTR
対応メディア	microSD メモリーカード microSDHC メモリーカード (microSDXC メモリーカードは非対応)	SD メモリーカード SDHC メモリーカード (SDXC メモリーカードは非対応)

2.8. 入力装置

2.8.1. キー入力

	SNAKE	CTR SPR	FTR	備考
十字ボタン	あり			
A / B / X / Y ボタン	あり			SNAKE の X ボタンをかなり強い力で押し込むと、C スティックが下方向に入力されたような動作になることがあります。そのため、C スティックを使用するアプリケーションでは、以下のような回避方法を推奨します。 ● 長押しするような操作を X ボタンに割り当てない。 ● X ボタンを長押しするシーンでは C スティックの値を取得しない。 ● C スティックの遊びを調整する。 なお、開発機および Newニンテンドー3DS で上記の現象を発生させるには、Newニンテンドー3DS LL で発生させるよりも強い力で長押しする必要があります。
L / R ボタン	あり			
ZL / ZR ボタン	あり	拡張スライドパッド接続時は使用可能	なし	SNAKE に搭載されている ZL / ZR ボタンは「ニンテンドー3DS専用拡張スライドパッド」に搭載されている ZL / ZR ボタンと互換性があり、アプリケーションからは常時装着された拡張スライドパッドとして扱うことができます。
スライドパッド	あり			
C スティック / スライドパッド(R)	あり	拡張スライドパッド接続時は使用可能	なし	SNAKE に搭載されている C スティックは「ニンテンドー3DS専用拡張スライドパッド」に搭載されているスライドパッド(R)と互換性があり、アプリケーションからは常時装着された拡張スライドパッドとして扱うことができます。
START / SELECT ボタン	あり			SELECT ボタンは DS との互換性のために残されており、アプリケーションでは START ボタンが押されたと判定されます。
HOME ボタン	あり			HOME メニューの起動を行います。 入力をアプリケーションの進行に使用することはできません。
電源ボタン	あり			電源の制御を行います。誤操作防止のために瞬間的な押下には反応しません。一定時間以上の押下で強制シャットダウン処理が開始されます。 SNAKE は蓋を閉じた状態で電源ボタンの操作が可能です。 入力をアプリケーションの進行に使用することはできません。

無線スイッチ	なし	あり	なし	SNAKE と FTR は HOME メニューから無線スイッチの ON/OFF を切り替えます。 入力をアプリケーションの進行に使用することはできません。
開閉検知スイッチ / スリープスイッチ	開閉検知スイッチ		スリープスイッチ	開閉検知スイッチは本体の開閉を検知するためのものです。FTR は本体を閉じることができないため、開閉検知スイッチではなくスライド式のスリープスイッチが搭載されています。 入力をアプリケーションの進行に使用することはできません。
サウンドボリューム	あり			スライド式のものが搭載されています。 入力をアプリケーションの進行に使用することはできません。
3D ボリューム	あり		なし	立体視表示時の表示調整に使用します。 入力をアプリケーションの進行に使用することはできません。

2.8.2. 加速度センサー

3 軸の加速度センサーが本体下部に搭載されています。アプリケーションで使用することができます。測定可能範囲は各軸 ±の方向に約 1.8 G、静止時ノイズは最大 ±0.02 G、感度は約 0.002 G、サンプリングレートは 100 Hz(デバイス単体での理論値)となっています。

2.8.3. タッチパネル

DS/DSi と同様に下画面にはタッチパネルが搭載されています。

抵抗膜方式の 1 点感知(マルチタッチなし)のタッチパネルで、性能は DSi 相当のものです。従来通り、液晶のドット単位で座標を取得することができます。

2.8.4. マイク

モノラルマイクが下画面の下部にあるボタン群の横に搭載されています。性能は DSi 相当のものです。従来通り、マイクアンプのゲインを 0.5 dB 刻み、10.5 dB から 70.0 dB の範囲で設定することができます。

マイクの感度の個体差は 0.5 dB(1.06 倍)以内です。

2.8.5. カメラ

内側に 1 つ、外側(左と右)に 2 つの計 3 つ、DSi と同性能のカメラが搭載されています。外側カメラの 2 つ、外側カメラ(L)と内側カメラの 2 つは同時に使用することができますが、3 つのカメラを同時に使用することはできません。

カメラの主なスペックは以下のとおりです。

絞り	F2.8(固定)
画角	枠外の表を参照(最大解像度で撮影時の値)
撮影可能範囲	20 cm～無限遠(パンフォーカス。マクロスイッチ非搭載)
最大解像度	VGA

最大フレームレート	30 fps（fps: 1 秒間のフレーム数）
出力フォーマット	YCrYCb（別途用意されている YUVtoRGB 回路を利用することで RGBA8888、RGB888、RGB565、RGBA5551 として出力可能）

	最小	平均	最大
対角	63.0°	66.0°	69.0°
水平	52.2°	54.9°	57.6°
垂直	40.4°	42.6°	44.8°

2.8.6. ジャイロセンサー

本体の傾きや回転速度を計測することのできる 3 軸のジャイロセンサーが本体下部に搭載されています。測定可能範囲は各軸±の方向に 1800 dps（Degrees Per Second）、静止時ノイズは最大 ±2.28 dps、感度は約 0.07 dps、サンプリングレートは 100 Hz となっています。

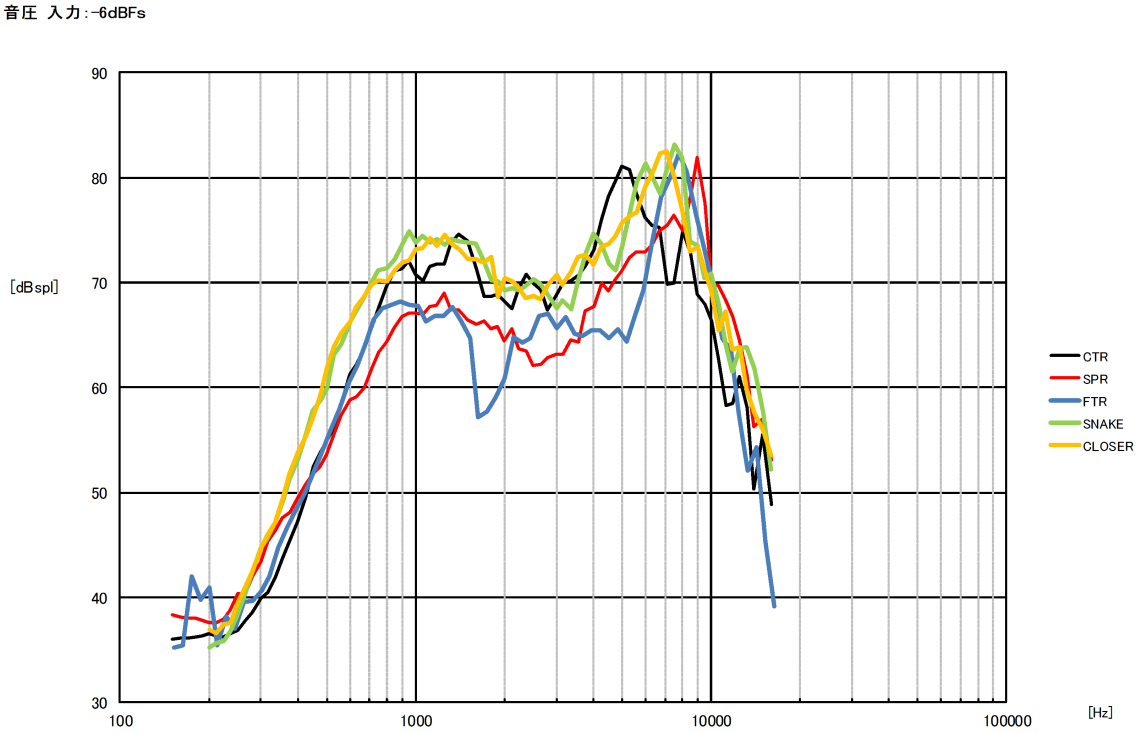
2.9. 出力装置

2.9.1. スピーカー

CTR、SPR、SNAKE、CLOSER では、上画面の左右にステレオスピーカーが配置されています。FTR では、上画面の左上にモノラルスピーカーが配置されています。

下図は、CTR / SPR / FTR / SNAKE / CLOSER に搭載されているスピーカーの音圧周波数特性をグラフにしたものです。

図 2-3. CTR(SPR / FTR / SNAKE / CLOSER)本体スピーカーの音圧周波数特性



2.9.2. ヘッドホン端子

ステレオ出力のミニジャック端子が搭載されています。DS/DSi にあった、マイク接続端子はありません。

マイク入力付のヘッドホンには対応していません。

	SNAKE	CTR	SPR	FTR
ヘッドホン端子の位置	下部手前の中央		下部手前の左側	

2.9.3. LED

カメラの状態、電池、充電、無線、3D 表示、おしらせの各状態を知らせる LED が搭載されています。

以下のように、搭載されている LED はプラットフォームごとに異なります。

	SNAKE	CTR	SPR	FTR
カメラの状態	なし	あり	あり	なし
電池	あり	あり	あり	あり
充電	あり	あり	あり	あり
無線	あり	あり	あり	あり
3D 表示	なし	あり	なし	なし
おしらせ	あり	あり	あり	あり

2.10. 通信装置

2.10.1. 無線通信モジュール

2.4 GHz 帯でワイヤレス通信可能な無線通信モジュールを搭載しています。

主な機能として、アプリケーションが意図的に行う通信（フォアグラウンド通信）とアプリケーションの裏でシステムが自動的に行う通信（バックグラウンド通信）の 2 つに大きく分けることができます。

無線通信の詳細については「3DS プログラミングマニュアル - 無線通信編」を参照してください。

- フォアグラウンド通信
  - インフラストラクチャ通信
  - ローカル通信
  - ダウンロードプレイ

- バックグラウンド通信
  - すれちがい通信
  - ダウンロードタスク
  - プレゼンス機能

2.10.2. 赤外線通信装置

赤外線受発光部と赤外線通信装置を搭載しています。

2.10.3. NFC（Near Field Communication）

SNAKE には非接触型の近距離無線通信（NFC：Near Field Communication）のアンテナが下画面の液晶パネル下に搭載されています。

2.11. その他

2.11.1. RTC (Real Time Clock)

時刻を保持し、計時を行います。

2.11.2. DSi との互換

DSi との互換性を備えており、DS ダウンロードプレイにも対応しています。

3. メモリ、ニンテンドー3DS 専用ゲームカード

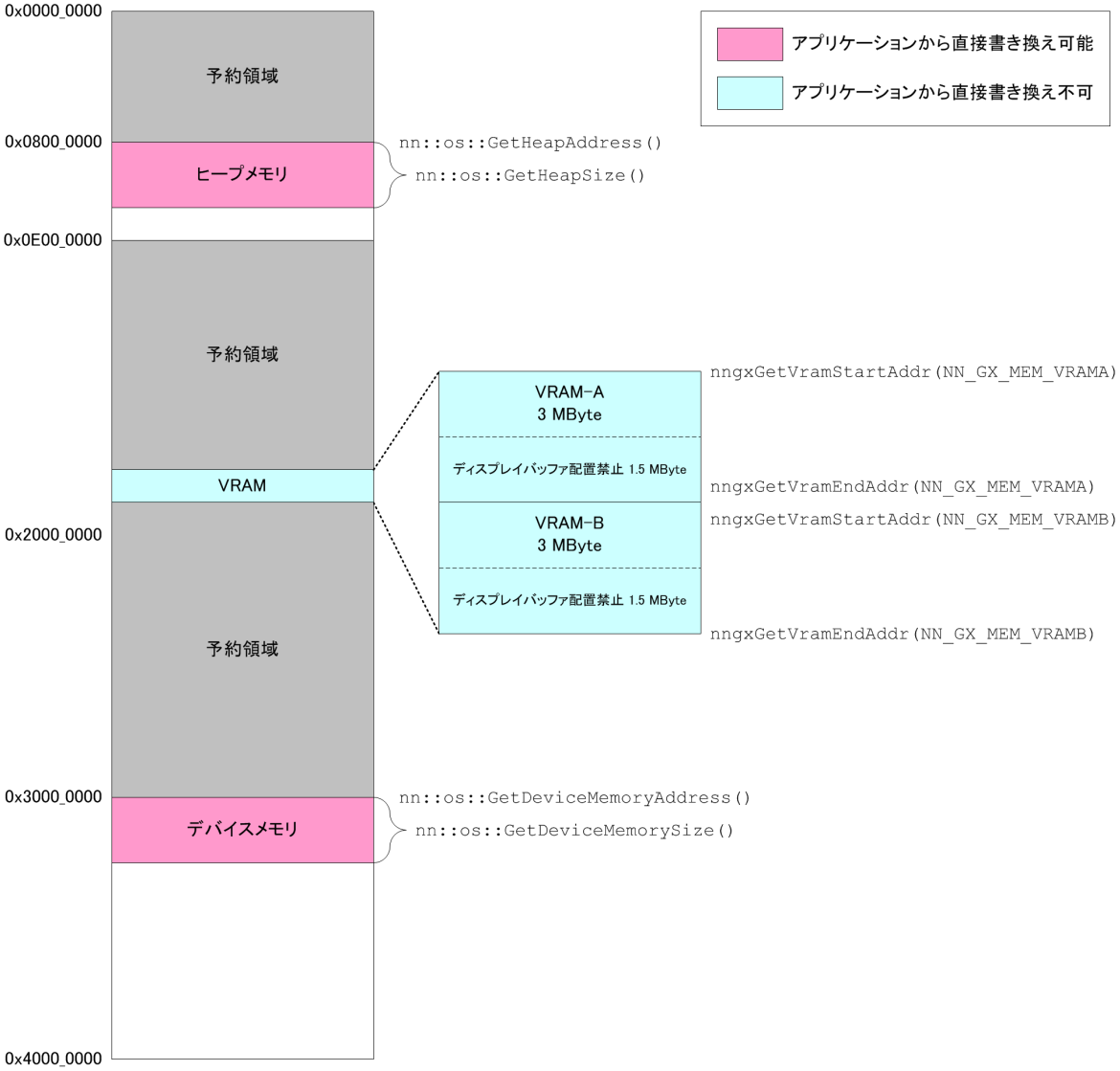
この章では、アプリケーションから扱うことのできるメモリ領域と、カードアプリが記録されるニンテンドー3DS 専用ゲームカードについて説明します。

3.1. メモリ

3.1.1. メモリマップ

アプリケーションから見たメモリマップを以下に示します。下図のメモリマップは製品版のものです。

図 3-1. メモリマップ(製品版)



アプリケーションからメモリへのアクセスは、仮想アドレスで行われます。

セキュリティ上の理由から、プログラムがロードされた領域以外のメモリ(ヒープメモリやデバイスメモリなど)にロードされたデータ列を実行することはできません。

0x4000_0000 以降は予約領域です。

3.1.2. デバイスメモリ

デバイスメモリとは、GPU などの周辺デバイスからのアクセスの際に OS がアドレスの整合性を保証しているメモリ領域のことです。グラフィックスで利用している GPU や、サウンドで利用している DSP などがアクセスするバッファには、このデバイスメモリに確保されていなければならないものもあります。逆に、デバイスメモリに確保してはいけないバッファもあり、基本的にアライメントが 4096 Byte、サイズが 4096 Byte の倍数でなければならないバッファがこれに該当します。

デバイスメモリとして確保するメモリ領域のサイズは `nn::os::SetDeviceMemorySize()` で指定することができます。指定の際は、`nn::os::DEVICE_MEMORY_UNITSIZE`（4096 Byte）の倍数単位で設定してください。CTR の場合はメインプログラムとヒープメモリ、デバイスメモリの合計が 64 MByte（開発機は 96 MByte）に収まる範囲で、SNAKE の場合はメインプログラムとヒープメモリ、デバイスメモリの合計が 124 MByte（開発機は 178 MByte）に収まる範囲でそれぞれ確保することができます。再度呼び出して、確保するメモリ領域のサイズを変更することも可能ですが、その場合は変更前後のサイズが 1 MByte（1048576 Byte）の倍数でなければなりません。この時、サイズ 0 へ変更する場合は変更前のサイズが 1 MByte の倍数でなくても問題ありません。サイズ 0 から変更する場合も同様に変更後のサイズは 1 MByte の倍数でなくても問題ありません。ただし、4096 Byte の倍数にする必要はあります。

確保される領域のアドレスが毎回同じである保証はありませんので、サイズを指定したあとに `nn::os::GetDeviceMemoryAddress()` で先頭アドレスを取得してからデバイスメモリにアクセスしてください。現在確保中のメモリ領域のサイズは `nn::os::GetDeviceMemorySize()` で取得することができます。

表 3-1. デバイスメモリ上に確保するバッファの種類と先頭アドレスのアライメント

デバイス	バッファの種類	アライメント	備考
GPU	テクスチャイメージ	128 Byte	
	頂点バッファ	1 ～ 4 Byte	頂点属性によって変化します。
	ディスプレイバッファ	16 Byte	
DSP	音源データ	32 Byte	音源データを格納する領域全体のサイズは 32 Byte の倍数でなければなりません。
CAMERA	受信用バッファ	64 Byte（推奨）	デバイスメモリ以外に確保したバッファを使用すると Panic で停止します。
Y2R	画像データ転送元 / 転送先バッファ	64 Byte（推奨）	デバイスメモリ以外に確保したバッファを使用すると Panic で停止します。

GPU がアクセスするバッファは VRAM 上に確保することもできます。

表 3-2. デバイスメモリ上に確保してはいけないバッファ

バッファ	備考
マイクのサンプリング結果を格納するバッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。
UDS ライブラリの受信バッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。
Socket ライブラリの受信バッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。

HTTP ライブラリの作業バッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。
DLP ライブラリの作業バッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。
ACT ライブラリの通信用バッファ	アライメントは 4096 Byte、サイズは 4096 Byte の倍数でなければなりません。

3.1.3. VRAM

VRAM-A と VRAM-B の 2 チャンネルで構成され、それぞれが 3 MByte の合計 6 MByte 搭載されています。

カラーバッファの確保など、GX(グラフィックス)ライブラリが使用するメモリ領域の管理はアプリケーションで行わなければなりませんが、CPU で VRAM 上のメモリ領域を読み込んだときに取得する内容が正確であるかどうかは保証されていません。VRAM 上のメモリ領域を確保する際には、`nngxGetVramStartAddr()`、`nngxGetVramEndAddr()`、`nngxGetVramSize()` で VRAM の開始アドレス、終了アドレス、サイズを取得し、要求されたメモリ領域の配置をアプリケーションで管理してください。

GX ライブラリやライブラリが使用するメモリ領域の管理方法については「5.5. GX ライブラリの初期化」や「3DS プログラミングマニュアル – グラフィックス基本編」を参照してください。

3.1.4. ヒープメモリ

ファイルの読み込みバッファなど、アプリケーションが自由に使うことのできるメモリ領域です。ヒープメモリの確保は `nn::os::SetHeapSize()` でサイズを指定することで行います。サイズは `nn::os::HEAP_UNITSIZE`（4096）の倍数のバイト数である必要があります。CTR の場合はメインプログラムとヒープメモリ、デバイスメモリとの合計が 64 MByte(開発機は 96 MByte)に収まる範囲で確保することができます。SNAKE の場合はヒープメモリの最大が 96 MByte かつ、メインプログラム、ヒープメモリ、デバイスメモリの合計が 124 MByte(開発機は 178 MByte)に収まる範囲で確保することができます。再度呼び出して、確保するメモリ領域のサイズを変更することも可能です。

利用状況に応じてヒープメモリ領域のサイズを拡大・縮小し、それに合わせてデバイスメモリ領域のリサイズを行う場合、ヒープメモリ領域の取り扱いは次の点に注意してください。

- ヒープメモリ領域は、拡大したのち、縮小するときは拡大したときの増加分と同じサイズ分を減少してください。
- ヒープメモリ領域は複数回に分けて拡大することが可能です。その場合は、縮小するときも拡大する前のサイズに戻していくように、段階的に実行してください。

上記に違反すると未割当の領域をデバイスメモリとして確保できない場合があります。

図 3-2. ヒープメモリを拡大・縮小後にデバイスメモリを拡大する場合の例

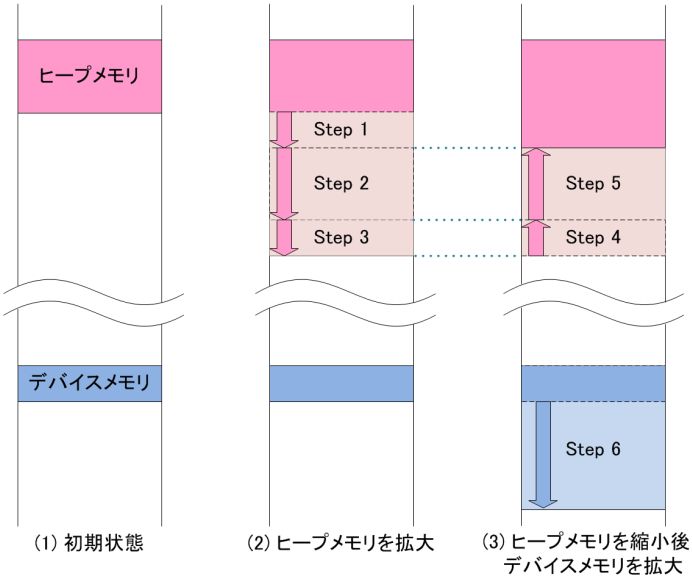


図 3-2 の例のように、ヒープメモリ領域を Step 1 、Step 2、Step 3 と段階的に拡大していた状態から、ヒープメモリ領域を減らしてデバイスメモリ領域を拡大する場合は、ヒープメモリ領域を拡大する前のサイズに戻すように、まず Step 3 での増加分と同じサイズ分を減少し(Step 4)、次にStep 2 での増加分と同じサイズ分を減少させます(Step 5)。

> **注意:** ヒープメモリ領域は可能な限り一括で必要なサイズを確保し、リサイズせずに使用することを推奨します。どうしてもリサイズが必要な場合のみ上述のように段階的に行ってください。ただし、細かい単位でリサイズを行うとメモリアクセスの著しいパフォーマンス低下を招く原因になります。

確保される領域のアドレスが毎回同じである保証はありませんので、ヒープメモリへのアクセスは、サイズを指定したあとに `nn::os::GetHeapAddr()` で先頭アドレスを取得してから行ってください。現在確保中のヒープメモリのサイズは `nn::os::GetHeapSize()` で取得することができます。

3.2. ニンテンドー3DS 専用ゲームカード

ニンテンドー3DS 専用ゲームカード(3DS カード)には、"CARD1" と "CARD2" の 2 種類があり、タイトルの仕様に応じた適切なものが使用可能です。

> **注意:** 原則として、バックアップメモリの容量が 512 KByte 以下のタイトルは CARD1 を、1 MByte 以上のタイトルは CARD2 を、それぞれ使用してください。

3.2.1. CARD1

CARD1 は、ROM(Read Only Memory)とバックアップメモリ(書き換え可能な不揮発性メモリ)の 2 つのデバイスを搭載しているカードです。

3.2.1.1. ROM

CARD1 の ROM の転送速度は使用する SDK のバージョンやデータサイズによって異なります。また、デバイスの個体差により若干のばらつきがありますので、転送速度に依存するようなアプリケーションの実装はしないでください。

ROM の容量は、現時点で 128 MByte ～ 4 GByte の 6 種類を用意しています。

表 3-3. ROM の容量(CARD1)

ROM の容量	転送速度	仕向け先による使用可能領域のサイズ		
		日本/北米/韓国向け	欧州向け	中国/台湾向け
128 MByte	※	93.5 MByte (98,041,856 byte)	91.5 MByte (95,944,704 byte)	61.5 MByte (64,487,424 byte)
256 MByte		219.0 MByte (229,638,144 byte)	217.0 MByte (227,540,992 byte)	187.0 MByte (196,083,712 byte)
512 MByte		470.0 MByte (492,830,720 byte)	468.0 MByte (490,733,568 byte)	438.0 MByte (459,276,288 byte)
1 GByte		921.5 MByte (966,262,784 byte)	919.5 MByte (964,165,632 byte)	889.5 MByte (932,708,352 byte)
2 GByte		1875.5 MByte (1,966,604,288 byte)	1873.5 MByte (1,964,507,136 byte)	1843.5 MByte (1,933,049,856 byte)
4 GByte		3783.0 MByte (3,966,763,008 byte)	3781.0 MByte (3,964,665,856 byte)	3751.0 MByte (3,933,208,576 byte)

※ 使用する SDK のバージョンやデータサイズによって異なります。詳しくは「3DS パフォーマンス TIPS」を参照してください。

3.2.1.2. バックアップメモリ

CARD1 のバックアップメモリの転送速度は、使用する SDK のバージョンやデータサイズによって異なります。また、デバイスの個体差により若干のばらつきがありますので、転送速度に依存するようなアプリケーションの実装はしないでください。また、書き換え回数についてはガイドラインを参照し、その規定に従ってください。

バックアップメモリの容量は、128 KByte と 512 KByte の 2 種類を用意しています。バックアップメモリに 1 MByte 以上の容量が必要な場合は、CARD2 を使用してください。

バックアップメモリの工場出荷時の内容は、全領域が 0xFF で埋められていることが保証されています。デバッグ時に、工場出荷状態をエミュレートする場合は全領域を 0xFF で埋めてください。

バックアップメモリに保存されるセーブデータは、ライブラリによるバックアップ領域の二重化を有効にすることができます。ただし、二重化を有効にした場合、実際にファイルの保存に使用することのできる容量は全容量の 40 % 程度になります。

表 3-4. バックアップメモリの容量(CARD1)

バックアップメモリの容量	転送速度	二重化有効時の使用可能領域のサイズ
128 KByte	※	50 KByte 以下
512 KByte		239 KByte 以下

※ 使用する SDK のバージョンやデータサイズによって異なります。詳しくは「3DS パフォーマンス TIPS」を参照してください。

3.2.2. CARD2

CARD2 は、ROM とバックアップメモリの両方の機能を持つ Writable メモリを搭載しているカードです。

3.2.2.1. Writable メモリ

CARD2 の Writable メモリの転送速度は、使用する SDK のバージョンやデータサイズによって異なります。デバイスの個体差により若干のばらつきがありますので、転送速度に依存するようなアプリケーションの実装はしないでください。また、書き換え回数についてはガイドラインを参照し、その規定に従ってください。

Writable メモリは、ROM 領域（書き換え不可）とバックアップ領域（書き換え可能）の 2 つの領域の境界を 1 MByte 単位で変更することができます。

バックアップ領域は、CARD1 のバックアップメモリと同様に、ライブラリによる二重化を選択することができます。ファイルの保存に使用することのできる容量についても、同様の計算で求めることができます。バックアップ領域として指定可能なサイズの上限は Writable メモリの容量の半分です。Writable メモリの容量は、512 MByte ～ 2 GByte の 3 種類を用意しています。

表 3-5. Writable メモリの容量（CARD2）

Writable メモリの容量	転送速度	仕向け先による使用可能領域のサイズ		
		日本/北米/韓国向け	欧州向け	中国/台湾向け
512 MByte	※	412.5 MByte (432,537,600 byte)	410.5 MByte (430,440,448 byte)	412.5 MByte (432,537,600 byte)
1 GByte		889.5 MByte (932,708,352 byte)	887.5 MByte (930,61,200 byte)	889.5 MByte (932,708,352 byte)
2 GByte		1843.5 MByte (1,933,049,856 byte)	1841.5 MByte (1,930,952,704 byte)	1843.5 MByte (1,933,049,856 byte)

※ 使用する SDK のバージョンやデータサイズによって異なります。詳しくは「3DS パフォーマンス TIPS」を参照してください。

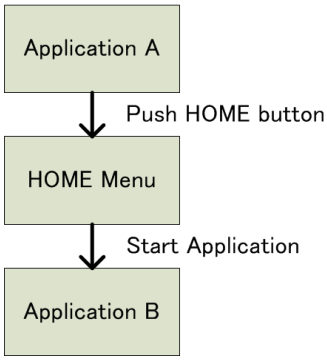
4. ソフトウェア構成

この章では 3DS のソフトウェア構成について説明します。

4.1. アプリケーション

アプリケーションから別のアプリケーションへと切り替えるには、HOME メニューを経由して再起動しなければなりません。

図 4-1. アプリケーションの切り替え



4.2. 標準アプリ、拡張アプリ

「2.5. 動作モード」で説明したように、SNAKE には CPU 性能等が異なる「標準モード」と「拡張モード」の 2 つの動作モードがあります。なお、CTR は動作モードが標準モードのみのプラットフォームとして扱います。

これに合わせるように、アプリケーション種別には SNAKE でも標準モードで動作する「標準アプリ」と SNAKE では拡張モードで動作する「拡張アプリ」の 2 つが存在します。

表 4-1. アプリケーション種別による各プラットフォームでの動作モード

アプリケーション種別	SNAKE	CTR	備考
標準アプリ	標準モード	標準モード	過去の SDK でビルドされたアプリケーション含む
拡張アプリ	拡張モード	標準モード	

4.2.1. 標準アプリの作成方法

CTR-SDK のビルドシステムを利用している場合は、OMakefile に ″DESCRIPTOR = $(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc″ があればその行をコメントアウトしてください。

VSI-CTR（要 VSI-CTR Platform 3.0.0 以降）を利用している場合は、プロジェクトの[構成プロパティ] - [全般] - [拡張アプリケーションを作成] を ″いいえ″ に変更してください。

4.2.2. 拡張アプリの作成方法

CTR-SDK のビルドシステムを利用している場合は、OMakefile に ″DESCRIPTOR = $(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc″ を追加してください。

VSI-CTR(要 VSI-CTR Platform 3.0.0 以降)を利用している場合は、プロジェクトの[構成プロパティ] - [全般] - [拡張アプリケーションを作成] を "はい" に変更してください。

4.2.3. 拡張モード使用時の注意事項

拡張モードでは L2 キャッシュが有効になりますが、L2 キャッシュは容量が大きいため nngxUpdaterBuffer() 等のパフォーマンスが低下する可能性があります。

このパフォーマンスの低下を防ぐため、拡張モードでは目的に合わせて API が分けられています。拡張モードで動作するアプリケーションにおいてグラフィックを扱う場合は、適切な API を使用するようにしてください。

4.2.4. 動作モードが切り替わるタイミング

SNAKE の動作モードは `nn::applet::Enable()` が呼び出されたときに切り替わります。そのため、標準アプリであっても、3DSロゴの表示中とアプリケーションに制御が移ってから `nn::applet::Enable()` を呼び出すまでは拡張モードで動作しています。

アプリケーションが起動しているかどうかにかかわらず、HOMEメニューが表示されているときの動作モードは拡張モードです。そのため、標準アプリが HOMEメニューの表示によって中断されると、標準モードから拡張モードに切り替わり、アプリケーションに戻ったときに標準モードに切り替わります。

また、「ゲームメモ」や「インターネットブラウザー」などのシステムアプレットはアプリケーションを中断しているため、HOMEメニューを表示したときと同じように、標準モードと拡張モードの切り替えが行われます。

表 4-2. アプリケーションの種別と動作モードが切り替わるタイミング

アプリケーション種別	ロゴ表示	nn::applet::Enable()	アプリケーション	HOMEメニュー	ライブラリアプレット	システムアプレット
拡張アプリ	拡張モード	拡張モード	拡張モード	拡張モード	拡張モード	拡張モード
標準アプリ	拡張モード	標準モード	標準モード	拡張モード	標準モード	拡張モード

4.2.5. SNAKE と CTR でアプリケーションの挙動を変化させる

3DS シリーズに SNAKE が追加されたことにより、動作しているハードウェアの種別によってアプリケーションの挙動を変化させる必要が出てくることもあります。

アプリケーションが動作しているハードウェアの種別が SNAKE であるかどうかは `nn::os::IsRunOnSnake()` で判断することができます。これにより、CTR の拡張スライドパッド相当の入力装置が標準搭載されている SNAKE では拡張スライドパッドの接続確認処理を省略するなど、ハードウェアの特性に合わせた実装が可能になります。

拡張アプリとして作成されたアプリケーションが SNAKE の拡張モードで動作しているかどうかは `nn::os::IsRunningAsExtApplication()` で判断することができます。ただし、この関数は実行コストが高く、`nninitStartUp()` 内で呼び出すことができないなどの制限があります。

ハードウェアとアプリケーションの組み合わせで、`nn::os::IsRunOnSnake()` と `nn::os::IsRunningAsExtApplication()` が返す値は以下のようになります。

表 4-3. ハードウェアとアプリケーションの組み合わせによる返り値の変化

ハードウェア	アプリケーション	IsRunOnSnake()	IsRunningAsExtApplication()
SNAKE	拡張アプリ	true	true
	標準アプリ	true	false
CTR	拡張アプリ	false	false
	標準アプリ	false	false

注意: 拡張モードで強化される項目(アプリケーションが使用できるメモリのサイズや CPU 性能)を利用できるかの判断には、必ず `nn::os::IsRunningAsExtApplication()` を使用してください。ただし、L2 キャッシュが有効になることにより追加されたキャッシュ操作関連の関数は標準モードでも使用することができます。

`nn::os::IsRunOnSnake()` は上記以外で SNAKE と CTR とで異なる仕様となっている機器を利用できるかの判断に使用してください。

4.2.6. SNAKE 専用タイトル

アプリケーションの動作対象を SNAKE のみにする場合、bsf ファイルに SNAKE 専用(SNAKEOnly を True)の設定をしてください。SNAKE 専用タイトルは CTR では起動しないため、CTR での動作を考慮する必要がなくなります。

補足: bsf ファイルについては、CTR-SDK のツール「ctr_makebanner」のリファレンスを参照してください。

4.3. SDK で提供されるもの

3DS では、キー入力などの各種デバイスを扱うためのライブラリだけでなく、HOME メニューやいつの間に通信などを利用するためのアプレットやライブラリが提供されます。

4.3.1. アプレット

HOME メニューのように、アプリケーションに特定の目的を持った機能を提供し、アプレットで提供されている機能については、アプリケーションの開発にかかるコストの削減などに効果を発揮します。

注意: 利用可能なアプレットは任天堂が提供するものに限られています。デベロッパが独自のものを開発することはできません。

4.3.2. ライブラリ

CTR-SDK では、搭載されたハードウェアを利用するためのライブラリが用意されています。それぞれのライブラリはライブラリ名に即した名前空間を持ち、複数のクラスやメンバ関数で構成されています。

ライブラリは主に C++ で記述されていますが、C 言語用のラッパー関数も用意されています。C 言語用のラッパー関数については関数リファレンスを参照してください。以降の説明では、関数名などを C++ での記述に合わせています。

表 4-4. ライブラリー覧

ライブラリ名	名前空間	説明
OS	nn::os	オペレーションシステムに関係するクラスをまとめたライブラリです。
RO	nn::ro	DLL 機能を提供するためのライブラリです。
APPLET	nn::applet	アプレットの起動、ふた閉じのスリープなどに対応するためのライブラリです。
FS	nn::fs	各メディア上のファイルにアクセスするためのライブラリです。
CX	nn::cx	データの圧縮、解凍を行うためのライブラリです。
MATH	nn::math	算術・数値演算の関数をまとめたライブラリです。
CRYPTO	nn::crypto	暗号化を行うライブラリです。
FND	nn::fnd	ヒープ、時間など、基礎的なクラスをまとめたライブラリです。
FONT	nn::font	フォントデータを利用した文字描画のためのライブラリです。
HID	nn::hid	デジタルボタン、スライドパッド、タッチパネル、加速度センサー、ジャイロセンサー、デバッグパッドからの入力を扱うライブラリです。
PTM	nn::ptm	電源とアラームを制御するライブラリです。
GX	nn::gx	GPU、LCD の制御を行うライブラリです。3D グラフィックスの描画には gl、nngx 関数を使用します。
GD	nn::gd	GX ライブラリの 3D グラフィックス描画関数を軽量化したライブラリです。
GR	nn::gr	3D グラフィックスコマンドの直接生成を支援するライブラリです。
MIC	nn::mic	マイクを使った自動サンプリングを行うライブラリです。
CAMERA	nn::camera	カメラを使った画像キャプチャを行うライブラリです。
Y2R	nn::y2r	YUVtoRGB 回路を利用して、YUV から RGB への変換を行うライブラリです。
QTM	nn::qtm	SNAKE でのみ利用可能なフェイストラッキングを行うためのライブラリです。
DSP	nn::dsp	サウンド再生に DSP を利用するためのライブラリです。
SND	nn::snd	サウンド再生を行うライブラリです。
AACDEC	nn::aacdec	AAC データのデコードを行うためのライブラリです。
AACENC	nn::aacenc	AAC データのエンコードを行うためのライブラリです。
NDM	nn::ndm	ネットワーク処理を行っているデーモンを制御するためのライブラリです。
UDS	nn::uds	無線通信モジュールを利用したローカル通信を行うライブラリです。
RDT	nn::rdt	UDS ライブラリを利用して、信頼性のあるデータ通信を行うライブラリです。
CEC	nn::cec	すれちがい通信の設定などを行うライブラリです。
DLP	nn::dlp	ニンテンドー 3DS ダウンロードプレイに対応するためのライブラリです。
AC	nn::ac	インフラストラクチャ通信の自動接続を行うライブラリです。
BOSS	nn::boss	ダウンロードタスクの登録などを行うライブラリです。
FRIENDS	nn::friends	フレンドの情報にアクセスするためのライブラリです。
NEWS	nn::news	おしらせを投稿するためのライブラリです。
ACT	nn::act	アカウントシステムに登録された情報の取得や、認証をするライブラリです。

EC	nn::ec	EC 機能を利用するためのライブラリです。
IR	nn::ir	本体間赤外線通信を利用するためのライブラリです。
NFP	nn::nfp	amiibo™(アミーボ)と呼ばれるキャラクター商品とアプリケーションを連携させるためのライブラリです。
ULCD	nn::ulcd	立体視表示で使用する左目、右目用のカメラ行列を算出するライブラリです。
JPEG	nn::jpeg	JPEG 形式のエンコードとデコードを行うライブラリです。
TPL	nn::tpl	複数枚のテクスチャをまとめた TPL ファイルを扱うためのライブラリです。
CFG	nn::cfg	本体設定で扱われている情報を取得するライブラリです。
NGC	nn::ngc	NG ワードリストによって NG ワードをチェックするためのライブラリです。
UBL	nn::ubl	受信拒否リストを管理するためのライブラリです。
PL	nn::pl	3DS 固有の機能(歩数情報など)を使用するためのライブラリです。
UTIL	nn::util	ユーティリティ関数のライブラリです。
ENC	nn::enc	文字コード変換を行うライブラリです。
ERR	nn::err	エラー処理のためのライブラリです。
ERREULA	nn::erreula	エラー・EULA アプレットを使用するためのライブラリです。
SWKBD	nn::swkbd	ソフトウェアキーボードアプレットを使用するためのライブラリです。
PHTSEL	nn::phtsel	写真選択アプレットを使用するためのライブラリです。
VOICESEL	nn::voicesel	音声選択アプレットを使用するためのライブラリです。
EXTRAPAD	nn::extrapad	拡張スライドパッド補正アプレットを使用するためのライブラリです。
WEBBRS	nn::webbrs	インターネットブラウザーを使用するためのライブラリです。
OLV	nn::olv	Miiverse アプリ、投稿アプリを使用するためのライブラリです。
SOCKET	nn::socket	(デバッグ用途のみ)ソケット通信を行うライブラリです。
SSL	nn::ssl	(デバッグ用途のみ)SSL による通信を行うライブラリです。
HTTP	nn::http	(デバッグ用途のみ)HTTP 通信を行うライブラリです。
DBG	nn::dbg	デバッグ補助のためのライブラリです。
HIO	nn::hio	(デバッグ用途のみ)Host IO を利用するためのライブラリです。
MIDI	nn::midi	(デバッグ用途のみ)MIDI を利用するためのライブラリです。

※ ライブラリの構成や名前は今後変更される場合があります。

4.4. エラー処理について

3DS では本体更新によりライブラリが更新されるため、アプリケーションを開発した時点では定義されていなかった値を処理結果として返す可能性があります。このようなケースに問題なく対応するためにも、処理の成否は必ず `nn::Result::IsSuccess()` もしくは `nn::Result::IsFailure()` で判定するようにしてください。

成否を確認せずに一致判定のみでエラー処理を行うことは、想定外のエラーが発生した際に意図しない成功または意図しない失敗と判定してしまう危険性があるため推奨しません。

補足: リファレンスにある症状とは異なるエラーが特定の個体でのみ発生する場合は、そのハードの故障を疑ってみてください。

5. アプリケーションの初期化と状態遷移のハンドリング

この章では、アプリケーションが最初に行わなければならない初期化処理とスリープなどの状態遷移への対処について説明します。

5.1. エントリ関数までの初期化処理

アプリケーションのエントリ関数が呼び出されるまでに行われている処理については、SDK に付属されている「システムプログラミングガイド」を参照してください。

5.2. エントリ関数

アプリケーションのエントリ関数は nnMain() で定義されています。

エントリ関数で最初に行うのは、アプリケーションで使用するライブラリの初期化です。主なライブラリの初期化については、次の節以降で説明します。

エントリ関数を抜けるとアプリケーションが終了します。エントリ関数内でメインループを構成し、アプリケーションが終了するまでエントリ関数から抜けないようにしてください。

5.3. APPLET ライブラリの初期化

APPLET ライブラリは、単にライブラリアプレットを利用するためだけでなく、HOME ボタンや電源ボタンが押されたときへの対処や蓋を閉じてスリープ状態へ移行するときへの対処を実装するために必要なライブラリです。APPLET ライブラリの初期化は、エントリ関数に実行が移るまでに行われています。アプリケーションで行わなければならない初期化処理は、APPLET ライブラリの各機能を有効にするために nn::applet::Enable() を呼び出すことです。

コード 5-1. APPLET ライブラリの初期化

```
void nn::applet::Enable(bool isSleepEnabled = true);
```

スリープ関連のコールバック設定が完了してから呼び出してください。*isSleepEnabled* には、呼び出しの時点でスリープ関連のコールバックを有効にするかどうかを指定します。

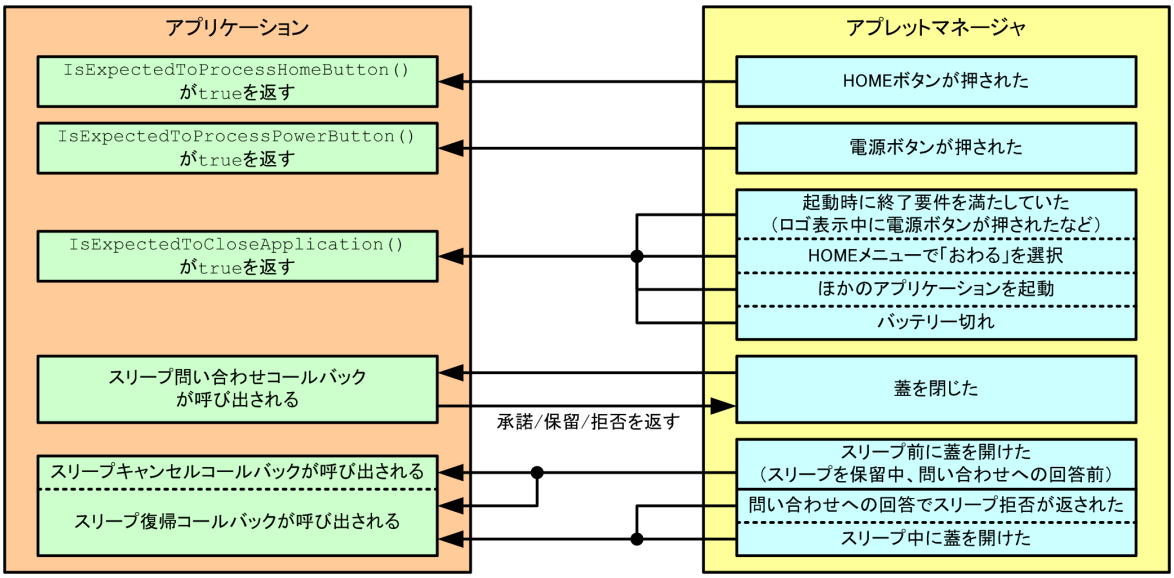
この関数を呼び出す前後では、スリープ関連のハンドリングを慎重に行う必要があります。スリープ関連のハンドリングについては、「5.3.3. スリープへの対処」を参照してください。

> **注意:** nn::applet::Enable() は nngxInitialize() や nn::dsp::Initialize()、nn::snd::Initialize() よりも前に呼び出さなければなりません。
>
> アプリケーションの起動中に電源ボタンが押されるなど、Enable() が呼び出されるまでに発生した要因で、アプリケーションの起動直後に終了させなければならない場合があります。そのため、この関数を呼び出した直後は、「5.3.1. アプリケーション終了要求への対処」に従ってアプリケーション終了要求に対処してください。その際、終了要求が来ていないことを確認するまで nngxInitialize() は呼び出さないでください。

以降の項では、アプリケーションの実装に必須となる、アプリケーション終了要求、HOME ボタン、スリープ(蓋閉じ)、電源ボタンへの対処とアプリケーションでのハンドリング例について説明します。対処が必要となる状態の変化は、すべてアプレットマネージャを介してアプリケーションに通知されます。

> **注意:** アプリケーション終了要求や電源ボタンへの対処を行わなかった場合、電源ボタンが一定時間以上押し続けられたときにアプリケーションが強制終了させられます。アプリケーションはこれらの状態変化に対応し、正常に処理が行われるように実装しなければなりません。

図 5-1. アプレットマネージャからの通知



5.3.1. アプリケーション終了要求への対処

アプレットマネージャは、アプリケーションを中断して表示している HOME メニューでの「おわる」の選択、ほかのアプリケーションの起動などが要因でアプリケーションが終了しなければならない状態にあるかどうかを `nn::applet::IsExpectedToCloseApplication()` で通知しますので、アプリケーションはこの関数を定期的(フレームごとなど)に呼び出さなければなりません。この関数が `true` を返したときは、速やかにアプリケーションを終了させる必要があります。また、描画の権限がない状態になる可能性があることに注意してください。

アプリケーションは独自の終了処理(4 秒以内)を行ったあとに `nn::applet::CloseApplication()` を呼び出してアプリケーションを終了させてください。描画の権限がない状態ではグラフィックス関連の処理が実行されませんので、`nngxInitialize()` や `nngxWaitCmdlistDone()` を呼び出すと処理がブロックされてしまいます。また、コマンドリクエストの終了割り込みも発生しませんので、これらの完了を待つことなく終了処理が行われるように実装してください。

`nngxFinalize()` は描画の権限がない状態でも呼び出すことができ、nngx 関数や gl 関数で確保したディスプレイバッファなどが自動的に解放されます。

コード 5-2. アプリケーションの終了に使用する関数

```
bool nn::applet::IsExpectedToCloseApplication(void);
nn::Result nn::applet::CloseApplication(const u8* pParam=NULL,
                size_t paramSize=0, nn::Handle handle=NN_APPLET_HANDLE_NONE);
```

`IsExpectedToCloseApplication()` は定期的に呼び出す以外にも、HOME メニューやライブラリアプレットからの復帰直後にも呼び出して、復帰待ちの間にアプリケーションを終了する要因が発生していないかどうかを確認しなければなりません。また、アプリケーションのロード中に終了要求が来た場合に備えて、APPLET ライブラリの初期化直後でも確認してください。オートセーブなど、アプリケーションの終了時に行いたい処理も、この関数が `true` を返したときに行ってください。

5.3.1.1. 終了処理の注意事項

CTR-SDK では、アプリケーションが任意のタイミングで nn::applet::CloseApplication() を呼び出しても、確保していたリソース類の解放が行われるように設計されていますが、その検証が十分に行われていません。安全にアプリケーションを終了するためにも、CloseApplication() を呼び出すまでに、以下の対応を踏まえて終了処理を実装してください。

> **補足:** これらの対応が不要となるように、今後のリリースで修正する予定です。

必須

nn::os::Alarm オブジェクトを作成していた場合は、Alarm オブジェクトのメンバ関数 Cancel() と Finalize() を順番に呼び出してください。

nn::os::Timer オブジェクトを作成していた場合は、Timer オブジェクトのメンバ関数 Stop() と Finalize() を順番に呼び出してください。

FS と UDS ライブラリは処理を止め、ライブラリの終了処理を行ってください。特に FS ライブラリでは、使用している各クラスの Finalize() を呼び出してください。NW4C などのように SDK 以外のパッケージでも、FS ライブラリを使用しているクラスについては確実に終了処理を行うようにしてください。

推奨

初期化した、すべてのライブラリの終了処理を行ってください。

5.3.2. HOME ボタンへの対処

HOME ボタンが押された場合、アプレットマネージャはアプリケーションが HOME メニューを起動しなければならない状態にあるかどうかを nn::applet::IsExpectedToProcessHomeButton() で通知しますので、アプリケーションはこの関数を定期的(フレームごとなど)に呼び出さなければなりません。この関数が true を返したときは、HOME メニューの起動を行う必要があります。アプリケーションはデバイスなどの動作を停止させ、すぐに HOME メニューを起動させてください。ただし、中断できない処理のために動作を停止できない場合は、後述する「HOME ボタン禁止アイコン」を表示して HOME メニューの起動を中止することができます。中止した場合は、nn::applet::ClearHomeButtonState() で HOME ボタンが押されたことを無効にしなければ、IsExpectedToProcessHomeButton() が true を返し続けることに注意してください。

コード 5-3. HOME メニューの起動に使用する関数

```
bool nn::applet::IsExpectedToProcessHomeButton(void);
bool nn::applet::ProcessHomeButton(void);
nn::applet::AppletWakeupState nn::applet::WaitForStarting(
        nn::applet::AppletId* pSenderId=NULL, u8* pParam=NULL,
        size_t paramSize=0, s32* pReadLen=NULL, nn::Handle *pHandle=NULL,
        nn::fnd::TimeSpan timeout=NN_APPLET_WAIT_INFINITE);
void nn::applet::ClearHomeButtonState(void);
void nn::applet::ProcessHomeButtonAndWait();
```

HOME メニューを起動するために必要な処理は nn::applet::ProcessHomeButton() を呼び出すだけで行われます。返り値が false の場合は HOME メニューからの復帰待ちを行う必要がありませんが、返り値が true のときは直後に nn::applet::WaitForStarting() を呼び出して HOME メニューからの復帰を待ってください。復帰するまで、キー入力の取得や描画などのデバイスの使用に制限がかかります。また、WaitForStarting() を呼び出したスレッドのみが停止することに注意してください。

優先度 16 もしくはより高い優先度（数値としては小さい）のスレッドが、HOME メニューへの遷移後も CPU を占有して動作するような実装にはしないでください。そのようなスレッドが存在した状態でゲームメモを起動すると、ゲームメモの起動中にフリーズします。なお、HOME メニューの表示中にアプリケーションのスレッドが動作し続けていると HOME メニュー等のパフォーマンスが低下しますので、HOME メニューへ遷移するときにはアプリケーションのすべてのスレッドを停止させることを推奨します。

`nn::applet::ProcessHomeButtonAndWait()` は `ProcessHomeButton()` の実行および待ち受けとスリープのハンドリングを行うラッパー関数です。

> **注意：** `ProcessHomeButton()` を呼び出すまでに、`nngxWaitCmdlistDone()` を呼び出すもしくはそれと同等の処理を行って GPU の描画コマンドの実行を確実に終了させておく必要があります。

> **補足：** `ProcessHomeButton()` を呼び出すことができるのは、GX ライブラリが使用可能である期間（`nngxInitialize()` の完了から `nngxFinalize()` を呼び出すまでの間）だけです。
>
> `ProcessHomeButton()` を呼び出す前にディスプレイバッファの設定とバッファスワップを行い、`nngxStartLcdDisplay()` を呼び出して LCD 出力を開始してください。LCD 出力が開始されていないと、黒画面のまま HOME メニューが起動する可能性があります。また、ディスプレイバッファの設定とバッファスワップを行っていない場合は、画面表示が不定なものになる可能性があります。

表 5-1. HOME メニュー表示中のデバイスへの対処

デバイス	HOME メニュー表示中のアプリケーションの対処
GPU/LCD	描画を止め、ディスプレイバッファを更新しないでください。また、HOME メニューが表示されるまでにグラフィックスの処理を完了させ、GPU が停止している状態でなければなりません。
デジタルボタンスライドパッド	特に対処は必要ありません。アプリケーションで取得する入力値に入力は反映されません。
タッチパネル	特に対処は必要ありません。アプリケーションで取得するサンプリング値に入力は反映されず、無効なサンプリング値（0）が返されます。
加速度センサー	特に対処は必要ありません。入力値を取得することができます。
ジャイロセンサー	入力値を取得することができます。復帰後にキャリブレーションを行わない場合は、HOME メニューの表示中も入力値を取得して補正が効いた状態にしておく必要があります。
サウンド	SND ライブラリの関数はメインスレッドおよびサウンドスレッドから呼び出されることを想定して設計されています。これらのスレッドからの呼び出しを行う場合は、HOME ボタン遷移時に注意することはありません。ただし、上記以外のスレッドから関数を呼び出す場合は、アプリケーションが中断されている状態で呼び出さないように、適切な対処が必要です。 なお、サウンドが停止するタイミングを制御することができませんので、ムービーのようにグラフィックスなどとの同期が必要な場合は再生状態を管理するなどの対処が必要です。
カメラ	特に対処は必要ありません。HOME メニューの起動までにカメラの終了処理を行っていない場合、カメラを使用する HOME メニューの機能は動作しません。
マイク	特に対処は必要ありません。
無線通信	ローカル通信以外であれば、特に対処は必要ありません。ただし、ブラウザなどを起動して無線通信を HOME メニューの表示中に行った場合、アプリケーションで実施していた通信が切断された状態で復帰します。 ローカル通信は継続することができます。ただし、HOME メニューの表示中に蓋が閉じられることを考慮する必要があり、蓋が閉じられたときにスリープする場合、ローカル通信を終了させていないとローカル通信の状態がエラーに遷移します。HOME メニューの起動までにローカル通信を終了させていない場合、ローカル通信を使用する HOME メニューの機能は動作しません。

NFC	NFP ライブラリを使用している場合は、HOME メニュー、及びアプレットに遷移する前に nn::nfp::Finalize() を呼び出して、NFP ライブラリの終了させてください。 詳細は「3DS プログラミングマニュアル – NFP 編」を参照してください。

HOME メニューから復帰した直後、アプリケーションが nngxSwapBuffers() を呼び出してディスプレイバッファを切り替えるまで、画面には HOME メニューへの遷移時にキャプチャされた画像が表示されています。ここで注意しなければならないのは、メインメモリや VRAM の内容は保護されていますが、GPU のレジスタ設定をアプリケーションで再設定しなければならないことです。頂点ロードアレイの設定やテクスチャユニットの設定だけでなく、フレームバッファやシェーダバイナリ、参照テーブルといった、すべてのレジスタ設定を再設定してください。

> **補足:** レジスタ設定を直接書き換えていなければ、nngxUpdateState(NN_GX_STATE_ALL) でレジスタ設定を復元することができます。

HOME ボタン押下の検知

HOME ボタン押下の検知は、nn::applet::SetHomeButtonCallback() で登録したコールバック関数への通知でも確認することができますが、基本的に nn::applet::IsExpectedToProcessHomeButton() を定期的に調べるように実装してください。

コード 5-4. HOME ボタンのコールバック関数の登録

```
void nn::applet::SetHomeButtonCallback(
        nn::applet::AppletHomeButtonCallback callback, uptr arg=0);
typedef bool (*nn::applet::AppletHomeButtonCallback)(
        uptr arg, bool isActive, nn::applet::CTR::HomeButtonState state);
```

コールバック関数の登録は nn::applet::SetHomeButtonCallback() で行います。登録されたコールバック関数が呼び出されたとき、引数 *arg* には SetHomeButtonCallback() の引数 *arg* に指定した値が格納されています。*isActive* には、アプリケーションが動作中であるかどうかが格納され、*state* には HOME ボタンの状態が格納されています。

HOME ボタンの状態は nn::applet::HomeButtonState 列挙子で定義されています。

表 5-2. HOME ボタンの状態

定義	説明
HOME_BUTTON_NONE	HOME ボタンは押されていない。
HOME_BUTTON_SINGLE_PRESSED	HOME ボタンがクリックされた。(200 ms 以上押し続けた)

HOME ボタンの状態は nn::applet::GetHomeButtonState() を呼び出すことでも確認することができます。押されたことが検知されると、HOME ボタンの状態は nn::applet::ClearHomeButtonState() が呼び出されるまで保持されます。ClearHomeButtonState() を呼び出すと、HOME ボタンの状態は HOME_BUTTON_NONE に戻ります。

コード 5-5. HOME ボタンの状態の確認

```
nn::applet::AppletHomeButtonState nn::applet::GetHomeButtonState(void);
```

コールバック関数で返す値によって、HOME ボタン押下の検出を有効にするか、無効にするかをアプリケーションで決定し、nn::applet::GetHomeButtonState() で取得する HOME ボタンの状態に反映するかどうかを制御することができます。true を返すと反映され、false を返すと反映されません。通常は *isActive* を返すように実装してください。

コールバック関数内ではフラグの制御などの軽い処理のみを実装し、HOME メニューの起動はコールバック関数内で行わないでください。

5.3.2.1. HOME メニューの起動までにできること

HOME ボタンが押されたことを検知してから HOME メニューを起動するまでの 0.5 秒以内であれば、アクションゲームで一時停止をかける、オートセーブを行うなどの処理を行うことができます。制限時間のあるパズルゲームなど、HOME メニューの背景にゲームの静止画像を表示したくない場合は、このタイミングで描画処理を行って画面を隠すことができますが、なるべくそのまま静止画像を表示するようにしてください。

アプリケーションの中断中に HOME メニューの上画面に表示されるキャプチャイメージを作る場合などでは、`nn::applet::IsExpectedToProcessHomeButton()` が `true` を返したことを確認してから描画処理を行いますが、`nn::applet::ProcessHomeButton()` で HOME メニューを表示する際には、GPU の処理が停止(グラフィックス処理を完了)している必要があります。実行中のグラフィックスコマンドがすべて完了するまで待つには、`nngxWaitCmdlistDone()` を呼び出します。また、LCD に表示される絵を確実に更新するためには、そのあとに `nngxWaitVSync()` を呼び出してください。

ユーザーが HOME メニューでアプリケーションを終了する可能性を考慮して、HOME メニューを起動する際には、HOME メニューで終了されても問題がないように実装してください。通常はアプリケーションに制御が戻ってから終了処理を行うことができますが、HOME メニュー表示中にスリープした状態でバッテリー切れになると終了処理を行えないことに注意してください。

5.3.2.2. HOME ボタン禁止アイコンの表示

セーブなどの中断できない処理を行っている最中に HOME ボタンが押されたことを検知したとき、その処理によってすぐに HOME メニューを表示できない場合は、HOME ボタン禁止アイコンを下画面の中央に表示し、HOME メニューの起動を中止することができます。

HOME ボタン禁止アイコンは、以下の仕様に従って実装してください。

- アイコン:CTR-SDK に付属している画像ファイル(HomeNixSign_Targa.tga)
- 表示位置:下画面の中央
- フェードイン:0.083 秒(60 フレーム換算で 5 フレーム)
- 表示時間:1 秒間(60 フレーム換算で 60 フレーム)
- フェードアウト:0.333 秒(60 フレーム換算で 20 フレーム)

表示演出中に HOME ボタンが押された場合は演出をそのまま続けてください。例えば、フェードアウト中に押されたときはフェードアウトの演出を続け、フェードインの演出に戻る必要はありません。また、表示演出中に HOME メニューの起動が可能になったとしても、HOME メニューを起動してはいけません。

5.3.2.3. スクリーンショットの投稿を禁止する

HOME メニューへの遷移時に作成されるキャプチャイメージを、ほかのアプリケーション(Miiverse などのシステムアプレット)で投稿できないようにすることができます。

コード 5-6. スクリーンショットの投稿の許可設定および設定値の取得

```
nn::Result nn::applet::SetScreenCapturePostPermission(
    const nn::applet::ScreenCapturePostPermission permission);
nn::Result nn::applet::GetScreenCapturePostPermission(
    nn::applet::ScreenCapturePostPermission* pPermission);
```

`SetScreenCapturePostPermission()` で投稿の可否を指定することができます。引数 *`permission`* に指定可能な値は `SCREEN_CAPTURE_POST_ENABLE` または `SCREEN_CAPTURE_POST_DISABLE` の 2 種類です。そのほかの値を指定することは禁止されています。なお、`nn::applet::RestartApplication()` でアプリケーションを再起動す

ると初期値(表 5-3 参照)に戻ります。

現在の設定は GetScreenCapturePosetPermission() で取得することができます。

表 5-3. スクリーンショットの投稿の許可設定および設定値の取得で使用する値

定義	説明
SCREEN_CAPTURE_POST_NO_SETTING	(指定禁止)設定の初期値で、投稿が許可された状態です。
SCREEN_CAPTURE_POST_NO_SETTING_OLD_SDK	(指定禁止)旧バージョンの SDK でリリースされたアプリケーションで取得される初期値で、投稿が許可された状態です。 カメラ使用時はスクリーンショットの投稿が禁止されます。ここでいうカメラ使用時とは、CAMERA ライブラリの nn::camera::Initialize() で初期化してから、nn::camera::Finalize() で終了するまでの期間を指します。
SCREEN_CAPTURE_POST_ENABLE	投稿が許可された状態です。
SCREEN_CAPTURE_POST_DISABLE	投稿が禁止された状態です。

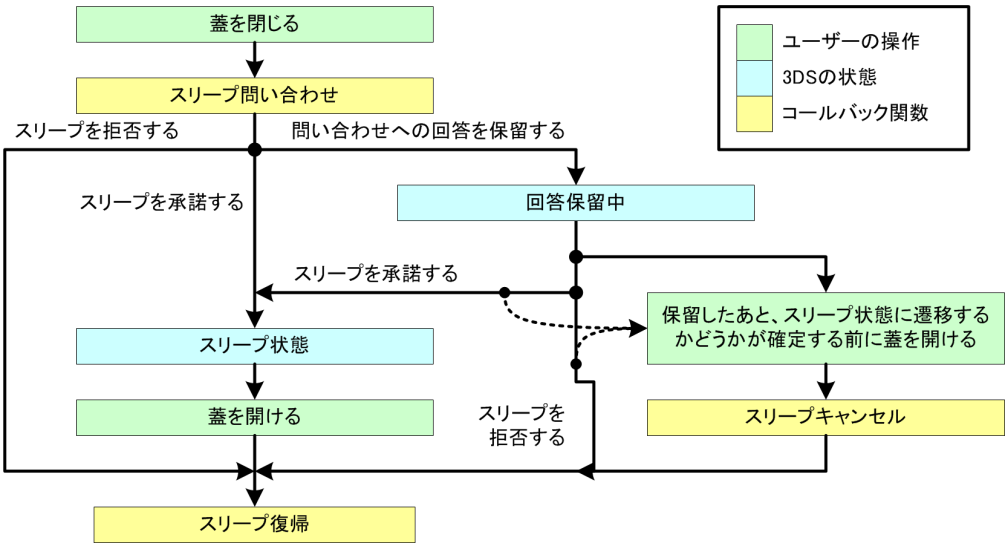
補足: CTR-SDK 7.x より前の SDK でリリースされたアプリケーションで、Miiverse へのスクリーンショットの投稿を禁止したい場合は弊社窓口までお問い合わせください。

5.3.3. スリープへの対処

3DS でも、DS と同様にアプリケーションの動作中に蓋が閉じられると、スリープ状態へ遷移するかどうかをアプリケーションで決定することができます。ただし 3DS では、HOME メニューを表示しているときなど、アプリケーションが中断中であってもスリープ状態への遷移が行われる可能性があります。

下図は、スリープの状態遷移を、関連するユーザーの操作、3DS の状態、コールバック関数の関係で示したものです。この項では、これらのコールバック関数を軸にスリープへの対処方法を説明し、そのあとにスリープに関連する情報を提供します。

図 5-2. スリープの状態遷移



5.3.3.1. スリープ問い合わせコールバック

アプリケーションの動作中に蓋が閉じられると、アプレットマネージャはアプリケーションに蓋閉じによるスリープ状態への移行を問い合わせます。問い合わせは登録されたコールバック関数に通知されます。

コード 5-7. スリープ問い合わせコールバック関数の登録

```
void nn::applet::SetSleepQueryCallback(
                nn::applet::AppletSleepQueryCallback callback, uptr arg=0);
typedef nn::applet::AppletQueryReply
                (*nn::applet::AppletSleepQueryCallback)(uptr arg);
```

コールバック関数の登録は nn::applet::SetSleepQueryCallback() で行います。登録されたコールバック関数が呼び出されたとき、引数 *arg* には SetSleepQueryCallback() の引数 *arg* に指定した値が格納されています。

問い合わせに対し、アプリケーションはコールバック関数の返り値ですぐにスリープ状態へ移行するかどうかをアプレットマネージャに通知します。

表 5-4. スリープ問い合わせコールバック関数の返り値に指定可能な値

定義	説明
REPLY_REJECT	スリープ状態への移行を拒否します。
REPLY_ACCEPT	スリープ状態への移行を承諾します。
REPLY_LATER	スリープ状態への移行を保留します。

蓋が閉じられた状態で無線通信を継続するときやサウンドの再生を継続するときには REPLY_REJECT を返してください。すぐにスリープ状態へ移行するときは REPLY_ACCEPT を返してもよいのですが、グラフィックスの処理などが停止しても問題ないタイミングでスリープ状態に移行するとは限りません。ほかの処理を停止してからスリープ状態に移行させるには、REPLY_LATER を返してスリープ状態への移行を一旦保留してください。ただし、スリープ状態に移行しても問題のないタイミングになったときは、REPLY_ACCEPT を引数に nn::applet::ReplySleepQuery() を速やかに呼び出し、イベントクラスなどでスリープからの復帰まで待機しなければなりません。

REPLY_LATER を返した時点で、LCD の電源がオフになるなどスリープに備えた状態になっています。また、アプレットマ

ネージャも HOME ボタンや電源ボタンの処理が行われない半停止状態になるため、ほかの処理の停止を行う場合、処理時間はできる限り短くしてください。ただし、後述するスリープキャンセルコールバックで保留中の蓋開けを適切にハンドリングしている場合は、数十秒程度であれば保留状態を続けてもかまいません。スリープキャンセルコールバックでハンドリングしていない場合、保留状態が 4 秒以下であればガイドラインに違反していないと判断します。4 秒以上スリープを禁止したい場合は、後述する REPLY_REJECT と nn::applet::EnableSleep() を組み合わせた、スリープ問い合わせコールバックの再発生を利用してください。

> **注意:** ローカル通信の初期化処理中(nn::uds::Initialize() の実行中)はスリープ状態への移行を承諾しないでください。無線通信の状態遷移のバッティングにより、正常に動作しない可能性があります。また、ローカル通信を終了させずにスリープ状態に移行した場合、UDS ライブラリは強制的にローカル通信を切断してエラー状態に遷移します。この状態になると、終了処理が行われるまで、一部の関数を除いて UDS ライブラリの関数はエラーを返すようになります。
>
> SND ライブラリの関数をメインスレッドおよびサウンドスレッドとは別のスレッドで呼び出している場合は、REPLY_ACCEPT でスリープ状態への移行を承諾したあと、スリープ状態から復帰するまで SND ライブラリの関数を呼び出すことがないよう、適切に制御する必要があります。
>
> なお、スリープ状態へ移行する間にサウンドスレッドの処理が停滞するなどして、通常 5 ms 周期で行われている nn::snd::WaitForDspSync() と nn::snd::SendParameterToDsp() の呼び出し周期が 100 ms を超えると、スリープ状態から復帰したときにサウンドスレッドが停止したままになることがあります。このスレッド停止はデバッグビルドでのみ、まれに起こることが報告されています。
>
> スリープ中にバッテリー切れになった場合はスリープから復帰せず、終了処理を行うことができません。また、スリープ中にゲームカードが抜ける場合もありますので、スリープ中に異常終了しても問題がないように実装してください。

なお、スリープ問い合わせコールバック関数をアプリケーションで登録していない場合は、REPLY_REJECT を返すコールバック関数を登録している状態と同じです。

アプリケーション中断中のコールバック

スリープ問い合わせコールバック関数は、HOME メニューの表示中など、アプリケーションが動作中でなくても呼び出されます。アプリケーションが動作中であるかどうかは、nn::applet::IsActive() で調べることができます。スリープ問い合わせコールバック関数内でアプリケーション中断中と判定された場合は、REPLY_ACCEPT を返してください。

コード 5-8. アプリケーション動作中の判定

```
bool nn::applet::IsActive(void);
```

返り値が true ならばアプリケーションは動作中、false ならば中断中です。

アプリケーションの動作が中断されている状態は Inactive 状態(動作している状態は Active 状態)といい、HOME メニューやライブラリアプレットが表示されている間のように、アプリケーションがその表示のための関数の呼び出しと終了待ちをしているスレッド以外は動作を継続することができます。また、アプリケーションの起動中、nn::applet::Enable() を呼び出すまでの間も、スリープ関連のコールバックは発生しませんが、Inactive 状態となります。

Inactive 状態では、スリープ問い合わせコールバックのほかにも、VSync のコールバックなども通常通り発生します。しかし、Inactive 状態ではメインスレッドが止まるように実装されているのが普通ですので、スリープ問い合わせコールバックを適切にハンドリングできず、意図しないデッドロック状態に陥る場合があります。例えば、メインスレッドでスリープのハンドリングを行っている場合、Inactive 状態で REPLY_LATER を返してしまうと、メインスレッドが止まっているためにスリープ状態への移行を承諾する機会が得られず、アプリケーションの状態が停滞してしまいます。

通信中などで常に REPLY_REJECT を返す場合を除き、HOME メニューの表示などで Inactive 状態になる前に FS ライブラリでのアクセスを止め、Inactive 状態でのスリープ問い合わせには常に REPLY_ACCEPT を返すような実装を推奨します。

なお、Inactive 状態でも動作するスレッドを作成する場合、適切にスリープのハンドリングを行っているならば Active 状態と同様の処理でも問題はありません。

スリープ対応の制御

nn::applet::EnableSleep() と nn::applet::DisableSleep() で、アプリケーションがスリープ状態への遷移に対応するかどうかを制御することができます。

コード 5-9. スリープ対応の制御

```
void nn::applet::EnableSleep(
                          bool isSleepCheck=nn::applet::SLEEP_IF_SHELL_CLOSED);
void nn::applet::DisableSleep(
                          bool isReplyReject=nn::applet::REPLY_REJECT_IF_LATER);
```

nn::applet::EnableSleep() はスリープ問い合わせコールバック関数で指定した返り値を有効にし、スリープ状態への移行シーケンスに移れるようにします。*isSleepCheck* に SLEEP_IF_SHELL_CLOSED を指定した場合は、関数を呼び出した時点での蓋の状態を確認し、蓋が閉じられていればスリープ問い合わせコールバック関数が呼び出されます。*isSleepCheck* に NO_SHELL_CHECK を指定した場合は、蓋の状態の確認とコールバック関数の呼び出しが行われないだけで、それ以外の動作に違いはありません。ちなみに、nn::applet::Enable(true) でスリープに関するコールバックを有効にしたときは、関数内で EnableSleep(NO_SHELL_CHECK) が実行されています。

isSleepCheck に SLEEP_IF_SHELL_CLOSED を指定して nn::applet::EnableSleep() を呼び出すと、蓋が閉じられているときにはスリープ問い合わせコールバックが発生することを利用して、スリープ問い合わせコールバック関数で REPLY_LATER ではなく REPLY_REJECT を返すと、スリープを拒否している間に必要な処理を行って、スリープ状態への遷移途中での蓋開けを考慮する必要がなくなります。つまり、スリープ問い合わせコールバックで REPLY_REJECT を返すと、それから蓋を閉じている間はスリープ問い合わせコールバックが発生しませんが、スリープ状態へ移行できるようになったときに EnableSleep(SLEEP_IF_SHELL_CLOSED) を呼び出すことで、まだ蓋が閉じられている場合に再びスリープ問い合わせコールバックを発生させることができます。なお、DisableSleep() を呼んでいなくても、EnableSleep() を任意のタイミングで呼び出しても構いません。ただし、アプレットマネージャの状態によっては関数の呼び出しから戻るまでに数ミリ秒以上かかる可能性がありますので、毎フレーム呼び出すことは推奨しません。

> **注意:** アプリケーションの起動中に蓋が閉じられていると、nn::applet::Enable() を呼び出すまでにスリープ問い合わせに対して REPLY_REJECT を返しているとみなされます。スリープに対応するアプリケーションは、起動中に蓋が閉じられていてもスリープ問い合わせコールバックが再発生するように、Enable() のあとに EnableSleep(SLEEP_IF_SHELL_CLOSED) を呼び出してください。

nn::applet::DisableSleep() はスリープ問い合わせコールバック関数で指定した返り値を無効にし、どのような返り値を指定しても REPLY_REJECT を返した状態にします。*isReplyReject* に REPLY_REJECT_IF_LATER を指定したときは、さらに nn::applet::ReplySleepQuery(REPLY_REJECT) を実行し、すでにスリープ状態へ移行しようとしていた場合にそれをキャンセルします。*isReplyReject* に NO_REPLY_REJECT を指定したときは、ReplySleepQuery(REPLY_REJECT) を実行しないだけで、それ以外の動作に違いはありません。

DisableSleep() が呼び出されたあとでも、蓋が閉じられたときはスリープ問い合わせコールバックが発生しますが、コールバック関数の返り値は無視され、常に REPLY_REJECT が返されたと見なされます。ただし、アプリケーションが中断中(nn::applet::IsActive() が false を返す状態)に呼び出されたスリープ問い合わせコールバックの返り値は有効となる(REPLY_REJECT 以外も返る)ことに注意してください。また、DisableSleep() は内部にカウンタを持っていませんので、複数回呼び出された場合でも、一度の EnableSleep() の呼び出しでコールバック関数の返り値が有効になります。

HOME メニューの表示(nn::applet::ProcessHomeButton())や電源メニューの表示(nn::applet::ProcessPowerButton())、ライブラリアプレットの起動など、アプリケーションが中断されるような関数を呼び出す場合は、必ずその前に DisableSleep() でスリープ問い合わせを拒否し、復帰したあとで EnableSleep() を呼び出してください。

> **注意:** アプリケーションの終了処理を行う前に DisableSleep(REPLY_REJECT_IF_LATER) を実行しなければ、終了処理の途中で蓋が閉じられると、スリープ状態で停止したままになる可能性があります。

通知状態の確認と保留への回答

アプリケーションでスリープ状態への遷移を保留しているかどうかは、nn::applet::IsExpectedToReplySleepQuery() で確認することができます。この関数が true を返したときは、アプリケーションはスリープ状態へ遷移しても構わないかどうかを判断し、保留していた回答を nn::applet::ReplySleepQuery() で確定しなければなりません。

スリープ状態の遷移をハンドリングする場合は、IsExpectedToReplySleepQuery() を定期的(フレームごとなど)に呼び出し、決められたタイミングでスリープ状態に遷移するような実装を推奨します。

コード 5-10. スリープ関連の通知状態の確認と保留への回答

```
bool nn::applet::IsExpectedToReplySleepQuery(void);
void nn::applet::ReplySleepQuery(nn::applet::AppletQueryReply reply);
```

5.3.3.2. スリープ復帰コールバック

蓋が開けられてスリープから復帰するときの通知はコールバック関数で受け取ります。

コード 5-11. スリープ復帰コールバック関数の登録

```
void nn::applet::SetAwakeCallback(nn::applet::AppletAwakeCallback callback,
                                  uptr arg=0);
typedef void (*nn::applet::AppletAwakeCallback)(uptr arg);
```

コールバック関数の登録は nn::applet::SetAwakeCallback() で行います。登録されたコールバック関数が呼び出されたとき、引数 *arg* には SetAwakeCallback() の引数 *arg* に指定した値が格納されています。

スリープ復帰コールバック関数内では、スリープからの復帰を待機しているイベントクラスをシグナル状態にするなどの簡単な処理のみを行ってください。このコールバック関数は、スリープ問い合わせコールバック関数で REPLY_REJECT を返すなど、蓋が閉じられてもスリープ状態に移行しない場合はすぐに呼び出されます。そのため、蓋が開けられたかどうかを、このコールバック関数が呼び出されることで判断することはできません。

> **注意:** スリープ状態に移行すると LCD 表示が OFF になります。そのため、スリープ復帰後に画面表示の準備ができた段階で nn::gx::StartLcdDisplay() を呼び出さなければ画面に何も表示されなくなります。ただし、画面の準備ができていない状態や、HOME メニューの表示中、ライブラリアプレットの実行中にスリープしてから復帰した状態で nn::gx::StartLcdDisplay() を呼び出すと、画面表示が一瞬乱れてしまう可能性があります。その場合は画面表示の準備が完了し、正常に表示できる状態になってから呼び出してください。

5.3.3.3. スリープキャンセルコールバック

蓋が閉じられてから、アプリケーションがスリープ状態への移行を行うまでに蓋が開けられた(スリープキャンセル)場合に呼び出されるコールバック関数を nn::applet::SetSleepCanceledCallback() で登録することができます。

コード 5-12. スリープキャンセルコールバック関数の登録

```
void nn::applet::SetSleepCanceledCallback(
                nn::applet::AppletSleepCanceledCallback callback, uptr arg=0);
typedef void (*nn::applet::AppletSleepCanceledCallback)(uptr arg);
```

このコールバック関数は、蓋が閉じられてからスリープ状態に移行するまでにデータをセーブするといった、時間のかかる処理を行っている間に、蓋が開けられたことを検知してスリープの処理を中断する場合などに利用します。

このコールバック関数内でなにもしなければ、スリープ状態への移行を保留したままになります。つまり、明示的に ReplySleepQuery(REPLY_REJECT) を呼び出さないと、スリープはキャンセルされたことになりません。また、このコールバック関数が呼び出された場合でも、スリープ復帰コールバック関数が呼び出されることに注意してください。

すぐにスリープ状態に移行する場合や処理にかかる時間が短い場合は、このコールバック関数を登録する必要はありません。このような場合は、スリープキャンセルコールバックの対応を無理に行わないことも検討してください。

通常、REPLY_LATER を返した場合にのみスリープキャンセルコールバックが発生します。ただし、高速に蓋の開閉を行った場合は、REPLY_ACCEPT や REPLY_REJECT を返したあとにスリープキャンセルコールバックが発生する可能性があります。スリープ問い合わせコールバックへの回答前に蓋開けが発生するケースは、高速な蓋の開閉を行わないと再現しないため、デバッグには困難が伴う可能性があります。

なお、スリープ関連のコールバックが同時に、並行して呼び出されることはありません。スリープキャンセルコールバックは、スリープ問い合わせコールバックとスリープ復帰コールバックの間に 0 または 1 回発生することが保証されています。

> **補足:** nn::applet::SetSleepCanceledCallback() の関数リファレンスに実装サンプルが掲載されています。

5.3.3.4. スリープ中に禁止されている処理

> **補足:** 現在、アプリケーションを実装する上でスリープ中に禁止されている処理はありません。

5.3.3.5. 蓋を閉じたときのデバイスの動作状態

スリープ問い合わせコールバックの返り値や nn::applet::ReplySleepQuery() の引数に REPLY_ACCEPT を指定するか REPLY_REJECT を指定するかで、蓋が閉じられた状態での動作が異なるデバイスが存在します。

表 5-5. 蓋を閉じたときのデバイスの動作状態

デバイス	REPLY_ACCEPT	REPLY_REJECT
LCD	ディスプレイバッファの更新および VSync 同期が停止し、LCD 表示が OFF になります。	ディスプレイバッファの更新が停止し、バックライトのみ OFF になります。
デジタルボタン	一切の入力を受け付けません。	CTR は L と R ボタンのみ入力を受け付けます。SNAKE は L / R / ZL / ZR ボタンの入力を受け付けます。
タッチパネル	サンプリング値を取得することができません。	無効なサンプリング値(0)を返します。

加速度センサー	歩数計として動作します。入力値を取得することができません。	歩数計として動作します。入力値を取得することができます。
ジャイロセンサー	停止します。	入力値を取得することができます。
サウンド	停止します。	通常はスピーカーからの出力を停止し、強制的にヘッドホンから出力されますが、スピーカーからも出力できるように設定可能です。
カメラ	停止します。	停止します。
マイク	停止します。	停止します。
無線通信	停止します。ローカル通信は事前に停止させておくことを推奨します。	停止しません。
赤外線通信	停止します。スリープ前に切断処理を実行し、完了まで待機することを推奨します。	停止しません。
NFC	停止します。	停止します。

メインスレッドを含め、アプリケーションで作成したスレッドはスリープ状態に移行すると停止します。スリープ状態に移行していなかった場合、スレッドの動作は継続していますがデバイス自体が停止しているとイベントなどの通知がなくなります。

> **注意:** ニンテンドーDS/DSi とは異なり、通常、蓋を閉じた状態ではスピーカーからサウンドが出力されないことに注意してください。スピーカーから出力されるようにするには、「10.5.7. 蓋閉じによるスリープを拒否した際のサウンド出力」を参照してください。

5.3.4. 電源ボタンへの対処

160 ミリ秒以上電源ボタンが押された場合、アプレットマネージャはアプリケーションで電源ボタンが押されたことに対応しなければならないかどうかを `nn::applet::IsExpectedToProcessPowerButton()` で通知しますので、アプリケーションはこの関数を定期的(フレームごとなど)に呼び出さなければなりません。この関数が `true` を返したときは、速やかに `nn::applet::ProcessPowerButton()` を呼び出してください。

コード 5-13. 電源ボタンへの対応に使用する関数

```
bool nn::applet::IsExpectedToProcessPowerButton(void);
bool nn::applet::ProcessPowerButton(void);
void nn::applet::ProcessPowerButtonAndWait();
```

`nn::applet::ProcessPowerButton()` を呼び出すと、「電源メニュー」を表示するために、システムに制御が移ります。電源メニューが表示されることにより、終了時にアプリケーションが特別な画面を表示する必要がありません。この関数を呼び出した直後に、アプリケーションは終了処理の開始が許可されるのを `nn::applet::WaitForStarting()` で待ち受ける必要があります。`WaitForStarting()` から処理が戻ったあとは、`nn::applet::IsCloseApplication()` が `true` を返す状態になっていますので、「5.3.1. アプリケーション終了要求への対処」を参照してアプリケーションの終了処理を行ってください。

`nn::applet::ProcessPowerButtonAndWait()` は `ProcessPowerButton()` の実行および待ち受けとスリープのハンドリングを行うラッパー関数です。

> **補足:** ProcessPowerButton() を呼び出すことができるのは、GX ライブラリが使用可能である期間(nngxInitialize() の完了から nngxFinalize() を呼び出すまでの間)だけです。
>
> ProcessPowerButton() を呼び出す前にディスプレイバッファの設定とバッファスワップを行い、nngxStartLcdDisplay() を呼び出して LCD 出力を開始してください。LCD 出力が開始されていないと、黒画面のまま電源メニューが起動する可能性があります。また、ディスプレイバッファの設定とバッファスワップを行っていない場合は、画面表示が不定なものになる可能性があります。

5.3.5. アプリケーションのハンドリング例

スリープや HOME ボタンのハンドリングは、以下に挙げるポイントを踏まえて実装してください。

- メインループなど、定期的に呼ばれる場所で、以下の判定を行ってください。
  - nn::applet::IsExpectedToProcessHomeButton() の判定
    true が返されたときは、nn::applet::ProcessHomeButton() を呼び出す
    (HOME メニューを表示したくないもしくはできない場合は、ガイドラインに従って HOME メニュー禁止アイコンを表示する)
  - nn::applet::IsExpectedToProcessPowerButton() の判定
    true が返されたときは、nn::applet::ProcessPowerButton() を呼び出す
  - nn::applet::IsExpectedToCloseApplication() の判定
    true が返されたときは、アプリケーションの終了処理を行う
- nn::applet::ProcessHomeButton() や nn::applet::ProcessPowerButton() を呼び出す前に、描画コマンドの実行を完了させておいてください。実行中の描画コマンドは nngxWaitCmdlistDone() で完了を待つことができます。
- nn::applet::ProcessHomeButton() や nn::applet::ProcessPowerButton() を呼び出したあとは、必ず nn::applet::WaitForStarting() を呼び出してください。
- nn::applet::Enable() や nn::applet::WaitForStarting() から制御が戻ってきたときは、必ず nn::applet::IsExpectedToCloseApplication() の判定を行ってください。

スリープや HOME ボタンのハンドリングのコード例を以下に示します。

コード 5-14. ハンドリングのコード例

```
nn::os::LightEvent sAwakeEvent(true);
nn::os::LightEvent sTransitionEvent(true);
nn::os::CriticalSection sFileSystemCS(WithInitialize);

// スリープ問い合わせコールバック
nn::applet::AppletQueryReply mySleepQueryCallback(uptr arg)
{
    sAwakeEvent.ClearSignal();
    return (nn::applet::IsActive() ? REPLY_LATER : REPLY_ACCEPT);
}

// スリープ復帰コールバック
void myAwakeCallback(uptr arg)
{
    sAwakeEvent.Signal();
}

// スリープ
```

```
void sleepApplication()
{
    // スリープ中にFSライブラリによるアクセスが起こらないようにロックする
    // ロックできなかった場合は次のフレームに再試行する
    if (sFileSystemCS.TryEnter())
    {
        _app_prepareToSleep();
        nn::applet::ReplySleepQuery(REPLY_ACCEPT);
        sAwakeEvent.Wait();
        _app_recoverFormSleep();
        sFileSystemCS.Leave();
        nn::gx::StartLcdDisplay();
    }
}

// アプリケーション終了
void exitApplication()
{
    _app_finalize();
    nn::applet::CloseApplication();
}

// メインループ
void nnMain()
{
    // イベントがシグナル状態のときは通常動作
    sAwakeEvent.Signal();
    sTransitionEvent.Signal();
    // コールバックの設定
    nn::applet::SetSleepQueryCallback(mySleepQueryCallback);
    nn::applet::SetAwakeCallback(myAwakeCallback);
    nn::applet::Enable(true);
    // アプリケーションロード中の終了要求に対応
    if (nn::applet::IsExpectedToCloseApplication())
    {
        exitApplication();
    }
    // アプリケーションロード中に蓋が閉じられていた場合に対応
    nn::applet::EnableSleep(SLEEP_IF_SHELL_CLOSED);

    while (true)
    {
        _app_exec();

        // スリープ問い合わせへの返答
        if (nn::applet::IsExpectedToReplySleepQuery())
        {
            if (_app_isRejectSleep())
            {
                // スリープできない状況ならば拒否を通知する
                nn::applet::ReplySleepQuery(REPLY_REJECT);
            } else {
                sleepApplication();
```

```
        }
    }
    // アプリケーションの終了要求
    if (nn::applet::IsExpectedToCloseApplication())
    {
        exitApplication();
    }
    // HOME ボタン
    if (nn::applet::IsExpectedToProcessHomeButton())
    {
        if (_app_isSuppressedHomeButton())
        {
            _app_drawSuppressedHomeButtonIcon();
            nn::applet::ClearHomeButtonState();
        } else {
            // HOME メニューやライブラリアプレットなどの起動のように、
            // メインスレッドを止めるような状態遷移の場合は、
            // イベントを非シグナル状態にする
            sTransitionEvent.ClearSignal();
            // HOME メニュー表示中にスリープしたときに、
            // FSライブラリによるアクセスが起こらないようにロックする
            // ロックできなかった場合は次のフレームに再試行する
            if (sFileSystemCS.TryEnter())
            {
                _app_prepareToHomeButton();
                nn::applet::ProcessHomeButtonAndWait();
                sFileSystemCS.Leave();
                if (nn::applet::IsExpectedToCloseApplication())
                {
                    exitApplication();
                }
                sTransitionEvent.Signal();
                // GPU のレジスタ設定を戻す必要があります
                _app_recoverGpuState();
            }
        }
    }
    // 電源ボタン
    if (nn::applet::IsExpectedToProcessPowerButton())
    {
        // FSライブラリによるアクセスが起こらないようにロックする
        // ロックできなかった場合は次のフレームに再試行する
        if (sFileSystemCS.TryEnter())
        {
            nn::applet::ProcessPowerButtonAndWait();
            sFileSystemCS.Leave();
            if (nn::applet::IsExpectedToCloseApplication())
            {
                exitApplication();
            }
            // GPU のレジスタ設定を戻す必要があります
            _app_recoverGpuState();
        }
```

```
        }
    }
}
// メインスレッド外でFSライブラリによるアクセスを行う場合は
// 以下のようにロックしてからアクセスする
{
    // スリープ状態の場合、起きるまでスレッドを止める
    sAwakeEvent.Wait();
    // メインスレッドに制御が戻ってくるまでスレッドを止める
    s_TransitionEvent.Wait();
    {
        os::CriticalSection::ScopedLock lock(sFileSystemCS);
        //
        // ここに FS にアクセスする処理を書く
        //
    }
}
```

5.3.6. アプリケーションの再起動

nn::applet::RestartApplication() を呼び出すことで、HOME メニューに戻ることなくアプリケーションを再起動することができます。再起動後のアプリケーションに渡すパラメータを指定することができ、指定されたパラメータは nn::applet::GetStartupArgument() で取得することができます。

コード 5-15. アプリケーションの再起動に使用する関数

```
nn::Result nn::applet::RestartApplication(
                const void* pParam = NULL,
                size_t paramSize = NN_APPLET_PARAM_BUF_SIZE);
bool nn::applet::GetStartupArgument(
                void* pParam,
                size_t paramSize = NN_APPLET_PARAM_BUF_SIZE);
```

pParam にはパラメータのバイト列を、*paramSize* にはパラメータのバイトサイズを指定します。バイト列の最大長は NN_APPLET_PARAM_BUF_SIZE です。

通常、nn::applet::RestartApplication() は制御を戻さずにアプリケーションを再起動させます。この関数によって再起動したかどうかは、nn::applet::GetStartupArgument() が true を返すかどうか(渡したパラメータを取得できたかどうか)で判断することができます。

5.3.7. 本体設定へのジャンプ

アプリケーションから本体設定のインターネット設定、ペアレンタルコントロール、データ管理の各画面へとジャンプすることができます。

コード 5-16. 本体設定へのジャンプ

```
nn::Result nn::applet::JumpToInternetSetting(void);
nn::Result nn::applet::JumpToParentalControls(
                nn::applet::AppletParentalControlsScene scene =
                nn::applet::CTR::PARENTAL_CONTROLS_TOP);
nn::Result nn::applet::JumpToDataManagement(
                nn::applet::AppletDataManagementScene scene =
```

```
                nn::applet::CTR::DATA_MANAGEMENT_STREETPASS);
bool nn::applet::IsFromMset(nn::applet::AppletMsetScene* pScene = NULL);
```

インターネット設定へは nn::applet::JumpToInternetSetting()、ペアレンタルコントロールへは nn::applet::JumpToParentalControls()、データ管理へは nn::applet::JumpToDataManagement() でそれぞれジャンプすることができます。アプリケーションを一旦終了させてから本体設定にジャンプしますので、事前に終了処理を行ってください。これらの関数によるジャンプに失敗したときはフェイタルエラーとなります。また、関数の成否に関わらず、アプリケーションに制御は戻りません。

ペアレンタルコントロールには複数の設定項目があり、引数でどの項目にジャンプするかが選択可能です。

表 5-6. ジャンプ先の画面の指定

定義	ジャンプ先の画面
PARENTAL_CONTROLS_TOP	ペアレンタルコントロールのトップ画面
PARENTAL_CONTROLS_COPPACS	ペアレンタルコントロールの COPPACS の認証処理画面
DATA_MANAGEMENT_STREETPASS	データ管理画面のすれちがい通信設定画面

上記の関数でジャンプした本体設定を終了すると、アプリケーションが再起動されます。本体設定を終了したあとの再起動では、nn::applet::IsFromMset() が true を返します。本体設定のどのシーンにジャンプしていたのかを判別する場合は、引数 *pScene* にジャンプ先シーンを格納する変数を指定します。IsFromMset() は nn::applet::Enable() よりもあとに呼び出してください。

5.3.8. 起動時に取得可能な初期パラメータ

アプリケーションから再起動した場合や本体設定にジャンプして再起動された場合など、アプリケーションがどのような経緯で起動されたのかを取得する関数には、以下のものがあります。

表 5-7. 起動時の初期パラメータを取得する関数

関数	取得可能な初期パラメータ
nn::applet::GetStartupArgument()	nn::applet::RestartApplication() で指定したパラメータ
nn::applet::IsFromMset()	アプリケーションからジャンプしていた本体設定の画面
nn::news::IsFromNewsList()	おしらせの種類や指定されたパラメータ。おしらせリストで「ソフトを起動」を選んだ場合に通知される
nn::frineds::IsFromFriendList()	フレンドのフレンドキー。フレンドリストで「プレイ中のソフトに参加する」を選んだ場合に通知される

5.3.9. ニンテンドーeショップへのジャンプ

アプリケーションからニンテンドーeショップのページにジャンプすることができます。

コード 5-17. ニンテンドーeショップのインストール確認

```
bool nn::applet::IsEShopAvailable();
```

本体更新の適用状態によっては、ユーザーの 3DS 本体にニンテンドーeショップがインストールされていない場合があります。必ず nn::applet::IsEshopAvailable() でニンテンドーeショップがインストール済みであることを確認してからジャンプしてください。

> **補足:** ニンテンドーeショップが本体に存在しなかった場合は、インターネットで本体更新を行う必要があることをユーザーに通知することをおすすめします。
>
> 例えば、「ニンテンドーeショップを使用できません。ニンテンドーeショップを使用するには、インターネットで本体更新をしてください。」といったメッセージを表示してください。

> **補足:** ダウンロードアプリの場合はインストール状況の確認は不要です。

コード 5-18. ニンテンドーeショップへのジャンプ(詳細ページへのジャンプ)

```
nn::Result nn::applet::JumpToEShopTitlePage(bit32 uniqueId);
```

ジャンプ後に表示されるページは、引数 *uniqueId* で指定されたユニーク ID を持つタイトルの詳細ページです。なお、指定されたタイトルが検索公開タイトル(タイトル名の検索で表示されるタイトル)でなければ、そのページは表示されません。また、引数 *uniqueId* にニンテンドーeショップに登録されていないタイトルのユニーク IDや不正な値を指定した場合は、ニンテンドーeショップ上でエラーが表示されます。

コード 5-19. ニンテンドーeショップへのジャンプ(パッチページへのジャンプ)

```
nn::Result nn::applet::JumpToEShopPatchPage(bit32 uniqueId);
```

認証サーバやアカウントサーバから、nn::act::ResultApplicationUpdateRequired などのアプリケーションの更新を必要とするエラーが返ってきた場合に、ニンテンドーeショップのタイトルのパッチページへ誘導するために使用します。

ジャンプ後に表示されるページは、引数 *uniqueId* で指定されたユニーク ID を持つタイトルのパッチページです。

アプリケーションの更新をリマスターで運用している場合は、ニンテンドーeショップにタイトルのパッチページが存在しません。その場合、タイトルが存在しないエラーが表示され、ニンテンドーeショップ起動画面に遷移します。リマスターで運用している場合は、 nn::applet::JumpToEShopTitlePage() を使用してください。

ニンテンドーeショップにジャンプするこれらの関数は、呼ばれた時にアプリケーションが終了しますので、事前に終了処理を行ってください。なお、ニンテンドーeショップを終了してもアプリケーションは再起動されません。

> **補足:** 本関数を使用してジャンプを行う場合には、ジャンプ先のタイトルを OMAS に申請してください。申請についての詳細は、ガイドラインの「eコマース」を参照してください。

5.3.10. 電子説明書へのジャンプ

HOME メニューの「説明書」をタッチしたときに起動する電子説明書に、アプリケーションからジャンプすることができます。

コード 5-20. 電子説明書へのジャンプ

```
void nn::applet::JumpToManual();
```

ジャンプ後、アプリケーションの取扱説明書が表示されます。

この関数はアプリケーションを中断してジャンプしますので、呼び出した後は nn::applet::WaitForStarting() で HOME メニューからの復帰を待ってください。電子説明書が終了すると、HOME メニューがアプリケーションを復帰させます。

5.4. FS ライブラリの初期化

多くの場合、メディア上のファイルにアクセスするために FS ライブラリの初期化を次に行うことになるでしょう。FS ライブラリの初期化は nn::fs::Initialize() を呼び出して行います。メディア上のファイルへのアクセスには、必ず FS ライブラリで用意されているクラスを使用してください。

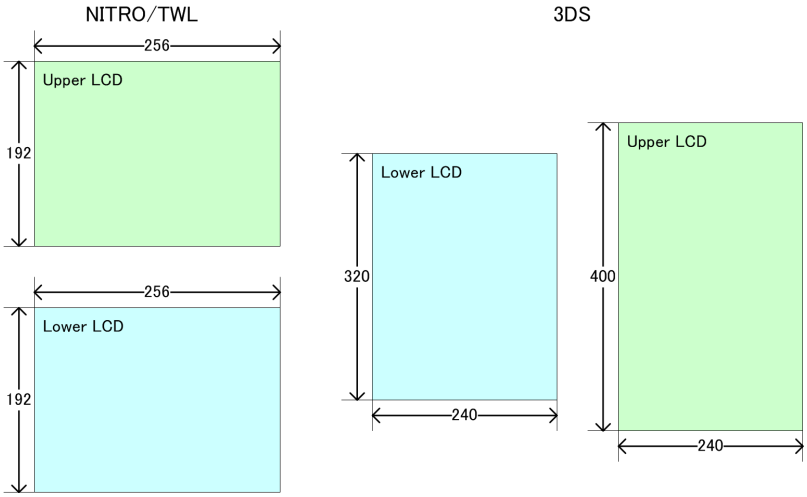
5.5. GX ライブラリの初期化

LCD への表示や 3D グラフィックスの描画を行うには GX ライブラリを使用します。GX ライブラリの初期化は nngxInitialize() を呼び出して行います。初期化にはライブラリからのメモリ領域確保要求と解放要求を処理する関数の指定が必要になります。

nngxInitialize() を呼び出したあとは、グラフィックスコマンドの実行のためのコマンドリストオブジェクトの作成や、LCD に表示するためのディスプレイバッファやレンダーバッファの確保などを行います。

3DS では、図 5-3 のように LCD の配置されている方向や解像度が NITRO/TWL と異なっている点に注意が必要です。

図 5-3. LCD の配置と解像度



ここでは、サンプルプログラムでの実装を例に説明します。使用されている関数や設定の詳細については「3DS プログラミングマニュアル - グラフィックス基本編」や SDK の関数リファレンスを参照してください。また、立体視表示については「3DS プログラミングマニュアル - グラフィックス応用編」を参照してください。

5.5.1. ライブラリの初期化

nngxInitialize() の呼び出しには、ライブラリからのメモリ確保と解放の要求を受け付けるアロケータとデアロケータの指定が必要です。

コード 5-21. GX ライブラリの初期化

```
void SetupNngxLibrary(const uptr fcramAddress, const size_t memorySize)
{
    InitializeMemoryManager(fcramAddress, memorySize);

    if (nngxInitialize(GetAllocator, GetDeallocator) == GL_FALSE)
    {
        NN_PANIC("nngxInitialize() failed.\n");
    }
}
```

5.5.1.1. アロケータ

グラフィックス処理で必要となるテクスチャイメージデータやレンダーバッファなど、GPU がアクセスするメモリ領域は nngxInitialize() で指定したアロケータを介して確保されます。

アロケータには 4 つの引数が渡されます。

第 1 引数(確保先のメモリ空間)

第 1 引数はメモリ領域を確保するメモリ空間の指定で、GLenum 型で受け取ります。受け取った値によって、確保先のメモリ空間が異なります。

NN_GX_MEM_FCRAM が渡された場合、メモリ領域はメインメモリから確保するのですが、その領域はデバイスメモリから確保されなければなりません。デバイスメモリとは、周辺デバイスとメインプロセスの双方からアクセスする際に OS がアドレスの整合性を保証しているメインメモリの一部です。デバイスメモリの詳細や確保の方法については、「3.1.2. デバイスメモリ」を参照してください。

NN_GX_MEM_VRAMA または NN_GX_MEM_VRAMB が渡された場合、メモリ領域の確保先は、前者が VRAM-A、後者が VRAM-B となります。それぞれの VRAM の先頭アドレスとサイズは nngxGetVramStartAddr() と nngxGetVramSize() に VRAM-A ならば NN_GX_MEM_VRAMA、VRAM-B ならば NN_GX_MEM_VRAMB を渡して呼び出すことで取得することができます。

第 2 引数(メモリ領域の用途)

第 2 引数はメモリ領域の用途指定で、GLenum 型で受け取ります。受け取った値によって、確保するメモリ領域のアライメントが異なります。

NN_GX_MEM_TEXTURE が渡された場合、メモリ領域はテクスチャイメージデータとして使用されます。アライメントはフォーマットに関係なく 128 Byte です。

NN_GX_MEM_VERTEXBUFFER が渡された場合、メモリ領域は頂点バッファとして使用されます。アライメントは格納するデータ型によって 1、2、4 Byte と異なりますが、データ型まではアロケータに渡されないため、最大の 4 Byte でメモリ領域を確保するように実装することを推奨します。

NN_GX_MEM_RENDERBUFFER が渡された場合、メモリ領域はレンダーバッファ(カラー、デプス、ステンシル)として使用されます。アライメントは 1 ピクセルあたりのビット数(16、24、32)によって 32、96、64 Byte と異なりますが、フォーマットまではアロケータに渡されません。そのため、アプリケーションで使用するレンダーバッファのフォーマットを固定するか、アライメントを最小公倍数の 192 Byte にしてメモリ領域を確保するように実装してください。

NN_GX_MEM_DISPLAYBUFFER が渡された場合、メモリ領域はディスプレイバッファとして使用されます。アライメントはフォーマットに関係なく 16 Byte です。**VRAM に確保する場合は最後尾から 1.5 MByte には確保しないでください。**

NN_GX_MEM_COMMANDBUFFER が渡された場合、メモリ領域はコマンドリストとして使用されます。アライメントは 16 Byte です。

NN_GX_MEM_SYSTEM が渡された場合、メモリ領域はライブラリのシステム領域として使用されます。アライメントは確保サイズによって 1、2、4 Byte と異なりますが、実装を簡単にするためにも、最大の 4 Byte でメモリを確保するように実装することを推奨します。

第 3 引数(オブジェクトの名前)

第 3 引数はオブジェクトの名前(ID)で、GLuint 型で受け取ります。第 2 引数が NN_GX_MEM_SYSTEM 以外である場合に渡され、メモリを管理する際の一情報として使用します。

第 4 引数(メモリ領域のサイズ)

第 4 引数は確保するメモリ領域のサイズで、GLsizei 型で受け取ります。指定されたサイズのメモリ領域を確保してください。

アプリケーションはこれらの引数から、適切なアライメントとサイズでメモリ領域を確保し、その先頭アドレスをライブラリに void* 型で返します。4 つの引数と確保したメモリ領域の先頭アドレスを組にして記憶しておくなど、後述するデアロケータでのメモリ領域の解放も含めて、メモリの管理はアプリケーションで行わなければなりません。

5.5.1.2. デアロケータ

アロケータで確保したメモリ領域が解放される際に、nngxInitialize() で指定したデアロケータが呼び出されます。デアロケータに渡される引数は 4 つで、第 1 から第 3 引数までは確保時にアロケータに渡された引数と同じ値が渡され、第 4 引数にはメモリ領域の先頭アドレスが渡されます。

アプリケーションはこれらの引数からアロケータで確保したメモリ領域を特定し、解放しなければなりません。

5.5.2. コマンドリストオブジェクトの作成

nngxInitialize() の呼び出しが完了したら、次はグラフィックス処理で呼び出される gl、nngx 関数の実行に必要となるコマンドリストオブジェクトを作成しなければなりません。

まず、nngxGenCmdlists() でコマンドリストのオブジェクトを作成し、nngxBindCmdlist() でカレントのコマンドリストに指定したあと、nngxCmdlistStorage() で 3D コマンドバッファとコマンドリクエストを蓄積するためのメモリ領域を確保します。

コード 5-22. コマンドリストオブジェクトの作成

```
void CreateCmdList()
{
    nngxGenCmdlists(1, &m_CmdList);
    nngxBindCmdlist(m_CmdList);
    nngxCmdlistStorage(256*1024, 128);
    nngxSetCmdlistParameteri(NN_GX_CMDLIST_RUN_MODE, NN_GX_CMDLIST_SERIAL_RUN);
}
```

このコード例では 3D コマンドバッファが 256 KByte、コマンドリクエストが最大 128 個蓄積可能なコマンドリストを 1 つだけ確保していますが、複数のコマンドリストを確保し、フレーム単位などで切り替えて使用することもできます。その際、3D コマンドを蓄積した順にコマンドリストを実行しなければならないことに注意しなければなりません。

5.5.3. ディスプレイバッファ、レンダーバッファの確保

LCD 表示に必要なディスプレイバッファと、レンダーターゲットとなるレンダーバッファを確保します。

フレームバッファを確保するには、フレームバッファオブジェクトにカラーバッファ、デプスバッファ、ステンシルバッファの各レンダーバッファを関連付ける必要があります。カラーバッファにはカラー情報が、デプスバッファにはデプス情報が、ステンシルバッファにはステンシル情報がレンダリング時に書き込まれます。ステンシルバッファを利用する場合は、デプスバッファとバッファを共有しなければならないことに注意してください。

上画面と下画面でフォーマットが同じ、かつ平行してレンダリングする必要がない場合は、メモリリソースを節約するためにもフレームバッファオブジェクトとレンダーバッファは上画面用と下画面用で共有することを推奨します。その際、バッファの幅と高さは両方の画面が収まるような設定を使用してください。

コード 5-23. レンダーバッファの確保

```
void CreateRenderbuffers(
        GLenum format, GLsizei width, GLsizei height)
{
    glGenFramebuffers(1, m_FrameBufferObject);
    glGenRenderbuffers(2, m_RenderBuffer);

    glBindRenderbuffer(GL_RENDERBUFFER, m_RenderBuffer[0]);
    glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMA, format,
        width, height);
    glBindFramebuffer(GL_FRAMEBUFFER, m_FrameBufferObject);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
        GL_RENDERBUFFER, m_RenderBuffer[0]);
    glBindRenderbuffer(GL_RENDERBUFFER, m_RenderBuffer[1]);
    glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMB,
        GL_DEPTH24_STENCIL8_EXT, width, height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
        GL_RENDERBUFFER, m_RenderBuffer[1]);
}
```

このコード例ではカラーバッファとデプス・ステンシルバッファを VRAM-A と VRAM-B に分けて確保しています。

通常のフレームバッファアーキテクチャではカラーバッファの内容をそのまま表示することができますが、3DS のカラーバッファのフォーマットはブロックフォーマットであるため、リニアフォーマットに変換しなければ LCD に表示することができません。そのため、3DS ではディスプレイバッファをカラーバッファと LCD の間に挟み、カラーバッファからディスプレイバッファへのコピーとフォーマット変換を行います。その際、ハードウェアによるアンチエイリアスと上下反転を行うことができます。

初期化の段階ではディスプレイバッファの確保のみを行います。

コード 5-24. ディスプレイバッファの確保

```
void CreateDisplaybuffers(
        GLenum format0, GLsizei width0, GLsizei height0, GLenum area0,
        GLenum format1, GLsizei width1, GLsizei height1, GLenum area1)
{
    // 上画面(DISPLAY0)
    nngxActiveDisplay(NN_GX_DISPLAY0);
    nngxGenDisplaybuffers(2, m_Display0Buffers);
    nngxBindDisplaybuffer(m_Display0Buffers[0]);
    nngxDisplaybufferStorage(format0, width0, height0, area0);
    nngxBindDisplaybuffer(m_Display0Buffers[1]);
    nngxDisplaybufferStorage(format0, width0, height0, area0);
    nngxDisplayEnv(0, 0);
```

```
    // 下画面(DISPLAY1)
    nngxActiveDisplay(NN_GX_DISPLAY1);
    nngxGenDisplaybuffers(2, m_Display1Buffers);
    nngxBindDisplaybuffer(m_Display1Buffers[0]);
    nngxDisplaybufferStorage(format1, width1, height1, area1);
    nngxBindDisplaybuffer(m_Display1Buffers[1]);
    nngxDisplaybufferStorage(format1, width1, height1, area1);
    nngxDisplayEnv(0, 0);
}
```

このコード例では LCD ごとにディスプレイバッファを 2 つ用意してマルチバッファリングを行っています。マルチバッファリングで複数確保しなければならないのはディスプレイバッファだけです。

ディスプレイバッファはメインメモリ(デバイスメモリ)上に確保することができます。ディスプレイバッファのフォーマットとカラーバッファのフォーマットで、ピクセル当たりに必要なビット数がディスプレイバッファ＞カラーバッファとなる組み合わせはカラーバッファからのコピー時にエラーが発生します。

LCD 表示に必要な初期化処理は以上です。各種バッファの確保が行われるように、この時点で一度 `nngxRunCmdlist()` を呼び出しておいてください。

5.6. メモリ管理

アプリケーションに割り当てられたヒープメモリ、デバイスメモリ、VRAM はアプリケーション自身で管理しなければなりません。CTR-SDK では、アプリケーションでメモリを管理するためのクラスを用意しています。

5.6.1. メモリブロック

メモリブロックは NITRO/TWL や Revolution ではアリーナと呼ばれていた機能に相当し、ある程度の大きさを持つ領域を確保したあと、後述する FND ライブラリのヒープクラスなどでメモリを切り出して使用することに適しています。FND ライブラリで定義されているヒープクラスは、メモリブロックを引数にインスタンスを生成することもできます。

メモリブロックは 4096 Byte 単位で確保されます。通常、アプリケーションに割り当てられたヒープメモリから、デバイスメモリに確保できない作業メモリやスタックなどを確保するために利用します。また、メモリブロックを利用することで、アプリケーションの実装を簡略化できるライブラリも存在します。

メモリブロックの確保は `nn::os::MemoryBlock` クラスやスタック用の `nn::os::StackMemoryBlock` クラス、`nn::os::StackMemory` クラスのインスタンスを生成して行います。これらのインスタンスを生成する前に `nn::os::InitializeMemoryBlock()` を呼び出してメモリブロックとして使用するメモリ領域を指定しなければなりません。

コード 5-25. メモリブロックとして使用するメモリ領域の指定

```
void nn::os::InitializeMemoryBlock(uptr begin, size_t size);
```

begin にはメモリ領域の先頭アドレス、*size* にはメモリ領域のサイズを指定します。どちらも `nn::os::MEMORY_BLOCK_UNITSIZE`（4096 Byte）のアライメントでなければなりません。

メモリブロックは `nn::os::MemoryBlock` クラス(スタック用は `nn::os::StackMemoryBlock` クラス)のインスタンスを確保したいメモリ領域のサイズで生成するか、空のインスタンスを生成したあとに `Initialize()` を呼び出して初期化することで確保されます。サイズの指定は 4096 Byte 単位で行わなければなりません。

確保したメモリブロックの先頭アドレスは `GetAddress()` で、サイズは `GetSize()` で取得することができます。また、スタック用には、スレッドに直接渡すことができるように `GetStackBottom()` と `GetStackSize()` のインターフェースが

用意されています。スタック用にはありませんが、SetReadOnly() と IsReadOnly() で読み込み専用属性の設定と取得を行うことができます。

不要になったメモリブロックはメンバ関数の Finalize() を呼び出すことで明示的に解放することができます。

> **補足:** nn::os::SetupHeapForMemoryBlock() は一部のサンプルデモで利用しているために残されている関数です。メモリブロックを利用するために呼び出すことは推奨しません。
>
> nn::os::MemoryBlock クラスと nn::os::StackMemoryBlock クラスは、今後の更新で互換性のない変更が行われる可能性がありますので、アプリケーションでの使用を推奨しません。

5.6.2. フレームヒープ

フレームヒープは、初期化時に指定されたメモリ領域から、指定サイズのメモリ領域を切り出して使用するメモリ管理クラスです。アライメントの指定ではバイト境界のほかに、その符号によってメモリ領域をヒープの先頭から確保するのか、末尾から確保するのかを選択することができます。ヒープの先頭からメモリ領域を確保している場合に限り、最後に確保したメモリ領域のサイズを変更することができます。

確保したメモリ領域は個別に解放することができず、すべてのメモリ領域か、先頭から確保したもの、末尾から確保したもののように一括して解放することになります。

フレームヒープのインスタンスは、ロック操作を指定するクラステンプレート(nn::fnd::FrameHeapTemplate)、ロック操作をしない nn::fnd::FrameHeap、スレッドセーフ(ロック操作は nn::os::CriticalSection)な nn::fnd::ThreadSafeFrameHeap のいずれかで行うことができます。

5.6.3. ユニットヒープ

ユニットヒープは、初期化時に指定されたメモリ領域から、一定サイズのメモリ領域(ユニット)を切り出して使用するメモリ管理クラスです。アライメントの指定は初期化時に行い、デフォルトでは 4 Byte が指定されます。連続して確保したメモリ領域が、連続するメモリ領域から確保されるとは限りません。
確保したメモリ領域は個別に解放することができます。確保していたメモリ領域を一括で解放する場合は、Invalidate() を呼び出したあとに Finalize() を呼び出して、ヒープの終了処理を行います。そのヒープを続けて利用する場合には Initialize() を呼び出して再構築してください。

ユニットヒープのインスタンスは、ロック操作を指定するクラステンプレート(nn::fnd::UnitHeapTemplate)、ロック操作をしない nn::fnd::UnitHeap、スレッドセーフ(ロック操作は nn::os::CriticalSection)な nn::fnd::ThreadSafeUnitHeap のいずれかで行うことができます。

5.6.4. 拡張ヒープ

拡張ヒープは、初期化時に指定されたメモリ領域から、指定サイズのメモリ領域を切り出して使用するメモリ管理クラスです。アライメントの指定ではバイト境界のほかに、空いているメモリ領域をヒープの先頭から探して確保するのか、末尾から探して確保するのかを符号で選択することができます。また、確保したメモリ領域のサイズを変更することができます。

確保したメモリ領域は個別に解放することができます。確保と解放を繰り返すと、GetTotalFreeSize() で取得した空き容量より小さなサイズでもメモリ領域を確保できなくなる可能性があります。これはヒープ内に指定されたサイズの連続したメモリ領域がないためです。GetAllocatableSize() であれば、連続するメモリ領域として確保することのできる最大サイズを取得することができます。

拡張ヒープのインスタンスは、ロック操作を指定するクラステンプレート(nn::fnd::ExpHeapTemplate)、ロック操作をしない nn::fnd::ExpHeap、スレッドセーフ(ロック操作は nn::os::CriticalSection)な

nn::fnd::ThreadSafeExpHeap のいずれかで行うことができます。

5.7. DLL 機能（RO ライブラリ）

DLL 機能とは、動的にメモリに読み込んだモジュールを使用する機能です。この機能を利用することで、アプリケーションのコード部分が使用するメモリ量を削減したり、アプリケーション起動までの時間を短縮したりすることができます。

CTR-SDK では、DLL 機能を利用するためのライブラリとして RO ライブラリを提供します。

5.7.1. DLL 機能に関する用語

CTR-SDK で提供されている DLL 機能に関する用語を解説します。これらの用語の意味や用いられ方は、一般的なものと異なる場合があります。

モジュール
実行コードを意味のある単位でまとめたものです。

静的モジュール
エントリ関数（nnMain()）が含まれ、起動時にロードされるモジュールです。静的モジュールのサイズを小さくすることで起動時間を短縮することができます。

動的モジュール
動的にロードして実行することのできるモジュールです。機能ごとに分けられた動的モジュールを入れ替えることでメモリ消費量を削減することができます。

シンボル
変数や関数のモジュール内での位置を示す情報です。

参照（インポート）
ほかのモジュール内のシンボルで示された変数や関数を使用すること、およびその宣言です。

解決
参照するシンボルが利用可能な状態になることです。

公開（エクスポート）
シンボルで示された変数や関数を、ほかのモジュールで使用できるようにすること、およびその宣言です。

公開種別
シンボルをどのような形式で公開するかを示します。名前、インデックス、オフセットの 3 形式あります。

5.7.2. RO ライブラリの特徴と制限

一般的な DLL 機能の実装と比較して、RO ライブラリには以下のような特徴があります。

- 3 種類の公開種別
  シンボルを解決するための方式として、オフセット、インデックス、名前の 3 つの形式のいずれか 1 つを使用することができます。どの形式を使用するかはシンボルごとに選択できますが、CTR-SDK のビルドシステムではモジュール単位での選択のみをサポートしています。
  公開種別の指定方法については「CTR-SDK ビルドシステムマニュアル（DLL 編）」または「ビルドシステム構築ガイド（DLL 編）」を参照してください。

- モジュール間の参照を自動で解決
  モジュールをロードするだけでモジュール間の参照を自動的に解決し、関数や変数が利用可能となります。公開種別が名前またはインデックスのシンボルであれば、シンボルのポインタを手動で取得することもできます。
- モジュールのメモリへのロードはアプリケーションで行う
  動的モジュールのメモリ上への読み込みや配置の調整はアプリケーションで行う必要があります。
- C++ で記述されたコードに対応
  C++ で記述されたコードを動的モジュールにすることができます。また、C++ のシンボルをモジュール間で参照/解決することができます。ただし、公開種別が名前のシンボルのポインタを手動で取得する場合はマングルされた名前を使用する必要があります。
- モジュールの境界を越えた C++ 例外の受け渡し
  あるモジュールで throw された例外を、異なるモジュールで catch することができます。

RO ライブラリには以下の上限および制約があります。

- 1 つのモジュールが名前で公開できるシンボルは最大 65535 個です。
- シンボルを名前で公開する場合の最大の名前長は 8192 文字です。
- 動的モジュールに C の標準関数やそれに類するものを含めることはできません。
- 動的モジュールで 8 Byte を超えるアライメントを指定した静的な変数/定数を定義した場合、そのアライメント制約は無視されます。
- 同時にロードできる動的モジュールの上限数は 64 個です。
- weak シンボルなど C/C++ 言語非標準の機能を使用した場合、意図した動作にならない場合があります。

スタティックライブラリを動的モジュールとして使用する場合は、ライブラリの作成者に動的モジュールとして使用可能かどうかを確認してください。

> **注意:** SDK で提供されているスタティックライブラリを動的モジュールとして使用することは禁止されています。
>
> 同じスタティックライブラリを複数の動的モジュールに組み込まないでください。グローバル変数などが複数存在するなど、ライブラリの意図しない動作による不具合が発生する可能性があります。

5.7.3. RO ライブラリで使用するファイル

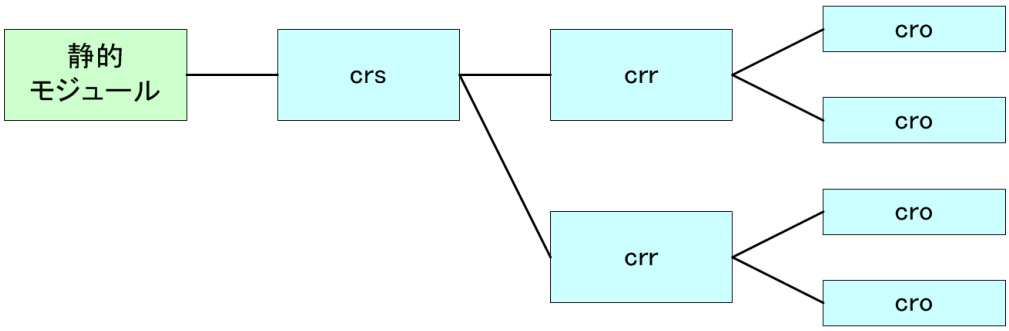
RO ライブラリでは DLL 機能を実現するために、以下のファイル形式で作成されたデータを使用しています。

表 5-8. RO ライブラリで使用するファイル

ファイル形式(拡張子)	説明
crs	静的モジュールの参照/公開情報が格納されているファイルです。実行コードは含まれていません。なお、静的モジュールはアプリケーションに 1 つしか存在しないため、crs ファイルも 1 つしか存在しません。
crr	1 つ以上の動的モジュールの管理情報が格納されているファイルです。ROM アーカイブ内の特定のディレクトリ(/.crr/)直下に配置されていなければなりません。
cro	動的モジュールの実行コードと参照/公開情報が格納されているファイルです。

各ファイルの関係を図にすると以下のようになります。

図 5-4. RO ライブラリで使用するファイルの関係



通常のアプリケーションでは、アプリケーションで使用するすべての cro の情報を含んだ crr を作成し、crr を 1 つだけ使用すれば十分です。crr が使用するメモリも最適化したい場合や追加プログラムの配信を行う場合は、複数の crr を使用します。

> **補足:** 各ファイルの作成方法については「CTR-SDK ビルドシステムマニュアル(DLL 編)」または「ビルドシステム構築ガイド(DLL 編)」を参照してください。

5.7.4. 公開と参照

ヘッダファイルが共有されている場合、モジュール間での関数の呼び出しや変数の参照は通常のソースコードと同様に記述するだけです。シンボルが公開されているかどうかを特に意識する必要はありません。

ヘッダファイルが共有されていない場合、ほかのモジュールの関数を呼び出したり変数を参照したりするには、モジュールのソースコードで明示的に公開と参照を宣言する必要があります。

明示的な公開と参照の宣言には、以下の定義をソースコードに記述します。

表 5-9. 明示的な公開と参照の宣言で使用する定義

定義	説明
NN_DLL_EXPORT	ほかのモジュールに公開する関数や変数の宣言に記述します。
NN_DLL_IMPORT	ほかのモジュールで公開されている関数や変数の宣言に記述します。

コード 5-26. 明示的な公開と参照の宣言のコード例

```
// 明示的な公開の宣言のコード例
NN_DLL_EXPORT int g_Variable;
NN_DLL_EXPORT void Function()
{
    // (略)
}

// 明示的な参照の宣言のコード例
extern NN_DLL_IMPORT int g_Variable;
extern NN_DLL_IMPORT void Function();
```

> **補足:** 公開を明示的に宣言したシンボルは必ず公開され、モジュール内で使用されていなくてもデッドストリップされません。

5.7.5. 公開種別による違い

シンボルの公開種別によって、以下のような違いがあります。

表 5-10. 公開種別による違い

項目	名前	インデックス	オフセット
参照情報のサイズ	大	小	小
公開情報のサイズ	大	小	0
モジュールのロードに要する時間	長	中	短
ポインタの手動取得	可	可	不可
参照するモジュールを先に作成	可	不可	不可
ビルドステップ	通常	通常	多

名前＞インデックス＞オフセットの順に高機能となりますが、ファイルサイズや処理時間といったコストがかかります。

通常はオフセットを選択し、シンボルのポインタを手動で取得する場合やライブラリのように動的モジュールを使用する場合は、必要に応じてインデックスや名前を選択します。

> **補足:** 静的モジュールのシンボルについても公開種別の指定を行うことができます。

5.7.6. 動的モジュールの特別な関数

動的モジュールのソースコード内に特定の名前の関数が公開されている場合、モジュールの初期化処理などのタイミングで、RO ライブラリがその関数を呼び出します。

コード 5-27. 特別な関数

```
extern "C" NN_DLL_EXPORT void nnroProlog();
extern "C" NN_DLL_EXPORT void nnroEpilog();
extern "C" NN_DLL_EXPORT void nnroUnresolved();
```

`nnroProlog()` は、`DoInitialize()` によるモジュールの初期化処理がすべて完了したあとに呼び出されます。モジュール固有の初期化処理を実装することができます。

`nnroEpilog()` は、`DoFinalize()` によるモジュールの終了処理がすべて完了したあとに呼び出されます。モジュール固有の終了処理を実装することができます。

`nnroUnresolved()` は、モジュールから外部のシンボルを使用したときに、そのシンボルが未解決だった場合に呼び出されます。未解決状態のシンボルを呼び出したことを検知する処理を実装することができます。

これらの関数の実装は必要に応じてアプリケーションで行ってください。

> **補足:** nnroUnresolved() は静的モジュールでも使用可能ですが、参照が解決されていたシンボルが未解決になった場合には機能しないという制約があります。

5.7.7. 基本の処理フロー

ここでは、動的モジュールを使用する際の基本の処理フローを順を追って説明します。

5.7.7.1. RO ライブラリの初期化

RO ライブラリの初期化は nn::ro::Initialize() で行います。初期化には crs ファイルが必要ですので、事前にメモリに読み込んでおかなければなりません。なお、crs ファイルの内容を格納するバッファは、デバイスメモリ以外のメモリ領域に確保し、その先頭アドレスが nn::ro::RS_ALIGNMENT (4096 Byte) のアライメント、バッファサイズが nn::ro::RS_UNITSIZE (4096 Byte) の倍数でなければならないことに注意してください。

crs ファイルが格納されているバッファは RO ライブラリの管理下に置かれるため、RO ライブラリの終了処理を行うまで、内容を書き換えたり、バッファを解放したりしないでください。

5.7.7.2. 管理情報の登録

動的モジュールを使用する前に、それを管理する crr ファイルを読み込み、nn::ro::RegisterList() で管理情報の登録を行わなければなりません。なお、crr ファイルの内容を格納するバッファは、デバイスメモリ以外のメモリ領域に確保し、その先頭アドレスが nn::ro::RR_ALIGNMENT (4096 Byte) のアライメント、バッファサイズが nn::ro::RR_UNITSIZE (4096 Byte) の倍数でなければならないことに注意してください。

crr ファイルが格納されているバッファは RO ライブラリの管理下に置かれるため、nn::ro::RegisterList() で返された nn::ro::RegistrationList クラスのポインタで Unregister() を呼び出して管理情報の登録を解除するまで、内容を書き換えたり、バッファを解放したりしないでください。

5.7.7.3. 動的モジュールのロード

動的モジュールのロードは nn::ro::LoadModule() で行います。cro ファイルはアプリケーションで事前にメモリに読み込んでおかなければなりません。なお、cro ファイルの内容を格納するバッファは、デバイスメモリ以外のメモリ領域に確保し、その先頭アドレスが nn::ro::RO_ALIGNMENT_LOAD_MODULE (4096 Byte) のアライメント、バッファサイズが nn::ro::RO_UNITSIZE_LOAD_MODULE (4096 Byte) の倍数でなければならないことに注意してください。

ほかにも .data/.bss セクションとして使用されるバッファを用意する必要があります。このバッファは先頭アドレスが nn::ro::BUFFER_ALIGNMENT (8 Byte) のアライメントでなければなりません。バッファのサイズは、nn::ro::GetSizeInfo() に cro ファイルの内容(先頭から REQUIRED_SIZE_FOR_GET_SIZE_INFO バイト)を渡して取得した、nn::ro::SizeInfo 構造体の bufferSize メンバの値以上でなければなりません。なお、SizeInfo 構造体にはロード後に使用可能となるメモリ領域の情報も格納されています。ロード後に解放されるメモリ領域のサイズが bufferSize よりも大きいならば、このバッファをそのメモリ領域から捻出することができます。

引数 *fixLevel* での指定によって、動的モジュールの機能やロード後に解放されるメモリ領域が以下のように異なります。指定を省略した場合は FIX_LEVEL_1 が指定されたものとして処理されます。

表 5-11. fixLevel の指定による動的モジュールの動作の違い

項目	FIX_LEVEL_0	FIX_LEVEL_1	FIX_LEVEL_2	FIX_LEVEL_3
アンロード後の再利用	可	不可	不可	不可
以降にロードされるモジュールに含まれているシンボルとの自動リンク	可	可	不可	不可

以降にロードされるモジュールと、このモジュールに含まれているシンボルとの自動リンク	可	可	可	不可
このモジュールに含まれているシンボルのアドレス取得	可	可	可	不可
現在までにロードされているモジュールに含まれているシンボルとの自動リンク	可	可	可	可
ライブラリの管理下となる領域の最終アドレス	fix0End	fix1End	fix2End	fix3End

ロード後に解放されるメモリ領域のサイズは FIX_LEVEL_0＜FIX_LEVEL_1＜FIX_LEVEL_2≦FIX_LEVEL_3 となります。また、シンボルの公開種別が名前であれば、解放されるメモリ領域のサイズはほかの公開種別よりも大きくなります。

引数 *doRegister* には通常 true を指定して自動リンク対象にし、ほかの動的モジュールがロードされたときに、ロード時点で未解決だったシンボルへの参照が自動的に解決されるようにします。false を指定した場合、これまでにロードされている動的モジュールに含まれているシンボルへの参照は解決されますが、ほかの動的モジュールがロードされたときに自動的に解決されません。この場合、nn::ro::LoadModule() で返される nn::ro::Module クラスのポインタで Link() を呼び出し、手動で解決する必要があります。

5.7.7.4. 動的モジュールの使用開始

動的モジュールに含まれている関数や変数を利用する前に、nn::ro::LoadModule() で返された nn::ro::Module クラスのポインタで DoInitialize() を呼び出し、動的モジュール内のグローバルオブジェクトの構築や nnroProlog() に実装された初期化処理を実行する必要があります。

初期化処理でほかの動的モジュールのグローバルオブジェクトを参照していなければ、DoInitialize() はモジュールのロード直後に呼び出すことができますが、相互にグローバルオブジェクトを参照している場合はロードまでで処理を止めておき、すべての動的モジュールをロードしたあとに DoInitialize() を呼び出してください。

動的モジュールの名前を Module クラスの GetName() で取得することができます。この関数で返される名前はビルド時に指定したモジュール名（CTR-SDK のビルドシステムであれば TARGET_MODULE で指定した名前）と同じです。また、ロードされている動的モジュールを nn::ro::FindModule() で検索する際の名前にも使用します。

動的モジュールから外部への参照がすべて解決済みであるかどうかは、Module::IsAllSymbolResolved() で判断することができます。なお、動的モジュールへの参照を含めて、そのモジュールの解決済みの参照をすべて未解決状態にするには Module::Unlink() を呼び出します。参照を再度解決させる場合は Module::Link() を呼び出します。

シンボルのポインタの手動取得には、nn::ro::GetPointer() ですべてのモジュールのシンボルから名前で検索する方法と、nn::ro::Module::GetPointer() で特定の動的モジュールに含まれているシンボルから名前またはインデックスで検索する方法があります。また、nn::ro::GetAddress() では、名前で検索したシンボルのアドレスを取得することができます。

5.7.7.5. 動的モジュールの使用終了

動的モジュールが不要になり、そのモジュールが使用していたメモリ領域を解放するには、まず Module クラスの DoFinalize() で動的モジュール内のグローバルオブジェクトの破棄や nnroEpilog() に実装された終了処理を実行する必要があります。そして終了処理が完了したあとに、Module クラスの Unload() で動的モジュールをアンロードします。

アンロードによって動的モジュールはロード前の状態に戻る（ライブラリの管理下からメモリを解放し、参照をすべて未解決状態にする）ため、cro ファイルの内容を格納していたバッファを解放することができます。

アンロードした動的モジュールを再度使用する場合は、基本的に cro ファイルの読み込みからやり直す必要があります。た

だし、引数 *fixLevel* に FIX_LEVEL_0(FIX_LEVEL_NONE)を指定してロードしていた場合は、ある条件を満たせば cro ファイルを読み込み直すことなく、使用していたバッファなどを nn::ro::LoadModule() でそのまま再利用することができます。再利用の条件は、静的変数の初期値を使用しない、もしくは nnroProlog() ですべての静的変数を初期化することです。これは、アンロードを実行しても動的モジュールを使用したときに書き換えられた静的変数がロード前の状態に戻らないためです。

5.7.7.6. 管理情報の登録解除

管理情報で使用していたメモリ領域を解放するには、nn::ro::RegisterList() で返された nn::ro::RegistrationList クラスへのポインタで Unregister() を呼び出し、管理情報の登録を解除する必要があります。

管理情報の登録が解除されると、crr ファイルの内容を格納していたバッファがライブラリの管理下から解放されます。なお登録解除後は、その crr ファイルが管理している動的モジュールを新たにロードできなくなりますが、ロード済みの動的モジュールに対する処理は正常に行うことができます。

5.7.7.7. RO ライブラリの終了処理

RO ライブラリの終了処理は nn::ro::Finalize() で行います。

nn::ro::Finalize() を呼び出したあとは、crs ファイルの内容を格納していたバッファだけでなく、crr ファイルや cro ファイルを格納していたバッファなど、RO ライブラリで使用していたメモリのすべてを解放することができるようになります。

5.7.8. 動的モジュールの列挙、探索

nn::ro::Module::Enumerate() で、自動リンクでロードされたかどうかに関係なく、すべての動的モジュールを列挙することができます。Enumerate() の引数には、nn::ro::Module::EnumerateCallback を継承したクラスを渡します。動的モジュールごとに、動的モジュールへのポインタを引数に operator() が呼び出されます。

nn::ro::Module::Find() で動的モジュールを探索することができます。指定された文字列と名前が一致する動的モジュールが見つかればモジュールへのポインタを返し、見つからなければ NULL を返します。ただし、探索の対象となるのは、自動リンクでロードされた動的モジュールのみです。

5.7.9. 動的モジュールが使用しているメモリ領域の情報

動的モジュールが使用しているメモリ領域の情報を、nn::ro::Module::GetRegionInfo() で取得することができます。メモリ領域の情報は、引数 *pri* に渡した nn::ro::RegionInfo 構造体に格納されます。

6. 入力装置からの入力

本体に搭載されているデジタルボタン、スライドパッド、タッチパネル、加速度センサー、ジャイロセンサー、マイク、カメラなどの入力装置からの入力は、それぞれ対応するライブラリとクラスによって簡単にアプリケーションに取り込むことができます。

6.1. デバイスを利用するライブラリについて

アプリケーションから本体に搭載されている各種入力装置などのデバイスにアクセスするには、CTR-SDK で提供されるライブラリを使用しなければなりません。デバイスを利用するライブラリの多くは、関数呼び出しの返り値に `nn::Result` クラスが返されます。

関数呼び出しとその処理が成功したかどうかは、`nn::Result` クラスの `IsSuccess()` または `IsFailure()` を呼び出して判定します。`IsSuccess()` が `true` を返した場合は成功、`false` を返した場合は失敗です。`IsFailure()` はその逆で、`true` を返した場合は失敗、`false` を返した場合は成功です。

`nn::Result` クラスにはエラーの重要度や概要、エラーの発生したモジュールの情報が記録されています。処理結果の成否判定に加え、これらの詳細な情報から発生したエラーへの対処方法を決定しなければならない場合もあります。

6.2. HID ライブラリ

HID ライブラリでは、デジタルボタン（十字ボタン、A/B/X/Y/L/R ボタン、START ボタン）、スライドパッド、タッチパネル、加速度センサー、ジャイロセンサー、デバッグパッド、拡張スライドパッドからの入力を扱うことができます。

HID ライブラリの初期化は `nn::hid::Initialize()` を呼び出して行います。初期化関数を重複して呼び出した場合は、何も行わずにエラーを返します。

ライブラリの初期化後、各デバイスの入力は対応するクラスを介して取得することができます。デバイスの種類によって、入力のサンプリングが開始されるタイミングやサンプリング周期が異なります。サンプリングの終了タイミングは、加速度センサーとジャイロセンサーは生成したクラスを破棄したとき、拡張スライドパッドは `nn::hid::ExtraPad::StopSampling()` を呼び出したとき、そのほかは HID ライブラリを終了したときです。

> **補足：** SNAKE に搭載されている C スティック、ZL ボタン、ZR ボタンは CTR に拡張スライドパッドを常時装着された状態と同じように処理することができます。その際、C スティックはスライドパッド（R）に対応し、ZL ボタンと ZR ボタンはそのまま ZL ボタンと ZR ボタンに対応します。
>
> C スティックについての情報および拡張スライドパッドとの相違点については「6.2.7. C スティック」を参照してください。

表 6-1. 入力の種類と扱うクラス、サンプリング周期の一覧

種類	入力を扱うクラス	サンプリング開始の契機	周期
デジタルボタンとスライドパッド	`nn::hid::PadReader`	HID ライブラリの初期化	4 ms
タッチパネル	`nn::hid::TouchPanelReader`	HID ライブラリの初期化	4 ms
加速度センサー	`nn::hid::AccelerometerReader`	Reader クラスの生成	平均 10 ms
ジャイロセンサー	`nn::hid::GyroscopeReader`	Reader クラスの生成	平均 10 ms

デバッグパッド	nn::hid::DebugPadReader	HID ライブラリの初期化（接続していた場合）	16 ms
拡張スライドパッド	nn::hid::ExtraPadReader	ExtraPad クラスの StartSampling()	8 ～ 32 ms
C スティック、ZL / ZR ボタン	nn::hid::ExtraPadReader	ExtraPad クラスの StartSampling()	8 ～ 32 ms

補足： SNAKE に搭載されている C スティック、ZL ボタン、ZR ボタンに対して、ライブラリではサンプリング周期を 8～32 ms の間で設定することができますが、ハードウェアに設定可能なサンプリング周期である 10～21 ms の間で設定することを推奨します。

HID ライブラリの使用を終了するときは nn::hid::Finalize() を呼び出してください。なお、nn::hid::ExtraPadReader クラスを使用している場合は、先に nn::hid::ExtraPad::Finalize() を呼び出さなければなりません。

6.2.1. デジタルボタンとスライドパッド

nn::hid::PadReader クラスの Read() または ReadLatest() を呼び出すことで、デジタルボタンとスライドパッドの入力を nn::hid::PadStatus 構造体に取得することができます。Read() はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、ReadLatest() は常に最新のサンプリング結果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

nn::hid::PadStatus 構造体の hold メンバにはサンプリング時に押されているボタンが、trigger メンバにはサンプリング時に押されたボタンが、release メンバにはサンプリング時に離されたボタンがビットにマッピングされてそれぞれ記録されています。ReadLatest() では、trigger、release の両メンバがその時点の状態で評価されますので、呼び出しと呼び出しの間の変化は考慮されません。スライドパッドの入力で十字ボタンの入力をエミュレーションしたビットもあります。スライドパッド自体の入力は stick メンバに 2 軸の座標値で記録されています。

表 6-2. ボタンと定義の対応

定義	対応するボタン
BUTTON_UP	十字ボタン（上）
BUTTON_DOWN	十字ボタン（下）
BUTTON_LEFT	十字ボタン（左）
BUTTON_RIGHT	十字ボタン（右）
BUTTON_A	A ボタン
BUTTON_B	B ボタン
BUTTON_X	X ボタン
BUTTON_Y	Y ボタン
BUTTON_L	L ボタン
BUTTON_R	R ボタン

BUTTON_START	START ボタンまたは SELECT ボタン(通常動作時)
BUTTON_SELECT_FOR_DEBUGGING	SELECT ボタン(デバッグ用途のみ)
BUTTON_EMULATION_UP	スライドパッドによる十字ボタン(上)のエミュレーション入力
BUTTON_EMULATION_DOWN	スライドパッドによる十字ボタン(下)のエミュレーション入力
BUTTON_EMULATION_LEFT	スライドパッドによる十字ボタン(左)のエミュレーション入力
BUTTON_EMULATION_RIGHT	スライドパッドによる十字ボタン(右)のエミュレーション入力

nn::hid::EnableSelectButton() と nn::hid::DisableSelectButton() で SELECT ボタンのサンプリングの有効 / 無効を切り替えることができます。

コード 6-1. SELECT ボタンのサンプリングの有効 / 無効の切り替え

```
bool nn::hid::EnableSelectButton();
void nn::hid::DisableSelectButton();
```

SELECT ボタンのサンプリングを有効に切り替えることができたときに、EnableSelectButton() は true を返します。デバッグモードに設定されていない場合は必ず false を返し、有効になりません。SELECT ボタンのサンプリングが有効になると、BUTTON_SELECT_FOR_DEBUGGING を返すようになり、START ボタンとの押し分けが可能になります。

スライドパッドは触れていない状態でも座標値を返します。そのため、nn::hid::PadReader クラスの SetStickClamp() と SetStickClampMode() を呼び出して、入力に適切な遊びとクランプの方法を設定してください。クランプの方法には円形クランプ(STICK_CLAMP_MODE_CIRCLE)と十字形クランプ(STICK_CLAMP_MODE_CROSS)、最小クランプ(STICK_CLAMP_MODE_MINIMUM)の 3 種類を用意しています。現在のスライドパッドのクランプ方法は GetStickClampMode() で、クランプで使用する値は GetStickClamp() で、それぞれ取得することができます。デフォルトのクランプ方法には円形クランプが設定されています。スライドパッドからの入力座標を(x, y)、原点からの距離を d、クランプの下限値と上限値を min と max、クランプ後の座標を(x', y')、原点からの距離を d' とすると、それぞれ以下のように座標をクランプします。

円形クランプ

d <= min

(x', y') = (0, 0)

min < d < max

(x', y') = ((d − min) / d * x, (d − min) / d * y)

d > max

(x', y') = ((max − min) / d * x, (max − min) / d * y)

十字形クランプ

x < 0

(x', y') = (x + min, y) ただし、x + min は 0 以上にならない

x >= 0

(x', y') = (x − min, y) ただし、x − min は 0 以下にならない

y < 0

(x', y') = (x, y + min)　ただし、y + min は 0 以上にならない

y >= 0

(x', y') = (x, y - min)　ただし、y - min は 0 以下にならない

d' > (max - min)

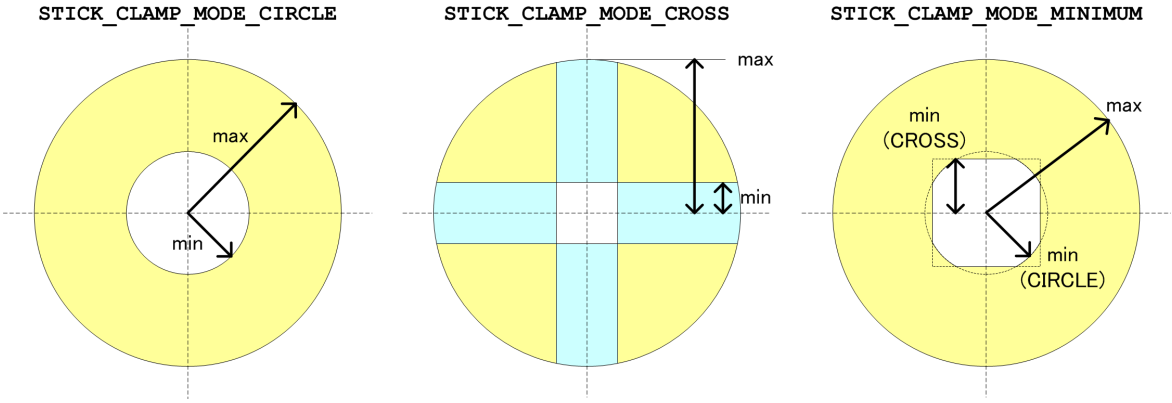
(x', y') = ((max - min) / d' * x', (max - min) / d' * y')

最小クランプ

最小クランプは円形クランプと十字形クランプを組み合わせたクランプ方法です。内側(最小値)のクランプは、ともに下限値に設定された円形クランプと十字形クランプそれぞれでクランプされる領域が小さな方(同じ入力座標でクランプされない方)を適用します。外側(最大値)のクランプは円形クランプと同じ方法で行われます。

スライドパッドの入力に対してクランプ後の座標が変化する範囲を図示すると、図 6-1 のようになります。

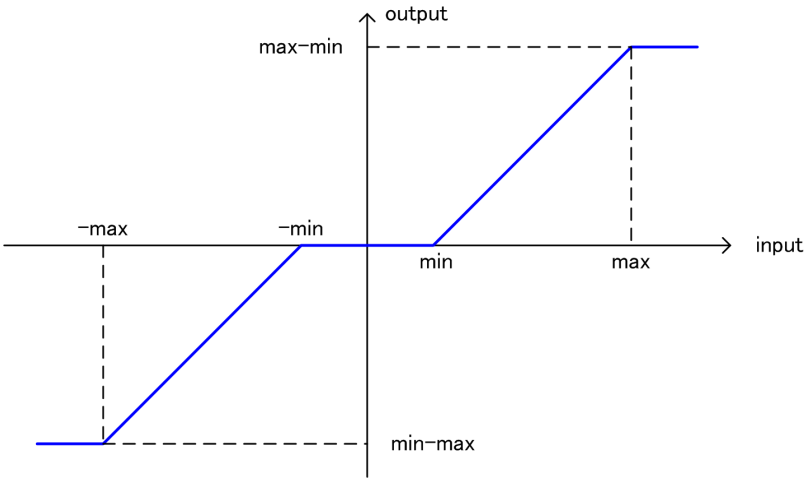
図 6-1. 円形クランプと十字形クランプの有効入力範囲



STICK_CLAMP_MODE_CROSS の x, y 軸近辺は x, y どちらかの座標(x 軸近辺なら x 座標)のみが変化します。

1 つの座標軸に注目して入力座標と出力座標をグラフにすると、どのクランプ方法でも以下のようになります。

図 6-2. 入力座標と出力座標の関係



どのクランプ方法でも、min までは 0、max 以上は ±(max - min) を出力し、min と max の間は 0 から ±(max - min) へと入力に比例して出力されます。

クランプ後に取り得る座標はどの方法でも (max - min) を半径とする円になりますが、どのような入力が必要なのかで採用する方法を決定することができます。STICK_CLAMP_MODE_CIRCLE は原点と入力座標の角度が保持されますので、細かく方向を指定する用途に向いています。STICK_CLAMP_MODE_CROSS は x, y 軸近辺で片方の座標のみが出力されますので、メニューを選択するときのように軸方向の入力を重視する用途に向いています。

スライドパッドの座標値の正規化

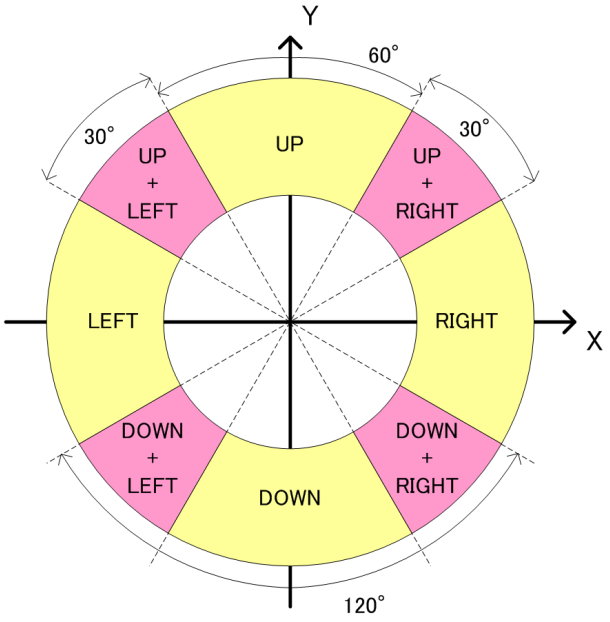
nn::hid::PadReader クラスの NormalizeStick() は、Read() または ReadLatest() で取得したスライドパッドの座標値を (max - min) を 1.0 とする -1.0 ～ +1.0 の範囲で、f32 型の浮動小数点数に正規化します。

NormalizeStickWithScale() は、感度調節機能を付加したスライドパッドの座標値の正規化を行います。感度の調節は、SetNormalizeStickScaleSetting() で設定された *scale*(デフォルトは 1.5)と *threshold*(デフォルトは 141)をもとに行われます。正規化後の値は、*threshold* 未満ならば (1/*scale*) 倍になり、*threshold* 以上ならば徐々に ±1.0 へと近づきます。この機能によって、スライドパッドの可動範囲が *scale* 倍あるように振る舞うことになります。

スライドパッドの十字ボタンエミュレーション

スライドパッドの入力で十字ボタンをエミュレーションしたビット(BUTTON_EMULATION_*)は、下限値を 40、上限値を 145 に設定した円形クランプを適用したあとの座標値が原点から見てどの方向にあるのかで判定されます。
SetStickClamp() および SetStickClampMode() による設定は影響しません。x, y 軸の正負それぞれに上下左右を割り振り、軸を中心に 120 度の範囲に座標があれば対応するビットが ON となります。そのため、範囲が重なる 30 度の部分は斜め方向の入力と判定されます。

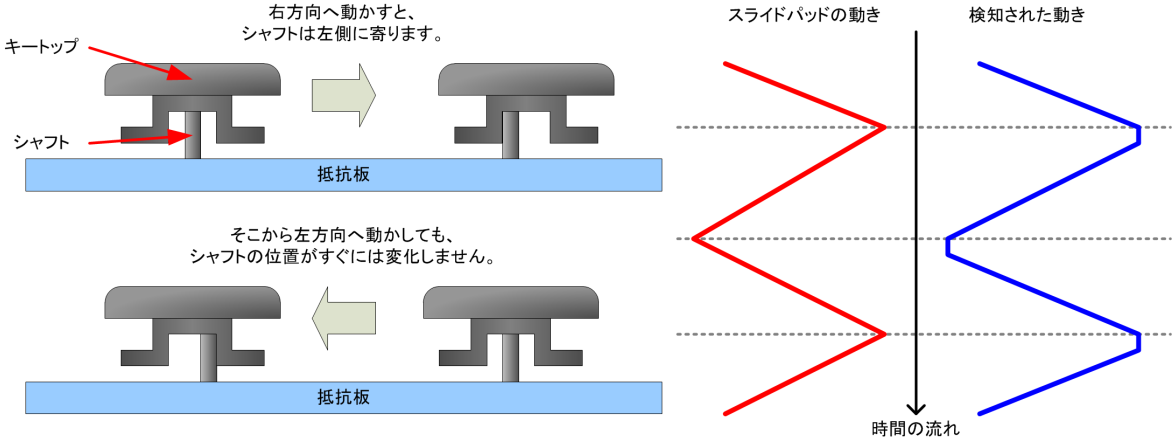
図 6-3. BUTTON_EMULATION_ の判定範囲*



6.2.1.1. スライドパッドのハードウェア特性

3DS に搭載されているスライドパッドは、抵抗板と接触しているシャフトとキートップの接合部分に遊びがある構造のため、ゲームキューブや Wii のコントローラに搭載されているアナログスティックに比べて、入力方向の反転に対する応答に若干の遅れが発生します。

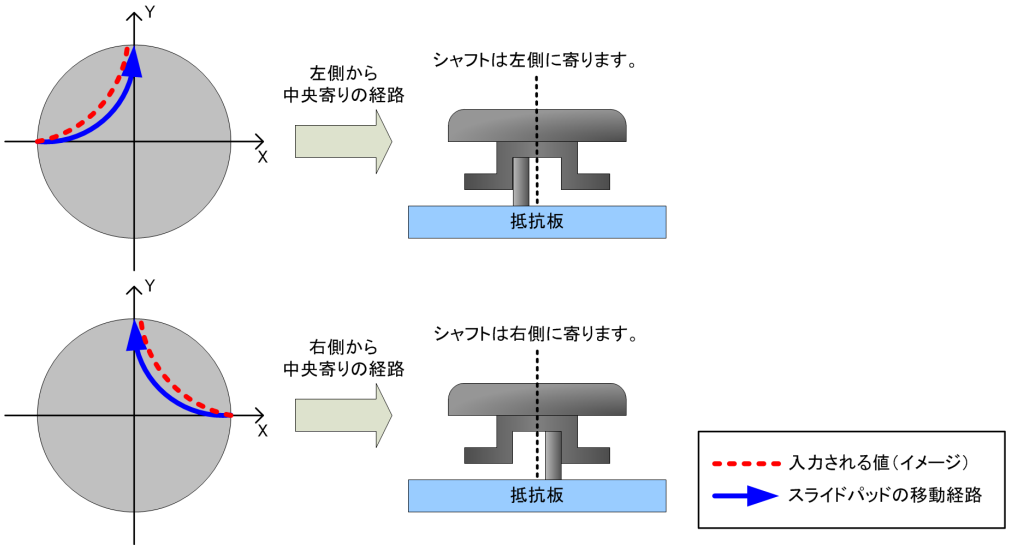
図 6-4. スライドパッドの構造による反転入力への応答の遅れ



また、スライドパッドを同じ位置に移動させたときに、その移動経路によって得られる値が異なる可能性（経路依存性）があります。この経路依存性もスライドパッドの構造が原因で発生しています。

例えば、左端から中央寄りの経路をたどった場合はシャフトが左寄りになるために -X 方向寄りの値が得られる傾向があり、右端から中央寄りの経路をたどった場合はシャフトが右寄りになるために +X 方向寄りの値が得られる傾向があります。この傾向は上下（±Y）方向にも当てはまります。

図 6-5. スライドパッドの経路依存性



アプリケーションは、入力値に対しての判定閾値に余裕を持たせるなど、この経路依存性に考慮した対応を行うことを推奨します。

6.2.2. タッチパネル

`nn::hid::TouchPanelReader` クラスの `Read()` または `ReadLatest()` を呼び出すことで、タッチパネルの入力を `nn::hid::TouchPanelStatus` 構造体に取得することができます。`Read()` はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、`ReadLatest()` は常に最新のサンプリング結果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

nn::hid::TouchPanelStatus 構造体の x、y メンバには、(下画面を手前にしたときの)LCD の左上隅を原点とするタッチパネルの入力座標がドット単位で記録されています。GX ライブラリでの LCD の座標とは座標軸が異なる点には注意が必要です。また、接触の難しい外周部の 5 ドットはクランプされた座標値で返されますので、実際に返される座標値は x が 5 ～ 314、y が 5 ～ 234 の範囲となります。

コード 6-2. タッチパネルの入力座標を LCD の座標に変換するコード例

```
x = nn::gx::DISPLAY1_WIDTH - touchPanel.y;
y = nn::gx::DISPLAY1_HEIGHT - touchPanel.x;
```

touch メンバにはタッチパネルにタッチペンが接触しているかどうかが記録されています。このメンバが 0 ならばタッチペンは接触しておらず、1 ならば接触しています。

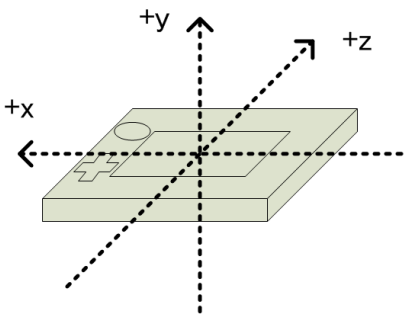
6.2.3. 加速度センサー

nn::hid::AccelerometerReader クラスの Read() または ReadLatest() を呼び出すことで、加速度センサーの入力を nn::hid::AccelerometerStatus 構造体に取得することができます。Read() はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、ReadLatest() は常に最新のサンプリング結果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

加速度センサーの電源は、AccelerometerReader クラスのインスタンスが作成されたときにオンになり、インスタンスが破棄されたときにオフになります。そのため、加速度センサーを使用するシーンの間だけインスタンスを作成する実装はバッテリーの節約には有効です。ただし、毎フレーム呼び出される関数でインスタンスの作成と破棄を繰り返すような実装は、頻繁に電源のオンとオフを切り替えるため、無駄な電力の消費や予期せぬ誤動作の一因となります。

nn::hid::AccelerometerStatus 構造体の x、y、z メンバには加速度センサーの 3 軸の入力が記録されています。十字ボタンの右から左への方向が x 軸の正方向、下 LCD の画面に垂直で下から上への方向が y 軸の正方向、十字ボタンの下から上への方向が z 軸の正方向です。3 軸の入力はそのままでは加速度を示す値ではありませんので、nn::hid::AccelerometerReader クラスの ConvertToAcceleration() で加速度(単位:G)に変換してからアプリケーションで利用してください。

図 6-6. 加速度センサーの 3 軸の方向



加速度センサーのゼロ点に最大で、経年変化により 0.05 G、温度特性により 0.08 G の計 0.13 G のズレが生じる可能性があります。アプリケーションで加速度センサーを利用する場合は、このゼロ点のズレを考慮した仕様にしてください。経年変化によるズレはキャリブレーションによりほぼ解消することができますが、温度特性によるズレはキャリブレーションを行っても短時間で再び生じることがあります。0.08 G のズレは、角度にしておよそ 5 度の傾きに相当します。この精度以上の繊細な動きを検知する場合は、ユーザーが違和感を感じたときに、すぐキャリブレーションを行えるようにするなどの配慮をしてください。

> **補足:** HOME メニュー表示中に Y ボタンと B ボタンを 3 秒間押し続けることで、加速度センサーのキャリブレーションを行うことができます。

また、感度には最大で ± 8 % のズレが生じる可能性があります。ゼロ点のズレによるオフセットも加味して、入力の最大値には測定可能範囲の限界値(約 1.8 G)を 10 % 軽減した値(約 1.62 G)を使用し、この値以上の入力をトリガにするような設計にはしないでください。

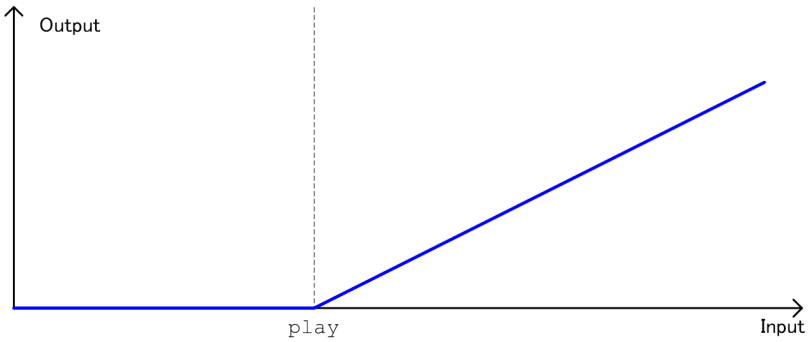
> **注意:** 加速度センサーの出力値には ±0.02 G の静止ノイズが重畳します。スピーカーからサウンドを出力している場合は、さらにスピーカーから伝わる振動が ±0.05 G のノイズとして重畳します。そのため、微小な加速度を検知する際には、これらのノイズによる影響を考慮する必要があります。
>
> 振動ノイズは 1kHz 帯付近で最大となり、再生音量を調節することで影響を小さくすることができます。

`nn::hid::AccelerometerReader` クラスの `SetSensitivity()` を呼び出すことで入力の遊びと検出感度を設定することができ、`GetSensitivity()` を呼び出すことでそれらの設定を取得することができます。どちらの設定も 0 ～ `MAX_OF_ACCELEROMETER_SENESITIVITY` の範囲で指定し、デフォルトでは遊びが 0、検出感度が `MAX_OF_ACCELEROMETER_SENESITIVITY` が指定されています。検出感度に 0 を指定すると 0 しか返さなくなります。`MAX_OF_ACCELEROMETER_SENESITIVITY` を指定すると、デバイスで検出した値そのままを返します。

加速度センサーからの入力値を 3 軸の座標値として見たとき、軸ごとに加速度の変化量が入力の遊び以下であれば遊びの範囲と判定します。遊びの範囲内での出力値は 0 になり、出力値の変化はまったくありません。

遊びを `play` として、加速度センサーからの入力値と出力値の関係をグラフにすると以下のようになります。この遊びの設定は、手ブレなどの小さな変位による誤検知を軽減するために使用します。

図 6-7. 遊びの設定による入力値と出力値の関係



`nn::hid::AccelerometerReader` クラスでは、検出感度と遊びを適用後の出力値に対してオフセットによる修正を行ったあと、軸回転による回転処理を行うことができます。なお、出力値の軸回転は任意の傾きを加速度センサーに与えることに利用できます。

コード 6-3. 加速度センサーの出力値のオフセット

```
class nn::hid::AccelerometerReader
{
    void EnableOffset();
    void DisableOffset();
    bool IsEnableOffset() const;
    void SetOffset(s16 x, s16 y,s16 z);
    void GetOffset(s16* pX, s16* pY, s16* pZ) const;
    void ResetOffset();
```

```
}
```

出力値へのオフセット修正は、EnableOffset() で有効(デフォルト)に、DisableOffset() で無効にすることができます。現設定が有効か無効かは IsEnableOffset() の返り値が true であるかどうかで判断することができます。オフセット修正が有効になっている場合、設定されたオフセット値の各成分が出力値の各成分から減算されます。

オフセット値の設定は SetOffset() で行います。各軸のオフセットを値で指定することができます。オフセット値の各成分は加速度(単位: G)ではなく、変換前のサンプリング値で指定することに注意してください。現設定の取得は GetOffset() で行います。オフセット値を初期値(すべての成分が 0)に戻すには ResetOffset() を呼び出してください。

補足: SetOffsetFromBaseStatus() は CTR-SDK から削除されました。

注意: オフセット値の設定を加速度センサーのキャリブレーションを行う目的で使用しないでください。

コード 6-4. 加速度センサーの軸回転

```
class nn::hid::AccelerometerReader
{
    void EnableAxisRotation();
    void DisableAxisRotation();
    bool IsEnableAxisRotation() const;
    void SetAxisRotationMatrix(const nn::math::MTX34 mtx);
    void GetAxisRotationMatrix(nn::math::MTX34* pMtx) const;
    void ResetAxisRotationMatrix();
}
```

出力値の軸回転は、EnableAxisRotation() で有効に、DisableAxisRotation() で無効(デフォルト)にすることができます。現設定が有効か無効かは IsEnableAxisRotation() の返り値が true であるかどうかで判断することができます。軸回転が有効になっている場合、設定された回転行列による回転処理が出力値に対して行われます。

回転行列(3x4 行列)は、SetAxisRotationMatrix() で設定、GetAxisRotationMatrix() で現設定の取得、ResetAxisRotationMatrix() で単位行列(軸回転なし)に戻すことができます。

出力値へのオフセット修正と軸回転の 2 つを有効にしたときは、オフセット修正を適用後に軸回転が適用されます。

注意: 軸回転の設定を加速度センサーのキャリブレーションを行う目的で使用しないでください。

6.2.3.1. アプリケーション独自のキャリブレーションを実装する場合の注意点

注意: アプリケーション独自のキャリブレーションを実装することは禁止となりました。

6.2.4. ジャイロセンサー

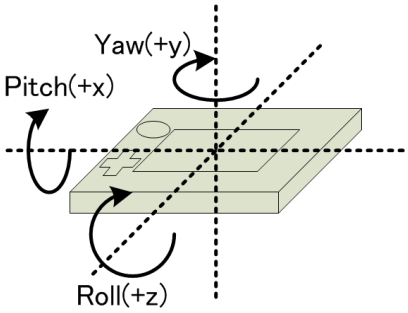
nn::hid::GyroscopeReader クラスの Read() または ReadLatest() を呼び出すことで、ジャイロセンサーの入力を nn::hid::GyroscopeStatus 構造体に取得することができます。Read() はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、ReadLatest() は常に最新のサンプリング結

果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

ジャイロセンサーの電源は、GyroscopeReader クラスのインスタンスが作成されたときにオンになり、インスタンスが破棄されたときにオフになります。そのため、ジャイロセンサーを使用するシーンの間だけインスタンスを作成する実装はバッテリーの節約には有効です。ただし、毎フレーム呼び出される関数でインスタンスの作成と破棄を繰り返すような実装は、頻繁に電源のオンとオフの切り替えるため、無駄な電力の消費や予期せぬ誤動作の一因となります。

nn::hid::GyroscopeStatus 構造体の speed、angle、direction メンバにはジャイロセンサーの角速度、回転角、三次元姿勢が記録されています。角速度と回転角は 3 次元のベクトルで、x 成分にピッチ方向、y 成分にヨー方向、z 成分にロール方向の値が記録されています。角速度は 360 dps を 1.0 とする値、回転角は 360 度を 1.0 とする値です。下 LCD 画面の長辺を軸に手前側を持ち上げて回転させる方向がピッチの正方向、中央を軸に水平に時計回りで回転させる方向がヨーの正方向、短辺を軸に十字ボタン側を持ち上げて回転させる方向がロールの正方向です。

図 6-8. ジャイロセンサーの 3 要素



角速度の感度には最大で ± 8 % のズレが生じる可能性があります。ゼロ点のズレによるオフセットも加味して、入力の最大値には測定可能範囲の限界値(約 1800 dps)を 10 % 軽減した値(約 1620 dps)を使用し、この値以上の入力をトリガにするような設計にはしないでください。

三次元姿勢は 3x3 行列で記録されています。三次元姿勢は角速度を元に計算されており、計算時に参照する角速度の倍率を nn::hid::GyroscopeReader::SetDirectionMagnification() で変更することができます。倍率に 1.0 を指定した場合は、検出された角速度をそのままを計算に使用します。倍率に 2.0 を指定した場合は、検出された角速度を 2 倍して計算に使用します。

nn::hid::GyroscopeReader クラスは静止状態を検知し、ゼロ点オフセットの自動キャリブレーション(ゼロ点ドリフトの補正)を行っています。補正処理の設定はすべての方向に一様に適用されます。

コード 6-5. ジャイロセンサーのゼロ点ドリフトの補正

```
class nn::hid::GyroscopeReader
{
    void EnableZeroDrift();
    void DisableZeroDrift();
    bool IsEnableZeroDrift() const;
    f32 GetZeroDriftEffect() const;
    void ResetZeroDriftMode();
    void SetZeroDriftMode(const nn::hid::ZeroDriftMode& mode);
    void GetZeroDriftMode(nn::hid::ZeroDriftMode& mode) const;
}
```

ゼロ点ドリフトの補正は、EnableZeroDrift() で有効(デフォルト)に、DisableZeroDrift() で無効にすることができます。補正の有効・無効は IsEnableZeroDrift() の返り値が true であるかどうかで確認することができます。補正

処理の働き具合は GetZeroDriftEffect() の返り値で確認することができます。補正が無効の場合は負の値が返りますが、有効の場合は 0 以上の値が返ります。0 は補正が行われなかった(動きを検知した)ことを示し、1.0 に近づくほど安定している(動きが少ない)ことを示します。

ゼロ点ドリフトの補正モードによって補正が行われる変動値の幅が異なり、本体が安定していると判断される角速度の幅が変化します。ゼロ点ドリフトの補正モードは、SetZeroDriftMode() で設定、GetZeroDriftMode() で現設定の取得、ResetZeroDriftMode() で初期値(GYROSCOPE_ZERODRIFT_STANDARD)に戻すことができます。

表 6-3. ジャイロセンサーのゼロ点ドリフトの補正モード

定義	説明
GYROSCOPE_ZERODRIFT_LOOSE	補正する変動幅が広くなり、等速運動を検知しない場合があります。
GYROSCOPE_ZERODRIFT_STANDARD	標準的な補正を行います。(デフォルト)
GYROSCOPE_ZERODRIFT_TIGHT	補正する変動幅が狭くなり、緩やかな動きを検知することができます。

ゼロ点ドリフトの補正を無効にした場合、ゼロ点のオフセットに最大で 50 dps のズレが生じることがあります。極めてゆっくりとした動きを検知するために無効に設定した場合は、このズレの発生に十分留意してください。

> **補足:** GYROSCOPE_ZERODRIFT_LOOSE 以外では静止状態でも補正処理の効きが悪くなる場合がありますので、本体を動かさないと思われるシーンでは GYROSCOPE_ZERODRIFT_LOOSE による補正を掛け、使用時に元の補正モードに戻すなどの実装を推奨します。

ゼロ点の遊び補正を設定することで、角速度の小さな変動に反応させないようにすることができます。ゼロ点の遊び補正の設定はすべての方向に一様に適用されます。

コード 6-6. ジャイロセンサーのゼロ点の遊び補正

```
class nn::hid::GyroscopeReader
{
    void EnableZeroPlay();
    void DisableZeroPlay();
    bool IsEnableZeroPlay() const;
    f32 GetZeroPlayEffect() const;
    void SetZeroPlayParam(f32 radius);
    void GetZeroPlayParam(f32& radius) const;
    void ResetZeroPlayParam();
}
```

ゼロ点の遊び補正は、EnableZeroPlay() で有効に、DisableZeroPlay() で無効(デフォルト)にすることができます。遊び補正の有効・無効は IsEnableZeroPlay() の返り値が true であるかどうかで確認することができます。遊び補正のかかり具合は GetZeroPlayEffect() の返り値で確認することができます。遊び補正が無効の場合は負の値が返りますが、有効の場合は 0 以上の値が返ります。角速度が設定された遊びの値に近くなるほど返り値は 0 に近づき、0 の場合は遊びによる補正が行われなかったことを示します。

遊び補正で 0 となる角速度の絶対値は、SetZeroPlayParam() で設定、GetZeroPlayParam() で現設定の取得、ResetZeroPlayParam() で初期値(0.005)に戻すことができます。絶対値は 360 dps を 1.0 とする値で設定します。

ゼロ点に関する 2 つの機能が有効である場合は、先にゼロ点ドリフトの補正が行われ、次に遊びによる補正が行われます。

nn::hid::GyroscopeReader クラスでは、ゼロ点ドリフトの補正と遊び補正を適用後の角速度に対して、軸回転による回転処理を行うことができます。静止時の出力値をもとに回転行列を計算することで、アプリケーションにジャイロセンサーのキャリブレーションを実装することができます。ほかにも、出力値の軸回転は任意の傾きをジャイロセンサーに与えることに利用できます。

コード 6-7. ジャイロセンサーの軸回転

```
class nn::hid::GyroscopeReader
{
    void EnableAxisRotation();
    void DisableAxisRotation();
    bool IsEnableAxisRotation() const;
    void SetAxisRotationMatrix(const nn::math::MTX34 mtx);
    void GetAxisRotationMatrix(nn::math::MTX34* pMtx) const;
    void ResetAxisRotationMatrix();
}
```

角速度の軸回転は、EnableAxisRotation() で有効に、DisableAxisRotation() で無効(デフォルト)にすることができます。現設定が有効か無効かは IsEnableAxisRotation() の返り値が true であるかどうかで判断することができます。軸回転が有効になっている場合、設定された回転行列による回転処理が角速度に対して行われます。

回転行列(3x4 行列)は、SetAxisRotationMatrix() で設定、GetAxisRotationMatrix() で現設定の取得、ResetAxisRotationMatrix() で単位行列(軸回転なし)に戻すことができます。

nn::hid::GyroscopeReader クラスは加速度センサーの入力値を利用して、ジャイロセンサーの三次元姿勢の補正を行っています。そのためジャイロセンサーを使用するときは、加速度センサーも使用することになります。コンストラクタで引数 *pAccelerometerReader* に NULL を渡した(オーバーロードの呼び出した)場合は、デフォルト設定の nn::hid::AccelerometerReader クラスのインスタンスを内部で生成して利用します。入力の遊びや検出感度を変更したい場合は、設定を変更した nn::hid::AccelerometerReader クラスのインスタンスへのポインタを渡してください。

コード 6-8. 加速度による三次元姿勢の補正

```
class nn::hid::GyroscopeReader
{
    GyroscopeReader(nn::hid::AccelerometerReader* pAccelerometerReader = NULL,
                    nn::hid::Gyroscope& gyroscope = nn::hid::GetGyroscope());

    void EnableAccRevise();
    void DisableAccRevise();
    bool IsEnableAccRevise() const;
    f32 GetAccReviseEffect() const;
    void SetAccReviseParam(f32 revise_pw, f32 revise_range);
    void GetAccReviseParam(f32& revise_pw, f32& revise_range) const;
    void ResetAccReviseParam();
}
```

加速度による補正は、EnableAccRevise() で有効(デフォルト)に、DisableAccRevise() で無効にすることができます。補正が有効か無効かは IsEnableAccRevise() の返り値が true であるかどうかで確認することができます。補正のかかり具合は GetAccReviseEffect() の返り値で確認することができます。補正が無効の場合は常に 0 が返りますが、有効の場合は 0 以上の値が返ります。返り値が 1.0 に近づくほど三次元姿勢(GyroscopeStatus.direction)の方向を加速度の方向に近づくように補正したことを示します。

加速度による補正のパラメータ(重みと有効範囲)は、SetAccReviseParam() で設定、GetAccReviseParam() で現設定の取得、ResetAccReviseParam() で初期値(重み 0.03、有効範囲 0.4)に戻すことができます。*revise_pw* には加速度の重みを 0.0 ～ 1.0 の範囲の値で指定します。値が大きいほど急激な補正がかかります。*revise_range* には補正に使用する加速度の範囲(1.0 を中心に±*revise_range*)を指定します。例えば、0.4 を指定した場合は 0.6 G ～ 1.4 G の範囲ならば補正計算に加速度を使用します。補正のパラメータはすべての方向に一様に適用されます。

加速度による補正が有効な状態でジャイロセンサーに角速度の軸回転を適用する場合は、ジャイロセンサーと加速度センサーの軸回転で適用する回転行列には同じ行列を指定してください。コンストラクタでデフォルト設定のインスタンスを利用している場合はライブラリ内で同じ行列となるように調整していますが、独自設定のインスタンスを利用している場合は注意が必要です。異なる行列が指定された場合は、加速度の補正が不適切に働く可能性があります。

6.2.5. デバッグパッド(デバッグ専用コントロールパッド)

nn::hid::DebugPadReader クラスの Read() または ReadLatest() を呼び出すことで、デバッグパッドのデジタルボタンと 2 本のアナログスティックの入力を nn::hid::DebugPadStatus 構造体に取得することができます。Read() はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期(16 ms)よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、ReadLatest() は常に最新のサンプリング結果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

nn::hid::DebugPadStatus 構造体の hold メンバにはサンプリング時に押されているボタンが、trigger メンバにはサンプリング時に押されたボタンが、release メンバにはサンプリング時に離されたボタンがビットにマッピングされてそれぞれ記録されています。ReadLatest() では、trigger、release の両メンバがその時点の状態で評価されますので、呼び出しと呼び出しの間の変化は考慮されません。leftStickX, leftStickY メンバには左アナログスティックの x 軸と y 軸、rightStickX, rightStickY メンバには右アナログスティックの x 軸と y 軸の値が −1.0 ～ +1.0 の範囲で記録されています。

アナログスティックのクランプ方法は SetStickClampMode() で設定、GetStickClampMode() で現在の設定の取得を行うことができます。左右のアナログスティックに対して、個別に設定することはできません。クランプの方法には、遊びのある円形クランプ(STICK_CLAMP_MODE_CIRCLE_WITH_PLAY)と遊びのない円形クランプ(STICK_CLAMP_MODE_CIRCLE_WITHOUT_PLAY)の 2 種類があります。

表 6-4. デバッグパッドのデジタルボタンと定義の対応

定義	対応するボタン
DEBUG_PAD_BUTTON_UP	十字ボタン(上)
DEBUG_PAD_BUTTON_DOWN	十字ボタン(下)
DEBUG_PAD_BUTTON_LEFT	十字ボタン(左)
DEBUG_PAD_BUTTON_RIGHT	十字ボタン(右)
DEBUG_PAD_BUTTON_A	A ボタン
DEBUG_PAD_BUTTON_B	B ボタン
DEBUG_PAD_BUTTON_X	X ボタン
DEBUG_PAD_BUTTON_Y	Y ボタン
DEBUG_PAD_TRIGGER_L	L トリガー
DEBUG_PAD_TRIGGER_R	R トリガー

DEBUG_PAD_TRIGGER_ZL	ZL トリガー
DEBUG_PAD_TRIGGER_ZR	ZR トリガー
DEBUG_PAD_BUTTON_PLUS	+(プラス)ボタン
DEBUG_PAD_BUTTON_MINUS	-(マイナス)ボタン
DEBUG_PAD_BUTTON_HOME	HOME ボタン

6.2.6. 拡張スライドパッド

拡張スライドパッドは CTR に装着して使用する周辺機器です。主な特徴は、CTR 本体に搭載されている入力装置に加え、拡張スライドパッドに搭載されたスライドパッド(以下スライドパッド(R)と表記し、本体に搭載されたものはスライドパッドと表記する)とデジタルボタン(ZL/ZR ボタン)が使用できる点です。

拡張スライドパッドに搭載されている入力装置からの入力は、赤外線通信により CTR 本体へ送信されます。サンプリング周期として指定可能な値は、1 ms 間隔で 8 ms から 32 ms までの範囲です。具体的な値の設定方法については、「6.2.6.4. サンプリングの開始と状態取得」を参照してください。

補足: アプリケーションで拡張スライドパッドを利用する際の処理フローについては、「付録:拡張スライドパッドを利用する場合のフロー図」を参照してください。

SNAKE に搭載されている C スティック、ZL ボタン、ZR ボタンの詳細、拡張スライドパッドとの相違点については、「6.2.7. C スティック」を参照してください。

注意: CTR では、赤外線通信を用いて拡張スライドパッドと通信します。
アプリケーションから拡張スライドパッドを使用する際、赤外線通信を利用する他の機能を使用中ならば、その機能を先に終了させてください。

赤外線通信は以下の機能で利用されています。

- 本体間赤外線通信
- NFP (CTR のみ)

6.2.6.1. ハードウェアの内部ステート

拡張スライドパッドのハードウェアの内部ステートとしては、CTR との通信が可能な「アクティブ状態」と、一切の通信を行わない「スタンバイ状態」の 2 種類のステートしかありません。

表 6-5. ハードウェアの内部ステート

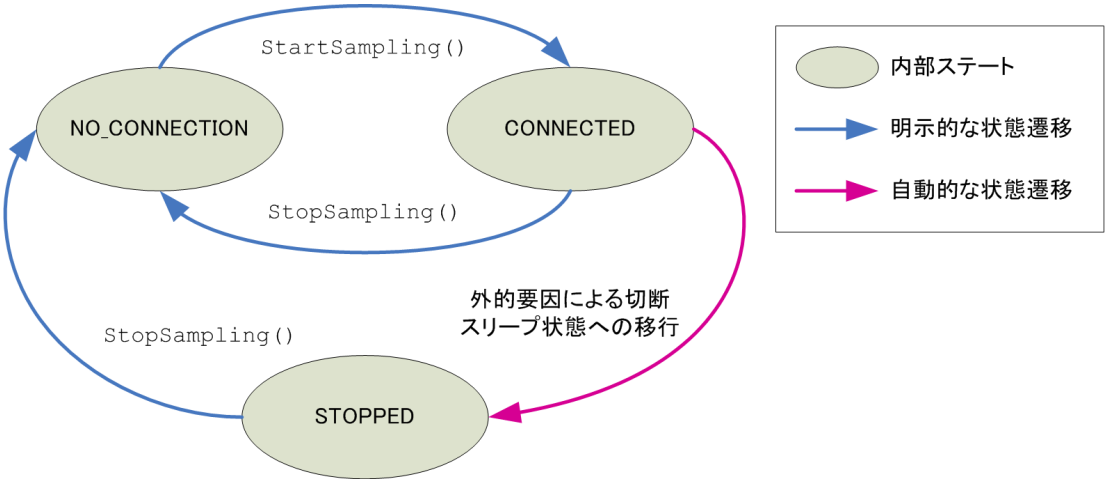
内部ステート	説明
アクティブ状態	CTR との通信が可能な状態です。この状態でなければ、CTR との接続ができません。 電池の挿入直後はこの状態になります。
スタンバイ状態	CTR との通信を一切行わない状態です。未使用時の電池消費を最小限に抑え、電池を長持ちさせます。 ボタン入力や赤外線通信が行われていない状態が 5 分間続くと、この状態になります。拡張スライドパッドのデジタルボタン(ZL、ZR、R)のいずれかを押すことで、アクティブ状態に復帰します。

> **補足:** 拡張スライドパッドには内部ステートを示すインジケータが存在しません。また、ライブラリから内部ステートを取得することもできません。

6.2.6.2. ソフトウェアの内部ステート

ソフトウェア(ライブラリ)の内部ステートは以下のように遷移します。

図 6-9. ソフトウェアの内部ステート遷移図(拡張スライドパッド)



以下のように、内部ステートは拡張スライドパッドとの接続状態を示しています。

表 6-6. ソフトウェアの内部ステート

内部ステート	説明
NO_CONNECTION	拡張スライドパッドとの接続が確立していない状態です。 初期化直後も同じ内部ステートです。
CONNECTED	拡張スライドパッドとの接続が確立され、サンプリングが行われている状態です。
STOPPED	確立していた拡張スライドパッドとの接続が、外的要因(電池の消耗、本体からの離脱など)によって切断されている状態です。 CTR 本体がスリープ状態に移行したときも、この内部ステートに遷移します。

内部ステートを示す定義は、`nn::hid::ExtraPad::ConnectionState` 列挙子に「`CONNECTION_STATE_*`」で定義されています。

6.2.6.3. 初期化

拡張スライドパッドからの入力のサンプリングを開始する前に、`nn::hid::ExtraPad` クラスの初期化関数を呼び出す必要があります。

コード 6-9. 拡張スライドパッドの初期化に使用する関数

```
static void nn::hid::ExtraPad::Initialize(
                void* workingMemory,
                size_t workingMemorySize);
```

`workingMemory` には、赤外線通信のためのバッファを指定します。指定するバッファは、サイズが `nn::hid::CTR::ExtraPad::WORKING_MEMORY_SIZE` (12288 Byte)、先頭アドレスのアライメントが

nn::hid::CTR::ExtraPad::WORKING_MEMORY_ALIGNMENT (4096 Byte) でなければなりません。また、**デバイスメモリから確保したバッファは使用できません**。

初期化が完了した時点で nn::hid::ExtraPadReader クラスを使用することができるようになります。

6.2.6.4. サンプリングの開始と状態取得

サンプリングの開始と状態取得には、以下の関数を使用します。

コード 6-10. サンプリングの開始と状態取得に使用する関数

```
class nn::hid::ExtraPad
{
    static nn::Result StartSampling(s32 samplingThreadPriority, s32 period);
    static ConnectionState GetConnectionState();
    static bool IsSampling();
}
```

拡張スライドパッドのサンプリングは nn::hid::ExtraPad::StartSampling() の呼び出しで行います。初期化が完了する前に呼び出すとエラー(nn::hid::MakeResultNotInitialized() と同値)が返されます。この関数の処理には、拡張スライドパッドとの接続に成功した場合は 50 ～ 200 ms、失敗した場合は最大 100 ms かかります。

samplingThreadPriority には、サンプリング用のスレッドのスレッド優先度を指定します。指定可能な値の範囲は 0～31 です。優先度が高い(0 に近い)ほど、安定したサンプリングが期待できます。

period には、サンプリング周期を 1 ミリ秒単位で指定します。指定可能な値の範囲は 8～32 です。

サンプリングを開始する際に、アプリケーションのリソースを消費してサンプリング用のスレッド(スレッド優先度は *samplingThreadPriority* で指定された値)が作成されます。このスレッドは *period* で指定したサンプリング周期で拡張スライドパッドからのサンプリングデータの受信処理を行い、約 1 秒ごとに拡張スライドパッドとの通信を持続するための送信処理を行います。サンプリング周期が高いほど単位時間あたりの処理量が大きくなり、周期の長さにほぼ反比例してアプリコア、システムコアの負荷が大きくなります。そのため、負荷の軽減を目的とする場合、サンプリング周期を下げることが最も有効となります。

> **注意:** ほかの処理における負荷が非常に高く、それがサンプリングスレッドの動作周期に影響を与えるほどである場合、入力の遅延や拡張スライドパッドの切断が発生する可能性があります。

拡張スライドパッドがアクティブ状態でなければサンプリングを開始することができません。ライブラリからはアクティブ状態であるかどうかを判断することができませんので、StartSampling() が拡張スライドパッドが見つからないという返り値(nn::hid::MakeResultNoConnection() と同値)を返したときは、ユーザーに拡張スライドパッドのデジタルボタン(R/ZR/ZL)のいずれかの入力を促すなどして、アクティブ状態に復帰させてから再接続を行ってください。

なお、接続状態で StartSampling() を呼び出すと、一度切断処理を行ってから接続処理が行われます。

> **補足:** 拡張スライドパッドの常時検出のために、1 秒間に 1 回の頻度で StartSampling() を呼び出した場合の負荷は 32 ms 周期でサンプリングする処理よりも小さいことがわかっています。

nn::hid::ExtraPad::GetConnectionState() は、拡張スライドパッドとの接続状態を内部ステートで返します。外的要因などで切断され、CONNECTION_STATE_STOPPED が返されたときは、切断処理を行ってから再接続を行ってください。

nn::hid::ExtraPad::IsSampling() は、サンプリング中かどうかを返します。true を返したときはサンプリングが

行われていますので、nn::hid::ExtraPadReader クラスで取得した nn::hid::ExtraPadStatus 構造体には拡張スライドパッドの入力が反映されています。true を返したときは内部ステートも接続状態(CONNECTED)であることが保証されていますが、接続処理や切断処理の途中などで、内部ステートが接続状態であっても false を返す可能性があることに注意してください。

> **補足:** 拡張スライドパッドのサンプリングが開始されると nn::hid::PadReader クラスは使用できなくなりますが、拡張スライドパッドが接続されていなくても、nn::hid::ExtraPadReader クラスで取得した nn::hid::ExtraPadStatus 構造体には本体のデジタルボタンやスライドパッドからの入力が反映されています。
>
> そのため、拡張スライドパッドに対応するアプリケーションは nn::hid::ExtraPadReader クラスと nn::hid::ExtraPadStatus 構造体を使用し、接続状態によって使用するクラスや構造体を切り替える必要はありません。

6.2.6.5. サンプリングの終了

サンプリングを終了するには、nn::hid::ExtraPad::StopSampling() を呼び出します。

コード 6-11. サンプリングの終了に使用する関数

```
static nn::Result nn::hid::ExtraPad::StopSampling();
```

切断処理に成功すると、サンプリング用のスレッドが破棄され、内部ステートが未接続状態(NO_CONNECTION)になります。この関数は、初期化されていない状態で呼び出されない限り、必ず処理に成功します。

6.2.6.6. サンプリング結果の取得

サンプリング結果は、nn::hid::ExtraPadReader クラスの Read() または ReadLatest() で取得する nn::hid::ExtraPadStatus 構造体に反映されます。Read() はサンプリング結果を新しい順に取得することができますが、取得したサンプリング結果は再度取得することはできません。そのため、サンプリング周期よりも早い周期で呼び出したときはサンプリング結果を取得することができません。一方、ReadLatest() は常に最新のサンプリング結果だけを取得することができます。サンプリング結果は再度取得することができますので、サンプリング周期よりも早い周期で呼び出したときでもサンプリング結果を取得することができます。

コード 6-12. nn::hid::ExtraPadStatus 構造体

```
struct nn::hid::ExtraPadStatus {
    AnalogStickStatus stick;
    AnalogStickStatus extraStick;
    bit32 hold;
    bit32 trigger;
    bit32 release;
    u8    batteryLevel;
    bool  isConnected;
    NN_PADDING2;
};
```

stick には、nn::hid::PadStatus 構造体と同じく、本体側のスライドパッドの入力が反映されます。

extraStick には、拡張スライドパッドのスライドパッド(R)の入力が反映されます。

hold、trigger、release メンバは nn::hid::PadStatus 構造体と同じですが、以下のボタンが追加されます。

表 6-7. 拡張スライドパッドで追加されるボタン

定義	対応するボタン
BUTTON_ZL	拡張スライドパッドの ZL ボタン
BUTTON_ZR	拡張スライドパッドの ZR ボタン
BUTTON_EMULATION_R_UP	スライドパッド(R)による十字ボタン 上のエミュレーション入力
BUTTON_EMULATION_R_DOWN	スライドパッド(R)による十字ボタン 下のエミュレーション入力
BUTTON_EMULATION_R_LEFT	スライドパッド(R)による十字ボタン 左のエミュレーション入力
BUTTON_EMULATION_R_RIGHT	スライドパッド(R)による十字ボタン 右のエミュレーション入力

補足: 拡張スライドパッドの装着時に R ボタンの押下が困難になるため、拡張スライドパッドにも R ボタンが実装されています。ただし、押下された R ボタンが本体のものなのか拡張スライドパッドのものなのかはアプリケーションから判別することができません。

拡張スライドパッドのサンプリングが行われていない状態では、nn::hid::ExtraPadReader クラスは 4 ms のサンプリング周期で nn::hid::ExtraPadStatus 構造体に本体側の入力を反映しています。しかし、拡張スライドパッドのサンプリングが行われている状態では、本体側の入力のサンプリング周期も、nn::hid::ExtraPad::StartSampling() で指定したサンプリング周期(8～32 ms)になります。

batteryLevel には、拡張スライドパッドの電池残量が 0 または 1 の 2値で格納されます。0 が返されるようになってからでも、1 日程度の連続動作が可能です。

注意: 直前まで電池残量に 1 を返していても、一定以上消耗している電池を抜き差しすると、拡張スライドパッドが再起動しない可能性があります。電池の抜き差し後に通信不可能になった場合は、新しい電池に交換してください。なお、この現象は電池の抜き差し時にのみ発生します。電池を入れたまま使用している場合は、電池残量が 0 になるまで動作します。

isConnected には、サンプリング中かどうかをライブラリ内で調べた結果が格納されます。

補足: 拡張スライドパッドの入力がアプリケーションによって取得できる状態になるまでの所要時間(入力遅延)は、アプリケーションのほかの処理が十分小さい場合で、約 2 ms ～(2 ms + サンプリング周期)になります。ユーザーの入力を拡張スライドパッドが取得するまでに 0 ～サンプリング周期の時間がかかり、そのあとの通信と CTR 側での処理に約 2 ms がかかります。

6.2.6.7. スライドパッドのクランプ処理

本体側のスライドパッドのクランプ処理で使用する関数は、nn::hid::PadReader クラスのメンバ関数と同じ関数名で定義されています。また、その動作も同じです。

スライドパッド(R)のクランプ処理で使用する関数は、以下のように関数名の「Stick」が「ExtraStick」になっていること以外、本体側のスライドパッドと同様の設定を個別に行うことができます。

コード 6-13. スライドパッド(R)のクランプ処理に使用する関数

```
class nn::hid::ExtraPadReader
{
    void SetExtraStickClamp(s16 min, s16 max);
    void GetExtraStickClamp(s16* pMin, s16* pMax) const;
    StickClampMode GetExtraStickClampMode() const;
    void SetExtraStickClampMode(StickClampMode mode);
    f32 NormalizeExtraStick(s16 x);
    void NormalizeExtraStickWithScale(
            f32* normalized_x, f32* normalized_y, s16 x, s16 y);
    void SetNormalizeExtraStickScaleSettings(f32 scale, s16 threshold);
    void GetNormalizeExtraStickScaleSettings(f32* scale, s16* threshold) const;
}
```

6.2.6.8. イベント通知

接続状態の変化とサンプリング完了を通知するイベント(nn::os::LightEvent クラス)を登録することができます。

コード 6-14. イベント通知に使用する関数

```
class nn::hid::ExtraPad
{
    static void RegisterConnectionEvent(nn::os::LightEvent* pLightEvent);
    static void UnregisterConnectionEvent();
    static void RegisterSamplingEvent(nn::os::LightEvent* pLightEvent);
    static void UnregisterSamplingEvent();
}
```

「～ConnectionEvent()」が接続状態の変化を、「～SamplingEvent()」がサンプリング完了を通知するイベントの登録と登録解除を行う関数です。イベントを登録すると、サンプリング用のスレッドの処理がわずかに増加します。

接続状態の変化を通知するイベントを登録したときの処理負荷は無視できるほどわずかです。しかし、サンプリング完了を通知するイベントを登録すると、サンプリング周期ごとに処理が行われるため、CPU の処理に負荷がかかります。そのため、イベント通知を使用しないアプリケーションは、これらの関数を使用しないことを推奨します。

なお、接続状態の変化は nn::hid::ExtraPad クラスの GetConnectionState() や IsSampling()、nn::hid::ExtraPadStatus 構造体の isConnected メンバの変化でも判断できます。

6.2.6.9. 補正アプレット

拡張スライドパッドのスライドパッド(R)の操作感覚を補正するため、「拡張スライドパッド補正アプレット」というライブラリアプレットが用意されています。拡張スライドパッドに対応するアプリケーションは、拡張スライドパッド補正アプレットを呼び出すシーンを用意するなど、ガイドラインの指示に従ってください。

拡張スライドパッド補正アプレットについては、「12.1.7. 拡張スライドパッド補正アプレット」を参照してください。

6.2.6.10. スライドパッドとスライドパッド(R)の相違点

スライドパッドおよびスライドパッド(R)からサンプリングされた入力座標は、キートップを意図的に動かしていない場合でも電気的揺らぎなどにより変化することがあります。スライドパッドとスライドパッド(R)とでは、キートップの位置を完全に固定した場合の値の変化量およびその確率が以下のように異なります。特に、スライドパッド(R)は値が変化する確率が高くなっていることに注意してください。

表 6-8. スライドパッドおよびスライドパッド(R)の入力座標の変化の確率

変化量	スライドパッド X 座標	スライドパッド Y 座標	スライドパッド(R) X 座標	スライドパッド(R) Y 座標
−3	0.000006 %	0.000006 %	0.002 %	0.001 %
−2	0.000178 %	0.000178 %	0.349 %	0.352 %
−1	0.003541 %	0.003541 %	3.485 %	3.273 %
±0	99.949837 %	99.949837 %	92.326 %	92.743 %
+1	0.003541 %	0.003541 %	3.491 %	3.285 %
+2	0.000178 %	0.000178 %	0.346 %	0.345 %
+3	0.000006 %	0.000006 %	0.002 %	0.001 %

単位時間内に値が変化する確率は

単位時間内に取得したサンプリングデータの数 × 上記表の確率

となります。

キートップから指を離し、キートップが中央に位置している場合、値は 0 から変化することはありません。これは、値の変化量を加味してもクランプの下限値を超えることがないためです。

6.2.7. C スティック

SNAKE に搭載されている C スティックは、ライブラリ上では拡張スライドパッドに搭載されているスライドパッド(R)と区別することなく利用することができますが、ハードウェアが異なります。そのため、ハードウェア特性の違いなどを考慮する必要があります。

6.2.7.1. C スティックのハードウェア特性

SNAKE に搭載されている C スティックは、樹脂製の軸状部品に加えられた荷重によって生じるわずかな変形(ひずみ)を検知し、変形の大きさを電圧変化の大小で読み取るアナログ入力装置です。

軸部分にかかる荷重によって C スティックからの出力値が定まるため、以下の点に留意する必要があります。

- ユーザーの力の強さによっては、最大値を入力し続けることが困難である可能性がある。
- 一定以上の荷重がかかると C スティックからの出力値が変動しない(出力値が飽和する)ポイントがある。
- C スティックを真上から強く押し込んだ場合、C スティックからの出力値が不定となる。
- 物理的に X ボタンとの距離が近いため、X ボタンを強い力で長押しすると C スティックからの出力値が変動する可能性がある。

また、個々の本体に搭載されるデバイスには性能にばらつきがあるため、以下の点にも留意する必要があります。

- 感度の個体差が分解能に影響する。

最大値を入力するために必要な荷重による影響

C スティックで検知可能な荷重範囲は、最大値が約 240 gf となっています。そのため、ユーザーによっては最大荷重をかけ続けることが困難になる可能性があります。

なお、最小荷重の値はクランプ方式および入力方向により異なり、C スティックで検知可能な荷重範囲をまとめると以下のようになります。

表 6-9. C スティックで検知可能な荷重範囲

クランプ方式	検知可能な最少荷重		検知可能な最大荷重
	上下左右の 4 方向	斜め 45 度方向	
円形(min=40, max=145)	約 68 gf	約 72 gf	約 240 gf
十字(min=36, max=145)	約 63 gf	約 90 gf	
最小(min=40, max=145)	約 63 gf	約 72 gf	

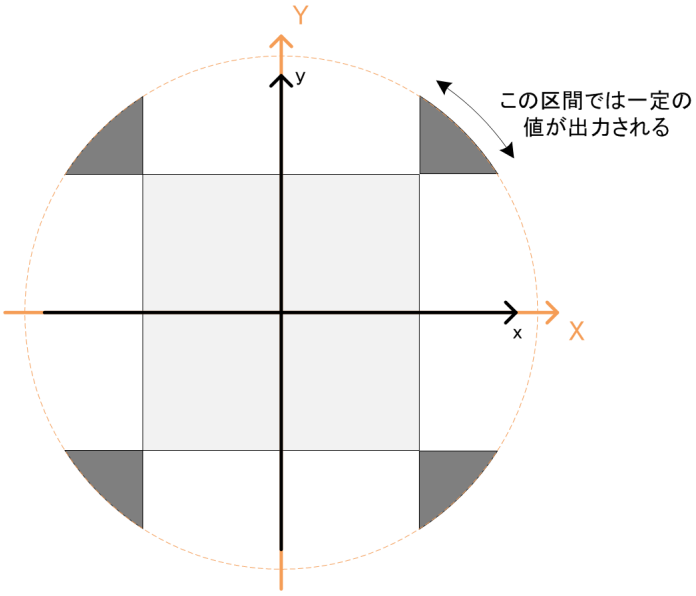
出力値が飽和することによる影響

C スティックに対して過大な荷重がかかった場合、一定の荷重を超えると、それ以上の大きな荷重をかけても C スティックからの出力値が変動しなくなります。これを「出力値の飽和」と呼びます。

C スティックへの入力は、独立した 2 軸(x 軸と y 軸)で取得しています。両軸ともに出力値が飽和する状況下では、ユーザーによる C スティックの操作の変化を検知できなくなります。

たとえば、大きく円を描くように力を込めて C スティックを動かすと、C スティックの入力軸と SNAKE 本体の上下左右方向が一致している場合では、下図のように C スティックの入力軸から見て±45 度にある 4 方向に C スティックからの出力値が変動しない区間が発生します。

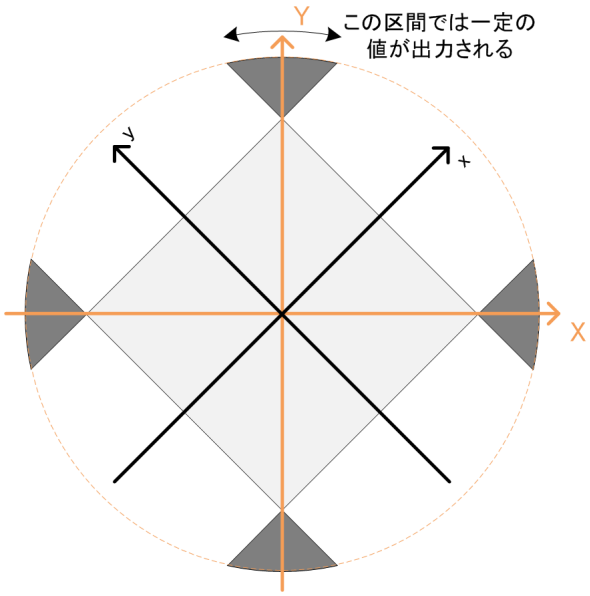
図 6-10. 出力値の飽和(C スティックの入力軸が SNAKE 本体の上下左右方向と一致している場合)



オレンジ色の XY 軸は SNAKE 本体の上下左右方向、黒色の xy 軸は C スティックの入力軸を示し、オレンジ色の円は入力荷重、薄い灰色部分(中央)が C スティックのデバイスから得られる未加工の出力値が取り得る範囲、濃い灰色部分が両軸ともに出力値が変動しない範囲です。なお、図解のために実際よりも変動しない範囲を大きく描いています。

C スティックの入力軸は SNAKE 本体の上下左右方向から 45 度回転させた状態で配置されており、上記のような症状による、ユーザーから見た違和感を低減させています。

図 6-11. 出力値の飽和(C スティックの入力軸が SNAKE 本体の上下左右方向から 45 度回転している場合)



上記の入力軸の回転はシステムによって吸収されるため、アプリケーションはライブラリから取得した C スティックの入力に回転を加える必要はありません。

真上から強く押し込んだ場合の挙動

軸部分が変形するため、以下の症状が発生する可能性があります。

- 上下左右方向に入力が加えられていない状態でも、C スティックからの出力値が変動する。
- ユーザーが入力した方向と異なる方向への入力として処理される。

X ボタンを長押しすることによる影響

X ボタンにかなりの荷重をかけたときに、C スティックを操作していないにもかかわらず、C スティックからの出力値が変動することがあります。この現象は、X ボタンを強く押すことでボタン周りの本体表面が変形し、その影響で C スティックの軸部分まで変形することが原因で発生します。

この現象は X ボタンを連打するケースでは発生せず、X ボタンを長押しするケースでもかなりの荷重をかけなければ発生しません。また、開発機および Newニンテンドー3DS でこの現象を発生させるには、Newニンテンドー3DS LL で発生させるよりも強い力で長押しする必要があります。

なお、この現象による C スティックからの出力値の変動範囲は、実用上考えられる範囲で、以下のようになっています。

表 6-10. X ボタンの長押しによる C スティックからの出力値の変動範囲

クランプ方式	X 軸	Y 軸
円形(min=40, max=145)	0～10	0～-35
十字(min=36, max=145)	0	0～-38
最小(min=40, max=145)	0～10	0～-37

感度の個体差による分解能への影響

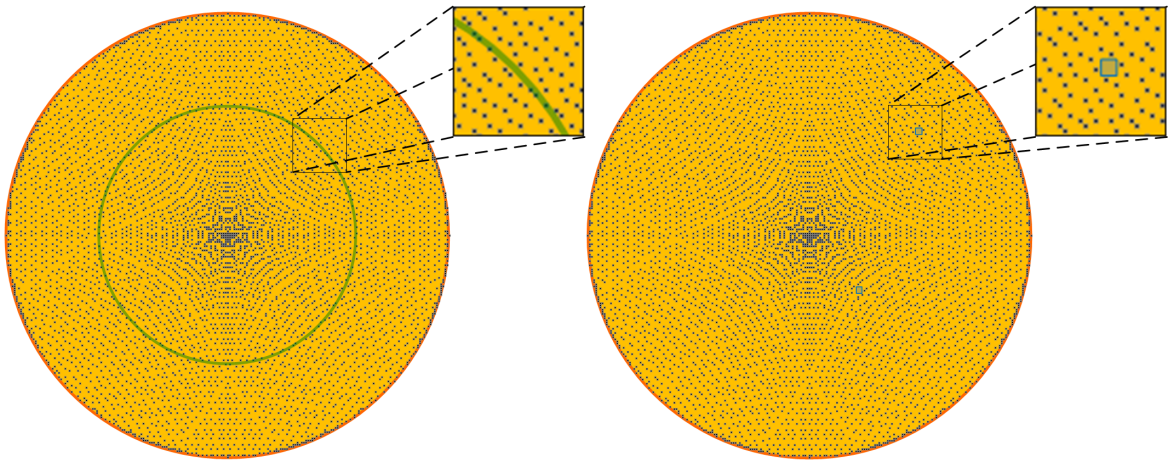
C スティックの感度には個体差があり、この個体差を吸収するために、生産工程でキャリブレーションを行っています。このキャリブレーションにより、キャリブレーション後の分解能が低い個体と高い個体が存在することになります。

個体差やクランプモードで変わりますが、クランプされたあとの C スティックからの出力値には、範囲内のすべての値が出力されるわけではありません。分解能が高い個体であっても連続して値が変化するとは限らず、分解能が低い個体はその傾向が顕著になります。ただし、中点および最外周については出力されることが保証されています。

そのため、出力値をごく狭い範囲に収めることを求めるような実装や滑らかな値変化を期待した判定を行うことは推奨しません。

以下の図は、分解能が最も低い個体で円形クランプをかけた際の出力値の分布と、推奨しないごく狭い範囲の例を図示したものです。オレンジ色の範囲がアプリケーションで使用可能な値の範囲、黒い点が実際に出力される値です。隣接する黒い点が連続した値であることは保証されていないことに注意してください。なお、左の緑の円は C スティックからの出力値をベクトル長に換算した値で判定する範囲が狭い場合、右の青い四角は C スティックからの出力値そのままで判定する範囲が狭い場合です。つまり、入力方向によっては求められるベクトル長を満たす点が存在しない可能性や、その範囲内の値を出力する点が存在しない可能性があるということです。

図 6-12. C スティックからの出力値の分布とごく狭い範囲の指定例（円形クランプ、分解能が最も低い個体を想定した場合）



6.2.7.2. 拡張スライドパッドとの相違点

HID ライブラリで SNAKE の C スティックへの入力を利用する場合、以下の点で拡張スライドパッドとは仕様や挙動に違いがあります。

- 設定可能なサンプリング周期が異なる。
- 外的要因による切断が発生しない。
- スリープ復帰時の挙動が異なる。
- 拡張スライドパッド補正アプレットでは中点補正および感度補正が行われない。

設定可能なサンプリング周期が異なる

SNAKE の C スティック、ZL ボタン、ZR ボタンに対して、ハードウェア的に設定可能なサンプリング周期は 10～21 ms です。

`nn::hid::ExtraPad::StartSampling()` では 8～32 ms の範囲で設定可能ですが、ライブラリ内で 10 ms 以下は 10 ms、21 ms 以上は 21 ms としてハードウェアからサンプリングし、関数で設定された周期でデータを返します。

そのため、ハードウェアのサンプリング周期に合わせて 10～21 ms の範囲でサンプリング周期を設定することを推奨します。

外的要因による切断が発生しない

赤外線通信によって接続されていた拡張スライドパッドでは外的要因(取り外しや電池切れ)による切断が発生しましたが、SNAKE の C スティック、ZL ボタン、ZR ボタンは本体に搭載されていますので、外的要因によって nn::hid::ExtraPad::GetConnectionState() が nn::hid::ExtraPad::CONNECTION_STATE_STOPPED を返すことはありません。

電池切れがないため、サンプリングを行っている間は nn::hid::ExtraPadStatus 構造体の batteryLevel には常に 1 が設定されます。

スリープ復帰時の挙動が異なる

nn::hid::ExtraPad::StopSampling() を呼び出さずにスリープ状態に遷移した場合、拡張スライドパッドと C スティックでは、以下のように挙動が異なります。

- 拡張スライドパッド
  スリープによって赤外線通信が停止することにより、スリープから復帰直後のサンプリングリクエストが失敗するためにサンプリングループを抜け、ライブラリは拡張スライドパッドが停止したとみなします。ただし、内部ステートの変化は赤外線通信の状態に依存するため、停止したことを検出する前の内部ステートは安定しません。
  具体的には、nn::hid::ExtraPad::GetConnectionState() はおおむね CONNECTION_STATE_STOPPED を返しますが、タイミングによっては CONNECTION_STATE_CONNECTED を返します。
- C スティック
  スリープから復帰直後もサンプリングリクエストが成功し、接続状態を維持したままになります。
  具体的には、nn::hid::ExtraPad::GetConnectionState() は常に CONNECTION_STATE_CONNECTED を返します。

また、nn::hid::ExtraPad::IsSampling() が返す値は内部ステートによって変化するため、nn::hid::ExtraPad::IsSampling() も同様のタイミングで値が変化します。

この挙動への対応として、スリープ状態に遷移する前に nn::hid::ExtraPad::StopSampling() を呼び出してサンプリングを停止することを推奨します。nn::hid::ExtraPad::StopSampling() を呼び出していた場合は、スリープから復帰した際に nn::hid::ExtraPad::StartSampling() でサンプリングを再開してください。アプリケーションに制御が戻る前にスリープ状態に遷移する可能性がある、HOMEメニュー、ライブラリアプレットの呼び出し前後も同様に対応することを推奨します。

拡張スライドパッド補正アプレットでは中点補正および感度補正が行われない

CTR で拡張スライドパッド補正アプレットを呼び出した場合と異なり、SNAKE でアプレットを呼び出した場合は C スティックの動作を確認するモードとなり、C スティックの中点補正を行う方法がメッセージによってユーザーに案内されます。

これは、以下のタイミングで C スティックの中点補正が自動的に行われるためです。

- 電源投入時
  C スティックを入力状態にしたまま電源が投入されると正常に補正されません。そのような場合でも、後述のスリープ復帰時の補正で正常な補正を行うことができます。
- スリープ復帰時
  蓋を閉じた状態では物理的に C スティックの操作を行うことができないため、C スティックを入力状態にしたままスリープ操作が行われることはまずありません。そのため、この補正が行われた場合は正常な補正が行われたとみなすことができます。
- サンプリング中(ユーザーによる操作が行われていない間)
  C スティックからの出力値がクランプの下限値以下の状態で行われる補正のため、アプリケーションでこの補正の影響を考慮する必要はありません。

注意: スリープ復帰時の中点補正はスリープから復帰したことをトリガーに行われます。そのため、アプリケーションでスリープを拒否すると蓋を開けても中点補正が行われません。なお、補正アプレットで表示されるメッセージでは、ユーザーに蓋を閉じてスリープすることを促します。

たとえば、以下のような理由でスリープを拒否するアプリケーションでは、スリープを拒否している間は中点補正を行えなくなることに注意してください。

- 蓋を閉じても通信を切断したくない
- サウンドモードで蓋を閉じたときに、ヘッドホンが接続されていたらスリープしない

6.3. MIC ライブラリ

MIC ライブラリでは、自動サンプリングによるマイクからの音声入力を扱うことができます。

MIC ライブラリの初期化は、`nn::mic::Initialize()` を呼び出して行います。この関数の処理が成功した時点でマイクの使用は可能となりますが、アプリケーションでマイクのサンプリングを行うには、サンプリング結果を格納するバッファの確保やマイクアンプのゲイン設定、マイク電源の制御など、サンプリングのための準備が必要です。

6.3.1. バッファの確保

サンプリング結果を格納するバッファはアプリケーションで確保してください。バッファは先頭アドレスが `nn::mic::BUFFER_ALIGNMENT`（4096 Byte）アライメント、サイズが `nn::mic::BUFFER_UNITSIZE`（4096）の倍数で、**デバイスメモリ以外から確保**しなければなりません。確保したバッファは、`nn::mic::SetBuffer()` で MIC ライブラリに渡します。バッファの終端部分の 4 Byte には、管理用のメモリ領域が確保されるため、実際にサンプリング結果が格納に使用されるバッファのサイズは、指定のサイズから管理用のメモリ領域のサイズを差し引いたものとなります。サンプリング結果の格納に使用されるバッファのサイズは、`nn::mic::GetSamplingBufferSize()` を呼び出して取得することもできます。

バッファが確保されている状態で、`nn::mic::SetBuffer()` を呼び出した場合はエラーとなります。バッファを確保しなおす場合は、先に `nn::mic::ResetBuffer()` でバッファを呼び出してから `nn::mic::SetBuffer()` を呼び出してください。

6.3.2. マイクアンプのゲイン設定

マイクアンプのゲイン（増幅率）設定は、マイクからの入力音声をどれくらいの倍率で増幅するかの設定です。

ゲインの取得と設定は `nn::mic::GetAmpGain()` と `nn::mic::SetAmpGain()` の呼び出しで行います。ゲインは 0 ～ 119 の範囲の値で 0.5 dB 刻みに設定することができ、0 が 10.5 dB（約 3.4 倍）、119 が 70.0 dB（約 3162 倍）に対応しています。NITRO でマイクアンプのゲインに設定可能だった 4 段階の倍率は以下の表のように、ゲインの設定値に変換して使用することができます。

表 6-11. NITRO で設定可能だった倍率とゲインの設定値

倍率	dB	ゲインの設定値
20 倍	26.0	31
40 倍	32.0	43
80 倍	38.0	55
160 倍	44.0	67

マイクアンプのゲイン設定は、ライブラリの初期化時に AMP_GAIN_DEFAULT_VALUE に設定されています。

6.3.3. マイク電源の制御

マイク電源(マイクアンプの電源)の制御は nn::mic::SetAmp() の呼び出しで行います。引数に true を渡して呼び出すことでマイク電源が ON になります。マイク電源を ON にした直後とスリープ状態から復帰した直後はマイクからの入力が安定しないため、1 秒間のサンプリング結果が強制的に無音となります。

現在の設定を取得するには nn::mic::GetAmp() を呼び出してください。

6.3.4. サンプリングの開始

ここまでの処理によってマイクのサンプリングを開始する準備は整いました。nn::mic::StartSampling() の呼び出しでマイクのサンプリングを開始することができ、サンプリングは自動で行われます。

コード 6-15. マイクサンプリングの開始

```
nn::Result nn::mic::StartSampling(
        nn::mic::SamplingType type, nn::mic::SamplingRate rate, s32 offset,
        size_t size, bool loop);
```

type には取得するデータの種別を指定します。データの種別は以下の 4 種類から選択することができ、ビット幅と符号の有無などに違いがあります。

表 6-12. マイクサンプリングデータの種別

type	ビット幅	符号の有無	サンプリング値の範囲	無音を示すサンプリング値
SAMPLING_TYPE_8BIT	8	なし	0 ～ 255	128
SAMPLING_TYPE_16BIT	16	なし	0 ～ 65535	32768
SAMPLING_TYPE_SIGNED_8BIT	8	あり	-128 ～ 127	0
SAMPLING_TYPE_SIGNED_16BIT	16	あり	-32768 ～ 32767	0

rate にはサンプリングレート(周波数)を指定します。サンプリングレートは以下の 4 種類から選択することができます。

表 6-13. マイクのサンプリングレート

rate	サンプリングレート
SAMPLING_RATE_32730	32.73 kHz(32728.498046875 Hz)
SAMPLING_RATE_16360	16.36 kHz(16364.2490234375 Hz)
SAMPLING_RATE_10910	10.91 kHz(10909.4993489583 Hz)
SAMPLING_RATE_8180	8.18 kHz(8182.12451171875 Hz)

offset にはサンプリングデータの格納開始位置(バッファの先頭からのオフセット)を指定します。0 以上、nn::mic::OUTPUT_OFFSET_ALIGNMENT (2 Byte) のアライメントで指定しなければなりません。

size には 1 回のサンプリングで格納するサンプリング結果のバイト数を nn::mic::CTR::OUTPUT_UNITSIZE (2) の

倍数で指定します。(*offset* + *size*)がバッファ確保時に指定したサイズ以下になるように指定しなければなりません。

loop には *size* で指定したバイト数分のサンプリング結果を格納したときに、サンプリングを継続するのか、停止するのかを指定します。true を指定した場合はサンプリングを継続し、バッファをリングバッファ(*offset* から *size* バイトの領域)のように扱います。

サンプリング中に nn::mic::StartSampling() を呼び出した場合は、サンプリングを中止してから新たなサンプリングを開始します。蓋が閉じられた状態で呼び出した場合は nn::mic::ResultShellClose を返し、サンプリングは開始されません。サンプリング中に蓋が閉じられると、自動的にサンプリングを停止し、開けられたときに再開します。

nn::mic::AdjustSampling() を呼び出すことでサンプリング中にサンプリングレートを変更することができます。サンプリングレートを変更すると、現在の格納位置から変更後のサンプリングデータを続けて格納します。

nn::mic::IsSampling() を呼び出すことでサンプリング中かどうかを取得できます。しかし、サンプリング中かどうかをデバイスに直接問い合わせるため処理が重く、毎フレーム呼び出すような処理には適していません。

nn::mic::SetLowPassFilter() の引数 *enable* に true を渡して呼び出すことで、マイク入力にカット周波数がサンプリングレートの 45 % のローパスフィルタを適用することができます。デフォルトの設定は false(適用しない)です。ただし、サンプリングレートに SAMPLING_RATE_32730 を指定してサンプリングしたデータは、オリジナルデータにローパスフィルタが適用済みのため、この関数でローパスフィルタを適用しても効果はありません。

6.3.4.1. サウンド処理との同期

サウンド処理を行っている DSP とマイク処理を行っているプロセッサは別のデバイスですので、完全に同期させることができません。ただし、ズレの補正を行うことで同期に近い状態にすることは可能です。

タイマーのズレに関しては、一定時間ごとにマイクのサンプリング数と経過時間やサウンドの再生時間とのズレを計算して補正を行なってください。また、サンプリング周波数のズレについては、Voice::SetPitch() で調整して補正を行ってください。

6.3.5. サンプリング結果の取得

nn::mic::GetLastSamplingAddress() の呼び出しで、最新のサンプリング結果が格納されたアドレスを取得することができます。

nn::mic::GetBufferFullEvent() の呼び出しで取得することのできる nn::os::Event クラスのインスタンスは、バッファにこれ以上のサンプリング結果を格納することができなくなったときにシグナル状態になる手動リセットイベントです。このイベントは *offset* の位置から *size* で指定されたバイト数のサンプリング結果を格納したときにシグナル状態となり、ClearSignal() で解除されるまでシグナル状態を維持します。

入力として得られる値の範囲には本体によって個体差があり、サンプリング種別ごとの入力値の保証範囲は、TYPE_*_GUARANTEED_INPUT_MIN(MAX) で定義されています。マイク入力の判定に、入力保証範囲外の値を期待するような実装はしないでください。

表 6-14. サンプリング種別ごとのマイク入力値の保証範囲

サンプリング種別	下限値	上限値
SAMPLING_TYPE_8BIT	27	228
SAMPLING_TYPE_16BIT	7105	58415
SAMPLING_TYPE_SIGNED_8BIT	-101	100
SAMPLING_TYPE_SIGNED_16BIT	-25663	25647

デフォルトでは、上記の保証範囲内の値になるように、マイクの入力値はクランプ処理されています。より広い入力範囲を必要とする場合は、nn::mic::SetClamp() でクランプ処理を無効に設定することができますが、保証範囲外の入力値が取得できない可能性があることに注意してください。

6.3.5.1. マイク入力あり判定禁止領域

nn::mic::GetForbiddenArea() にマイクアンプのゲイン設定とサンプリング種別を渡し、マイク入力ありと判定してはいけないサンプリング結果の下限値と上限値を取得してください。取得した下限値から上限値の間の値となったサンプリング結果をマイク入力ありと判定しないでください。取得可能な値を表 6-15 に示します。

「表 6-15. マイク入力あり判定禁止領域」の中で薄い灰色で示してある、ゲインが 68 以上(44.5 dB 以上)の場合は、本体スピーカーの出力音や各種ボタンの操作音、タッチパネル操作音、その他の外部からの雑音などの影響が、マイク入力に対して顕著に表れます。そのため、ゲインが 68 以上の場合は、振幅レベルによるマイク入力あり/なしの判定に使用するには不適切です。さらに、濃い灰色で示してある、ゲインが 104 以上(62.5 dB 以上)の場合は、ノイズ成分がマイク入力の範囲全域に現れるため、マイク入力が不正であってもかまわない場面でのみ使用するようにしてください。

表 6-15. マイク入力あり判定禁止領域

サンプリング種別	ゲイン	dB	マイク入力あり判定禁止領域(ノイズ成分)
SAMPLING_TYPE_8BIT	0 ～ 31	10.5 ～ 26.0	125 ～ 131
	32 ～ 43	26.5 ～ 32.0	123 ～ 133
	44 ～ 55	32.5 ～ 38.0	119 ～ 137
	56 ～ 67	38.5 ～ 44.0	112 ～ 144
	68 ～ 80	44.5 ～ 50.5	96 ～ 160
	81 ～ 91	51.0 ～ 56.0	77 ～ 179
	92 ～ 103	56.5 ～ 62.0	18 ～ 238
	104 ～ 119	62.5 ～ 70.0	0 ～ 255
SAMPLING_TYPE_16BIT	0 ～ 31	10.5 ～ 26.0	32000 ～ 33536
	32 ～ 43	26.5 ～ 32.0	31488 ～ 34048
	44 ～ 55	32.5 ～ 38.0	30464 ～ 35072
	56 ～ 67	38.5 ～ 44.0	28672 ～ 36864
	68 ～ 80	44.5 ～ 50.5	24576 ～ 40960
	81 ～ 91	51.0 ～ 56.0	19712 ～ 45824
	92 ～ 103	56.5 ～ 62.0	4608 ～ 60928
	104 ～ 119	62.5 ～ 70.0	0 ～ 65535
SAMPLING_TYPE_SIGNED_8BIT	0 ～ 31	10.5 ～ 26.0	- 3 ～ + 3
	32 ～ 43	26.5 ～ 32.0	- 5 ～ + 5
	44 ～ 55	32.5 ～ 38.0	- 9 ～ + 9
	56 ～ 67	38.5 ～ 44.0	- 16 ～ + 16
	68 ～ 80	44.5 ～ 50.5	- 32 ～ + 32

	81 ～ 91	51.0 ～ 56.0	- 51 ～ + 51
	92 ～ 103	56.5 ～ 62.0	- 110 ～ + 110
	104 ～ 119	62.5 ～ 70.0	- 128 ～ + 127
SAMPLING_TYPE_SIGNED_16BIT	0 ～ 31	10.5 ～ 26.0	- 768 ～ + 768
	32 ～ 43	26.5 ～ 32.0	- 1280 ～ + 1280
	44 ～ 55	32.5 ～ 38.0	- 2304 ～ + 2304
	56 ～ 67	38.5 ～ 44.0	- 4096 ～ + 4096
	68 ～ 80	44.5 ～ 50.5	- 8192 ～ + 8192
	81 ～ 91	51.0 ～ 56.0	- 13056 ～ + 13056
	92 ～ 103	56.5 ～ 62.0	- 28160 ～ + 28160
	104 ～ 119	62.5 ～ 70.0	- 32768 ～ + 32767

6.3.6. サンプリングの停止

開始されているサンプリングの停止は、nn::mic::StopSampling() の呼び出しで行います。この関数の呼び出しではサンプリングが停止するだけで、マイクの電源は OFF になりません。

6.3.7. MIC ライブラリの終了

アプリケーションの終了時など、マイク入力の使用を終了する場合は、サンプリングを停止したあとに以下の手順で終了処理を行ってください。

1. nn::mic::SetAmp() に false を渡して呼び出し、マイクの電源を OFF にします。
2. nn::mic::ResetBuffer() を呼び出し、確保していたバッファを解放します。
3. nn::mic::Finalize() を呼び出して MIC ライブラリを終了させます。

時間を空けてサンプリングを行う場合、1. の処理のみを行ってマイクの電源を OFF にし、バッテリーの消費を抑えることができます。

マイクアンプのゲイン設定は、ライブラリの終了時に AMP_GAIN_DEFAULT_VALUE に設定されます。

6.4. CAMERA ライブラリ、Y2R ライブラリ

CAMERA ライブラリでは、本体に搭載されているカメラを扱うことができます。カメラでキャプチャされた画像は YUV フォーマットでしか取得することができません。RGB フォーマットへの変換には、GPU のネイティブフォーマットへの変換にも対応している YUVtoRGB 回路の使用をお勧めします。YUVtoRGB 回路の扱いには Y2R ライブラリを利用してください。

6.4.1. 初期化

CAMERA ライブラリは nn::camera::Initialize() を、Y2R ライブラリは nn::y2r::Initialize() を呼び出すことで初期化が行われます。カメラも YUVtoRGB 回路も、これら初期化関数の呼び出しだけでは、まだ利用することができません。それぞれの設定やデータの送受信のための準備などが必要となります。

6.4.2. 撮影環境の設定

撮影環境に関して設定可能な項目について説明します。設定に使用する関数の呼び出しには対象とするカメラを指定する必要があり、以下の設定値から指定します。

表 6-16. 対象となるカメラの指定

設定値	対象となるカメラ
SELECT_NONE	カメラ指定なし。スタンバイ状態の設定などに使用します。
SELECT_OUT1	外側カメラ(R)。
SELECT_IN1	内側カメラ。
SELECT_IN1_OUT1	外側カメラ(R)と内側カメラ。
SELECT_OUT2	外側カメラ(L)。
SELECT_OUT1_OUT2	外側カメラ(R)と外側カメラ(L)。
SELECT_IN1_OUT2	外側カメラ(L)と内側カメラ。
SELECT_ALL	すべてのカメラ。外側カメラ(R)、外側カメラ(L)、内側カメラの 3 つ。

SELECT_OUT1_OUT2 を設定したときは特にステレオカメラと呼び、2 つのカメラで取得したキャプチャ画像を立体視表示に利用することができます。その際は、2 つのカメラの撮影環境がなるべく同じになるように設定してください。

補足: ステレオカメラを利用した立体視表示やキャリブレーションの方法については、「3DS プログラミングマニュアル - グラフィックス応用編」を参照してください。

注意: 撮影環境の設定をキャプチャ中に行うと、キャプチャ画像がずれてしまう可能性がありますので、キャプチャを一時的に停止した状態で設定を行ってください。

SELECT_ALL によるカメラの指定は、外側のカメラが 1 つであったときと現在とでは対象となるカメラが異なることに注意してください。

カメラの再起動処理中に撮影環境の設定が行われると、ライブラリ内で処理が長時間ブロックされることがありますので注意してください。

カメラが撮影して得られる画像データには、カメラ個体のばらつき、環境光によるばらつき、被写体自体の色のばらつきを含むことになります。たとえば、カメラ個体のばらつきには画角、解像度、色再現性、回転、ディストーション等を含みます。そのため、同じアングル、同じカメラ設定で撮影して得られる画像は、本体ごとに異なります。

解像度

カメラがキャプチャするイメージの解像度を nn::camera::SetSize() の呼び出しで設定することができます。解像度はトリミングされる前のイメージサイズで、設定可能な解像度には以下のものがあります。

表 6-17. 解像度

設定値	解像度	説明
SIZE_VGA	640 x 480	VGA

SIZE_QVGA	320 x 240	QVGA
SIZE_QQVGA	160 x 120	QQVGA
SIZE_CIF	352 x 288	CIF(アスペクト比 11: 9)
SIZE_QCIF	176 x 144	QCIF(アスペクト比 11: 9)
SIZE_DS_LCD	256 x 192	DS の LCD の解像度
SIZE_DS_LCDx4	512 x 384	DS の LCD の解像度を幅、高さともに 2 倍にした解像度
SIZE_CTR_TOP_LCD	400 x 240	3DS の上 LCD の解像度
SIZE_CTR_BOTTOM_LCD	320 x 240	3DS の下 LCD の解像度。QVGA と同じです

SIZE_CIF、SIZE_QCIF、SIZE_CTR_TOP_LCD でキャプチャした画像は、4 : 3 でキャプチャした画像の左右が切られた形で取得することになります。

nn::camera::SetDetailSize() を呼び出すことで、上記以外の解像度を設定することができます。出力画像の幅と高さ、元画像(VGA サイズ)から切り出す領域(クロップ領域)を指定することで、任意の解像度でキャプチャ画像を取得することができます。必ずクロップ領域の幅と高さが出力画像以上になるように指定してください。トリミングを行わない場合は出力画像の幅と高さを乗じた値が 128 の倍数でなければなりません。また、互換性維持のため、必ず引数 *cropX0* は偶数かつ、*cropX1* - *cropX0* + 1 が 4 の倍数になるように指定してください。

ステレオカメラを利用して立体視表示をするときは、2 つのカメラの解像度を同じ解像度に設定してください。

鮮明度

カメラの鮮明度を nn::camera::SetSharpness() の呼び出しで設定することができます。設定可能な鮮明度は -4 ～ +5 の範囲です。

露光

カメラの露光を nn::camera::SetExposure() の呼び出しで設定することができます。設定可能な露出量は -5 ～ +5 の範囲です。

自動露出機能は nn::camera::SetAutoExposure() の呼び出しで有効・無効を制御することができます。カメラ起動直後は露出値が不安定になるため、自動露出機能を有効にしておくことを推奨しています。現在の設定は nn::camera::IsAutoExposure() の呼び出しで取得することができます。自動露出機能が無効に設定されていても、nn::camera::SetExposure() で露出を設定すると有効に切り替わることに注意してください。

nn::camera::SetAutoExposureWindow() を呼び出すことで、自動露出機能による露出の自動計算の基準となる領域を設定することができます。領域はキャプチャ画像の最大サイズである VGA(640 × 480)内の部分領域として指定し、開始座標と幅、高さは以下の範囲内で変更することができます。また、表の右端 3 列はそれぞれ、内側カメラ、外側カメラ(R)、外側カメラ(L)の初期設定値です。

表 6-18. 露出の基準領域の指定

引数	設定	設定値の範囲	IN1	OUT1	OUT2
startX	開始座標(横方向)	0 ～ 600(40 ピクセル単位)	80	0	0
startY	開始座標(縦方向)	0 ～ 450(30 ピクセル単位)	60	0	0
width	幅	40 ～ 640(40 ピクセル単位) *startX* との和が 640 以内	480	640	640

height	高さ	30 ～ 480(30 ピクセル単位) *staryY* との和が 480 以内	360	480	480

フレームレート

1 秒間にキャプチャする画像の枚数(fps)を nn::camera::SetFrameRate() の呼び出しで設定することができます。設定可能なフレームレートには以下のものがあります。ステレオカメラを利用するときは、2 つのカメラに対して同じ固定フレームレートを設定してください。

表 6-19. フレームレート

設定値	フレームレート
FRAME_RATE_15	15 fps 固定
FRAME_RATE_15_TO_5	明るさに応じて 15 fps から 5 fps の間で自動的に変化
FRAME_RATE_15_TO_2	明るさに応じて 15 fps から 2 fps の間で自動的に変化
FRAME_RATE_10	10 fps 固定
FRAME_RATE_8_5	8.5 fps 固定
FRAME_RATE_5	5 fps 固定
FRAME_RATE_20	20 fps 固定
FRAME_RATE_20_TO_5	明るさに応じて 20 fps から 5 fps の間で自動的に変化
FRAME_RATE_30	30 fps 固定
FRAME_RATE_30_TO_5	明るさに応じて 30 fps から 5 fps の間で自動的に変化
FRAME_RATE_15_TO_10	明るさに応じて 15 fps から 10 fps の間で自動的に変化
FRAME_RATE_20_TO_10	明るさに応じて 20 fps から 10 fps の間で自動的に変化
FRAME_RATE_30_TO_10	明るさに応じて 30 fps から 10 fps の間で自動的に変化

ホワイトバランス

ホワイトバランスを nn::camera::SetWhiteBalance() の呼び出しで設定することができます。設定可能なホワイトバランスには以下のものがあります。

表 6-20. ホワイトバランス

設定値	エイリアス指定	説明
WHITE_BALANCE_AUTO	WHITE_BALANCE_NORMAL	オートホワイトバランス
WHITE_BALANCE_3200K	WHITE_BALANCE_TUNGSTEN	タングステン光(白熱電球)
WHITE_BALANCE_4150K	WHITE_BALANCE_WHITE_FLUORESCENT_LIGHT	白色蛍光灯
WHITE_BALANCE_5200K	WHITE_BALANCE_DAYLIGHT	太陽光
WHITE_BALANCE_6000K	WHITE_BALANCE_CLOUDY	くもり
	WHITE_BALANCE_HORIZON	夕焼け

WHITE_BALANCE_7000K	WHITE_BALANCE_SHADE	日陰

自動調整機能は、nn::camera::SetWhiteBalance() でホワイトバランスを WHITE_BALANCE_AUTO に設定すると有効になり、それ以外に設定すると無効になることに注意してください。

WHITE_BALANCE_AUTO で設定されるホワイトバランスのまま、自動調整機能の有効・無効だけを制御したい場合は nn::camera::SetAutoWhiteBalance() を使用してください。WHITE_BALANCE_AUTO 以外が設定されている場合、nn::camera::SetAutoWhiteBalance() を使用することはできません。自動調整機能の現在の設定は nn::camera::IsAutoWhiteBalance() の呼び出しで取得することができます。

nn::camera::SetAutoWhiteBalanceWindow() を呼び出すことで、ホワイトバランスの自動調節機能によるホワイトバランスの自動計算の基準となる領域を設定することができます。領域はキャプチャ画像の最大サイズである VGA(640 × 480)内の部分領域として指定し、開始座標と幅、高さは以下の範囲内で変更することができます。また、表の右端 3 列はそれぞれ、内側カメラ、外側カメラ(R)、外側カメラ(L)の初期設定値です。

表 6-21. ホワイトバランスの基準領域の指定

引数	設定	設定値の範囲	IN1	OUT1	OUT2
startX	開始座標(横方向)	0 ～ 600(40 ピクセル単位)	0	0	0
startY	開始座標(縦方向)	0 ～ 450(30 ピクセル単位)	0	0	0
width	幅	40 ～ 640(40 ピクセル単位) *startX* との和が 640 以内	640	640	640
height	高さ	30 ～ 480(30 ピクセル単位) *staryY* との和が 480 以内	480	480	480

補足: 入力画像に変化が少ない場合やコントラストが少ない画像(無地の壁など)の場合は、ホワイトバランスの自動調整機能の働きが弱くなることがあります。

撮影モード

被写体に合わせて、撮影モードを nn::camera::SetPhotoMode() の呼び出しで設定することができます。設定可能な撮影モードには以下のものがあります。

表 6-22. 撮影モード

設定値	撮影モード	説明
PHOTO_MODE_NORMAL	補正なし	カメラの設定は補正されません。
PHOTO_MODE_PORTRAIT	ポートレートモード	人物の撮影に向いた設定です。
PHOTO_MODE_LANDSCAPE	風景モード	風景の撮影に向いた設定です。
PHOTO_MODE_NIGHTVIEW	暗視モード	暗所での撮影に向いた設定です。
PHOTO_MODE_LETTER	文字モード	QR コードや文字の撮影に向いた設定です。

撮影モードを変更すると、下表のようにコントラストやゲイン、鮮明度、露光、ホワイトバランスが上書きされるほか、外側カメラ

と内側カメラの露出の基準領域が変更されます。「全体」の設定は開始座標が (0, 0)、幅と高さが 640 と 480 です。「中心」の設定は開始座標が (80, 60)、幅と高さが 480 と 360 です。

表 6-23. 撮影モードで調整される撮影環境

撮影モード	コントラスト	ゲイン	鮮明度	露光	ホワイトバランス	外側カメラ	内側カメラ
補正なし	NORMAL	通常	0	0	NORMAL	全体	中心
ポートレートモード	LOW	通常	-2	0	NORMAL	中心	中心
風景モード	NORMAL	通常	+1	0	DAYLIGHT	全体	中心
暗視モード	NORMAL	最大	-1	+2	NORMAL	全体	中心
文字モード	HIGH	通常	+2	+2	NORMAL	全体	中心

反転処理

出力画像に適用される反転処理を nn::camera::FlipImage() の呼び出しで設定することができます。設定可能な反転処理には以下のものがあります。

表 6-24. 反転処理

設定値	適用される反転処理
FLIP_NONE	反転処理なし
FLIP_HORIZONTAL	左右反転
FLIP_VERTICAL	上下反転
FLIP_REVERSE	180 度回転(上下、左右ともに反転した状態)

エフェクト

出力画像に適用される特殊効果(エフェクト)を nn::camera::SetEffect() の呼び出しで設定することができます。設定可能なエフェクトには以下のものがあります。

表 6-25. エフェクト

設定値	エフェクト
EFFECT_NONE	特殊効果なし
EFFECT_MONO	モノクロ調
EFFECT_SEPIA	セピア調(黄土色)
EFFECT_NEGATIVE	ネガポジ反転
EFFECT_NEGAFILM	フィルム調のネガポジ反転。EFFECT_NEGATIVE の U と V の順番を入れ替えています
EFFECT_SEPIA01	セピア調(赤褐色)

エフェクトを適用した上でほかの設定を変更すると、エフェクトの効果が変わります。セピア調に設定した上で鮮明度を下げれば印象が柔らかくなり、フィルム調のネガポジ反転に設定した上でホワイトバランスの色温度を上げれば赤みを強調することができます。

コントラスト

コントラスト(ガンマカーブ)を nn::camera::SetContrast() の呼び出しで設定することができます。設定可能なコントラストには以下のものがあります。

表 6-26. コントラスト

設定値	コントラスト
CONTRAST_PATTERN_*n*(*n* は 01 ～ 11)	コントラストのパターン No.*n*
CONTRAST_HIGH	デフォルトよりもコントラスト比が高くなる設定(パターン No.7)
CONTRAST_NORMAL	デフォルトの設定(パターン No.6)
CONTRAST_LOW	デフォルトよりもコントラスト比が低くなる設定(パターン No.5)

レンズ補正

レンズ補正とは、光量の差によって画像の中心と周辺の明るさに差が出てしまう現象(周辺光量低下)への対策として、画像の周辺の明るさを補正して中心の明るさに近づける処理のことです。レンズ補正は nn::camera::SetLensCorrection() の呼び出しで設定することができます。設定可能なレンズ補正の値には以下のものがあります。

表 6-27. レンズ補正

設定値	レンズ補正
LENS_CORRECTION_OFF	レンズ補正の設定を無効にします
LENS_CORRECTION_ON_70	レンズ補正を 70 に設定します
LENS_CORRECTION_ON_90	レンズ補正を 90 に設定します
LENS_CORRECTION_DARK	デフォルトよりも画像の周辺が暗くなる設定(LENS_CORRECTION_OFF)
LENS_CORRECTION_NORMAL	デフォルトの設定(LENS_CORRECTION_ON_70)
LENS_CORRECTION_BRIGHT	デフォルトよりも画像の周辺が明るくなる(LENS_CORRECTION_ON_90)

コンテキスト

コンテキストを利用すると、複数の撮影環境の設定をまとめて切り替えることができます。コンテキストの切り替えは、nn::camera::SwitchContext() を呼び出して行います。コンテキストはカメラごとに A と B の 2 つ(計 6 パターン)があり、外側カメラがコンテキスト A、内側カメラがコンテキスト B のように独立して設定することができます。

コンテキストによって切り替えられる撮影環境は、解像度、反転処理、エフェクトの 3 項目です。

ノイズフィルタ

画面の明るさが明から暗またはその逆に変化したとき、カメラモジュールはノイズフィルタ機能によって自動的にキャプチャ画像のノイズを除去します。この機能が有効になっていると、ステレオカメラ使用時に片方のカメラのキャプチャ画像だけがぼやけて見える状況になることがあります。それを防ぐために nn::camera::SetNoiseFilter() の引数 *on* に

false を渡して呼び出して、ノイズフィルタを無効にしてください。*on* に true を渡した場合は有効になります。どのカメラも、デフォルトでノイズフィルタが有効に設定されています。

撮影環境の一括設定

撮影環境を一括して変更する関数が用意されています。

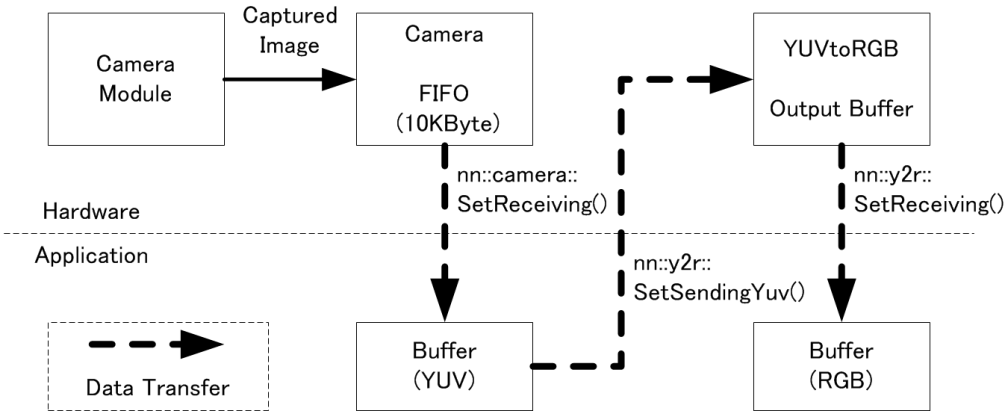
nn::camera::SetPackageParameterWithoutContext() は、コンテキスト指定のない撮影環境すべて(解像度、反転処理、エフェクト以外)を一括して設定することができます。

nn::camera::SetPackageParameterWithContext() は、コンテキスト指定のある撮影環境すべて(解像度、反転処理、エフェクト)を一括して設定することができます。

6.4.3. キャプチャの設定

カメラがキャプチャした画像は数ラインずつ FIFO に書き込まれ、アプリケーションは用意したバッファに YUV フォーマットのデータを受け取ります。アプリケーションは次に、受け取ったデータを YUVtoRGB 回路に送信し、RGB フォーマットに変換されたデータを受け取ります。この一連のデータの流れを図にすると以下のようになります。

図 6-13. キャプチャされたデータの流れ



ここでは、YUV フォーマットのキャプチャ画像をアプリケーションのバッファに受け取るまでに必要な設定について説明します。

ポートについて

ステレオカメラのように 2 つのカメラを同時に使用できるようになり、キャプチャした画像を取得するときに、どのカメラから画像を取得するのかを指定しなければならなくなりました。3 つあるカメラのうち、内側カメラと外側カメラ(R)が 1 つのポートに接続されており、外側カメラ(L)は単独で別のポートに接続されています。

キャプチャの設定を行う関数のほとんどがポートの指定を必要とし、そのポートの指定には以下の列挙子を使用します。

表 6-28. ポートの指定

列挙子	説明
PORT_NONE	ポートを指定しません。
PORT_CAM1	内側カメラと外側カメラ(R)が接続されているポートを指定します。
PORT_CAM2	外側カメラ(L)が接続されているポートを指定します。
PORT_BOTH	両方のポートを指定します。

トリミング

トリミングは、キャプチャ画像の一部を切り取ってアプリケーションが必要とするサイズにする処理です。カメラの解像度と必要なキャプチャ画像のサイズが異なる場合に使用します。

トリミングは nn::camera::SetTrimming() の呼び出しで有効・無効を制御することができます。トリミングを有効にすると、nn::camera::SetTrimmingParams() または nn::camera::SetTrimmingParamsCenter() でトリミングの範囲を設定することができます。トリミングが有効であるかどうかは nn::camera::IsTrimming() の呼び出しで判定することができ、現在のトリミング範囲は nn::camera::GetTrimmingParams() の呼び出しで取得することができます。

> **注意:** トリミングの設定はキャプチャの開始前に行ってください。

トリミングの位置と範囲を nn::camera::SetTrimmingParams() で設定する場合は、トリミング開始位置(x1, y1)とトリミング終了位置(x2, y2)を指定します。(x1, y1)はトリミングの範囲に含まれますが、(x2, y2)は含まれません。これらの設定値には以下の制限があります。

- トリミング開始位置の座標 x1 と y1 は偶数でなければなりません。
- x1 < x2, y1 < y2 を満たさなければなりません。
- トリミング後の画像の幅(x2 – x1)と(y2 – y1)は偶数でなければなりません。
- トリミング後の画像の幅と高さを乗じた値が 128 の倍数でなければなりません。

nn::camera::SetTrimmingParamsCenter() で設定する場合は、トリミングサイズの幅(trimWidth)と高さ(trimHeight)、カメラ解像度の幅(camWidth)と高さ(camHeight)を指定し、キャプチャ画像の中心を基準にトリミングを行います。この設定から計算される(x1, y1)と(x2, y2)は以下のコードで表すことができます。計算後の位置が nn::camera::SetTrimmingParams() での制限にかからないように注意してください。

コード 6-16. 中心を基準にしたトリミング設定の位置計算

```
x1 = (camWidth - trimWidth) / 2;
y1 = (camHeight - trimHeight) / 2;
x2 = x1 + trimWidth;
y2 = y1 + trimHeight;
```

バッファに転送されるキャプチャ画像のサイズはトリミング後のサイズです。トリミング後のサイズが同じであれば、カメラの解像度が異なる場合でも転送にかかるコストは変わりません。ただし、高い解像度でキャプチャした画像に対してトリミングを行うと、視野角の違いによって映る範囲が狭まってしまいます。

転送バイト(ライン)数

カメラでキャプチャした画像は、ハードウェア上の FIFO に分割して格納され、数回の転送でアプリケーションが用意したバッファに書き込まれます。FIFO の容量は 10 KByte と決まっています。また、一回の転送で転送されるバイト数には、以下の条件があります。

- 一回の転送バイト数は 256 Byte の整数倍でなければなりません。
- 一回の転送バイト数は 10 KByte(10240 Byte)以下でなければなりません。
- 総転送バイト数は一回の転送バイト数の整数倍でなければなりません。

総転送バイト数はトリミング後の画像の幅と高さに 1 ピクセルあたりのバイト数である 2 を乗じた値です。

一回の転送で転送されるバイト数をライン数で設定するには、nn::camera::SetTransferLines() を呼び出します。nn::camera::GetMaxLines() で返されるライン数は、そのまま一回の転送で転送されるライン数として設定することができますが、上記の条件を満たすライン数が見つからなければ関数内で停止しますので注意してください。

nn::camera::SetDetailSize() で任意の解像度を設定しているなど、ライン数で設定することができないときは、nn::camera::SetTransferBytes() を呼び出して一回の転送で転送されるバイト数を設定してください。nn::camera::GetMaxBytes() で返されるバイト数は、そのまま一回の転送で転送されるバイト数として設定することができますが、幅と高さに 1 ピクセルあたりのバイト数である 2 を乗じた値が 256 の整数倍でなければ関数内で停止しますので注意してください。

> **注意:** 転送バイト(ライン)数の設定はキャプチャの開始前に行ってください。バッファエラー回避のため、GetMaxLines() と GetMaxBytes() は FIFO の容量を 5 KByte とみなして計算します。

現在の設定は nn::camera::GetTransferBytes() の呼び出しで取得することができます。

受信バッファ

1 フレームのキャプチャ画像を受信するために必要なバッファのサイズは nn::camera::GetFrameBytes() で取得することができます。バッファの先頭アドレスのアライメントは 4 Byte ですが、64 Byte 未満のアライメントでは転送速度が落ちる可能性があります。**デバイスメモリ上に確保したバッファのみ指定可能です。**

6.4.4. キャプチャの開始

転送とキャプチャの開始前に nn::camera::Activate() を呼び出してキャプチャに使用するカメラを起動します。同じポートに接続されている、内側カメラと外側カメラ(R)を同時に起動することはできません。別のポートに接続されている外側カメラ(L)は、内側カメラまたは外側カメラ(R)のどちらかと同時に起動することができます。

カメラの起動を確認してから、nn::camera::SetReceiving() を呼び出して転送を開始し、nn::camera::StartCapture() を呼び出してキャプチャを開始してください。

コード 6-17. nn::camera::SetReceiving()

```
void nn::camera::SetReceiving(nn::os::Event* pEvent, void* pDst,
        nn::camera::Port port, size_t imageSize, s16 transferUnit);
```

pEvent には転送完了の通知を受け取るイベントを指定します。*pDst* には受信バッファの先頭アドレスを nn::camera::BUFFER_ALIGNMENT (4 Byte) アライメントで指定します。*port* にはポートを指定します。*imageSize* には 1 フレームのキャプチャ画像のバイトサイズ(受信バッファのサイズ)を指定してください。*transferUnit* には nn::camera::GetTransferBytes() で返される一回の転送で転送されるデータのバイトサイズを指定してください。

nn::camera::IsFinishedReceiving() の呼び出しで、1 フレーム分のキャプチャ画像の転送が終了したかどうかを取得することができます。

FIFO への書き込みエラーなど、キャプチャの途中でエラーが発生したかどうかを確認するには、nn::camera::GetBufferErrorInterruptEvent() で取得することのできるエラーイベントがシグナル状態になっていないかをチェックしてください。エラーイベントはバッファエラーが発生したときにシグナル状態になる、nn::os::Event クラスの自動リセットイベントです。また、このエラーイベントはカメラの誤作動による再起動処理が発生したときにもシグナル状態となります。エラーから復帰するには、転送、キャプチャの順に再開してください。

nn::camera::IsBusy() の呼び出しでキャプチャ画像を取り込んでいる最中であるかどうかを取得することができます。また nn::camera::GetVsyncInterruptEvent() では、カメラの VSYNC 割り込みの発生でシグナル状態となるイベントを取得することができます。カメラの VSYNC に同期した処理や、フレーム転送が行われていない期間に撮影環境を変更する処理などに利用することができます。

ステレオカメラを利用して立体視表示をするときなどに、カメラの VSYNC 割り込みのタイミングを同期させたい場合は

nn::camera::SynchronizeVsyncTiming() を呼び出してください。この関数は 2 つのカメラの VSYNC 割り込みのタイミングが同期するように試みます。同期した状態を継続させる処理ではありませんので、ズレが生じた場合には再び呼び出す必要があります。また、カメラを一度スタンバイ状態にしてから復帰させるため、関数を呼び出した直後の 4 フレームの画像は極端に暗くなることがあります。2 つのカメラの設定を同じにしていても、それぞれのカメラからキャプチャ画像が送出されるタイミングにはズレがあります。また、同じ設定、固定フレームレートで撮影していても、nn::camera::Activate() を呼び出してカメラを起動したときや撮影環境の設定を変更したときは VSYNC 割り込みのタイミングに大きなズレが生じることがあります。

キャプチャ画像(YUV4:2:2 フォーマット)は、以下のデータの並びで出力されます。

+ 0 Byte	+ 1 Byte	+ 2 Byte	+ 3 Byte
Y (n)	U (n)	Y (n + 1)	V (n)

6.4.5. YUVtoRGB 回路の設定

YUVtoRGB 回路は YUV フォーマットのデータをハードウェアで RGB フォーマットに変換することができます。また、GPU のネイティブフォーマットであるブロックフォーマットでの出力機能も備えています。しかし、YUVtoRGB 回路は 1 つしか搭載されていませんので、ステレオカメラなどで複数のカメラからのキャプチャ画像を変換するときには、排他処理などで複数の変換要求が同時に発生しないようにしなければなりません。

ここでは、YUVtoRGB 回路による変換に関係する設定項目を説明します。

入力フォーマット

入力する YUV データのフォーマットは nn::y2r::SetInputFormat() を呼び出して設定します。現在の設定は nn::y2r::GetInputFormat() で取得することができます。入力可能なフォーマットには以下のものがあります。

表 6-29. 入力フォーマット

設定値	フォーマット
INPUT_YUV422_INDIV_8	YUV4:2:2 の Y、U、V を個別に 8 bit で入力します
INPUT_YUV420_INDIV_8	YUV4:2:0 の Y、U を個別に 8 bit で入力します
INPUT_YUV422_INDIV_16	YUV4:2:2 の Y、U、V を個別に 16 bit(要パディング)で入力します
INPUT_YUV420_INDIV_16	YUV4:2:0 の Y、U を個別に 16 bit(要パディング)で入力します
INPUT_YUV422_BATCH	YUV4:2:2 の Y、U、V をまとめて 32 bit で入力します

それぞれのデータ形式は以下のようになっています。

YUV4:2:2/YUV4:2:0 個別入力(8 bit)

成分	+ 0 Byte	+ 1 Byte	+ 2 Byte	+ 3 Byte
Y	Y(n)	Y (n + 1)	Y (n + 2)	Y (n + 3)
U	U (n)	U (n + 1)	U (n + 2)	U (n + 3)
V	V (n)	V (n + 1)	V (n + 2)	V (n + 3)

YUV4:2:2/YUV4:2:0 個別入力(16 bit padding)

成分	+ 0 Byte	+ 1 Byte	+ 2 Byte	+ 3 Byte
Y	Y (n)	padding	Y (n + 1)	padding
U	U (n)	padding	U (n + 1)	padding
V	V (n)	padding	V (n + 1)	padding

YUV4:2:2 一括入力

成分	+ 0 Byte	+ 1 Byte	+ 2 Byte	+ 3 Byte
YUV	Y (n)	U (n)	Y (n + 1)	V (n)

カメラのキャプチャ画像は YUV 一括のフォーマットでのみ取得することができますので、ほとんどの場合、キャプチャ画像の変換時には INPUT_YUV422_BATCH を指定することになります。

1 ラインの幅と入力ライン数

変換するデータ(入力データ)の 1 ラインの幅は nn::y2r::SetInputLineWidth() で設定します。現在の設定は nn::y2r::GetInputLineWidth() で取得することができます。1 ラインの幅は 1024 までの 8 の倍数でなければなりません。

入力ライン数は nn::y2r::SetInputLines() で設定します。現在の設定は nn::y2r::GetInputLines() で取得することができます。

出力フォーマット

YUVtoRGB 回路で変換されたデータは出力バッファに格納されます。出力されるデータのフォーマット設定は nn::y2r::SetOutputFormat() を呼び出して行います。現在の設定は nn::y2r::GetOutputFormat() で取得することができます。出力フォーマットには以下のフォーマットを指定することができます。

表 6-30. 出力フォーマット

設定値	フォーマット
OUTPUT_RGB_32	32 bit RGB(RGBA8888)
OUTPUT_RGB_24	24 bit RGB(RGB888)
OUTPUT_RGB_16_555	16 bit RGB(RGBA5551)
OUTPUT_RGB_16_565	16 bit RGB(RGB565)

それぞれのデータ形式は以下のようになっています。

32 bit RGB(OUTPUT_RGB_32)

+0 Byte	+1 Byte	+2 Byte	+3 Byte
A [7 : 0]	B (n)	G (n)	R (n)

24 bit RGB(OUTPUT_RGB_24)

+0 Byte	+1 Byte	+2 Byte
R (n)	G (n)	B (n)

16 bit RGB(OUTPUT_RGB_16_555)

+0 Byte		+1 Byte		+2 Byte		+3 Byte	
5 bit	5 bit	5 bit	1 bit	5 bit	5 bit	5 bit	1 bit
R (n)	G (n)	B (n)	A [7]	R (n + 1)	G (n + 1)	B (n + 1)	A [7]

16 bit RGB(OUTPUT_RGB_16_565)

+0 Byte		+1 Byte	+2 Byte		+3 Byte
5 bit	6 bit	5 bit	5 bit	6 bit	5 bit
R (n)	G (n)	B (n)	R (n + 1)	G (n + 1)	B (n + 1)

アルファ成分が出力されるフォーマットの場合、nn::y2r::SetAlpha() の呼び出しで設定されたアルファ値が使用されます。設定されたアルファ値は、OUTPUT_RGB_32 のフォーマットでは 0 ～ 7 bit が、OUTPUT_RGB_16_555 のフォーマットでは 7 bit 目が使用されます。現在のアルファ値は nn::y2r::GetAlpha() の呼び出しで取得することができます。

ブロックアライメント

出力バッファに格納されるデータの並び(ブロックアライメント)は nn::y2r::SetBlockAlignment() を呼び出して行います。現在の設定は nn::y2r::GetBlockAlignment() で取得することができます。ブロックアライメントには以下のものを指定することができます。

表 6-31. ブロックアライメント

設定値	ブロックアライメント
BLOCK_LINE	水平ラインフォーマット。通常のリニアフォーマットです。出力フォーマットが 24 bit RGB または 32 bit RGB の場合、バイトオーダーの関係で、そのままでは OpenGL 標準フォーマットのテクスチャとして利用することができません。
BLOCK_8_BY_8	8x8 ブロックフォーマット。GPU のネイティブフォーマットであるブロックフォーマットと同じデータの並びです。ネイティブフォーマットのテクスチャイメージとして使用することができます。

注意: BLOCK_8_BY_8 を指定した場合は、入力イメージの高さ(縦のライン数)も 8 の倍数でなければなりません。

出力バッファ

1 フレームの出力データを受信するために必要なバッファのサイズは nn::y2r::GetOutputImageSize() で取得することができます。バッファの先頭アドレスのアライメントは 4 Byte ですが、64 Byte 未満のアライメントでは転送速度が落ちる可能性があります。**デバイスメモリ上に確保したバッファのみ指定可能です。**

変換係数

YUV フォーマットから RGB フォーマットへの変換に使用する係数を標準的な変換係数から選択します。変換係数の設定は、nn::y2r::SetStandardCoefficient() の呼び出しで行うことができます。

カメラから出力された画像を変換するための変換係数には、カメラモジュールが将来的に変更される可能性を考慮し、nn::camera::GetSuitableY2rStandardCoefficient() で取得した変換係数のタイプを選択してください。

標準的な変換係数のタイプには以下の 4 つがあります。

表 6-32. 標準的な変換係数のタイプ

設定値	変換係数のタイプ(値の範囲)
COEFFICIENT_ITU_R_BT_601	ITU-R BT.601(0 ≦ Y, U, V ≦ 255)
COEFFICIENT_ITU_R_BT_709	ITU-R BT.709(0 ≦ Y, U, V ≦ 255)
COEFFICIENT_ITU_R_BT_601_SCALING	ITU-R BT.601(16 ≦ Y ≦ 235, 16 ≦ U, V ≦ 240)
COEFFICIENT_ITU_R_BT_709_SCALING	ITU-R BT.709(16 ≦ Y ≦ 235, 16 ≦ U, V ≦ 240)

回転

フォーマット変換の際に回転を適用することができます。nn::y2r::SetRotation() で回転の角度を指定することができ、nn::y2r::GetRotaion() で現在の回転角度を取得することができます。回転なしを含めると、指定可能な回転角度は以下の 4 つです。

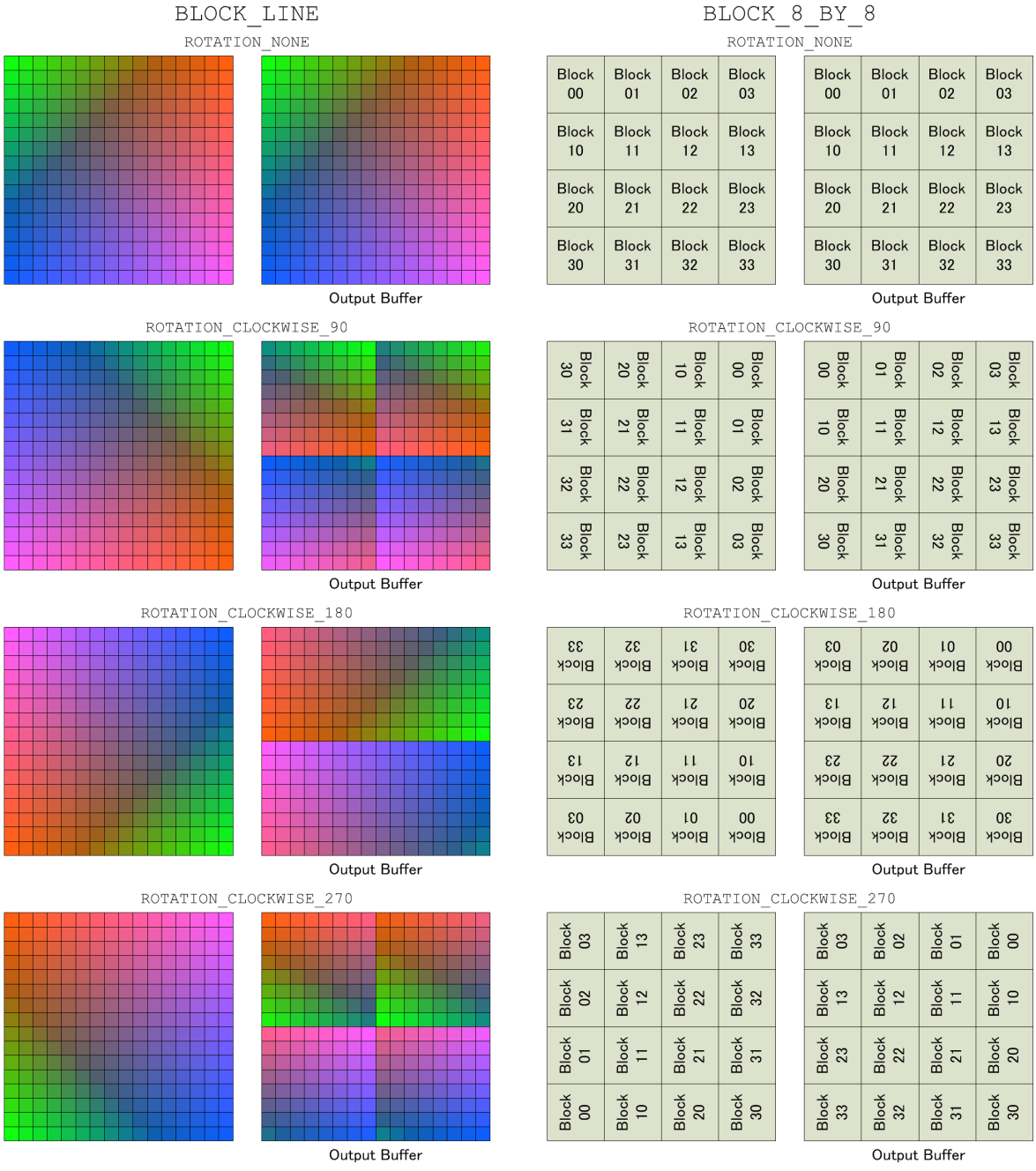
表 6-33. 回転角度

設定値	回転角度
ROTATION_NONE	回転なし
ROTATION_CLOCKWISE_90	時計回りに 90 度
ROTATION_CLOCKWISE_180	180 度
ROTATION_CLOCKWISE_270	時計回りに 270 度(反時計回りに 90 度)

回転が適用されると、変換後のデータは正しいイメージデータの並びではありません。そのため、アプリケーションは 1 フレームのデータを受信したあと、データの並びを修正しなければなりません。

> **補足:** 今後、転送時に正しいデータの並びで出力されるように修正される可能性があります。

図 6-14. ブロックアライメントと回転による出力データの並びの違い



一括設定

`nn::y2r::SetPackageParameter()` で YUVtoRGB 回路の設定を一括して変更することができます。設定の一括取得は `nn::y2r::GetPackageParameter()` で行うことができます。

6.4.6. フォーマット変換の開始

フォーマットの変換はデータの送受信と平行して行われるため、変換の開始までにデータ送受信の準備を行わなければなりません。

入力データの送信準備は、YUV 一括が `nn::y2r::SetSendingYuv()`、Y のみが `nn::y2r::SetSendingY()`、U のみが `nn::y2r::SetSendingU()`、V のみが `nn::y2r::SetSendingV()` で行われます。引数には入力データが格納されているバッファ(**デバイスメモリ上に確保したバッファのみ指定可能です**)と、転送データの総サイズと 1 ライン分の

入力データのサイズをバイト単位で指定します。総サイズが 1 ライン分のサイズの整数倍でなければならないことに注意してください。また、いずれの関数でも 1 ライン分の入力データが転送されたときに加算されるオフセット値(*transferStride*)を指定することができます。

出力データの受信準備は nn::y2r::SetReceiving() で行います。引数には出力データを格納するバッファ(**VRAM 上のバッファは指定できません。デバイスメモリ上に確保したバッファのみ指定可能です。また nn::y2r::BUFFER_ALIGNMENT (4 Byte) アライメントが必要です。**)と、受信データの総サイズと受信データの 1 回分の転送サイズをバイト単位で指定します。パフォーマンスを上げるために、転送サイズには 8 ライン分の出力データのサイズを指定することを推奨します。受信データの総サイズに指定する値は nn::y2r::GetOutputImageSize() で取得することができます。8 ライン分のデータサイズに指定する値には、nn::y2r::GetOutputFormatBytes() で取得した 1 ピクセルのバイトサイズと 1 ラインの幅とを乗算した値に 8 を乗算した値を指定します。また、1 回の転送ごとに加算されるオフセット値(*transferStride*)を指定することができます。1 ラインごとにオフセットを加えたい場合は、1 回分の転送サイズに 1 ライン分の出力データのサイズを指定してください。

データ送受信の準備を行うタイミングとしては、nn::camera::IsFinishedReceiving() もしくは nn::camera::SetReceiving() で指定したイベントのシグナル状態でキャプチャ画像の受信が終了したことを確認した直後の、入力データが確定した時点が最適です。ただし、nn::y2r::IsBusyConversion() でフォーマット変換中でないことを確認してからデータ送受信の準備を行ってください。

データ送受信の準備が整ったあとは nn::y2r::StartConversion() でフォーマット変換を開始することができます。フォーマット変換の開始と同時にデータの送受信も開始されます。キャプチャ処理中のエラーの発生などで、フォーマット変換を強制的に停止しなければならないときは nn::y2r::StopConversion() を呼び出してください。

フォーマット変換中であるかどうかは nn::y2r::IsBusyConversion() で取得することができます。入力データの送信が完了しているかどうかは、YUV 一括が nn::y2r::IsFinishedSendingYuv()、Y のみが nn::y2r::IsFinishedSendingY()、U のみが nn::y2r::IsFinishedSendingU()、V のみが nn::y2r::IsFinishedSendingV() で取得することができます。出力データの受信が完了しているかどうかは、nn::y2r::IsFinishedReceiving() で取得することができます。

1 フレーム分のフォーマット変換およびデータの送受信が終了しているかどうかは、上記の関数を組み合わせて判断することもできますが、データ転送完了時に発生する割り込みの通知を nn::y2r::GetTransferEndEvent() で取得したイベントクラス(自動リセットイベント)で受け取ることで判断することもできます。イベントクラスを取得する前に nn::y2r::SetTransferEndInterrupt() による割り込み通知の許可(デフォルトは許可された状態です)が必要です。現在、通知が許可されているかどうかは nn::y2r::GetTransferEndInterrupt() で取得することができます。

発生することはまれですが、カメラと YUVtoRGB 回路を同時に使用しているときに、カメラのデータ転送中にバッファエラーが発生すると、その復帰処理で nn::camera::SetReceiving() の処理と、YUVtoRGB 回路のデータ転送処理の特定のタイミングと重なったときに nn::y2r::GetTransferEndEvent() で取得した転送完了イベントがシグナル状態にならず、nn::y2r::IsBusyConversion() も true を返さなくなるハードウェアの不具合があります。そのため転送完了イベントのシグナル状態を待つときには、必ずタイムアウト時間を指定するようにしてください。

タイムアウト時間には変換にかかる時間よりも長い時間を指定します。変換にかかる時間は入力画像のサイズと出力フォーマットに依存し、VGA 画像を 16 bit RGB で出力する場合はおよそ 13 msec、VGA 画像を 24 bit RGB で出力する場合はおよそ 16 msec です。

なお、この不具合が発生したときは nn::y2r::StopConversion() を呼び出して、変換を強制終了してください。YUVtoRGB 回路が再度使用できるようになります。

不具合の発生確率はカメラのバッファエラーの頻度に比例しますので、バッファエラーが起こりづらい条件でカメラを使用してください。バッファエラーが頻発する場合は、カメラのフレームレートを下げるなどの対応をしてください。また、nn::camera::SetReceiving() を呼び出すスレッドの優先度は高めに設定してください。

6.4.7. シャッター音の再生

スピーカー音量が 0 でもシャッター音を強制的に再生し、カメラランプ(LED)の一時消灯を行います。

> **補足:** SNAKE および FTR にはカメラランプが搭載されていません。

コード 6-18. シャッター音の再生

```
nn::Result nn::camera::PlayShutterSound(nn::camera::ShutterSoundType type);
```

type にはシャッター音の種類を以下から選択して指定します。

表 6-34. シャッター音の種類

設定値	シャッター音の種類
SHUTTER_SOUND_TYPE_NORMAL	通常のシャッター音
SHUTTER_SOUND_TYPE_MOVIE	動画撮影を開始するときに鳴らす音
SHUTTER_SOUND_TYPE_MOVIE_END	動画撮影を終了するときに鳴らす音

6.4.8. キャプチャの終了

キャプチャの終了処理は以下の手順で行ってください。下記の手順で終了処理を行わなければ、HOME メニュー上でサウンドのノイズが発生する可能性があります。

1. nn::y2r::StopConversion() を呼び出してフォーマット変換を停止します。
2. nn::camera::StopCapture() を呼び出してキャプチャを停止します。
3. nn::camera::Activate(SELECT_NONE) で全カメラをスタンバイ状態にします。
4. nn::camera::Finalize() と nn::y2r::Finalize() を呼び出して CAMERA ライブラリと Y2R ライブラリを終了させます。

6.4.9. スリープへの対応

初期化時を含め、蓋が閉じられた状態で CAMERA ライブラリや Y2R ライブラリの関数を呼び出すと、蓋が閉じられていることを示す nn::camera::ResultIsSleeping(nn::y2r::ResultIsSleeping)が返されます。この返り値は蓋が閉じられていれば、スリープ状態でなくても返されることに注意してください。

カメラはスリープとは関係なく、本体の蓋が閉じられることにより動作が停止するようになっています。キャプチャ中に蓋が閉じられた場合、蓋が開けられたタイミングでキャプチャが再開されますが、蓋が閉じられたときに、ライブラリ内部で nn::camera::Activate(SELECT_NONE) と同等の処理が行われるため、キャプチャを再開したときに Activate() によるカメラ起動直後の画像が不安定な期間が存在することに注意してください。また、蓋閉じのタイミングによっては、IsBusy() が常に true を返す状態になる可能性があります。この状態は蓋開け時に解除されますが、蓋閉じからスリープに移行する処理を実装する際は、IsBusy() によるポーリングをしないように注意してください。

Y2R ライブラリによる RGB 変換の最中にスリープに移行すると、変換は強制的に終了されますが、スリープから復帰したときに再開されません。そのため、スリープに対応する場合、変換中(IsBusyConversion() が true を返すとき、あるいは GetTransferEndEvent() で取得したイベントを待っている最中)はスリープに移行せず、変換の完了を確認してからスリープに移行するように実装してください。特に、GetTransferEndEvent() で取得したイベントを待っている最中にスリープに移行すると、スリープから復帰したあとはイベントがシグナル状態にならないことに注意してください。

6.4.10. 簡易カメラ機能との競合

アプリケーションがカメラを利用しているときに HOME メニューを表示した場合、L ボタンとは R ボタンの同時押しで起動する簡易カメラ機能はカメラが使用中であるというダイアログを表示して起動をキャンセルします。

カメラが使用中であるかどうかは CAMERA ライブラリと Y2R ライブラリのいずれかが初期化されているかどうかで判断されます。そのため、カメラを使用するアプリケーションが起動時にライブラリの初期化を行っている場合、カメラを使用していない場面で HOME メニューを表示したときでもカメラが使用中であると判断されてしまいます。また、ムービーの再生など、YUV 画像を RGB 画像に変換するために Y2R ライブラリのみを使用している場合でも、カメラを使用していると見なされて簡易カメラ機能の起動がキャンセルされます。

なるべく、カメラや YUVtoRGB 回路を使用する場面の前後で初期化と終了処理を行い、使用していないときは CAMERA ライブラリや Y2R ライブラリが初期化したままにならないようにしてください。

7. ファイルシステム

この章では、各メディアに対するアクセスで使用する FS ライブラリと、メディアごとの注意点などについて説明します。

7.1. FS ライブラリ

3DS に搭載されている記憶装置（3DS カード、SD カード）に対するアクセスは、必ず FS ライブラリを介して行わなければなりません。

7.1.1. 初期化

FS ライブラリの初期化は `nn::fs::Initialize()` を呼び出して行います。すでに初期化されているときに、再度呼び出しても問題はありません。

ファイル、ディレクトリに対する関数の呼び出しは、FS ライブラリの初期化後でなければエラーとなります。

7.1.2. 終了

FS ライブラリの使用を終了する場合は、開いていたファイルやディレクトリを閉じ、すべてのアーカイブのマウントを解除してください。

アプリケーションの終了時やスリープ状態への移行時にも FS ライブラリの使用を終了もしくは中断しなければなりません、詳細については「5.3.1.1. 終了処理の注意事項」、「5.3.3.4. スリープ中に禁止されている処理」を参照してください。

7.1.3. パスの指定について

パス名（ファイル名やディレクトリ名）の指定は**絶対パス**でなければなりません。パスの区切りには "/"（スラッシュ）を用います。

FS ライブラリの各関数でパス名を指定する際の文字列には、ワイド文字列とマルチバイト文字列（ASCII 文字のみ）を使用することができます。ただし、マルチバイト文字列で指定すると、ワイド文字列への変換でライブラリがスタック上に大きなバッファを確保するため、スタックの大きさに注意しなければなりません。そのため、特別な理由がない限りはワイド文字列で指定するようにしてください。

7.1.4. ファイルへのアクセス

メディア上のファイルへのアクセスに使用するクラスは、以下の 3 種類から目的に応じて選択します。

表 7-1. ファイルへのアクセスに使用できるクラス

クラス	説明
`nn::fs::FileInputStream`	読み込み専用でファイルを開きます。
`nn::fs::FileOutputStream`	書き込み専用でファイルを開きます。指定したファイルが存在していなかった場合、初期化時の指定でファイルを新規に作成して開くことができます。
`nn::fs::FileStream`	アクセスモードの指定によって、読み込みと書き込みの両用でファイルを開くことができます。

> **注意:** ファイルアクセスのクラスには、Initialize() と TryInitialize() のように、関数名の先頭に Try の付加された関数とそうでない関数が存在します。行われる処理に違いはありませんが、アプリケーションで使用する関数には、Try の付加されている関数を使用してください。
>
> 基本的に、FS ライブラリの関数は実行が完了するまで処理を戻しません。また、ファイルへのアクセスはリクエストの到着順に処理されるため、優先度の高いスレッドからのファイルアクセスであっても、先にリクエストされたファイルアクセスが完了するまで処理されません。

7.1.4.1. nn::fs::FileInputStream クラス

ファイルからデータを読み込むには、このクラスを使用してファイルを開いてください。読み込むファイルの指定は、コンストラクタの引数や、Initialize() または TryInitialize() の引数で行います。引数ありのコンストラクタや Initialize() でファイルを開くときに、存在しないファイルを指定した場合はアプリケーションが停止します。TryInitialize() でファイルを開いたときに限り、返り値でエラーを判定することができます。正常に開くことのできなかったインスタンスでファイルにアクセスした場合や、すでにファイルを開いているインスタンスで新たにファイルを開こうとした場合はアプリケーションが停止します。

ファイルのサイズは GetSize() または TryGetSize() で取得することができます。読み込みに使用するバッファのサイズを決定する際などに利用することができます。

ファイルの読み込みは Read() または TryRead() で行います。引数として、ファイルの内容をコピーするバッファとバッファのサイズを渡してください。返り値には実際にバッファにコピーされたバイト数が返され、ファイルの終端に達していた場合は 0 を返します。バッファはなるべく 4 Byte アライメントで確保してください。デバイスやメモリ領域によって変化しますが、4 Byte アライメントではないバッファかつ、カレント位置が 4 Byte 単位で変化しない読み込みの速度はかなり遅くなります。

ファイルの読み込み開始位置(カレント位置)は GetPosition() または TryGetPosition() で取得、SetPosition() または TrySetPosition() で設定することができます。カレント位置はファイルの先頭からのバイト数です。また、Seek() または TrySeek() では基準となる位置(ベース位置)と、そこからのオフセットで設定することもできます。設定されたカレント位置がファイルの先頭から 4 の倍数ではない場合、デバイスやメモリ領域によって変化しますが、読み込み速度はかなり遅くなりますので注意してください。カレント位置がファイルの先頭より前に設定された場合や、ファイルの終端以降に設定された場合は次のアクセス時にアプリケーションが停止します。

表 7-2. ベース位置の指定

設定値	説明
POSITION_BASE_BEGIN	ファイルの先頭を基準にカレント位置を設定します。
POSITION_BASE_CURRENT	ファイルのカレント位置を基準にカレント位置を設定します。
POSITION_BASE_END	ファイルの終端を基準にカレント位置を設定します。

ファイルへのアクセスを終了するときは Finalize() を呼び出し、ファイルを閉じてください。

以下のコード例では、TryInitialize() で正常にファイルを開くことができたかを確認してから読み込みを行っています。

コード 7-1. ファイル読み込みのコード例

```
nn::fs::FileInputStream fis;
nn::Result result = fis.TryInitialize(L"rom:/test.txt");
if (result.IsSuccess())
{
    s64 fileSize;
    result = fis.TryGetSize(&fileSize);
    NN_LOG("FileSize=%lld\n", fileSize);
    buf = heap.Allocate(fileSize);
    s32 ret;
    result = fis.TryRead(&ret, buf, fileSize);
    ...
    heap.Free(buf);
}
fis.Finalize();
```

7.1.4.2. nn::fs::FileOutputStream クラス

ファイルにデータを書き込むには、このクラスを使用してファイルを開いてください。書き込み先のファイルの指定は、コンストラクタの引数や、Initialize() または TryInitialize() の引数で行います。TryInitialize() でファイルを開いたときに限り、返り値でエラーを判定することができます。ファイルの指定時に *createIfNotExist* 引数に true を渡すと、指定のファイルが存在していない場合でもサイズが 0 のファイルを新規に作成して開くことができます。引数ありのコンストラクタや Initialize() でファイルを開くときに存在しないファイルを指定した場合、*createIfNotExist* 引数に false を渡しているとアプリケーションが停止します。正常に開くことのできなかったインスタンスでファイルにアクセスした場合や、すでにファイルを開いているインスタンスで新たにファイルを開こうとした場合はアプリケーションが停止します。

ファイルに書き込みを行う前に SetSize() または TrySetSize() でファイルサイズを指定することができます。書き込みの途中でファイルサイズを元のサイズより小さく変更した場合、ファイルの書き込み位置が調整されます。GetSize() または TryGetSize() は現在のファイルサイズを返します。

> **補足：** TryRead() は、TryInitialize() で作成されたサイズが 0 のファイルを正常に読み込みます。そのため、TryInitialize() の *createIfNotExist* 引数に true を渡し、あとから TrySetSize() でファイルサイズを設定する場合は注意が必要です。
>
> TryInitialize() と TrySetSize() の間にゲームカードが抜かれるなどした場合、次回読み込み時に TryRead() を呼び出すと、サイズが 0 のファイルを読み込んだ状態になります。
>
> 固定長のファイルを扱うため、TryRead() 前に TryGetSize() でサイズを確認しないアプリケーションは、*createIfNotExist* 引数によるファイル作成ではなく、ファイルサイズが指定可能な TryCreateFile() でファイル作成することを推奨します。

ファイルへの書き込みは Write() または TryWrite() の呼び出しで行います。引数として、ファイルに書き込む内容が格納されているバッファの先頭アドレスと書き込む内容のバイト数を渡してください。返り値には実際にファイルに書き込まれたバイト数が返されます。ファイルの終端を越えるような書き込みを行った場合、可能ならばファイルを拡張して書き込みます。バッファはなるべく 4 Byte アライメントで確保してください。デバイスやメモリ領域によって変化しますが、4 Byte アライメントではないバッファからの書き込み速度はかなり遅くなります。また、ファイルの書き込みと同時に、ファイルキャッシュの内容をメディアに書き込む(フラッシュする)かどうかを、引数 *flush* で指定することができます。メディアの磨耗や書き込み中のカード抜けによるデータ破壊を防ぐためにも、*false* を指定しておき、ファイルを閉じる前に Flush() または TryFlush() を呼び出してフラッシュを行うことを推奨します。ただし、ファイルの書き込みと同時にフラッシュを行っていない場合は、ファイルを閉じるまでに必ずフラッシュを行わなければならないことに注意してください。

ファイルの書き込み開始位置(カレント位置)は GetPosition() または TryGetPosition() で取得、

SetPosition() または TrySetPosition() で設定することができます。カレント位置はファイルの先頭からのバイト数です。また、Seek() または TrySeek() ではベース位置と、そこからのオフセットで設定することもできます。設定されたカレント位置がファイルの先頭から 4 の倍数ではない場合、デバイスやメモリ領域によって変化しますが、書き込み速度はかなり遅くなりますので注意してください。カレント位置がファイルの先頭より前に設定された場合や、ファイルの終端以降に設定された場合は次のアクセス時にアプリケーションが停止します。

ファイルへのアクセスを終了するときは Finalize() を呼び出し、ファイルを閉じてください。

以下のコード例では、TryInitialize() で正常にファイルを開くことができたかを確認してから書き込みを行っています。

コード 7-2. ファイル書き込みのコード例

```
nn::fs::FileOutputStream fos;
nn::Result result = fos.TryInitialize(L"sdmc:/test.txt", true);
if (result.IsSuccess())
{
    s32 ret;
    result = fos.TryWrite(&ret, buf, sizeof(buf));
}
fos.Finalize();
```

7.1.4.3. nn::fs::FileStream クラス

このクラスではファイルを読み書き両用で開くことができます。初期化時の引数指定を除いて、各メンバ関数の動作は前 2 つのクラスの動作と違いはありません。

コンストラクタおよび Initialize()、TryInitialize() の引数で、アクセスするファイルの指定とアクセスモードの指定を行います。アクセスモードの指定は以下のフラグの組み合わせで行います。

表 7-3. アクセスモードの指定フラグ

フラグ	説明
OPEN_MODE_READ	読み込み可能状態でファイルを開きます。
OPEN_MODE_WRITE	書き込み可能状態でファイルを開きます。読み込みも可能です。
OPEN_MODE_CREATE	OPEN_MODE_WRITE と組み合わせることで、初期化時に指定されたファイルが存在しない場合にファイルを新規作成します。

7.1.5. ディレクトリへのアクセス

nn::fs::Directory クラスを用いることで、ディレクトリとそのエントリ（ディレクトリとファイル）の情報を取得することができます。

7.1.5.1. nn::fs::Directory クラス

このクラスではディレクトリを開き、ディレクトリに格納されているエントリの一覧を取得することができます。ディレクトリの指定は、コンストラクタの引数や、Initialize() または TryInitialize() の引数で行います。引数ありのコンストラクタや Initialize() でディレクトリを開くときに、存在しないディレクトリを指定した場合はアプリケーションが停止します。TryInitialize() でディレクトリを開いたときに限り、返り値でエラーを判定することができます。正常に開くことのできなかったインスタンスでディレクトリにアクセスした場合や、すでにディレクトリを開いているインスタンスで新たにディレクトリを開こうとした場合はアプリケーションが停止します。

メディアのルートディレクトリを指定する場合、SD カードならば "sdmc:/" のように、末尾に "/"(スラッシュ)が必要です。ルートディレクトリ以外の指定ではスラッシュは不要ですが、付けていても開くことができます。

エントリ情報の取得は Read() の呼び出しで行います。引数として、エントリ情報(nn::fs::DirectoryEntry 構造体)の配列と配列の要素数を渡してください。返り値には実際に配列に格納されたエントリの数が返されます。すべてのエントリを取得し終わると、以降 Read() は 0 を返し続けます。

ディレクトリへのアクセスを終了するときは Finalize() を呼び出し、ディレクトリを閉じてください。

以下のコード例では、TryInitialize() で正常にディレクトリを開くことができたかを確認してからエントリの取得を行っています。

コード 7-3. ディレクトリアクセスのコード例

```
nn::fs::Directory dir;
nn::fs::DirectoryEntry entry[ENTRY_MAX];
nn::Result result = dir.TryInitialize(L"sdmc:/TestDirectory");
if (result.IsSuccess())
{
    s32 readCount;
    while (true)
    {
        result = dir.TryRead(&readCount, entry, ENTRY_MAX);
        if (readCount == 0) break;
        ...
    }
}
dir.Finalize();
```

7.1.6. ファイルとディレクトリの操作

ファイル名とディレクトリ名の変更、ファイルとディレクトリの作成と削除を行うことができます。

7.1.6.1. ファイルの作成

nn::fs::CreateFile() または nn::fs::TryCreateFile() の呼び出しで、指定されたサイズのファイルを作成することができます。nn::fs::CreateFile() の呼び出しではエラーが発生するとアプリケーションが停止しますが、nn::fs::TryCreateFile() の呼び出しでは返り値でエラーを判定することができます。

すでに存在しているファイルと同じ名前のファイルを作成しようとした場合はエラーとなります。同一ファイル名のファイル作成以外でエラーが発生したときには、不要なファイルが作成されている可能性がありますので、そのファイルの削除を行ってください。その際、不要なファイルが作成されていなかった場合には nn::fs::ResultNotFound が返されます。

7.1.6.2. ファイル名の変更

nn::fs::RenameFile() または nn::fs::TryRenameFile() の呼び出しでファイル名を変更することができます。nn::fs::RenameFile() の呼び出しではエラーが発生するとアプリケーションが停止しますが、nn::fs::TryRenameFile() の呼び出しでは返り値でエラーを判定することができます。

開かれた状態のファイルに対して呼び出した場合や、すでに存在しているファイルと同じ名前(自身を含む)に変更しようとした場合はエラーとなります。

7.1.6.3. ファイルの削除

nn::fs::DeleteFile() または nn::fs::TryDeleteFile() の呼び出しでファイルを削除することができます。
nn::fs::DeleteFile() の呼び出しではエラーが発生するとアプリケーションが停止しますが、
nn::fs::TryDeleteFile() の呼び出しでは返り値でエラーを判定することができます。

開かれた状態のファイルに対して呼び出した場合はエラーとなります。

7.1.6.4. ディレクトリの作成

nn::fs::CreateDirectory() または nn::fs::TryCreateDirectory() の呼び出しでディレクトリを作成することができます。nn::fs::CreateDirectory() の呼び出しではエラーが発生するとアプリケーションが停止しますが、nn::fs::TryCreateDirectory() の呼び出しでは返り値でエラーを判定することができます。

存在していないディレクトリの配下にディレクトリを作成しようとした場合や、すでに存在しているディレクトリと同じ名前のディレクトリを作成しようとした場合はエラーとなります。

7.1.6.5. ディレクトリ名の変更

nn::fs::RenameDirectory() または nn::fs::TryRenameDirectory() の呼び出しでディレクトリ名を変更することができます。nn::fs::RenameDirectory() の呼び出しではエラーが発生するとアプリケーションが停止しますが、nn::fs::TryRenameDirectory() の呼び出しでは返り値でエラーを判定することができます。

開かれた状態のディレクトリに対して呼び出した場合や、すでに存在しているディレクトリと同じ名前(自身を含む)に変更しようとした場合はエラーとなります。

7.1.6.6. ディレクトリの削除

nn::fs::DeleteDirectory() または nn::fs::TryDeleteDirectory() の呼び出しでディレクトリを削除することができます。削除するディレクトリ内にエントリが残されていると削除することができません。
nn::fs::DeleteDirectory() の呼び出しではエラーが発生するとアプリケーションが停止しますが、
nn::fs::TryDeleteDirectory() の呼び出しでは返り値でエラーを判定することができます。

開かれた状態のディレクトリに対して呼び出した場合はエラーとなります。

nn::fs::TryDeleteDirectoryRecursively() は、指定されたディレクトリにエントリが残されていても、エントリを再帰的に削除することで指定されたディレクトリを完全に削除しようとします。削除の途中でエラーが発生した場合、この関数はそのエラーを返します。

7.1.7. SD カードの状態チェック

SD カードの挿抜状態の確認や挿入・排出を通知するイベントを登録・解除する関数、書き込み可能かを確認する関数が用意されています。

コード 7-4. SD カードの挿抜状態の確認、挿入・排出の通知、書き込み可能の確認

```
bool nn::fs::IsSdmcInserted();
void RegisterSdmcInsertedEvent(nn::os::LightEvent* p);
void UnregisterSdmcInsertedEvent();
void RegisterSdmcEjectedEvent(nn::os::LightEvent* p);
void UnregisterSdmcEjectedEvent();
bool nn::fs::IsSdmcWritable();
nn::Result nn::fs::GetSdmcSize(s64* pTotal, s64* pFree);
```

nn::fs::IsSdmcInserted() を呼び出すことで、SD カードの挿抜状態を確認することができます。壊れた SD カードや空の SD カードアダプタのような SD カード以外のものでも、SD カードスロットに機器が挿入されていれば true を返しま

す。この関数呼び出しによる挿抜状態の確認処理は負荷が高いため、SD カードの挿抜を待つ場合は以下の関数を利用し、イベントクラスで待機するようにしてください。

`nn::fs::RegisterSdmcInsertedEvent()` と `nn::fs::UnregisterSdmcInsertedEvent()` は、SD カード挿入の通知を受け取る `nn::os::LightEvent` クラスの登録と登録の解除を行います。

`nn::fs::RegisterSdmcEjectedEvent()` と `nn::fs::UnregisterSdmcEjectedEvent()` は、SD カード排出の通知を受け取る `nn::os::LightEvent` クラスの登録と登録の解除を行います。

`nn::fs::IsSdmcWritable()` は、SD カードが挿入された状態で、かつ書き込みが可能ならば `true` を返します。

`nn::fs::GetSdmcSize()` は、SD カードの総容量(*pTotal*)と空き容量(*pFree*)を返します。

7.1.8. レイテンシエミュレーション

アクセス速度の変化に対するデバッグ用に、いつの間に通信などのバックグラウンド処理で行われるファイルシステムへのアクセスと競合した場合に発生する、アプリケーションからのファイルアクセスの遅延を擬似的に再現する機能を以下の関数で有効にすることができます。

コード 7-5. レイテンシエミュレーションの初期化

```
void nn::fs::InitializeLatencyEmulation(void);
```

レイテンシエミュレーションの有効/無効は、Config ツール「Debug Setting」の「Debug Mode」で切り替えます。この項目が「enable」に設定され、さらにこの関数によって初期化されている場合にのみ、アプリケーションの動作時にファイルアクセスが「FS Latency Emulation」で指定された時間(ミリ秒単位)遅延するエミュレーション機能が有効になります。

7.1.9. アクセス優先度の設定

ファイルシステムのアクセス優先度がサポートされています。これにより、複数のスレッドから複数のファイルアクセスを行った場合に実行順序が調整され、設定したアクセス優先度の高いものほど優先的に処理されるようになります。

アクセス優先度には、ストリーミング再生のようにリアルタイム性を要求する(一定の周期で一定の処理量を必要とする)ファイルアクセスに適した設定が用意されています。この設定では、ファイルアクセスにかかる遅延時間を最小にすることができます。

7.1.9.1. アクセス優先度の種類

設定可能なアクセス優先度は、優先度の高い順に以下のものが用意されています。

表 7-4. アクセス優先度の種類

種類	定義	説明
リアルタイム優先度	`PRIORITY_APP_REALTIME`	ストリーミングデータの読み込みなど、読み込みが遅れるとユーザーエクスペリエンスに影響のあるファイルアクセスをするために使用される、特殊な優先度です。 ただし、使用する際には、いくつかの制限があります。
通常優先度	`PRIORITY_APP_NORMAL`	モデルデータやシーンデータの読み込みや、各種セーブデータへのアクセスなど、一般的なファイルアクセスをするために使用される優先度です。

低優先度	PRIORITY_APP_LOW	通常優先度より低く、オートセーブなど、ファイルアクセスが比較的空いているときに実行されればよいファイルアクセスをするために使用される優先度です。

アクセス優先度の指定に使用する定義は、nn::fs::PriorityForApplication 列挙子に定義されています。

> **補足:** リアルタイム優先度の制限については、CTR-SDK の関数リファレンスを参照してください。

7.1.9.2. アクセス優先度の設定対象

アクセス優先度の設定対象となる範囲には以下のものがあります。

表 7-5. アクセス優先度の設定対象と設定に使用する関数

設定対象	設定に使用する関数
ファイルシステム全体	nn::fs::SetPriority()
アーカイブ	nn::fs::SetArchivePriority()
ファイルストリーム、ディレクトリ	各クラスの TrySetPriority() または SetPriority()

設定対象の「ファイルシステム全体」とは、そのアプリケーションからのアクセス全般と、セーブデータのフォーマットのようにアーカイブ名を指定せずに行うアーカイブへのアクセスが対象です。なお、アクセス優先度の設定を明示的に行っていない場合は通常優先度が適用されます。

「アーカイブ」は、マウントされているアーカイブに対して、アーカイブ名を指定して行うアクセスが対象です。アクセス優先度は、アーカイブごとに異なる設定が可能です。アクセス優先度の設定を明示的に行わなければ、マウント時点のファイルシステム全体の設定が適用されます。マウント後にシステム全体の設定を変更しても、アーカイブの設定には影響しません。同様に、アーカイブの設定を変更しても、ファイルシステム全体の設定には影響しません。

「ファイルストリーム、ディレクトリ」は、ファイルストリームのオブジェクト(nn::fs::FileStream など)やディレクトリのオブジェクト(nn::fs::Directory)を介して行うアクセスが対象です。アクセス優先度は、同じファイルやディレクトリであっても、オブジェクトごとに異なる設定が可能です。アクセス優先度の設定を明示的に行わなければ、オブジェクト作成時点のアーカイブの設定が適用されます。オブジェクト作成後にアーカイブの設定を変更しても、オブジェクトごとの設定には影響しません。同様に、オブジェクトごとの設定を変更しても、アーカイブの設定には影響しません。

7.1.9.3. 注意点

アクセス優先度はファイルシステム内での優先度を変更するものです。ファイルアクセス関数を呼び出すスレッドの優先度が高くなければ、リアルタイム優先度であっても、ファイルへのアクセスがすぐに行われません。そのため、処理の目的(ファイルアクセスの優先度)に合わせて、ファイルアクセス関数を呼び出すスレッドの優先度を高くする必要があります。

また、要求されたファイルアクセスが完了する順序は保証されていません。アクセス優先度の高いファイルアクセスよりも、あとから要求されたアクセス優先度の低いファイルアクセスの方が先に完了する可能性があります。複数のファイルに平行してアクセスする場合は、ファイルアクセスの終了順序に依存するような実装にはしないでください。

ファイルアクセスの性能設計を実測値をもとに行わないでください。アクセス時間の見積もりについては、CTR-SDK の関数リファレンスを参照してください。

7.1.10. ファイル_ディレクトリの同時オープン数の制限

ファイルシステムを用いて同時にオープンできるファイルおよびディレクトリの数には限りがあります。

安全に動作させるために、アプリケーションは同時に開くファイルとディレクトリの数を下記の制限以内に抑えてください。

- SD カードを直接参照するアーカイブと拡張セーブデータアーカイブのファイルは合わせて 4 個まで
- セーブデータアーカイブのファイルは 10 個まで(ほかのアプリケーションのセーブデータを含む)
- 全部合わせて 10 個まで

> **補足:** ROM アーカイブは上記の制限に含まれません。MountRom() での指定に準じます。

7.2. ROM アーカイブ

ROM アーカイブは、ビルド時に作成される ROMFS を参照するための読み取り専用のアーカイブです。ROM アーカイブはアプリケーションで明示的にマウントしなければなりません。アプリケーションがカードアプリ、ダウンロードアプリのいずれであっても、マウントする手順や引数に違いはありません。

ROM アーカイブを使用する際は、アプリケーションで用意した作業メモリが必要となります。必要となる作業メモリのサイズは同時に開くことのできるファイルとディレクトリ数によって変化し、nn::fs::GetRomRequiredMemorySize() を呼び出すことで必要とされるメモリサイズを取得することができます。負の値が返されたときは、ROM アーカイブをマウントすることができません。アプリケーションは必要とされるサイズ以上のメモリ領域を nn::fs::WORKING_MEMORY_ALIGNMENT (4 Byte) アライメントで確保してから、nn::fs::MountRom() を呼び出して ROM アーカイブをマウントしてください。

コード 7-6. ROM アーカイブのマウント

```
s32 nn::fs::GetRomRequiredMemorySize(size_t maxFile, size_t maxDirectory,
                                     bool useCache = true);
nn::Result nn::fs::MountRom(const char* archiveName, size_t maxFile,
                            size_t maxDirectory, void* workingMemory,
                            size_t workingMemorySize, bool useCache = true);
```

maxFile と *maxDirectory* には、同時に開くことができるファイルの数とディレクトリの数をそれぞれ指定します。同時に開くことのできる数はファイル名の長さなどで変化することはなく、作業メモリのサイズにのみ依存しています。

useCache に true を指定するとメタデータをキャッシュし、ファイルを開くときやディレクトリの走査にかかる時間を短縮することができますが、必要となる作業メモリは大きくなります。

archiveName にはマウント先のアーカイブ名を指定します。この引数のないオーバーロードを呼び出したときは、アーカイブ名 "rom:" にマウントされます。

workingMemory と *workingMemorySize* には、作業メモリとそのサイズを渡します。

> **補足:** これらの関数のエラーハンドリングは不要です。関数内でエラーが発生するとエラー画面が表示され、アプリケーションにエラーは返されません。

7.2.1. アーカイブ名の指定について

アーカイブ名に使用することのできる文字はアーカイブ名の区切りである ":" を除いた一部の記号と半角の英数字で、大文字と小文字は区別されます。1 文字以上、区切り文字の ":" を含めて 8 文字以内で指定しなければなりません。

アーカイブ名の先頭には "$" を使用しないでください。アーカイブ名やファイル名、ディレクトリ名に使用できない文字や名前については関数リファレンスを参照してください。

7.3. セーブデータ

アプリケーション固有のセーブデータ領域は、カードアプリならばバックアップメモリに、ダウンロードアプリならば SD カード内のアーカイブファイルに存在します。セーブデータ領域はアーカイブとして FS ライブラリでアクセスすることができ、どのメディア(3DS カード、SD カード)にアプリケーションが存在していても、アクセスに使用する関数や引数に違いはありません。

コード 7-7. セーブデータ領域のマウント、フォーマット、コミット

```
nn::Result nn::fs::MountSaveData(const char* archiveName = "data:");
nn::Result nn::fs::MountSaveData(const char* archiveName, bit32 uniqueId);
nn::Result nn::fs::MountDemoSaveData(const char* archiveName, bit32 uniqueId,
                                     bit8 demoIndex);
nn::Result nn::fs::FormatSaveData(size_t maxFiles, size_t maxDirectories,
                                  bool isDuplicateAll = false);
nn::Result nn::fs::CommitSaveData(const char* archiveName = "data:");
```

nn::fs::MountSaveData() でセーブデータ領域を *archiveName* で指定したアーカイブ名でマウントします。引数に *uniqueId* を持つオーバーロードを呼び出すことで、ほかのアプリケーションのセーブデータをマウントすることができます。nn::fs::MountDemoSaveData では、体験版のセーブデータ領域をマウントすることができます。*demoIndex* には、体験版のインデックスを指定します。

> **補足:** セーブデータ領域のマウント時に *uniqueId* で指定する値は、それぞれのアプリケーションの RSF ファイルで指定されたユニーク ID です。ほかのアプリケーションのユニーク ID を指定する場合は、マウントする側のアプリケーションの RSF ファイルに、アクセスしたいアプリケーションのユニーク ID をあらかじめ指定する必要があります。

セーブデータ領域のマウントで nn::fs::ResultNotFormatted、nn::fs::ResultBadFormat、nn::fs::ResultVerificationFailed に属するエラーが返された場合は、nn::fs::FormatSaveData() でセーブデータ領域のフォーマットを行ったあとで再度マウントしてください。フォーマットの際には、ファイルとディレクトリの最大数を *maxFiles* と *maxDirectories* で指定します。指定可能な値に制限はありません。アーカイブ名の指定については「7.2.1. アーカイブ名の指定について」を参照してください。

> **注意:** 体験版を含むダウンロードアプリからカードアプリのセーブデータ領域をマウントする際は、3DS カードが挿入されていない場合と、挿入されている 3DS カードのセーブデータ領域がフォーマットされていない場合とが、同じエラー(nn::fs::ResultNotFound)を返すことに注意してください。
>
> 一度カードアプリを起動してセーブデータ領域のフォーマットを行わなければ、この状況から抜け出すことができませんので、「(メディア名)のセーブデータが見つかりません。まだ一度も起動していない場合はゲームを起動し、セーブデータを作成してからもう一度試してください。」のように、エラーの原因がどちらであっても違和感のないメッセージを表示することで対応してください。

> **補足:** メディアによっては返さないエラーがあります。
> 例えば、現状ではマウント時に CARD1 は nn::fs::ResultBadFormat を返すことがありますが、CARD2 やダウンロードアプリでは nn::fs::ResultBadFormat を返すことはありません。
> 上述のような場合であっても、エラーハンドリングにおいて将来の変更に対応できるようメディアに依存しない処理の記述を推奨します。

セーブデータ領域全体の二重化をライブラリでサポートしています。フォーマット時に *isDuplicateAll* に true を指定した場合、セーブデータ領域の全域に対して、ライブラリによる二重化が行われます。二重化を有効にしている状態でファイルの書き込みを行った場合は、セーブデータ領域のマウントを解除するまでに nn::fs::CommitSaveData() を呼び出さなければ更新が有効になりません。また、更新が有効になる前に電源が切断された場合などには、次回の起動時に元の古いセーブデータがマウントされます。

> **注意:** セーブデータがファイル間に依存関係のある複数のファイルで構成されている場合、それらのファイル間で矛盾がない状態で nn::fs::CommitSaveData() を呼び出す必要があります。

二重化を有効にした場合、セーブデータの保存に使用することのできる領域はバックアップ領域の物理的な容量の半分です。さらにそこからファイルシステムの管理領域が差し引かれます。実際に使用可能なバックアップ領域の容量は、関数リファレンスの「セーブデータ容量計算シート」で計算することができます。現在、ファイルを保存すると 512 Byte 単位で領域を使用します。

マウントされたセーブデータ領域はアーカイブとして扱うことができます。アーカイブ内には、自由にファイルを作成することができます。ファイル名とディレクトリ名は 16 文字まで、パスの最大長は 253 文字です。ファイル名とディレクトリ名に使用することのできる文字は、半角の英数字と一部の記号のみで、パスの区切りには "/(スラッシュ)" を用います。作成可能なファイルのサイズと一度に書き込むことのできるサイズには、セーブデータの保存に使用可能な領域の容量以外による制限はありません。

アーカイブの空き容量は nn::fs::GetArchiveFreeBytes() で取得することができます。

コード 7-8. アーカイブの空き容量の取得

```
nn::Result nn::fs::GetArchiveFreeBytes(s64* pOut, const char* archiveName);
```

pOut には空き容量を受け取る s64 型の変数へのポインタを、*archiveName* にはマウント時に指定したアーカイブ名をそれぞれ指定します。

アーカイブ内のファイルは、ハッシュ値による改竄チェックで保護されています。アクセスしたデータのハッシュ値が合わないときは、改竄されていると判断されてエラーが返されます。二重化を有効にしていない場合、ハッシュ不整合のエラーは、セーブ中の電源断やカード抜けでデータ破損が起こったデータへのアクセスでも発生します。

セーブデータ領域へのアクセスを完了し、マウントを解除するときはアーカイブ名を引数に nn::fs::Unmount() を呼び出してください。

コード 7-9. マウントの解除

```
nn::Result nn::fs::Unmount(const char* archiveName);
```

DSi ウェアと同様に、ダウンロードアプリのセーブデータ領域は、インポートと同時にインポートされたメディア上に自動的に作成されます。ダウンロードアプリが削除されたときは、そのセーブデータも一緒に削除されます。

7.3.1. セーブデータの巻き戻し対策

補足: CTR-SDK の関数リファレンスの「ファイルシステム：セーブデータ巻き戻し防止支援機能」を参照してください。

注意: セーブデータの巻き戻し対策機能を使用するアプリケーションは、基本的にバナースペックファイルでセーブデータのバックアップ機能に対応しないように(DisableSaveDataBackup を True に)設定してください。詳細については、「3DS オーバービュー」および ctr_makebanner の説明を参照してください。なお、セーブデータの巻き戻し対策機能の使用・不使用を、パッチの適用時に切り替えることは禁止されています。

7.3.2. ファイル、ディレクトリが存在しない場合のエラーハンドリング

アプリケーションでセーブデータの作成手順を工夫していても、以下のような操作で、マウントに成功した上でファイルが存在しない状態になり得ます。

- セーブデータ初回作成時などでセーブデータのフォーマットは完了している。
- フォーマット後のファイル、ディレクトリの作成中(セーブデータを二重化している場合は初回コミットまでの間)に下記理由でアプリケーションが強制的に終了した。
  - 電源ボタンを長押しした。
  - カードを抜いた。
  - 処理途中で電池切れとなった。
    特にスリープを許可しているアプリケーションにおいて、本体を閉じた状態のまま放置する。

上記のような場合には、ファイル、ディレクトリ操作時に nn::fs::ResultNotFound が返される可能性がありますので、ユーザーの手元でセーブデータを正常な状態に復帰できるようにしてください。

7.4. 拡張セーブデータ

SD カードに作成し、セーブデータとは別に管理されるデータ領域です。作成した本体でのみ使用可能なデータ領域ですので、本体内にのみ存在する情報(フレンド情報など)とアプリケーション独自のデータを結び付ける場合は、そのデータを拡張セーブデータに保存することを推奨しています。拡張セーブデータにはゲームの進行に必要なデータはなるべく保存しないでください。拡張セーブデータをシステムが自動的に作成することはありません。

コード 7-10. 拡張セーブデータのマウント、作成、削除

```
nn::Result nn::fs::MountExtSaveData(const char* archiveName,
                                    nn::fs::ExtSaveDataId id);
nn::Result nn::fs::CreateExtSaveData(nn::fs::ExtSaveDataId id,
                                     const void* iconData, size_t iconDataSize,
                                     u32 entryDirectory, u32 entryFile);
nn::Result nn::fs::DeleteExtSaveData(nn::fs::ExtSaveDataId id);
```

拡張セーブデータを使用するには、拡張セーブデータ ID を指定して nn::fs::MountExtSaveData() を呼び出し、拡張セーブデータをマウントしなければなりません。返り値に nn::fs::ResultNotFormatted、nn::fs::ResultNotFound、nn::fs::ResultBadFormat、nn::fs::ResultVerificationFailed が返された場合は、すでに拡張セーブデータが作成されているならば nn::fs::DeleteExtSaveData() で拡張セーブデータを削除し、nn::fs::CreateExtSaveData() で拡張セーブデータを作成してから再度マウントを行ってください。nn::fs::ResultOperationDenied に属するエラーが返された場合は、SDカードやファイルに書込めない状態に

なっているか、SDカードの接触不良などの可能性があります。詳細は関数リファレンス「ファイルシステム：エラーハンドリングについて」を参照してください。

archiveName に指定されたアーカイブ名で拡張セーブデータがマウントされます。アーカイブ名の指定については「7.2.1. アーカイブ名の指定について」を参照してください。

id には拡張セーブデータ ID を指定します。拡張セーブデータ ID とは拡張セーブデータを識別するための番号です。アプリケーションからアクセスすることのできる拡張セーブデータの拡張セーブデータ ID は、makerom で使用する RSF ファイルの "ExtSaveDataNumber"（1 つのみ指定可能）もしくは "AccessControlInfo" の "AccessibleSaveDataIds"（複数指定可能）に記述する必要があります。アプリケーションで作成する拡張セーブデータ ID には通常、弊社より発行されたユニーク ID を指定します。そのため、複数のアプリケーションで拡張セーブデータを共有する場合は、それらのアプリケーションのいずれかのユニーク ID を指定する必要があります。

拡張セーブデータを作成する際には、*iconData* と *iconDataSize* にデータ管理画面で表示されるアイコンデータ（ctr_makebanner32 で作成された icn ファイル）とそのサイズを指定します。また、*entryDirectory* と *entryFile* には拡張セーブデータに格納したいディレクトリとファイルの数をそれぞれ指定します。

マウントされた拡張セーブデータはアーカイブとして扱うことができます。アーカイブ内には、自由にファイルを作成することができますが、ファイル作成には `nn::fs::TryCreateFile()` を使用しなければならず、作成後はファイルサイズを変更することができません。ファイル名とディレクトリ名は 16 文字まで、パスの最大長は 248 文字です。ファイル名とディレクトリ名に使用することのできる文字は、半角の英数字と一部の記号のみで、パスの区切りには "/（スラッシュ）" を用います。アーカイブのサイズは可変です。

アーカイブ内のファイルは、ハッシュ値による改竄チェックで保護されています。アクセスしたデータのハッシュ値が合わないときは、改竄されていると判断されてエラーが返されます。ハッシュ不整合のエラーは、データ書き込み中の電源断やカード抜けでデータ破損が起こったデータへのアクセスでも発生します。セーブデータとは異なり、拡張セーブデータはライブラリによる二重化がサポートされていません。

データの保護機能はありませんので、ファイルを書き込んでいる途中で SD カードが抜かれたときは拡張セーブデータが高い確率で壊れてしまいます。壊れてしまった場合は `nn::fs::DeleteExtSaveData()` で削除してから、拡張セーブデータを作り直す必要があります。

拡張セーブデータへのアクセスが完了し、マウントを解除するときは、アーカイブ名を引数に `nn::fs::Unmount()` を呼び出してください。

ダウンロードアプリの削除は拡張セーブデータに影響しません。拡張セーブデータは本体設定のデータ管理画面で削除される可能性があります。

> **補足：** 拡張セーブデータの総サイズが 32 MByte まであれば、アプリケーションで自由に使用することができます。32 MByte を超えて使用したい場合は、弊社窓口までお問い合わせください。

7.4.1. 拡張セーブデータの用途

拡張セーブデータには以下のようなデータを保存するという用途があります。

- アプリケーション独自のデータ
- シリーズ共通のデータ（異なるバージョンのアプリケーションなどと共通で使いたいデータ）
- ダウンロードしたデータ（追加アイテム、追加コースなど）
- CTR 拡張バナーデータ（アプリケーションで作成、もしくはサーバーからダウンロード）

7.4.1.1. アプリケーション独自のデータ

ゲームの進行とは無関係で、あまり重要度の高くないデータや本体にのみ保存されている情報と結び付けたデータ、サイズの大きなユーザー作成のデータなどです。

7.4.1.2. シリーズ共通のデータ

異なるバージョンやナンバリングタイトルのアプリケーションと共通で使用するデータです。シリーズで共通の拡張セーブデータ ID を使用することで、シリーズ間でのデータの共有を行うことができます。

7.4.1.3. ダウンロードしたデータ

追加コースなど、ダウンロードタスクを登録してダウンロードしたデータです。ダウンロードタスクで取得したデータはタスクを登録したアプリケーションでなければアクセスできませんが、拡張セーブデータ内に移動させることで拡張セーブデータを共有するアプリケーションからもアクセスすることができるようになります。

7.4.1.4. CTR 拡張バナーデータ

CTR タイトルバナーのデータの一部を置き換えることができるデータです。置き換えることができるのは、上画面の下部でスクロール表示されるテキストと、差し替え用に決められている名前のテクスチャ(1 枚のみ)です。アイコンデータの差し替えには対応していません。

CTR 拡張バナーデータには、アプリケーションで作成するローカル拡張バナーと、サーバーからダウンロードする DL 拡張バナーの 2 種類が存在します。

ローカル拡張バナー

ゲームの進行状況に合わせてメッセージを表示したり、CTR タイトルバナーで表示されるテクスチャの一部を変更したりすることができます。テキストは最大 255 文字(全角半角問わず)、データのサイズは最大 128 KByte です。1 枚のテクスチャの中に複数の画像を入れ、複数モデルに貼り付けることは可能です。全言語で共通の COMMON データと、必要ならばリージョンで有効となっている言語分のデータをそれぞれ用意してください。

DL 拡張バナー

サーバーが用意したデータで表示の変更を行うことができます。テキストとテクスチャはローカル拡張バナーと同じ仕様です。DL 拡張バナーには有効期限(年月日)を設定することができます。

CTR 拡張バナーデータの表示について

CTR 拡張バナーデータのテキストは、DL 拡張バナーのテキスト、ローカル拡張バナーのテキストの順に表示されます。ローカル拡張バナーと DL 拡張バナーに同じ名前のテクスチャが含まれていた場合、DL 拡張バナーのテクスチャが優先されます。

7.4.2. 複数のゲームカードからのアクセス

1 台の 3DS と 1 枚の SD カードの組み合わせを複数人で共有し、それぞれが同じタイトルのゲームカードを所持している可能性があります。セーブデータは各ゲームカードのバックアップメモリに書き込まれるため、このような場合でも問題はありません。しかし、拡張セーブデータは SD カードに書き込まれるため、拡張セーブデータに保存するデータの構成によっては不具合を引き起こす可能性があることに注意してください。

具体的には、以下のような状況が考えられます。

- セーブデータ、拡張セーブデータの整合性が取れずにプログラムが暴走する。
- ファイル名を固定にしていると、別のゲームカードで作成された拡張セーブデータ内のファイルを、ユーザーの意図に関係なく上書きしてしまう。
- 開発者の意図に反して、1 台の 3DS で複数枚のゲームカードで遊ぶために、同じ枚数の SD カードが必要となってし

まう。

例えば、拡張セーブデータ内にファイルを作成する場合は、ディレクトリ名やファイル名にユニークな名前を付与することで不用意な上書きを回避することができます。ただし、名前の付与基準には、キャラクタ名などのユーザーが任意で付けられるものではなく、現在時刻を加工した文字列などを適用してください。また、セーブデータ内に拡張セーブデータと結び付けるための情報を書き込んだ場合は、その情報に該当する拡張セーブデータが SD カード上に存在しない可能性があることに注意してください。

> **注意:** CTR 拡張バナーデータに関しては、このような状況に対応することができません。そのため、どのゲームカードが挿入されたとしても、最後に作成されたローカル拡張バナーデータや DL 拡張バナーデータが有効になります。

7.4.3. 複数の拡張セーブデータへのアクセス

RSF ファイルのアクセス可能セーブデータ属性(″AccessControlInfo″ の ″AccessibleSaveDataIds″)に複数のユニーク ID(最大 6 個)を記述することで、以下のデータにアクセスすることができるようになります。

- ユニーク ID が記述されたユニーク ID と同じであるセーブデータ
- 拡張セーブデータ ID が記述されたユニーク ID と同じである拡張セーブデータ

> **注意:** 本体設定で拡張セーブデータが個別に消去される可能性がありますので、整合性が必要なデータはなるべく拡張セーブデータに保存しないでください。
>
> CTR 拡張バナーデータを使用するタイトルは、共通でアクセスする拡張セーブデータのほかに、自身のユニーク ID を拡張セーブデータ ID とする拡張セーブデータを作成してください。これは拡張バナーの表示に使用するファイルが、自身のユニーク ID で作成された拡張セーブデータの ″ExBanner″ ディレクトリになければならないためです。

> **補足:** RSF ファイルの記述方法については、CTR-SDK のツール「ctr_makerom」のリファレンスおよび「3DS プログラミングマニュアル - アプリケーション作成フロー編」を参照してください。

7.5. SD カードを直接参照するアーカイブ

`nn::fs::MountSdmcWriteOnly()` を呼び出して、挿入されている SD カードを直接参照するアーカイブをマウントすることができます。ただし、この関数を使用するには、RSF ファイルのファイルシステムアクセス権利属性(″AccessControlInfo″ の ″FileSystemAccess″)に ″- DirectSdmcWrite″ が記述されている必要があります。

コード 7-11. SD カードを直接参照するアーカイブのマウント

```
nn::Result nn::fs::MountSdmcWriteOnly(const char* archiveName = "sdmcwo:");
```

この関数でマウントされたアーカイブでは、ファイルやディレクトリの読み込みはできませんが、書き込まれたファイルが暗号化されないため、PC などでそのまま読み込むことができます。また、同じ SD カード上に存在する拡張セーブデータとは異なり、ファイルの作成後にサイズを変更することができます。

マウント時に返される可能性があるエラーは拡張セーブデータのマウント時よりも種類が少なく、エラーのハンドリングは拡張セーブデータのマウント時と同じ方法で行うことができます。ただし、読み込みが禁止されているため、一度書き込みを行わ

なければディレクトリやファイルの存在確認ができませんので、ディレクトリやファイルの作成時には nn::fs::ResultAlreadyExists に属するエラーをハンドリングする必要があります。

マウントを解除するときは、アーカイブ名を引数に nn::fs::Unmount() を呼び出してください。

> **補足:** アプリケーションから書き込みを行うことのできるパスは制限されています。制限や運用上の注意点はガイドラインの「ファイルシステム」を参照してください。

7.6. メディアごとの注意点

7.6.1. 3DS カード

アクセス速度の変化に対する検証

FS ライブラリによる 3DS カード上の ROM アーカイブへのアクセスは、メディアの状態が良好なときと劣化したときのパフォーマンスに大きな開きが発生することがあります。また、本体更新によりパフォーマンスが改善されることもあります。そのため 3DS カードでの販売を予定しているアプリケーションは、このアクセス速度の変化によって不具合が起こらないかどうかを、以下の検証を行って確認してください。

- CARD1 アプリケーションの場合
  1. Config ツールで「Debug Setting」の「Debug Mode」を「enable」に設定(レイテンシエミュレーションが有効な状態)して一通りプレイする。
  2. 開発カードスピードを Fast 設定にし、開発カードに書き込まれたアプリケーション(Release ビルド)を一通りプレイする。
- CARD2 アプリケーションの場合
  1. Config ツールで「Debug Setting」の「Debug Mode」を「enable」に設定(レイテンシエミュレーションが有効な状態)して一通りプレイする。
  2. PARTNER-CTR Debugger のカード制御ダイアログのカードエミュレーション制御タブでエミュレーションメモリスピード設定の転送速度を「Fast」に設定し、エミュレーションメモリ上にアプリケーション(Release ビルドの CCI ファイル)をロードした状態で一通りプレイする。
  3. 開発カードスピードを Fast 設定にし、開発カードに書き込まれたアプリケーション(Release ビルド)を一通りプレイする。

> **注意:** なるべく最新バージョンの PARTNER-CTR Debugger/Writer ソフトウェアで検証を行ってください。

7.6.2. SD カード

SD カードに保存される拡張セーブデータやダウンロードアプリを含め、SD カードへのアクセスが行われるアプリケーションの実装では以下の点に注意してください。

SD メモリカードを直接参照する際の注意点

nn::fs::MountSdmc() でマウント(引数省略時は "sdmc:"にマウント)された、SD メモリカードを直接参照するためのアーカイブはデバッグ用途専用です。製品版ではアクセスすることはできませんので注意してください。

スリープ中に SD カードが差し替えられる可能性があることに注意してください。マウントされていた SD カードが抜かれ、再挿入された SD カードをマウントするときは、nn::fs::Unmount() でアンマウントしてから再マウントを行わなければなり

ません。

ストリーミング再生時の注意点

SD カードの空き容量がほとんどない、著しく断片化が進んでいる、SD カードの劣化などの要因によりファイルアクセスの速度が低下することがあります。SD カードへアクセスする関数の多くはこの影響を受けて速度が低下し、特にファイルを新規作成するような関数(`nn::fs::CreateExtSaveData()` など)への影響は大きくなります。

そのため、このような関数をサウンドやムービーのストリーミング再生を行っているときに呼び出した場合、実行完了までファイルアクセスがブロックされるため、音切れやフレーム落ちの原因のひとつとなり得ることに注意してください。

リード・ディスターブへの対策

SD カードには読み出し回数に対するデータの保持能力に限界があり、それを超えて読み出そうとすると、データの一部が変わってしまう「リード・ディスターブ」という劣化現象が起こることがあります。リード・ディスターブ現象が起こる箇所によっては、SD カードのデータがすべて読み込めなくなる場合もあります。

特に気を付けなければならないのは、**SD カードからの読み出しはファイルのオープンを行うだけでも発生する**ことです。市場にはリード・ディスターブ現象への耐性が十分でない SD カードも流通していますので、この現象が発生するリスクを低減させるためにも、アプリケーション側でファイルの開閉処理をできる限り削減するようにしてください。たとえば、同じファイルに対して異なるオフセットで複数回のアクセスを行う場合は、アクセスの度にファイルの開閉処理を行わず、なるべく 1 回の開閉処理の間にすべてのアクセスを行うようにしてください。

また、SD カード上の同じ領域を繰り返し読み出すことでデータの維持が不安定になるため、基本的に書き換えが行われない ROM アーカイブや追加コンテンツへのアクセスがもっとも問題を起こしやすいと言えます。カードアプリであっても、将来的にダウンロードアプリとして販売する可能性がある場合には注意が必要です。

データを長く維持させる工夫としては、SD カードから特定のデータを繰り返し読み出す場合(短い波形データを繰り返しストリーミング再生する場合など)には、一度 SD カードからメモリ上のバッファにデータを読み出し、そのバッファのデータを使用するという方法があります。このとき、バッファのサイズを 16 KByte 以上にすることが効果的で、サイズを大きくすることでさらに改善していきます。

この対策は制限事項ではありませんので、メモリの空きに余裕がない状況でも無理に対応しなければならないものではありません。

> **補足:** NW4C サウンドライブラリを利用している開発者は、`nw::snd::SoundArchivePlayer` クラスのリファレンスに記載されている説明を参照してください。

7.7. セーブデータと拡張セーブデータの使い分け

アプリケーションを販売する際に選択可能なメディアとして 3DSカード(CARD2)が用意されるようになり、アプリケーション固有のセーブデータ領域として利用可能な容量が大きくなりました。そのため、以前は容量の関係で拡張セーブデータに保存せざるを得なかったデータをセーブデータとして保存することができるようになります。それに伴い、保存するデータの種類や用途によって、セーブデータと拡張セーブデータを適切に使い分ける必要があります。この節では、セーブデータと拡張セーブデータそれぞれの特性を説明し、どのようなデータをどちらに保存すべきかを説明します。

7.7.1. セーブデータと拡張セーブデータの特性

セーブデータと拡張セーブデータには、以下のような特性の違いがあります。

表 7-6. セーブデータと拡張セーブデータの特性

項目	セーブデータ	拡張セーブデータ
二重化機能	対応	非対応
バックアップ機能	対応(ダウンロードアプリのみ)。 原則有効(下記参照)	非対応
巻き戻し対策機能	対応(ダウンロードアプリのみ)。 利用する場合はバックアップ機能を無効にすること	非対応
保存先メディア	カードアプリ: カードのバックアップ領域 ダウンロードアプリ: SDカード	SDカード
容量確保のタイミング	カードアプリ: 最初から ダウンロードアプリ: インストール時	アプリケーションが確保したとき
削除	アプリケーション内でのみ可能	アプリケーション内、本体設定のデータ管理画面で可能
データ破壊	バックアップ、二重化によりデータ破壊のリスクを軽減可能	データ破壊のリスクは軽減できない
データ破壊時に必要な対応	二重化あり: アーカイブ全体が破壊された場合のハンドリングのみ必要 二重化なし: アーカイブ全体の破壊に加えて、ファイル単位、ディレクトリ単位での破壊に対するハンドリングが必要	アーカイブ全体の破壊に加えて、ファイル単位、ディレクトリ単位での破壊に対するハンドリングが必要
容量不足への対応	不要(確保した容量内での対応は必要)	ファイル作成のたびに必要

これまでに、以下のような理由でバックアップ機能を無効にしたアプリケーションがあります。

- 巻き戻し防止機能を使用している
- セーブデータ側だけバックアップ機能により巻き戻ることで、拡張セーブデータとセーブデータ間で不整合が生じる可能性がある
- バックアップからセーブデータを復元されると都合の悪いデータがある
  配信データやレアキャラなど、ほかのユーザーに譲渡したあとにセーブデータを復元させて、無限に増殖できてしまうなど

7.7.2. データの保存先を決定する際の方針

基本的に、消えてしまうことでゲームの進行が不可能になるようなデータや、重要度の高いデータはセーブデータに保存してください。消えてしまってもゲームの進行に問題のないデータや最終的に必要となる容量が不定となるデータは拡張セーブデータに保存することを推奨します。なお、拡張セーブデータは本体設定のデータ管理画面で消去することができるため、アプリケーションの制御が及ばないタイミングで消されることを考慮する必要があります。また、カードアプリのパッケージ版の場合、異なる本体でプレイする可能性や、前回アクセスした拡張セーブデータが保存されている SDカードとは異なるSDカードが挿入されている可能性があることにも注意が必要です。

以下のデータは、ユーザーのデータを保護するためにも、セーブデータに保存します。

- ゲームの進行データ
- ゲームの進行データとの整合性が必要なデータ
  拡張セーブデータに整合性が必要なデータを保存した場合、拡張セーブデータが削除されたことによりセーブデータの初期化が必要となる可能性やバックアップ機能を使用した際に整合性が保てなくなる可能性があります。そのため、ゲーム内の各所でセーブデータを巻き戻したり、拡張セーブデータを削除したりして整合性の確認をするなど、デバッグの作業量が膨大になることが考えられます。

以下のようなデータは、セーブデータに保存することを推奨します。

- ゲームを楽しむ上で重要と思われるデータ
  ステージ初クリア時のスクリーンショットやフレンドから送られてきた記念日のメッセージなど。
- 複数のファイル間で整合性が必要なデータ
  リプレイシーンを構成するテクスチャとコマンドデータなど。
  セーブデータの方が、ファイルごとに壊れる可能性は低くなります。ただし、1 つのファイルに結合して保存するならば、ファイル消失のリスクは拡張セーブデータでも二重化なしのセーブデータと同程度になります。

以下のようなデータは、消えてしまってもゲームの進行に影響がなければ、拡張セーブデータに保存することを推奨します。

- ネットから再ダウンロードが可能なデータやゲーム中に再作成が可能なデータ
- ほかのアプリケーションと共有するデータ
  特に複数のゲームを並行してプレイしながらも、互いに共有するデータは拡張セーブデータに保存することを推奨します。ただし、セーブデータのバックアップ機能を用いると、アイテム増殖などの不正を行うことができるようになる可能性があることに注意が必要です。
- ファイル数やサイズが不定のデータ
  ユーザーが作成したステージデータや楽譜データ、ユーザーが録画した動画など。
  拡張セーブデータは、保存可能なディレクトリ数とファイル数を領域の作成時に指定しますが、ファイルサイズには制限がありません。一方、セーブデータは保存可能なディレクトリおよびファイルの数を初期化時に指定するのは同様ですが、領域の容量が有限であるため、サイズ不定のデータを複数保存するような用途には向いていません。
- ほかのアプリケーションのセーブデータから移行するデータ
  ほかのアプリケーションのセーブデータへのアクセスは可能ですが、パッケージ版同士の組み合わせでは不可能なため、旧作からのデータ移行などでも拡張セーブデータを利用するようにしてください。ただし、ダウンロード専売のアプリケーション同士やダウンロード体験版からセーブデータを移行する場合はセーブデータへのアクセスを利用してください。

8. 時間

3DS で扱うことのできる時間には、システムクロックをもとに計測した時間であるチックとバッテリバックアップ付きで時刻を計時する RTC の 2 種類があります。

アプリケーションに時間の経過を知らせる機能としては、チックを利用したタイマとアラームがあります。

RTC は現在時刻を取得する機能のほかにも、設定された日時にアラームを鳴らす機能を持っています。

8.1. 時間を表すクラス

SDK 内で時間を指定する関数は、単位を間違えないためにも、その引数に nn::fnd::TimeSpan クラスを使うように設計されています。このクラスの内部では、時間はナノ秒単位 の 64 bit 整数として表されています。単位があいまいになるのを防ぐため、整数値からこの型への暗黙的な変換は用意されていませんが、0 だけはこの型への暗黙的な変換を行うことができます。

nn::fnd::TimeSpan クラスには、各単位(秒、ミリ秒、マイクロ秒、ナノ秒)で表された時間の整数値からインスタンスを生成する static メンバ関数(From*Seconds())と、インスタンスが保持している時間を各単位での値として s64 型で取得するメンバ関数(Get*Seconds())が用意されています。

また、このクラス同士であれば比較(==, !=, <, >, <=, >=)と加減算(+, -, +=, -=)を行うことができます。

コード 8-1. 各時間単位からのインスタンス生成関数と時間の取得関数

```
// From*Seconds()
static nn::fnd::TimeSpan FromSeconds(s64 seconds);
static nn::fnd::TimeSpan FromMilliSeconds(s64 milliSeconds);
static nn::fnd::TimeSpan FromMicroSeconds(s64 microSeconds);
static nn::fnd::TimeSpan FromNanoSeconds(s64 nanoSeconds);
// Get*Seconds()
s64 GetSeconds() const;
s64 GetMilliSeconds() const;
s64 GetMicroSeconds() const;
s64 GetNanoSeconds() const;
```

8.2. チック

チックは CPU が 268 MHz で動作しているときの 1 クロックの時間(約 3.73 ナノ秒)で、その値をチック値と表記することもあります。1 秒間のチック値は nn::os::Tick::TICKS_PER_SECOND に定数として定義されています。

> **注意:** SNAKE が拡張モードで動作している場合でも、1 チックは標準モードで動作している場合と同じ時間です。つまり、拡張モードでは 1 チックが 3 CPU サイクル分の時間となります。

チック値は nn::os::Tick クラスで扱うことができ、変換コンストラクタや変換演算子によって実際の時間を表す nn::fnd::TimeSpan クラスと相互に変換することができます。

コンストラクタには、s64 型で表されたチック値からインスタンスを生成するコンストラクタと、nn::fnd::TimeSpan クラスで表された時間をチック値に変換してインスタンスを生成するコンストラクタの 2 種類があります。

変換演算子には、チック値を s64 型の数値に変換するものと、nn::fnd::TimeSpan クラスに変換するものの 2 種類が

あります。nn::os::Tick クラス同士であれば加減算(+, -, +=, -=)を行うことができます。

上記以外にも、ToTimeSpan() を呼び出すことで、チック値から変換された時間を表す nn::fnd::TimeSpan クラスのインスタンスを生成することができます。

GetSystemCurrent() を呼び出すことで、システムが起動してから経過した時間をチック値(nn::os::Tick クラス)で取得することができます。

8.3. タイマ

タイマは、指定された時間が経過したことを通知する機能です。タイマはシグナル状態と非シグナル状態の 2 つの状態を持ち、指定された時間が経過すると非シグナル状態からシグナル状態に遷移します。タイマの作成個数は 8 個に制限されています。

タイマは nn::os::Timer クラスで定義されています。インスタンスを生成して Initialize() または TryInitialize() を呼び出して初期化します。初期化の際には、シグナルリセットの種類を手動リセットと自動リセットから選択することができます。手動リセットの場合、シグナル状態になるとその状態はクリアされるまで維持され、その間、そのタイマがシグナル状態になるのを待っているスレッドがすべて解放されます。自動リセットの場合、シグナル状態になると、そのタイマがシグナル状態になるのを待っているスレッドのうちで優先順位の一番高いスレッドだけが解放され、すぐに非シグナル状態に戻ります。

タイマの種類には、開始から指定された時間が経過すると 1 度だけシグナル状態に遷移するワンショットタイマと、指定された時間が経過したあとも一定間隔でシグナル状態に遷移する周期的タイマの 2 種類があります。ワンショットタイマで開始するには StartOneShot() を呼び出します。周期的タイマで開始するには StartPeriodic() を呼び出します。どちらのタイマも Stop() を呼び出して停止させることができます。

Wait() の呼び出しで、スレッドはタイマがシグナル状態になるまで待ちます。時間の経過を待たずにタイマをシグナル状態に遷移させるには Signal() を呼び出します。シグナル状態に遷移すると、シグナルリセットの種類が手動リセットの場合は、ClearSignal() を呼び出すまでシグナル状態が維持されます。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

タイマを使用するアプリケーションは「5.3.1.1. 終了処理の注意事項」を参照し、終了処理でインスタンスの破棄などを確実に行ってください。

8.4. アラーム

アラームは、指定された時間が経過すると登録されたハンドラを呼び出す機能です。アラームは内部でスレッドを作成してハンドラを呼び出します。そのため、使用する前に nn::os::InitializeAlarmSystem() でアラームシステムを初期化していなければなりません。

アラームは nn::os::Alarm クラスで定義されています。インスタンスを生成して Initialize() または TryInitialize() を呼び出して初期化します。

アラームの種類には、開始から指定された時間が経過すると 1 度だけハンドラを呼び出すワンショットアラームと、指定された時間が経過したあとも一定間隔でハンドラを呼び出す周期的アラームの 2 種類があります。ワンショットアラームを設定するには SetOneShot() を呼び出します。周期的アラームを設定するには SetPeriodic() を呼び出します。どちらのアラームも Cancel() を呼び出して解除することができます。

アラームを設定することができるか状況かどうかは CanSet() の呼び出しで取得することができます。アラームが設定されていて、今後ハンドラが呼び出される場合は false を返します。Cancel() で解除しても、一度ハンドラが呼び出されるま

で true は返されません。

ハンドラの型は以下のように定義されています。

コード 8-2. アラームハンドラの型定義

```
typedef void(* nn::os::AlarmHandler)(void *param, bool cancelled);
```

param にはアラームを設定したときの引数が渡されます。通常、*cancelled* には false が渡されますが、Cancel() でアラームが解除されたあとにハンドラが呼び出された場合は true が渡されます。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

> **注意:** 現在の実装では、アラームシステムは 2 つの内部スレッドで機能しています。そのため、処理に長い時間のかかるハンドラが多数登録されていると、アラームシステムが不安定になります。

アラームを使用するアプリケーションは「5.3.1.1. 終了処理の注意事項」を参照し、終了処理でインスタンスの破棄などを確実に行ってください。

8.5. RTC

RTC は Real Time Clock の略で、時刻の計時を行うハードウェアのことを指します。バッテリバックアップにより、本体の電源が落とされていても時刻の計時は継続されます。

時刻の設定は本体設定からのみ行うことができます。設定可能な日付は 2049/12/31 までですが、計時可能な日付は 2099/12/31 までです。時刻の設定から 50 年以上経たないと計時の上限に達することがないため、アプリケーション起動中の時刻の巻き戻りをチェックする必要はありません。

この節では、RTC で計時されている時刻をアプリケーションで利用する方法について説明します。

8.5.1. 日時を表すクラス

CTR-SDK では日時を表すクラスとして nn::fnd::DateTime クラスを用意しています。RTC で計時されている現在時刻は nn::fnd::DateTime::GetNow() の呼び出しで取得することができ、返されるのはこのクラスのインスタンスとなっています。

引数ありのコンストラクタでは、年月日と時分秒だけでなく、ミリ秒まで指定可能です。引数なしのコンストラクタでは 2000/01/01 00:00:00.000 を表すインスタンスが生成されます。

このクラスで扱うことのできる日時は nn::fnd::DateTime::MIN_DATE_TIME(1900/01/01 00:00:00.000)から nn::fnd::DateTime::MAX_DATE_TIME(2189/12/31 23:59:59.999)までです。2000/01/01 (Sat) を基準とするグレゴリオ暦ですべての年が計算され、1 日が正確に 86400 秒であるという前提で日時を扱います。

nn::fnd::DateTime クラス同士の減算では、2 つの日時の差異が nn::fnd::TimeSpan クラスで返されます。また、nn::fnd::DateTime クラスと nn::fnd::TimeSpan クラスとの加減算は、ある日時から指定時間が経過した(戻った)日時を nn::fnd::DateTime クラスで返します。

日時のパラメータには、年、月、日、曜日、時(24 時間制)、分、秒、ミリ秒があり、それぞれ Get*() で取得、曜日を除くパラメータは Replace*() で書き換えを行うことができます。各パラメータに対する Get*() は、曜日のみが列挙型(nn::fnd::Week)を、そのほかは s32 型を返します。Replace*() は書き換えたパラメータを持つ新しいインスタンスを返します。元のインスタンスのパラメータは書き換えられません。

nn::fnd::DateTimeParameters 構造体を引数に持つ GetParameters() と FromParameters() では、すべてのパラメータの取得および書き換えを一括して行うことができます。FromParameters() の呼び出しでは、構造体の曜日を示すメンバ変数は無視されます。無効な日時のパラメータで書き換えた場合の結果は不定です。有効な日時のパラメータであるかどうかは IsValidParameters() で確認することができます。構造体を引数にした関数では曜日も判定基準に入っていますので、構造体のパラメータ設定には注意が必要です。曜日が不明である場合は、曜日以外の各パラメータを引数に持つ関数で確認してください。

DateToDays() では、指定された日付が基準日(2000/01/01)から何日後であるかを取得することができます。無効な日付で問い合わせた場合の結果は不定です。有効な日付であるかどうかは IsValidDate() で確認することができます。また、指定された年がうるう年であるかどうかは IsLeapYear() で確認することができ、うるう年ならば 1 を返します。

DaysToDate() では、基準日からの経過日数から日付を取得することができます。DaysToWeekday() では、基準日からの経過日数から曜日を取得することができます。

8.5.2. RTC アラーム機能

RTC で計時されている時刻を利用した RTC アラーム機能が用意されています。この機能は、現在時刻に分単位の変化が起こったときに設定された時刻を過ぎているとアプリケーションへの通知を行います。時刻の設定によっては 1 分少々遅れて通知される可能性があります。そのため、目覚まし時計のように指定時刻を通知する用途には向いていますが、砂時計のように時間経過を通知する用途には向いていません。時間経過を通知する用途には、「8.3. タイマ」または「8.4. アラーム」を利用してください。

RTC アラーム機能は、PTM ライブラリを介して使用することができます。PTM ライブラリを使用するには、nn::ptm::Initialize() を呼び出して初期化を行う必要があります。PTM ライブラリの使用を終了するときは nn::ptm::Finalize() を呼び出してください。

コード 8-3. PTM ライブラリの初期化と終了

```
nn::Result nn::ptm::Initialize();
nn::Result nn::ptm::Finalize();
```

RTC アラーム機能では、設定された時刻を過ぎたときに、指定されたイベントがシグナル状態になることでアプリケーションへの通知が行われます。

> **注意:** スリープ中は通知を受け取ることができません。HOME メニューやライブラリアプレットを起動した状態では、HOME メニューやライブラリアプレットを起動したスレッドが停止されたまま、イベントがシグナル状態になります。グラフィックスやサウンドを操作する権限のない状態で通知を受け取るため、RTC アラームを待ち受けるスレッドの実装には注意が必要です。

イベントの指定やアラーム時刻の設定などは、以下の関数で行います。

コード 8-4. RTC アラーム機能の関数

```
nn::Result nn::ptm::RegisterAlarmEvent(nn::os::Event &event);
nn::Result nn::ptm::SetRtcAlarm(nn::fnd::DateTime datetime);
nn::Result nn::ptm::GetRtcAlarm(nn::fnd::DateTime *pDatetime);
nn::Result nn::ptm::CancelRtcAlarm();
```

RTC アラーム機能の通知を受け取るイベントは nn::ptm::RegisterAlarmEvent() で指定します。*event* に渡す nn::os::Event クラスのインスタンスは、アプリケーションで生成および初期化を行ってください。

アラーム時刻の設定は nn::ptm::SetRtcAlarm() で行います。*datetime* には、アラーム時刻を

nn::fnd::DateTime クラスで指定しますが、その時刻は分単位で設定してください。すでにアラーム時刻が設定されていた場合は nn::ptm::ResultOverWriteAlarm を返しますが、アラーム時刻の設定には成功しています。過去の時刻を設定しても、すぐには通知されません。

アラーム時刻の現在の設定は nn::ptm::GetRtcAlarm() で取得することができます。*pDatetime* には、アラーム時刻を受け取る nn::fnd::DateTime クラスのインスタンスへのポインタを渡してください。アラーム時刻が設定されていなかった場合は、nn::ptm::ResultNoAlarm を返します。

RTC アラームの設定をキャンセルする場合は nn::ptm::CancelRtcAlarm() を呼び出してください。アラーム時刻が設定されていなかった場合は、nn::ptm::ResultNoAlarm を返します。

8.5.3. 時刻改変のオフセット値の取り扱い

ユーザーが本体設定の「日付と時刻」で時刻を何秒進めたか、もしくは何秒戻したかを、秒単位のオフセット値の累積をハードウェアに記録しています。その情報は nn::cfg::GetUserTimeOffset() で取得することができます。詳細については「11.2.8. RTC 改変オフセット値」を参照してください。

同一の本体でアプリケーションを起動した場合に限り、このオフセット値を前回起動時の値と比較することで、ユーザーが時刻の設定を変更したかどうかを判別することができます。この判別の結果をアプリケーションで利用することができますが、別の本体でアプリケーションを起動した場合でもゲームの進行に不都合がないようにしてください。

8.6. チックと RTC の混在使用の禁止

チック(nn::os::Tick クラス)と RTC(nn::fnd::DateTime クラス)は同じように時間を扱っていますが、時間の進み具合や精度が異なります。つまり、同じ期間をチックと RTC で計測しても、その結果が同じ値になるとは限りません。そのため、チックで得た時間と RTC で得た時間を混在して使用してはいけません。

チックには個体差があり、温度変化の影響を大きく受けます。また、1 ヶ月に ±300 秒程度の誤差が生じる可能性があります。

RTC にも個体差があり、温度変化の影響を大きく受けます。また、1 ヶ月に ±60 秒程度の誤差が生じる可能性があります。なお、nn::fnd::DateTime::GetNow() は計算により補間された現在時刻を返すため、進む速度には 1 時間当たり ±1 秒程度の揺らぎがあります。

サウンドや動画のストリーミング再生など、ライブラリ内で時間を扱っている場合の多くは、その時間の進む速度がチックや RTC の速度とは異なります。そのため、アプリケーションでの演出とサウンド再生の同期を取りたい場合には、個々のライブラリの仕様を把握して、速度が一致する時間を使用してください。

9. スレッド

SDK で定義されているスレッドクラス(nn::os::Thread)には、スレッドの開始、スレッドの合流、パラメータ取得、パラメータ変更といった基本的な関数のみが定義されています。アプリケーションで使用するスレッドは、このスレッドクラスを継承したクラスでなければなりません。

9.1. 初期化と開始

コンストラクタで構築された時点では、スレッドはまだ初期化も開始もされていません。スレッドの初期化と開始は、スタック領域の管理をアプリケーションで行う場合は Start() を、ライブラリが自動的に行う場合は StartUsingAutoStack() を呼び出すことで行われます。Try で始まるメンバ関数(TryStart()、TryStartUsingAutoStack())を呼び出した場合は、リソース不足などが原因で失敗したときにエラーを返します。

スタック領域の管理をアプリケーションで行う場合、スタック領域としてメンバ関数に渡すオブジェクトには、スタックの底を uptr 型で返す GetStackBottom() というメンバ関数が定義されていなければなりません。スタック用のメモリブロックを確保する nn::os::StackMemoryBlock クラスと、static 領域や構造体の中に配置することのできる nn::os::StackBuffer クラスはその条件を満たしていますので、そのまま引数として渡すことができます。アプリケーションは、スレッドが終了するまでスタック領域が無効にならない(解放されない)ように注意しなければなりません。

9.1.1. システムコアでのアプリケーションのスレッドの実行

システムコアの CPU 時間の一部をアプリケーションに割り当て、システムコアでアプリケーションのスレッドを実行することができます。

システムコアでスレッドを実行するには、nn::os::SetApplicationCpuTimeLimit() でアプリケーションへの CPU 時間の割り当てを指定してから、*coreNo* に 1 を指定してスレッドを開始させます。CPU 時間を割り当てていない状態では、システムコアでスレッドを開始することができずにエラーとなります。

コード 9-1. アプリケーションへのシステムコアの CPU 時間の割り当て関数

```
nn::Result nn::os::SetApplicationCpuTimeLimit(s32 limitPercent);
s32        nn::os::GetApplicationCpuTimeLimit();
```

limitPercent には、アプリケーションに割り当てるシステムコアの CPU 時間の割合をパーセンテージで指定します。指定可能な値の範囲は 5～30 です。割り当てられる時間は 2 ミリ秒を一周期として、その周期の先頭から指定のパーセントに当たる時間です。つまり、25 を指定した場合は 0.5 ミリ秒(= 2 * 25 / 100 ミリ秒)がアプリケーションに割り当てられ、残りの 1.5 ミリ秒がシステムに割り当てられることになります。

現在割り当てられている CPU 時間の割合は nn::os::GetApplicationCpuTimeLimit() で取得することができます。初期状態では CPU 時間が割り当てられていないため、0 が返されます。

> **注意:** 一度アプリケーションに CPU 時間を割り当てると、以降 0 に戻すことができません。
>
> スレッドを実行していなくても、アプリケーションに割り当てられた期間はシステムの処理が行われません。そのため CPU 時間を割り当てた時点で、無線通信などのシステムコアで行われている処理の速度が低下します。

9.1.2. ManagedThread クラス

ManagedThread クラスは Thread クラスに機能を追加したユーティリティクラスです。スレッドとしての基本的な動作は、初期化と実行を行う関数が分離されていること以外は Thread クラスと同じです。

表 9-1. ManagedThread クラスで追加された機能

追加された機能	関連する関数
スタックに関する情報の取得	GetStackBufferBegin(), GetStackBufferEnd(), GetStackBufferSize(), GetStackBottom(), GetStackSize()
名前の保持	GetName(), SetName()
ID 取得の高速化	GetId(), GetCurrentId()
カレントスレッドに対応するインスタンスの取得	GetCurrentThread()
スレッドの列挙	Enumerate()
スレッドの探索	FindById(), FindByStackAddress()

Thread クラスとリソースを共有するため、ManagedThread クラスの総数は作成可能なスレッド数の制限に含まれます。また、ManagedThread を使用するために nn::os::ManagedThread::InitializeEnvironment() を呼び出した時点で、スレッドローカルストレージを 2 つ消費します。

9.2. スレッド関数

スレッドがどのような処理を行うのかは、スレッドの初期化と開始を行うメンバ関数に渡すスレッド関数で記述します。スレッド関数は 0 ～ 1 個の引数を受け取る、返り値なし(void 型)の関数で記述してください。

なお、Thread クラスの Start 系関数には、引数を受け取らないスレッド関数が使用可能なものや、スレッド関数に渡す引数の型をテンプレートで指定することのできるオーバーロードが用意されています。渡された引数はスレッドのスタックにコピーされるため、テンプレートで指定する引数の型はコピー可能でなければならないこと、その分スレッドで使用可能なスタックが減少することに注意してください。

9.3. 終了と破棄

スレッド関数の終了時や、親クラスである nn::os::WaitObject クラスの Wait*() による待ち状態からの解放時にスレッドは終了します。Join() はスレッドの終了を無条件で待ちますので、タイムアウトを考慮するなどの細かい制御が必要な場合は親クラスのメンバ関数(nn::os::WaitObject::WaitOne() など)でスレッドの終了を待ったあとに Join() を呼び出してください。Join() の呼び出しでブロックされないようにする場合も同じです。

スレッドが生きている(終了していない)かどうかは IsAlive() で取得することができます。

不要になったスレッドを破棄するには Finalize() を呼び出してください。その際は、必ず下記の手順に従ってください。

Start() または TryStart() でスレッドを開始した場合は、スレッドを破棄する前に Join() を明示的に呼び出さなければなりません。Detach() は呼び出さないでください。

StartUsingAutoStack() または TryStartUsingAutoStack() でスレッドを開始した場合は、スレッドを破棄する前に Join() または Detach() を明示的に呼び出さなければなりません。Detach() を呼び出したあとは、そのスレッドに対して Finalize() とデストラクタ以外の呼び出しを行うことができなくなります。

9.4. スケジューリング

スレッドには優先度を設定することができ、スレッドの実行スケジュールは優先度によって決定されます。優先度は 0 ～ 31 の範囲で指定することができ、0 が一番高く、31 が一番低いことを表しています。標準的なスレッドには 16(DEFAULT_THREAD_PRIORITY)を指定します。

Yield() の呼び出しは、同じ優先度を持つ、ほかのスレッドに実行権を譲渡します。同じ優先度のスレッドが存在しなければ何も起きません。

Sleep() の呼び出しは、指定時間の間、スレッドを休止状態にします。

実行中のスレッドに割り込みによるスケジューリングが発生した場合、中断されたスレッドはスレッドキューの先頭に入れられ、なるべくスレッドが切り替わらないように制御されます。実行中のスレッドの優先度より低いまたは同じスレッドを新規に作成して実行しようとした場合、システムプロセスの動作が割り込むことでスケジューリングが直ちに行われず、スレッドが切り替わらない可能性があります。そのため、確実にスレッドを切り替える必要がある場合は、イベントを使用して待つことを推奨します。

> **注意:** Sleep() に短い時間を指定して連続呼び出しを行うことは、システムコア側に高い負荷をかけ、システム全体のパフォーマンスを低下させる要因になります。

9.5. パラメータの取得と変更

スレッド自身が持つパラメータとしては、スレッド ID、優先度があります。

各パラメータの取得には Get*() と GetCurrent*() の 2 種類のメンバ関数があり、前者はインスタンスの、後者はカレントスレッドのパラメータを取得するメンバ関数です。パラメータの変更にも同じように、Change*() と ChangeCurrent*() の 2 種類のメンバ関数があります。

カレントスレッド(メインスレッド)のオブジェクトは GetMainThread() の呼び出しで取得することができます。

スレッド ID

スレッドごとに異なる ID(bit32 型)が振られています。GetId() または GetCurrentId() による取得のみで、変更はできません。

優先度

スレッドスケジューリングの優先度(s32 型)を設定することができます。現在の優先度は、GetPriority() または GetCurrentPriority() を呼び出して取得することができます。優先度を変更するには、ChangePriority() または ChangeCurrentPriority() を呼び出してください。

9.6. スレッドローカルストレージ

uptr 型を格納することができるスレッドローカルストレージが、スレッドごとに 16 スロット用意されています。スレッドローカルストレージの使用は nn::os::ThreadLocalStorage クラスのインスタンスを生成することで予約することができますが、16 スロットを超えて予約しようとすると予約に失敗し、アプリケーションは PANIC で強制停止してしまいます。

スレッドローカルストレージへの値の設定は SetValue() で行い、値の取得は GetValue() で行います。スレッドの開始時に、スレッドローカルストレージの全スロットには 0 が設定されます。

また、スレッド終了時に呼び出されるコールバック関数を、nn::os::ThreadLocalStorage クラスの引数ありのコンストラクタで登録することができます。コールバック関数が呼ばれる際、スレッドローカルストレージの値が引数として渡されます。

9.7. 同期オブジェクト

スレッドセーフでないライブラリや共有リソースなどへのアクセスは、アプリケーションでスレッド間の同期を取って調停しなければなりません。SDK ではクリティカルセクションなどの同期オブジェクトを用意しています。

9.7.1. クリティカルセクション

クリティカルセクションは排他制御のための同期オブジェクトです。クリティカルセクションに侵入できるスレッドを一つだけに限定することで、同一リソースへの複数スレッドからのアクセスを禁止することができます。クリティカルセクションは後述するミューテックスよりもメモリ使用量は多くなりますが、ほとんどの場合高速に動作します。また、空きメモリがある限り、作成することのできる個数に上限はありません。

クリティカルセクションは nn::os::CriticalSection クラスで定義されています。インスタンスを生成して Initialize() または TryInitialize() で初期化したあと、Enter() または TryEnter() の呼び出しでクリティカルセクションに侵入してロックしようとします。クリティカルセクションがすでにロックされていると、Enter() でロックしようとした場合は呼び出したスレッドの実行がクリティカルセクションのロックが解除されるまでブロックされます。再帰ロックが可能ですので、ロックしたスレッドからのロック要求はブロックされずにネスト回数が 1 増加します。TryEnter() でロックしようとした場合はロックに成功したかどうかだけを返し、スレッドの実行はブロックされません。

クリティカルセクションのロックを解除するには、Leave() を呼び出します。ロックしたスレッドからの呼び出しでのみネスト回数が 1 減少し、ネスト回数が 0 になるまではロックが解除されません。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

nn::os::CriticalSection::ScopedLock クラスを利用すると、オブジェクトの生成からスコープを出るまでの間、クリティカルセクションをロックします。ロックの解除はスコープから外れたときに自動で行われます。

9.7.1.1. スレッド優先度の逆転

クリティカルセクションはスレッドの優先度継承を行いません。そのため、優先度の低いスレッド(A)がロックしているクリティカルセクションに対して高い優先度のスレッド(B)がロックを要求している場合、A よりも高くて B よりも低い優先度のスレッド(C)に B が実行を妨げられてしまいます。つまり、事実上 B の優先度が下がってしまい、B と C の優先度が逆転する状況になってしまいます。

9.7.2. ミューテックス

ミューテックスは排他制御のための同期オブジェクトです。クリティカルセクションと同じように、同一リソースへの複数スレッドからのアクセスを禁止することができます。クリティカルセクションとの違いは、スレッドの優先度継承を実装していることと、作成個数が 32 個に制限されていることです。優先度継承によって、優先度の低いスレッドがロックしているミューテックスに対して高い優先度のスレッドがロックを要求すると、ロックしているスレッドの優先度が一時的にロックを要求しているスレッドと同じ優先度に引き上げられます。

ミューテックスは nn::os::Mutex クラスで定義されています。インスタンスを生成して Initialize() または TryInitialize() で初期化したあと、Lock() または TryLock() の呼び出しでミューテックスをロックしようとします。ミューテックスがすでにロックされていると、Lock() でロックしようとした場合は呼び出したスレッドの実行がミューテックスのロックが解除されてロックに成功するまでブロックされます。再帰ロックが可能なミューテックスですので、同じスレッドからのロック要求があった場合はネスト回数が 1 増加します。TryLock() でロックしようとした場合はタイムアウトまでにロックが成功したかどうかだけを返し、スレッドの実行はブロックされません。

ミューテックスのロックを解除するには、Unlock() を呼び出しますが、Unlock() はロックしているスレッドでのみ呼び出すことができます。また、ネスト回数を 1 減少させ、ネスト回数が 0 になるまではロックが解除されません。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

nn::os::Mutex::ScopedLock クラスを利用すると、オブジェクトの生成からスコープを出るまでの間、ミューテックスをロックします。ロックの解除はスコープから外れたときに自動で行われます。

9.7.3. イベント

イベントはイベントの発生を通知する単純な同期オブジェクトです。イベントにはシグナル状態と非シグナル状態の 2 つの状態があり、イベントの発生は非シグナル状態からシグナル状態に遷移することで表されます。スレッドがイベントの発生を待つことで、スレッド間の同期を取ることができます。イベントの作成個数は 32 個に制限されています。

イベントは nn::os::Event クラスで定義されています。インスタンスを生成して Initialize() で初期化します。初期化の際には、イベントの種類を手動リセットイベントと自動リセットイベントから選択することができます。手動リセットイベントの場合、シグナル状態になるとその状態はクリアされるまで維持され、その間、そのイベントを待っているスレッドがすべて解放されます。自動リセットイベントの場合、シグナル状態になるとそのイベントを待っているスレッドのうちで優先順位の一番高いスレッドだけが解放され、すぐに非シグナル状態に戻ります。

Wait() の呼び出しでスレッドはイベント待ちになり、イベントがシグナル状態になるまで実行がブロックされます。タイムアウト時間を指定することができ、タイムアウトまでにイベントが発生したかどうかを確認することができます。タイムアウト時間に 0 を指定した場合はすぐに制御が戻り、ブロックされずにイベントの発生を確認することができます。

イベントをシグナル状態にするには Signal() を呼び出してください。イベントが手動リセットイベントで初期化されている場合は ClearSignal() を呼び出すまでシグナル状態がクリアされません。自動リセットイベントで初期化されている場合はイベント待ちのスレッドを 1 つだけ解放してシグナル状態は自動的にクリアされますが、イベント待ちのスレッドがない場合はシグナル状態のままとなります。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

9.7.3.1. ライトイベント

ライトイベントはスレッド間でフラグの通知を行う単純な同期オブジェクトで、機能的にはイベントとほとんど違いがありません。

ライトイベントは nn::os::LightEvent クラスで定義されています。nn::os::WaitObject::WaitAny() などで複数の同期オブジェクトを待つことができない点を除いては nn::os::Event クラスよりも優れていますので、なるべく nn::os::LightEvent クラスを使用してください。また、空きメモリがある限り、作成することのできる個数に上限はありません。

引数なしのコンストラクタで生成したインスタンスは Initialize() で初期化しなければなりません。初期化の際には、イベントの種類を手動リセットイベントと自動リセットイベントから選択することができます。それぞれの動作については nn::os::Event クラスと同じです。イベントの種類は引数ありのコンストラクタでも指定することができます。引数ありで生成したインスタンスは Initialize() による初期化は不要です。

Wait() でタイムアウト指定ができないことを除いて、Wait()、Signal()、ClearSignal()、Finalize() の各メンバ関数は nn::os::Event クラスにある同名のメンバ関数と同じ動作をします。

nn::os::LightEvent クラスには、以下のメンバ関数が追加されています。

TryWait() によるフラグのチェックを行うことができます。フラグの状態を返り値で取得するだけでスレッドの実行はブロックされません。また、自動リセットイベントのインスタンスならば、フラグがセット(true = シグナル状態)されていた場合にだけフラグをクリア(false = 非シグナル状態)します。なお、TryWait() ではタイムアウト時間を指定することができます。

Pulse() はフラグがセットされるの待っているスレッドの解放を行います。自動リセットイベントならば最も優先度の高いスレッドを 1 つだけ解放し、フラグをクリアします。Signal() と異なり、待っているスレッドがなくてもフラグはクリアされます。

手動リセットイベントの場合は待っているスレッドすべてを解放してからフラグをクリアします。

9.7.4. セマフォ

セマフォはカウンタを持った同期オブジェクトです。セマフォの取得要求があるとカウンタが 1 減少し、取得を要求したスレッドはカウンタが 0 より大きくなるのを待ちます。セマフォが解放されるとカウンタは 1 増加し、カウンタが 0 より大きくなるとセマフォ待ちをしているスレッドのうちで優先順位の一番高いスレッドにセマフォが渡されます。セマフォは、同時にアクセス可能なスレッドの数を制限するようなリソースの管理に利用することができます。セマフォの作成個数は 8 個に制限されています。

セマフォは nn::os::Semaphore クラスで定義されています。インスタンスを生成して Initialize() または TryInitialize() で初期化します。初期化の際には、カウンタの初期値と最大値を指定します。

セマフォの取得要求は Acquire() または TryAcquire() の呼び出しで行います。カウンタが 0 以下のときに Acquire() を呼び出した場合、セマフォを取得できるまでスレッドの実行がブロックされます。TryAcquire() の呼び出しではタイムアウト時間を指定することができ、タイムアウトまでにセマフォを取得できたかどうかを確認することができます。タイムアウト時間に 0 を指定した場合はすぐに制御が戻り、ブロックされずにセマフォの取得を試すことができます。

取得していたセマフォは Release() の呼び出しで解放することができます。カウンタの増加値に 1 以外を指定することもできますが、これは初期値を 0 で作成したインスタンスに初期値を与えるときに利用するためです。セマフォの解放時は引数なしで呼び出し、カウンタの増加値が 1 になるようにしてください。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

9.7.4.1. ライトセマフォ

ライトセマフォはセマフォと同じ機能を持った同期オブジェクトです。ライトセマフォは nn::os::LightSemaphore クラスで定義されています。nn::os::WaitObject::WaitAny() などで複数の同期オブジェクトを待つことができない点を除いては nn::os::Semaphore クラスよりも優れていますので、なるべく nn::os::LightSemaphore クラスを使用してください。また、空きメモリがある限り、作成することのできる個数に上限はありません。

セマフォとの違いは TryInitialize() が定義されていないことです。nn::os::LightSemaphore クラスのメンバ関数は nn::os::Semaphore クラスにある同名のメンバ関数と同じ動作をします。

9.7.5. ブロッキングキュー

ブロッキングキューはスレッド間でメッセージのやり取りを安全に行うことのできる同期オブジェクトです。キューのバッファサイズを超えて要素が挿入されたときや、要素のないキューから要素を取り出そうとしたときにスレッドの実行がブロックされます。ブロッキングキューは、スレッドが多数のスレッドからのメッセージを待つような一対多、多対多でのメッセージのやり取りに利用することができます。このブロッキングキューは NITRO/TWL や Revolution の SDK ではメッセージキューと呼ばれていた機能に相当します。

ブロッキングキューには、nn::os::BlockingQueue クラスと nn::os::SafeBlockingQueue クラスの 2 種類の定義があります。前者は内部でスレッド間の同期にクリティカルセクションを使用していますので、優先度の逆転が起こるような状況ではデッドロックが発生する可能性があります。そのような可能性がある場合はミューテックスを使用している後者のクラスを使用してください。両者のメンバ変数やメンバ関数に違いはありません。

インスタンスを生成して Initialize() または TryInitialize() で初期化します。初期化の際には、キューのバッファとして uptr 型の配列とその要素数を指定します。

キューの末尾に要素(uptr 型)を挿入するには Enqueue() または TryEnqueue() を呼び出します。キューが一杯になっているときに Enqueue() を呼び出した場合、要素を挿入できるまでスレッドの実行がブロックされます。

TryEnqueue() の呼び出しでは要素を挿入できたかどうかに関係なく制御が戻るため、ブロックされずに要素の挿入を試すことができます。

キューの先頭に要素を挿入するには Jam() または TryJam() を呼び出します。キューが一杯になっているときに Jam() を呼び出した場合、要素を挿入できるまでスレッドの実行がブロックされます。TryJam() の呼び出しでは要素を挿入できたかどうかに関係なく制御が戻るため、ブロックされずに要素の挿入を試すことができます。

キューの先頭から要素を取り出すには Dequeue() または TryDequeue() を呼び出します。キューが空の状態で Dequeue() を呼び出した場合、要素を取り出せるまでスレッドの実行がブロックされます。TryDequeue() の呼び出しでは要素を取り出せたかどうかに関係なく制御が戻るため、ブロックされずに要素の取り出しを試すことができます。

GetFront() または TryGetFront() を呼び出した場合もキューの先頭の要素を取得することができますが、キューの状態を変更する(要素を取り出す)ことはありません。キューが空の状態で GetFront() を呼び出した場合、要素を取得できるまで(要素が挿入されるまで)スレッドの実行がブロックされます。TryGetFront() の呼び出しでは要素を取得できたかどうかに関係なく制御が戻るため、ブロックされずに要素の取得を試すことができます。

インスタンスを明示的に破棄する場合は Finalize() を呼び出してください。

9.7.6. ライトバリア

ライトバリアは複数のスレッドが到着するのを待つ同期オブジェクトです。初期化時に指定された数のスレッドが到着するまで早く到着したスレッドを待機させますが、このクラスを N 個のスレッドの中から M 個(M < N)の到着を待つために使用することはできません。

ライトバリアは nn::os::LightBarrier クラスで定義されています。引数なしのコンストラクタで生成したインスタンスは Initialize() で初期化しなければなりません。初期化の際には待ち合わせるスレッドの数を指定します。待ち合わせるスレッドの数は引数ありのコンストラクタでも指定することができます。引数ありで生成したインスタンスは Initialize() による初期化は不要です。空きメモリがある限り、作成することのできるライトバリアの個数に上限はありません。

Await() はほかのスレッドが到着を待ちます。この関数を呼び出したスレッドの数が待ち合わせるスレッドの数になるまで、スレッドの実行がブロックされます。

9.7.7. デッドロック

クリティカルセクション、ミューテックス、セマフォを利用して複数スレッド間の排他制御を行う際には、デッドロックに陥らないように注意しなければなりません。

例えば、リソースのアクセスに 2 つのミューテックス X と Y を必要とするスレッド A と B があると仮定します。A が X をロックし、B が Y をロックした場合、A と B は永久にブロックされたまま(デッドロック)になってしまいます。

デッドロックにならないようにする単純な方法としては、リソースを必要とするスレッドすべてが同じ順番でミューテックスのロックを要求することです。

9.8. リソースの上限

スレッドや同期オブジェクト、タイマなどは、アプリケーションで同時に生成可能なインスタンスの数が制限されています。

生成可能なインスタンスの数が制限されているクラスには、以下のクラスが該当します。

- nn::os::Thread クラス(参照:「9.1. 初期化と開始」～「9.6. スレッドローカルストレージ」)
- nn::os::Event クラス(参照:「9.7.3. イベント」)
- nn::os::Mutex クラス(参照:「9.7.2. ミューテックス」)
- nn::os::Semaphore クラス(参照:「9.7.4. セマフォ」)

- nn::os::Timer クラス(参照:「8.3. タイマ」)

上記のクラスには共通して、リソースの使用数や上限を取得する、以下のメンバ関数が定義されています。

コード 9-2. リソースの使用数、上限を取得するメンバ関数

```
static s32 GetCurrentCount();
static s32 GetMaxCount();
```

GetCurrentCount() は、インスタンスの現在の使用数を返します。

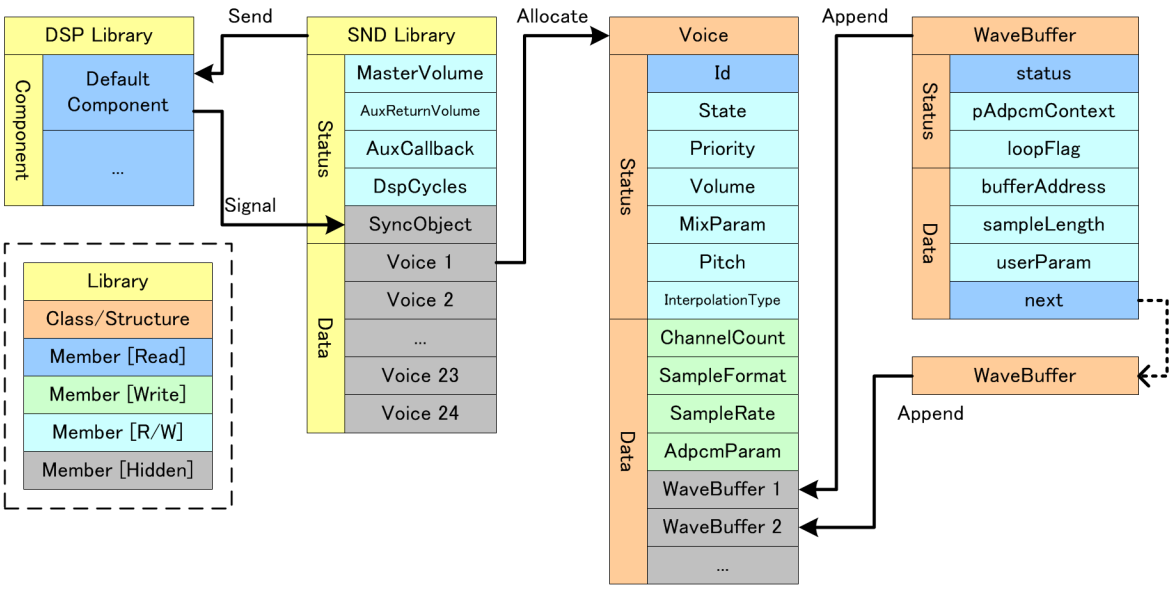
GetMaxCount() は、インスタンスの生成可能数の上限値を返します。

アプリケーションが起動した時点から、ライブラリなどが内部的にこれらのインスタンスを生成しているため、アプリケーションで使用可能なリソースはさらに制限される可能性があります。詳しくは、「システムプログラミングガイド」の「リソース上限」や各ライブラリの説明を参照してください。

10. サウンド

CTR-SDK のサウンドライブラリは、DSP にロードされたコンポーネントと通信することでサウンドの再生を制御します。サウンドライブラリには 24 個のボイスオブジェクトが用意されており、アプリケーションはそこから必要な数のボイスオブジェクトを確保し、ウェーブバッファ(音源データ情報)を登録することでサウンドの再生を行うことができます。

図 10-1. サウンド再生に関係するライブラリとクラスの関連図



10.1. 初期化

サウンドを再生するには DSP を使用しなければなりません。そのため、サウンド再生に使用する SND ライブラリを初期化する前に、DSP ライブラリの初期化とサウンド再生のためのコンポーネントのロードが必要となります。

DSP ライブラリの初期化は `nn::dsp::Initialize()` で行い、`nn::dsp::LoadDefaultComponent()` でサウンドの再生を行うコンポーネントをロードします。

SND ライブラリの初期化は `nn::snd::Initialize()` で行います。

コード 10-1. DSP、SND ライブラリの初期化

```
nn::Result result;
result = nn::dsp::Initialize();
result = nn::dsp::LoadDefaultComponent();
result = nn::snd::Initialize();
```

10.1.1. 出力バッファ数の設定

DSP から出力される最終的なサウンドデータの出力バッファの数を `nn::snd::SetOutputBufferCount()` で指定することができます。

コード 10–2. 出力バッファ数の設定

```
void nn::snd::SetOutputBufferCount(s32 count);
s32 nn::snd::GetOutputBufferCount();
```

count には 2 または 3 を指定することができます。デフォルトの設定は 2 です。

出力バッファの数を 2 に設定した場合は、音源データが実際に発音されるまでの遅延時間は最小となりますが、サウンドスレッドの処理に遅延が発生したときに発音が途切れることがあります。

出力バッファの数を 3 に設定した場合は、2 に設定した場合に比べて発音が約 5 ms 遅延しますが、サウンドスレッドの処理が多少遅延したときでも発音が途切れることがありません。

この関数は出力バッファの内容をクリアしますので、ライブラリの初期化直後などの無音状態で呼び出してください。

現在の設定は nn::snd::GetOutputBufferCount() で取得することができます。

10.2. ボイスオブジェクトの確保

SND ライブラリは、自身の持つ 24 個のボイスオブジェクトに結び付けられた音源データ情報をもとにサウンドを合成し、再生します。つまり、最大で 24 の音源データを同時に再生することができます。

ボイスオブジェクトは有限のリソースですので、アプリケーションは再生したい音源データに結びつけるボイスオブジェクトを、nn::snd::AllocVoice() でライブラリから確保しなければなりません。

コード 10–3. ボイスオブジェクトの確保

```
nn::snd::Voice* nn::snd::AllocVoice(
        s32 priority, nn::snd::VoiceDropCallbackFunc callback, uptr userArg);
```

priority には、確保するボイスオブジェクトの優先順位を 0 ～ nn::snd::VOICE_PRIORITY_NODROP(0x7FFF = 32767)の範囲で指定します。値が大きいほど優先度は高くなります。範囲外の値を指定した場合は、アサートで停止します。

priority の指定によって、すでにボイスオブジェクトが最大数まで確保されているときの動作が異なります。

priority が nn::snd::VOICE_PRIORITY_NODROP の場合、確保済みのボイスオブジェクトのうちで最低の優先順位が同じ nn::snd::VOICE_PRIORITY_NODROP ならばボイスオブジェクトの確保に失敗して NULL が返されます。それ以外の場合、確保済みのボイスオブジェクトのうちで最低の優先順位であるボイスオブジェクトが解放(ドロップ)されます。確保しようとしているボイスオブジェクトの優先順位が一番低い場合は確保に失敗します。

callback には、ライブラリが強制的にボイスオブジェクトをドロップしたときに呼び出されるコールバック関数を指定します。NULL を指定した場合はドロップ時にコールバックが発生しません。*userArg* にはコールバック時に渡される引数を指定します。不要な場合は NULL を渡してください。

コールバック関数の型は以下のように定義されています。

コード 10–4. ボイスオブジェクトのドロップコールバック関数

```
typedef void (* VoiceDropCallbackFunc)(nn::snd::Voice *pVoice, uptr userArg);
```

pVoice はドロップされたボイスオブジェクトへのポインタ、*userArg* は確保時に指定した引数です。

10.2.1. ボイスドロップ処理のモード

SND ライブラリは、ボイスオブジェクトが上限を超えて確保されようとしたときや、DSP 内部での処理負荷が増大する恐れがあるときに優先度の低いボイスオブジェクトを解放(ドロップ)します。

このうち、処理負荷による制御には 2 つのモードが用意されており、nn::snd::SetVoiceDropMode() で設定することができます。

コード 10-5. ボイスドロップ処理のモード設定

```
void nn::snd::SetVoiceDropMode(nn::snd::VoiceDropMode mode);
```

mode に VOICE_DROP_MODE_DEFAULT を指定した場合は、予測値のみを参考にしてドロップ処理を行います。

VOICE_DROP_MODE_REAL_TIME を指定した場合は、予測値に加えて実際の処理負荷を参考にしてドロップ処理を行います。ただし、このモードに設定する場合は出力バッファ数が 3 に設定されていなければなりません。

10.3. 音源データ情報の設定

音源データを格納するための領域(バッファ領域)は、デバイスメモリ上の領域から 32 Byte アライメントの連続した領域で確保しなければなりません。バッファ領域は 32 Byte アライメント、サイズが 32 Byte の倍数でなければなりません。その中から音源データのバッファを確保する際は、先頭アドレスが 32 Byte アライメントの連続した領域でなければならない制限はそのままですが、サイズは 32 Byte の倍数でなくてもかまいません。

バッファに書き込んだ音源データは、そのままではキャッシュにしか書き込まれていない場合があります。必ず nn::snd::FlushDataCache() を呼び出して、メモリに書き込まれている状態にしてください。

音源データは音源データ情報の構造体 nn::snd::WaveBuffer を介して DSP に渡します。音源データ情報は、音源データに関する情報を設定する前に、nn::snd::InitializeWaveBuffer() で初期化しなければなりません。再生を完了した音源データ情報を再利用する場合も同様です。

初期化した音源データ情報に設定する情報は、バッファの先頭アドレス(bufferAddress)、サンプル長(sampleLength)、ループ指定フラグ(loopFlag)の各メンバ変数に設定します。サンプルフォーマットが DSP ADPCM の場合は、ADPCM コンテキスト(pAdpcmContext)への設定も行ってください。ユーザーパラメータ(userParam)には任意のデータを設定することができます。そのほかのメンバは変更しないでください。

音源データに関する、そのほかの情報(チャンネル数、サンプルフォーマット、サンプリングレート、ADPCM パラメータの基本的な情報)はボイスオブジェクトに設定します。

チャンネル数は nn::snd::Voice::SetChannelCount() で設定することができます。モノラルのデータならば 1 を、ステレオのデータならば 2 を設定してください。それ以外の値は無効です。

サンプルフォーマットは nn::snd::Voice::SetSampleFormat() で設定することができます。SND ライブラリが対応している音源データのフォーマットは 8 bit / 16 bit PCM と DSP ADPCM の 3 フォーマットです。

表 10-1. サンプルフォーマット一覧

値	種別	ステレオ再生
SAMPLE_FORMAT_PCM8	8 bit PCM	interleaved
SAMPLE_FORMAT_PCM16	16 bit PCM	interleaved
SAMPLE_FORMAT_ADPCM	DSP ADPCM	不可

サンプルフォーマットが DSP ADPCM の場合、ADPCM パラメータを nn::snd::Voice::SetAdpcmParam() で設定してください。また、音源データのヘッダに格納されている ADPCM コンテキストデータの構造体へのアドレスを nn::snd::WaveBuffer の pAdpcmContext メンバ変数に設定し、設定したコンテキストデータはバッファの再生が終了するまで変更しないでください。

サンプリングレートは nn::snd::Voice::SetSampleRate() で設定することができます。音源データの周波数そのままの値を設定してください。

音声データの補間方法は nn::snd::Voice::SetInterpolationType() で設定することができ、現在の設定は nn::snd::Voice::GetInterpolationType() で取得することができます。対応している補間方法は以下の 3 種類です。デフォルトでは INTERPOLATION_TYPE_POLYPHASE が設定されています。

表 10-2. 補間方法一覧

値	補間方法
INTERPOLATION_TYPE_POLYPHASE	4 点による補間を行います。設定されたサンプルレートとピッチから、最適な係数が選択されます。
INTERPOLATION_TYPE_LINEAR	線形補間を行います。
INTERPOLATION_TYPE_NONE	補間を行いません。音源データの周波数が 32728 Hz ではない場合、再生されるサウンドにノイズが発生します。

音源データに関する情報をすべて設定したあと、nn::snd::Voice::AppendWaveBuffer() でボイスオブジェクトに音源データ情報を登録します。複数の音源データ情報を続けて登録することができますが、1 サウンドフレーム(約 4.889 ミリ秒)のうちに 1 つのボイスオブジェクトで再生される音源データ情報は最大で 4 つです。

登録後の音源データの状態は nn::snd::WaveBuffer の Status メンバ変数で確認することができます。ボイスオブジェクトの登録前ならば STATUS_FREE、再生待ちならば STATUS_WAIT、再生中ならば STATUS_PLAY、再生完了ならば STATUS_DONE です。このうち、STATUS_WAIT と STATUS_PLAY のときは、音源データがボイスオブジェクトの管理下におかれていますので、設定などを変更しないでください。

nn::snd::Voice::UpdateWaveBuffer() を呼び出して、登録されている音源データ情報を変更することができます。ただし、変更可能な情報はサンプル長(sampleLength)とループ指定フラグ(loopFlag)のみで、SendParameterToDsp() による更新のタイミングによっては指定されたサンプル長よりもあとのデータが再生されたり、すでに再生が完了しているために変更が無効になったりすることがあります。

登録された音源データ情報は nn::snd::Voice::DeleteWaveBuffer() で削除することができます。SendParameterToDsp() による更新で、状態が再生完了になるまで情報を書き換えないようにしてください。

DSP ADPCM の音源データをストリーム再生する場合、pAdpcmContext メンバ変数への設定はボイスオブジェクトに最初に登録される音源データのみでかまいません。そのあとに登録される音源データの pAdpcmContext メンバ変数が NULL の場合、コンテキストの更新が行われません。

10.4. サウンドスレッドの作成

アプリケーションでスレッド(サウンドスレッド)を作成し、そのスレッド内で DSP ライブラリから SND ライブラリへの通知を待つように設計してください。これは約 4.889 ミリ秒ごとに通知が行われるためです。また、作成するスレッドの優先度はなるべく高く設定し、スレッド内では重い処理を行わないようにしなければ、サウンドの再生が途切れる可能性があります。

サウンドスレッド内では、基本的に以下の処理をループさせることになります。

1. DSP ライブラリからの通知を nn::snd::WaitForDspSync() で待ちます。
2. アプリケーションは確保しているボイスオブジェクトに登録された音源データ情報の状態を確認し、再生が完了しているならば次に再生する音源データ情報を nn::snd::Voice::AppendWaveBuffer() で登録します。その際、サウンドスレッド内では音源データ情報の登録だけを行い、サウンドの再生に使用する音源データはあらかじめ別のスレッドでバッファにロードするようにしてください。
3. nn::snd::SendParameterToDsp() を呼び出し、前回の通知からこれまでに変更されたパラメータや新たに登録された音源データ情報などを DSP に対して送信し、サウンドの再生に反映させます。

注意: nn::snd::AllocVoice() と nn::snd::FreeVoice() を除いて、ボイスオブジェクトを操作する関数はスレッドセーフではありません。サウンドスレッド以外のスレッドで呼び出す場合は、クリティカルセクションなどで排他制御を行う必要があります。

10.5. サウンドの再生

サウンドの再生・停止・一時停止はボイスオブジェクトの状態を変更することで行います。状態の設定と取得はそれぞれ nn::snd::Voice::SetState() と nn::snd::Voice::GetState() で行うことができます。サウンドの再生を開始または一時停止状態を解除するときは STATE_PLAY を、停止するときは STATE_STOP を、一時停止するときは STATE_PAUSE を、それぞれ設定してください。

表 10-3. ボイスオブジェクトの状態

状態	説明
STATE_PLAY	サウンドの再生を指令する、または再生中です。
STATE_STOP	サウンドの停止を指令する、または停止しています。
STATE_PAUSE	サウンドの一時停止を指令する、または一時停止中です。STATE_PLAY に変更することで続きから再生されます。

状態の変更はすぐに反映されるわけではありません。nn::snd::SendParameterToDsp() によってパラメータが送信されるまで、つまりサウンドフレームの更新間隔(約 4.889 ミリ秒)の遅延が発生する可能性があります。

再生に関する設定は、状態以外にもいくつかあります。

優先順位

ボイスオブジェクトの優先順位を設定するには、nn::snd::Voice::SetPriority() を呼び出します。現在の設定を取得するには、nn::snd::Voice::GetPriority() を呼び出してください。再生は優先順位の高いボイスオブジェクト(同じ優先順位ならば後に確保したもの)から行われるため、処理負荷が要因となって優先順位の低いボイスが再生されない可能性があります。

マスターボリューム

マスターボリュームは SND ライブラリ全体のボリュームです。nn::snd::SetMasterVolume() で設定することができ、1.0 で等倍に設定されます。現在の設定を取得するには、nn::snd::GetMasterVolume() を呼び出してください。

ボリューム

ボイスオブジェクトそれぞれのボリュームです。nn::snd::Voice::SetVolume() で設定することができます。現在の設定を取得するには、nn::snd::Voice::GetVolume() を呼び出してください。ボリュームは 1.0 で等倍に設定されま

す。

ミックスパラメータ

ミックスパラメータは 4 チャンネル(FrontLeft / FrontRight / RearLeft / RearRight)それぞれのゲインのことです。

nn::snd::Voice::SetMixParam() で設定することができます。現在の設定を取得するには、nn::snd::Voice::GetMixParam() を呼び出してください。nn::snd::MixParam 構造体のそれぞれのメンバにゲインの値が格納されます。ゲインは 1.0 で等倍に設定されます。

サウンドの再生ボリュームは、マスターボリューム、ボリューム、ミックスパラメータの設定値が乗算されたものとなります。

ピッチ

ピッチは元の音源データのサンプリングレートに対する速度比のことです。nn::snd::Voice::SetPitch() で設定することができます。現在の設定を取得するには、nn::snd::Voice::GetPitch() を呼び出してください。ピッチは 1.0 で等倍に設定されます。サンプリングレートが 32 kHz の音源データに対して 0.5 を設定した場合、音源データは 16 kHz で再生されます。

問い合わせ

サウンドが再生中であるかどうかは nn::snd::Voice::IsPlaying() の呼び出しで判別することができます。

現在の再生位置は、nn::snd::Voice::GetPlayPosition() で取得することができます。返される値は、DSP によって次に処理されるメインメモリ上のデータの先頭位置(サンプル数)です。

AUX バス

A と B の 2 つの AUX バスを利用して、再生されるサウンドにディレイなどのエフェクトをかけることができます。AUX バスのボリュームは nn::snd::SetAuxReturnVolume() と nn::snd::GetAuxReturnVolume() で設定および取得を行うことができます。ボリュームは 1.0 で等倍に設定されます。ミックスパラメータには AUX バスのゲインを設定するメンバがありますので、AUX バスへ送られる音声データをチャンネルごとに調整することができます。

AUX バスでのサウンドの加工はそれぞれのバスに設定されたコールバック関数で行います。コールバック関数の設定は nn::snd::RegisterAuxCallback() で行うことができ、解除は nn::snd::ClearAuxCallback() で行うことができます。現在設定されているコールバック関数の取得は nn::snd::GetAuxCallback() で行うことができます。コールバック関数の型は以下のように定義されています。

コード 10–6. AUX バスのコールバック関数

```
typedef void(*AuxCallback)(nn::snd::AuxBusData* data, s32 sampleLength,
                           uptr userData);
```

data にはサンプル長(バイトサイズではありません)が *sampleLength* の音声データが渡されます。チャンネルごとにバッファのアドレスが設定されており、それぞれのバッファを書き換えることで加工した音声データを AUX バスの出力として再生することができます。

ボイスの補間方法

音源データのサンプリングレートから再生されるサウンドの周波数へと変換される際の補間方法を設定することができます。nn::snd::Voice::SetInterpolationType() と nn::snd::Voice::GetInterpolationType() で補間方法の設定と取得を行うことができます。

補間方法は、デフォルトの 4 点補間(INTERPOLATION_TYPE_POLYPHASE)、線形補間(INTERPOLATION_TYPE_LINEAR)、補間なし(INTERPOLATION_TYPE_NONE)から選択します。4 点補間では、ライブラリがサンプリングレートとピッチから最適な係数を選択します。

フィルタ

ボイスオブジェクトごとにフィルタを適用することができます。フィルタは nn::snd::Voice::SetFilterType() と nn::snd::Voice::GetFilterType() で設定および取得を行うことができます。設定可能なフィルタには、単極フィルタ(FILTER_TYPE_MONOPOLE)と双極フィルタ(FILTER_TYPE_BIQUAD)があり、デフォルトはフィルタなし(FILTER_TYPE_NONE)に設定されています。また、それぞれのフィルタの係数を設定する関数が用意されています。

単極フィルタの係数は nn::snd::Voice::SetMonoFilterCoefficients() と nn::snd::Voice::GetMonoFilterCoefficients() で設定および取得を行うことができます。係数そのままを設定することも、カットオフ周波数を指定して単極ローパスフィルタとして設定することもできます。

双極フィルタの係数は nn::snd::Voice::SetBiquadFilterCoefficients() と nn::snd::Voice::GetBiquadFilterCoefficients() で設定および取得を行うことができます。こちらは係数による設定のみに対応しています。

係数による設定では、ベースとなる周波数を 32728 Hz としてそれぞれのフィルタの係数を計算してください。

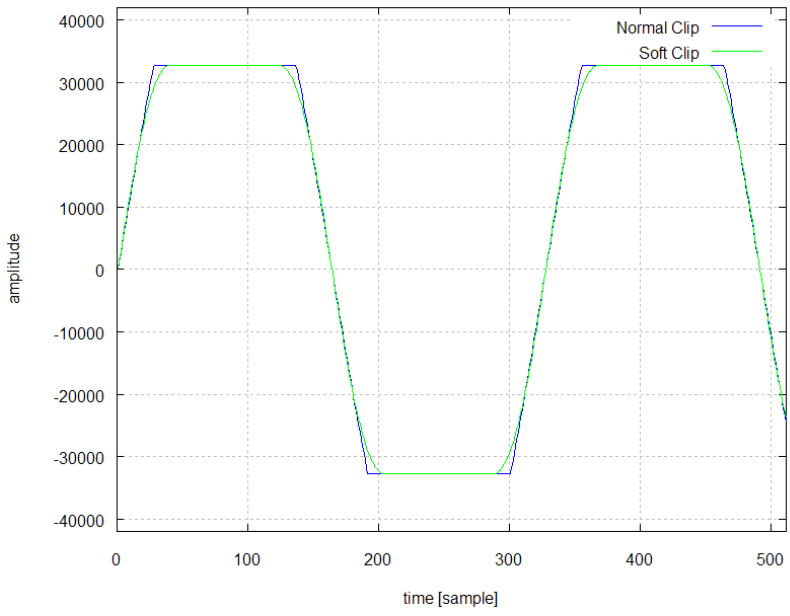
クリッピングモード

最終的なサウンド出力は 16 bit をオーバーフローした部分がクリッピングされたものになります。クリッピングの方法として、通常のクリッピングと高周波ノイズが軽減されるソフトクリッピングを利用することができます。クリッピング方法の指定は、nn::snd::SetClippingMode() でクリッピングモードの設定を行います。通常のクリッピングを指定する場合は CLIPPING_MODE_NORMAL を、ソフトクリッピングを指定する場合は CLIPPING_MODE_SOFT を指定します。現在の設定は nn::snd::GetClippingMode() で取得することができ、デフォルトはソフトクリッピングが設定されています。

ソフトクリッピングを用いた場合は非線形のクリップ処理が行われるため、通常のクリップ処理で発生する音割れを緩和することができますが、最大振幅に達していないサンプル値も修正されてしまいます。通常の音楽など、多くの場合はほとんど影響がありませんが、単一周波数のサイン波を入力した場合などに倍音成分が付加される可能性があります。

下図は、振幅の大きなサイン波がそれぞれのクリッピングモードでクリップ処理された結果の波形です。緑の線(Soft Clip)がソフトクリッピングモードによるクリップ処理の結果を示しています。

図 10-2. クリッピングモードによるクリップ処理結果の違い



自動フェードイン

再生開始時のボリュームを 0 から設定されたボリュームまで 1 サウンドフレームかけて変化させ、自動的にフェードインさせることができます。フェードイン処理は nn::snd::Voice::SetStartFrameFadeInFlag() でボイスオブジェクトごとに設定することができ、true に設定することで処理が行われるようになります。デフォルトは false(フェードイン処理を行わない)に設定されています。

bcwav ファイルの利用

nn::snd::Voice クラスの SetupBcwav() を使用することで、ctr_WaveConverter でコンバートされた波形ファイル(bcwav ファイル)を簡単にサウンドの再生に利用することができます。

コード 10-7. bcwav ファイルの利用

```
bool nn::snd::Voice::SetupBcwav(uptr addrBcwav,
  nn::snd::WaveBuffer* pWaveBuffer0, nn::snd::WaveBuffer* pWaveBuffer1,
  nn::snd::Bcwav::ChannelIndex channelIndex = nn::snd::Bcwav::CHANNEL_INDEX_L);
```

addrBcwav には、bcwav ファイルを読み込んだバッファの先頭アドレスを指定します。バッファに読み込まれた内容はそのまま音源データとして利用しますので、バッファは音源データと同様にデバイスメモリから 32 Byte アライメントの先頭アドレス、32 Byte の倍数で確保し、必ず読み込んだ内容を nn::snd::FlushDataCache() でメモリに書き込まれている状態にしてください。

pWaveBuffer0 と *pWaveBuffer1* には、初回再生時とループ再生時に使用する nn::snd::WaveBuffer 構造体へのポインタを指定します。構造体の実体はアプリケーションで用意しなければなりません。ループフラグの設定されていない音源であることが分かっている場合は、*pWaveBuffer1* に NULL を渡すことができます。

channelIndex には、音源データがステレオフォーマットである場合に、Voice クラスにどちらのチャンネルの音源データを割り当てるのかを指定します。音源データがモノラルである場合、この引数には CHANNEL_INDEX_L を指定しなければなりません。現在、ctr_WaveConverter でコンバートされたステレオの音源データは interleaved ではありません。そのため、ステレオの bcwav ファイルを再生するには、Voice クラスが 2 つ必要になります。

処理に成功し、再生準備が整った場合は true が返されます。bcwav ファイルのヘッダにないパラメータのうち、ボリュームとピッチには 1.0 が自動的に設定されます。ミックスパラメータの設定は行われません。

bcwav ファイルのヘッダ情報などにアクセスする場合は、nn::snd::Bcwav クラスを利用してください。

エフェクト(ディレイ、リバーブ)

サウンドの再生にエフェクトをかけるには、AUX バスのコールバック関数を利用してアプリケーションで実装する以外にも、ライブラリで用意されているエフェクトを適用することができます。

SND ライブラリでは、ディレイ(nn::snd::FxDelay クラス)とリバーブ(nn::snd::FxReverb クラス)を用意しています。これらのエフェクトを利用する場合は、それぞれのクラスのインスタンスを生成し、パラメータの設定とワークメモリの確保を行ってから、そのインスタンスとエフェクトを適用する AUX バスを指定して nn::snd::SetEffect() を呼び出します。1 つのバスには 1 種類のエフェクトのみ適用することができ、複数回呼び出した場合は最後に呼び出したエフェクトが適用されます。エフェクトの処理は DSP ではなく CPU で行われます。

エフェクトを解除する場合は、AUX バスを指定して nn::snd::ClearEffect() を呼び出してください。この関数はエフェクトのみを解除しますので、AUX バスのコールバック関数は解除されません。

> **補足:** エフェクトのパラメータについては、関数リファレンスを参照してください。

10.5.1. サウンド出力モード

SND ライブラリのサウンド出力のモードにはモノラル、ステレオ、サラウンドの 3 種類のモードが用意されており、`nn::snd::SetSoundOutputMode()` と `nn::snd::GetSoundOutputMode()` で設定と現設定の取得を行うことができます。

コード 10-8. サウンド出力モードの定義と設定および取得の関数

```
typedef enum
{
    OUTPUT_MODE_MONO       = 0,
    OUTPUT_MODE_STEREO     = 1,
    OUTPUT_MODE_3DSURROUND = 2
} nn::snd::OutputMode;

bool nn::snd::SetSoundOutputMode(nn::snd::OutputMode mode);
nn::snd::OutputMode nn::snd::GetSoundOutputMode(void);
```

`nn::snd::SetSoundOutputMode()` の返り値が `true` ならば処理に成功し、サウンド出力モードが *mode* で指定されたモードに設定されます。デフォルトのサウンド出力モードは `OUTPUT_MODE_STEREO`(ステレオ)に設定されています。

10.5.1.1. モノラル

全ボイスと AUX バスの出力がミックスされた 4 チャンネル(FrontLeft / FrontRight / RearLeft / RearRight)に対して、以下のように処理を行った結果を左右のスピーカーから出力します。

Output = (FrontLeft + FrontRight + RearLeft + RearRight) * 0.5

10.5.1.2. ステレオ

全ボイスと AUX バスの出力がミックスされた 4 チャンネル(FrontLeft / FrontRight / RearLeft / RearRight)に対して、以下のように処理を行った結果を左右のスピーカーからそれぞれ出力します。

OutputLeft = (FrontLeft + RearLeft)

OutputRight = (FrontRight + RearRight)

10.5.1.3. サラウンド

全ボイスと AUX バスの出力がミックスされた 4 チャンネル(FrontLeft / FrontRight / RearLeft / RearRight)に対して 3D サラウンド処理を行い、出力に空間的な広がりを与えます。3D サラウンド処理は仮想スピーカーの位置とかかり具合のパラメータからスピーカー出力を計算します。ボイス単位ではなく、ミックス後の出力に対して 3D サラウンド処理が行われますが、フロントチャンネル(FrontLeft と FrontRight)の出力に関しては、3D サラウンド処理の影響を受けないようにボイス単位でバイパスすることができます。

3D サラウンド処理が行われると、パンの位置を中心に設定したときよりも、左右いずれかの端に設定したときに音量が大きく感じられる傾向があります。この音量の違いが気になる場合は、左右の端におけるミックスパラメータの上限を 0.8 程度に制限するなどの補正を行うことで解消される可能性があります。

仮想スピーカーの位置

仮想スピーカー位置のモードを `nn::snd::SetSurroundSpeakerPosition()` で指定します。関数の呼び出しによるモードの指定に成功した場合は `true` が返されます。

コード 10-9. 仮想スピーカー位置の設定

```
bool nn::snd::SetSurroundSpeakerPosition(nn::snd::SurroundSpeakerPosition pos);
```

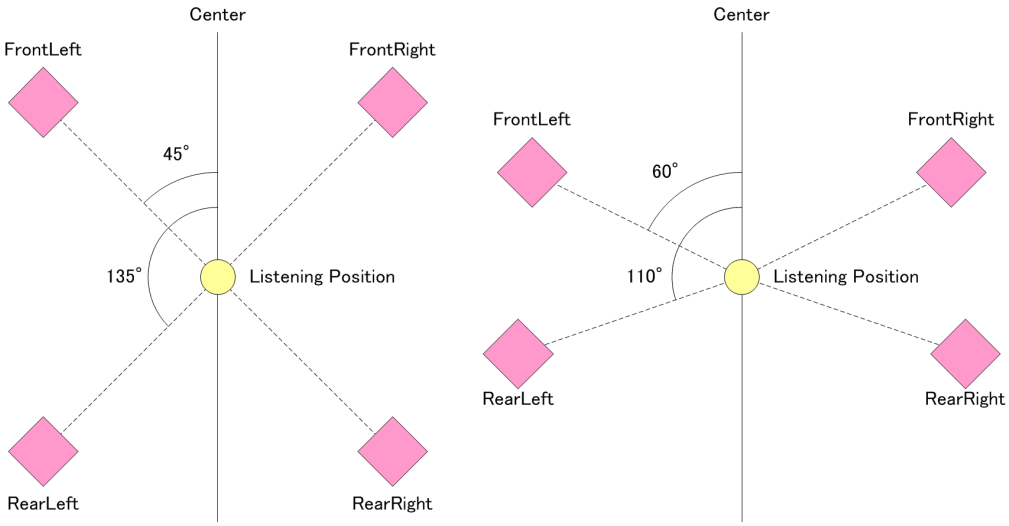
pos に指定する値は以下のモードから選択します。

表 10-4. 仮想スピーカー位置のモード

設定値	説明
SURROUND_SPEAKER_POSITION_SQUARE	Square モード。正方形を形作るように配置されます。
SURROUND_SPEAKER_POSITION_WIDE	Wide モード。やや左右に広がる形に配置されます。

仮想スピーカーのモードによって、4 つのチャンネルがどのような角度で左右対称に配置するのかが決定します。下図は、それぞれのモードで仮想スピーカーがどのような角度で配置されるのかを示したものです。

図 10-3. 仮想スピーカー位置のモードと仮想スピーカーの配置



SURROUND_SPEAKER_POSITION_SQUARE **SURROUND_SPEAKER_POSITION_WIDE**

かかり具合（サラウンドのデプス値）

nn::snd::SetSurroundDepth() で指定するデプス値によって、サラウンド効果の大小を変化させることができます。関数の呼び出しによるデプス値の指定に成功した場合は true が返されます。

コード 10-10. サラウンドのデプス値の設定

```
bool nn::snd::SetSurroundDepth(f32 depth);
```

depth には 0.0 ～ 1.0 の範囲でデプス値を指定します。0.0 で効果が最小、1.0 で効果が最大になります。デフォルトは 1.0 に設定されています。

3D サラウンド処理は、出力がスピーカーかヘッドホンかに応じて自動的に異なる処理を行います。このデプス値はスピーカーから出力されるときには有効ですが、ヘッドホンから出力されるときには無効となり、サラウンド効果が変化しません。

フロントバイパス設定

リアチャンネルの出力は必ず 3D サラウンド処理の影響を受けますが、フロントチャンネルの出力が 3D サラウンドの影響を受けないように、バイパスするかどうかをボイスオブジェクト単位で設定することができます。バイパスの設定は

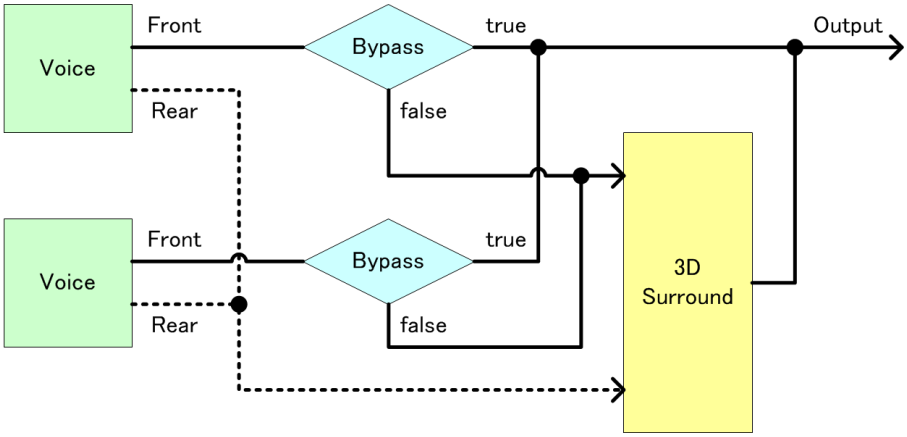
nn::snd::Voice クラスのメンバ関数 SetFrontBypassFlag() で行うことができます。

コード 10-11. ボイスオブジェクトのフロントバイパス設定

```
void nn::snd::Voice::SetFrontBypassFlag(bool flag);
```

フロントチャンネルの出力をバイパスさせる場合は、*flag* に true を渡してください。デフォルトは false(バイパスしない)に設定されています。

図 10-4. フロントバイパス設定と 3D サラウンド処理の影響



フロントバイパス設定は AUX バスのフロントチャンネルに対しても行うことができます。

コード 10-12. AUX バスのフロントバイパス設定

```
bool nn::snd::SetAuxFrontBypass(nn::snd::AuxBusId busId, bool flag);
```

busId には AUX バスの ID を、*flag* にはバイパスさせる場合に true を渡します。デフォルトは、全バスが false(バイパスしない)に設定されています。

10.5.2. ヘッドホンの接続状態

ヘッドホン端子にヘッドホンが接続されているかどうかを nn::snd::GetHeadphoneStatus() または nn::snd::UpdateHeadphoneStatus() で判断することができます。ヘッドホンが接続されているときは、返り値に true が返されます。

GetHeadphoneStatus() はサウンドスレッドで呼び出される SendParameterToDsp() 内で定期的に更新される状態からヘッドホンの接続状況を返すため、最大で 32 サウンドフレーム(約 160 ms)古い情報である場合があります。リアルタイム性が要求される状況では UpdateHeadphoneStatus() を呼び出してください。ただし、この関数は DSP の状態を更新するため、処理負荷が小さいものではないことに注意が必要です。

コード 10-13. ヘッドホンの接続状態

```
bool nn::snd::GetHeadphoneStatus();
bool nn::snd::UpdateHeadphoneStatus();
```

10.5.3. 出力される音声データの取得

最終的にスピーカーから出力される、ミックス済みの音声データをバッファに取り込むことができます。

コード 10-14. 出力される音声データの取得

```
bool nn::snd::GetMixedBusData(s16* pData,
                              s32 nSamplesPerFrame = NN_SND_SAMPLES_PER_FRAME);
```

pData には音声データを格納するバッファを指定します。バッファの先頭アドレスは nn::snd::MIXED_BUS_DATA_ALIGNMENT (4 Byte) アライメントでなければなりません。必要となるバッファのサイズは sizeof(s16) * *nSamplesPerFrame* * 2 で計算することができます。

nSamplesPerFrame には、1 チャンネルあたりのサンプル数(通常は NN_SND_SAMPLES_PER_FRAME)を指定します。

音声データの取り込みに成功した場合は true が返されます。音声データはステレオの 16 bit PCM データで、左右のチャンネルがインターリーブされて格納されます。ヘッダ情報は含まれていません。サンプリングレートはサウンド DSP のサンプリングレート(約 32728 Hz)です。

10.5.4. DSP ADPCM フォーマットへのエンコード

nn::snd::EncodeAdpcmData() を呼び出して、モノラルの 16 bit PCM データを DSP ADPCM フォーマットにエンコードすることができます。

コード 10-15. DSP ADPCM フォーマットへのエンコード

```
s32 nn::snd::GetAdpcmOutputBufferSize(s32 nSamples);
void nn::snd::EncodeAdpcmData(s16* pInput, u8* pOutput, s32 nSamples,
                              s32 sampleRate, s32 loopStart, s32 loopEnd,
                              nn::snd::DspsndAdpcmHeader* pInfo);
```

pInput には、エンコードに使用する 16 bit PCM データ(モノラル)が格納されているバッファの先頭アドレスを指定します。モノラルの音声データでなければエンコードすることができません。*nSamples* と *sampleRate* には、音声データのサンプル数とサンプリングレートを指定してください。

pOutput には、エンコード結果の DSP ADPCM データを格納するバッファを指定します。バッファの先頭アドレスは 4 Byte アライメントでなければなりません。必要となるバッファのサイズは、音声データのサンプル数を引数に呼び出した nn::snd::GetAdpcmOutputBufferSize() の返り値で求めることができます。

loopStart と *loopEnd* には、ループの開始と終了のサンプル位置を指定します。サンプル位置は、16 bit PCM データの先頭を 0 とする値で指定してください。

pInfo には、DSP ADPCM データのヘッダ情報を格納する構造体へのポインタを指定してください。

10.5.5. DSP ADPCM フォーマットのデコード

nn::snd::DecodeAdpcmData() を呼び出して、DSP ADPCM フォーマットの音声データをモノラルの 16 bit PCM データにデコードすることができます。

コード 10-16. DSP ADPCM フォーマットのデコード

```
void nn::snd::DecodeAdpcmData(const u8* pInput, s16* pOutput,
                              const nn::snd::AdpcmParam& param,
                              nn::snd::AdpcmContext& context, s32 nSamples);
```

pInput には、デコードする DSP ADPCM データが格納されているバッファの先頭アドレスを指定します。*param* と *context* には、DSP ADPCM のパラメータとコンテキストを、*nSamples* には、音声データのサンプル数を指定してください。

pOutput には、デコード結果の 16 bit PCM データ(モノラル)を格納するバッファを指定します。バッファの先頭アドレスは 4 Byte アライメントでなければなりません。

実行が完了したあとの *context* には、デコード終了時のコンテキスト情報が格納されます。

10.5.6. サンプル位置とニブル数の相互変換

PCM データのサンプル位置を DSP ADPCM のニブル数へ、または、その逆の変換を行う関数を用意しています。

コード 10-17. サンプル位置とニブル数の相互変換

```
u32 nn::snd::ConvertAdpcmPos2Nib(u32 nPos);
u32 nn::snd::ConvertAdpcmNib2Pos(u32 nNib);
```

10.5.7. 蓋閉じによるスリープを拒否した際のサウンド出力

蓋が閉じられたことによるスリープを拒否した場合、サウンドはヘッドホンに強制的に出力されますが、以下の関数でその挙動を制御することができます。

コード 10-18. 蓋閉じによるスリープを拒否した際のサウンド出力の制御

```
nn::Result nn::snd::SetHeadphoneOutOnShellClose(bool forceout);
```

引数 *forceout* に false を指定することで、蓋閉じ状態でもヘッドホンが接続されていなければスピーカーからサウンドが出力されるようになります。true を指定した場合は、ヘッドホンの接続状態に関係なく、ヘッドホンから出力されます。

> **注意:** 引数 *forceout* に false を指定する場合は、必ずスリープ要求を拒否してください。拒否しなかった場合、蓋が閉じられてからスリープ状態に遷移するまでの間、スピーカーからサウンドが出力されることになります。
>
> なお、スリープ要求を受け付けるコールバック関数内でこの関数を呼び出した場合、そのスリープ要求では設定が変更されず、次のスリープ要求から有効になります。

10.5.8. ノイズが発生する原因と対策

本体の仕様により、以下のようなノイズが発生することがあります。

10.5.8.1. スピーカーから「ザー」というノイズが発生する

原因は、スピーカーおよび筐体の機構上、500Hz 付近でビビリノイズが発生しやすい特性があり、500Hz 付近の帯域を含む音をパンを振って鳴らすとサラウンド処理がこの帯域の音を増幅してしまうためです。

以下の条件で発生します。

- スピーカーで鳴らしている。
- 400～600Hz のサイン波や、サイン波に近い音色を鳴らしている(フルートやホルンなど)。
- 音量を大きくしている。
- パンをめいっぱい振っている。
- サラウンドを ON にしている。

以下のような対応策が考えられます。

- パンを左右に極端に振らない。
- もう少しセンターに寄せ、2 つのスピーカーが「程良い」強さで鳴るようにすることで音量感を維持する。
- 400～600Hz の帯域をイコライザーでおさえる。

- 音色を変える。

10.5.8.2. 「ジジジ」という機械ノイズが発生する

原因は、スピーカーから大きな音量で音を出したときに発生する筐体の振動が 3Dボリュームに伝播し、3Dボリュームが筐体とぶつかるためです。

400Hz ～ 1kHz 程度のサイン波のような単音でこのノイズは顕著になり、音を出しているときに 3Dボリュームを指等で押さえてノイズが消えればこの症状です。

このノイズは、音を小さくするか、複合音にすることで目立たなくすることができます。

10.5.8.3. 「サー」というノイズが重畳する

原因は、スピーカーの構造上の問題です。

音量が大きく、400Hz ～ 1kHz 程度のサイン波のような単音でやや顕著になります。

このノイズは、音を小さくするか、複合音にすることで目立たなくすることができます。

10.5.8.4. 特定の状況でヘッドホンから「プツッ」というノイズが聞こえる

以下の状況で、ヘッドホンから「プツッ」というノイズが聞こえる本体があります。

- 電源ON/OFF時
- スリープ入/出時
- 互換モード遷移時

本体の仕様によるもので対策はありません。スライドボリュームを 0 にした状態で電源を ON にした際にノイズが発生する本体では上記のタイミングで必ずノイズが発生します。

なお、このノイズが発生する本体は量産実機と開発実機の中に混在していますが、PARTNER-CTR で発生することはありません。

10.6. 音源データ情報の再利用

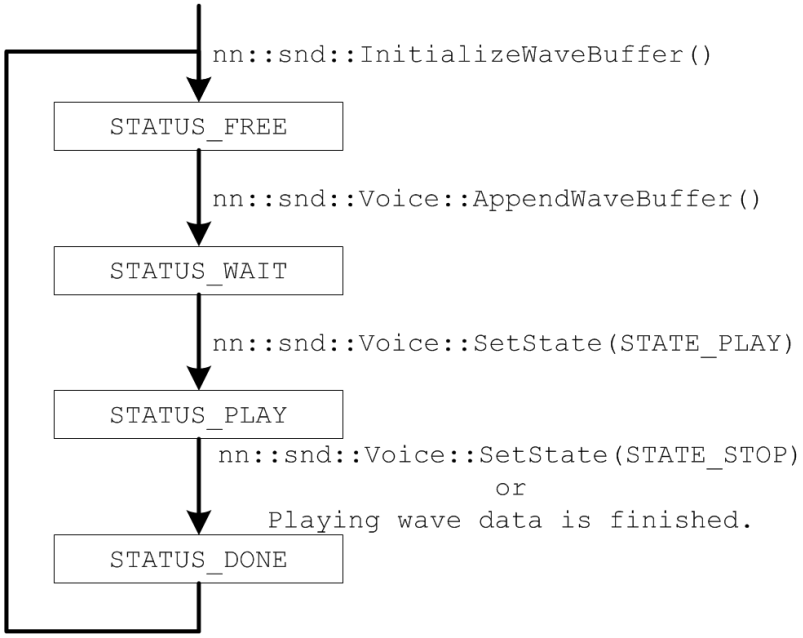
音源データの再生が完了した音源データ情報は、`nn::snd::InitializeWaveBuffer()` で初期化して再利用することができます。音源データの再生が完了しているかどうかは、音源データ情報の `status` メンバで判断することができます。

表 10-5. 音源データ情報の状態

状態	説明
`STATUS_FREE`	音源データ情報を初期化した直後の状態です。
`STATUS_WAIT`	音源データ情報をボイスオブジェクトに登録した直後の状態です。
`STATUS_PLAY`	音源データが再生されている状態です。
`STATUS_DONE`	音源データの再生が完了している状態です。停止させた場合も含みます。
`STATUS_TO_BE_DELETED`	`nn::snd::Voice::DeleteWaveBuffer()` により、登録されていた音源データの削除が予定されている状態です。 次の `nn::snd::SendParameterToDsp()` の実行時に `STATUS_DONE` に遷移します。

状態の遷移を図にすると、以下のようになります。

図 10-5. 音源データ情報の状態遷移



10.7. ボイスオブジェクトの解放

不要になったボイスオブジェクトは nn::snd::FreeVoice() で解放してください。ただし、ボイスオブジェクトの確保でライブラリからドロップ対象とされたボイスオブジェクトは、この関数で解放してはいけません。

10.8. 終了処理

サウンドの再生に利用していた DSP および SND のライブラリの終了処理は、以下の手順で行ってください。

1. nn::snd::Finalize() を呼び出して、SND ライブラリの使用を終了する。
2. nn::dsp::UnloadComponent() を呼び出して DSP にロードしていたコンポーネントをアンロードし、DSP を停止する。
3. nn::dsp::Finalize() を呼び出して、DSP ライブラリの使用を終了する。

コード 10-19. サウンドの終了処理

```
nn::snd::Finalize();
nn::dsp::UnloadComponent();
nn::dsp::Finalize();
```

11. 本体設定

この章では、本体設定で行ったユーザーに関する情報やサウンドの設定などを、アプリケーションから取得する方法について説明します。

11.1. 初期化

本体設定で扱われている情報や 3DS 本体の情報へのアクセスには CFG ライブラリを使用します。

一部の関数を除いて、CFG ライブラリの関数を使用するには、まず `nn::cfg::Initialize()` で初期化を行わなければなりません。初期化を行ったあとは、終了処理を行うまで CFG ライブラリの関数を呼び出すことができます。

コード 11-1. CFG ライブラリの初期化

```
void nn::cfg::Initialize(void);
```

11.2. 情報の取得

ライブラリの初期化後は、用意されている関数を呼び出すことで様々な情報にアクセスすることができるようになります。

11.2.1. ユーザー名

本体設定で設定された、ユーザーの名前を取得することができます。

コード 11-2. ユーザー名の取得

```
void nn::cfg::GetUserName(nn::cfg::UserName* pUserName);

struct nn::cfg::UserName
{
    wchar_t userName[CFG_USER_NAME_LENGTH];
    bool isNgUserName;
    NN_PADDING1;
};
```

pUserName には、ユーザー名を格納する `nn::cfg::UserName` 構造体へのポインタを指定します。

`nn::cfg::UserName` 構造体の `userName` メンバにはユーザー名がワイド文字列で格納され、`isNgUserName` メンバにはユーザー名に NG ワードが含まれている場合に `true` が格納されます。

NG ワードのチェックは、日本リージョンであれば日本語、北米リージョンであれば北米英語と本体設定言語、欧州リージョンであればイギリス英語と本体設定言語で行われます。

11.2.2. 誕生日

本体設定で設定された、ユーザーの誕生日を取得することができます。

コード 11-3. 誕生日の取得

```
void nn::cfg::GetBirthday(nn::cfg::Birthday* pBirthday);

struct nn::cfg::Birthday
{
    s8  month;
    s8  day;
};
```

pBirthday には、誕生日を格納する nn::cfg:Birthday 構造体へのポインタを指定します。

11.2.3. 国コード

本体設定で設定された、ユーザーの住んでいる国および地域を国コードで取得することができます。

コード 11-4. 国コードの取得

```
nn::cfg::CfgCountryCode nn::cfg::GetCountry(void);
```

定義されている国コードについては、ヘッダファイル(nn/cfg/CTR/cfg_CountryCode.h)を参照してください。

国コードと国名コード(ISO 3166-1 alpha-2)は、以下の関数で相互に変換することができます。

コード 11-5. 国コードと国名コードの相互変換

```
nn::Result nn::cfg::ConvertCountryCodeToIso3166a2(
    char* iso3166a2, nn::cfg::CfgCountryCode countryCode);
nn::Result nn::cfg::ConvertIso3166a2ToCountryCode(
    nn::cfg::CfgCountryCode* pCountryCode, const char* iso3166a2);
```

補足: nn::cfg::GetCountryCodeA2() は CTR-SDK から削除されます。

注意: 国コードと国名コードの相互変換を行う関数を使用するアプリケーションは、必ず返り値に nn::cfg::ResultNotFound が返された場合のハンドリングを行ってください。

11.2.4. 言語コード

本体設定で設定された、表示に使用する言語を言語コードで取得することができます。

コード 11-6. 言語コードの取得

```
nn::cfg::CfgLanguageCode nn::cfg::GetLanguage(void);
```

定義されている言語コードには、以下のものが存在します。

表 11-1. 言語コード

値	言語	言語名(ISO 639-1 alpha-2)
CFG_LANGUAGE_JAPANESE	日本語	ja
CFG_LANGUAGE_ENGLISH	英語	en

CFG_LANGUAGE_FRENCH	フランス語	fr
CFG_LANGUAGE_GERMAN	ドイツ語	de
CFG_LANGUAGE_ITALIAN	イタリア語	it
CFG_LANGUAGE_SPANISH	スペイン語	es
CFG_LANGUAGE_SIMP_CHINESE	中国語(簡体字)	zh
CFG_LANGUAGE_KOREAN	韓国語	ko
CFG_LANGUAGE_DUTCH	オランダ語	nl
CFG_LANGUAGE_PORTUGUESE	ポルトガル語	pt
CFG_LANGUAGE_RUSSIAN	ロシア語	ru
CFG_LANGUAGE_TRAD_CHINESE	中国語(繁体字)	zh

取得した言語コードは、以下の関数で言語名(ISO 639-1 alpha-2)に変換することができます。

コード 11-7. 言語コードから言語名への変換

```
const char* nn::cfg::GetLanguageCodeA2(CfgLanguageCode cfgLanguageCode);
```

cfgLanguageCode で指定した言語コードに対応する文字列がなかった場合は NULL を返します。

> **補足:** この関数は初期化を行っていない状態でも呼び出すことができます。

11.2.5. 簡易アドレス情報

本体設定で設定された、簡易アドレス情報(国名、地域名、緯度、経度)を取得することができます。

コード 11-8. 簡易アドレス情報の取得

```
void nn::cfg::GetSimpleAddress(nn::cfg::SimpleAddress* pSimpleAddress);

struct nn::cfg::SimpleAddress
{
    u32 id;
    wchar_t countryName[CFG_SIMPLE_ADDRESS_NUM_LANGUAGES]
                       [CFG_SIMPLE_ADDRESS_NAME_LENGTH];
    wchar_t regionName[CFG_SIMPLE_ADDRESS_NUM_LANGUAGES]
                      [CFG_SIMPLE_ADDRESS_NAME_LENGTH];
    u16 latitude;
    u16 longitude;
};
```

pSimpleAddress には、簡易アドレス情報を格納する nn::cfg::SimpleAddress 構造体へのポインタを指定します。

nn::cfg::SimpleAddress 構造体の countryName メンバと regionName メンバには国名と地域名がワイド文字列で格納され、latitude メンバと longitude メンバには緯度と経度が格納されます。

緯度(latitude メンバ)と経度(longitude メンバ)は 360° を 65536 で割った値(約 0.005°)を単位に表されます。北緯 0° ～90° が 0x0000～0x4000、南緯 0.005° ～90° が 0xFFFF～0xC000、東経 0° ～179.995° が

0x0000～0x7FFF、西経 0.005° ～180° が 0xFFFF～0x8000 にそれぞれ対応しています。

11.2.5.1. 簡易アドレス情報の ID

簡易アドレス情報の ID のみを取得する関数が用意されています。

コード 11-9. 簡易アドレス情報の ID の取得

```
void nn::cfg::GetSimpleAddressId(nn::cfg::SimpleAddressId* pSimpleAddressId);

struct nn::cfg::SimpleAddressId
{
    u32 id;

    nn::cfg::CfgCountryCode GetCountryCode(void) const;
    u8 GetRegionCode(void) const;
};
```

pSimpleAddressId には、nn::cfg::SimpleAddressId 構造体へのポインタを指定します。

nn::cfg::SimpleAddressId 構造体には、国コードを取得する GetCountryCode() メンバと地域コードを取得する GetRegionCode() メンバが用意されています。それぞれのメンバ関数で取得できる値については、「11.2.3. 国コード」を参照してください。

11.2.5.2. ID による簡易アドレス情報の取得

簡易アドレス情報の ID から、簡易アドレス情報を取得することができます。

コード 11-10. ID による簡易アドレス情報の取得

```
nn::Result nn::cfg::GetSimpleAddress(nn::cfg::SimpleAddress* pSimpleAddress,
                                     nn::cfg::SimpleAddressId simpleAddressId,
                                     uptr pWorkMemory, u32 workMemorySize);
```

simpleAdressId に指定した ID をもとに取得した簡易アドレス情報を *pSimpleAddress* に格納します。

この関数の動作にはワークメモリが必要です。nn::cfg::CFG_SIMPLE_ADDRESS_WORKMEMORY_SIZE 以上のサイズで確保したワークメモリとそのサイズを、*pWorkMemory* と *workMemorySize* のそれぞれに指定してください。

11.2.5.3. 簡易アドレス情報の ID の 3DS と Wii U 間での相互変換

以下の関数で、3DS の簡易アドレス情報の ID と Wii U の簡易アドレス情報の ID を相互に変換することができます。

コード 11-11. 簡易アドレス情報の ID の 3DS と Wii U 間での相互変換

```
nn::cfg::SimpleAddressId nn::cfg::ConvertToWiiUSimpleAddressId(
    nn::cfg::SimpleAddressId ctrSimpleAddressId);
nn::cfg::SimpleAddressId nn::cfg::ConvertToCtrSimpleAddressId(
    nn::cfg::SimpleAddressId wiiUSimpleAddressId);
```

11.2.6. リージョンコード

> **補足：** ここで扱うリージョンコードは、出荷時に 3DS 本体に設定されているものです。カードに設定するリージョンコードは bsf ファイルで設定し、本体とカードのリージョンコードが異なる場合やカードにリージョンコードが設定されていない場合にはソフトは起動しません。bsf ファイルについては、CTR-SDK のツール「ctr_makebanner」のリファレンスなどを参照してください。

本体の仕向地をリージョンコードで取得することができます。

コード 11-12. リージョンコードの取得

```
nn::cfg::CfgRegionCode nn::cfg::GetResion(void);
```

定義されているリージョンコードには、以下のものが存在します。

表 11-2. リージョンコード

値	仕向地	対応する文字列
CFG_REGION_JAPAN	日本	JPN
CFG_REGION_AMERICA	米州	USA
CFG_REGION_EUROPE	欧州および豪州	EUR
CFG_REGION_CHINA	中国	CHN
CFG_REGION_KOREA	韓国	KOR
CFG_REGION_TAIWAN	台湾	TWN

取得したリージョンコードは、以下の関数で対応するアルファベット 3 文字の文字列に変換することができます。

コード 11-13. リージョンコードから対応する文字列への変換

```
const char* nn::cfg::GetRegionCodeA3(CfgRegionCode cfgRegionCode);
```

cfgRegionCode に指定したリージョンコードに対応する文字列がなかった場合は NULL を返します。

11.2.7. サウンド出力モード

本体設定で設定された、サウンドの出力モードを取得することができます。

コード 11-14. サウンド出力モードの取得

```
nn::cfg::CfgSoundOutputMode nn::cfg::GetSoundOutputMode(void);
```

定義されているサウンド出力モードには、以下のものが存在します。

表 11-3. サウンド出力モード

値	サウンド出力モード
CFG_SOUND_OUTPUT_MODE_MONO	モノラル
CFG_SOUND_OUTPUT_MODE_STEREO	ステレオ
CFG_SOUND_OUTPUT_MODE_SURROUND	(3D)サラウンド

11.2.8. RTC 改変オフセット値

ハードウェアが保持している、ユーザーによる RTC 時刻の改変オフセット値を取得することができます。

コード 11-15. RTC 改変オフセット値の取得

```
nn::fnd::TimeSpan nn::cfg::GetUserTimeOffset(void);
```

改変オフセット値は秒単位で返されます。この値は本体設定の時刻設定で時刻が変更されたときに、何秒進め(戻し)たかの絶対値を累積したものです。この値の取り扱いについては、「8.5.3. 時刻改変のオフセット値の取り扱い」を参照してください。

11.2.9. ペアレンタルコントロール

写真の送受信やフレンドの追加などをアプリケーションで行う前に、本体設定の「保護者による使用制限」(以降、「ペアレンタルコントロール」と表記)で制限されていないかどうかを確認しなければなりません。ペアレンタルコントロールの設定が有効になっているかどうかは、nn::cfg::IsParentalControlEnabled() で確認することができます。

コード 11-16. ペアレンタルコントロール設定が有効になっているかどうかの確認

```
bool nn::cfg::IsParentalControlEnabled(void);
```

表 11-4. ペアレンタルコントロールの項目と制限対象となるアプリケーションの機能

項目名	制限対象となるアプリケーションの機能	参照先
年齢制限	対象外(システム側で制限します)	-
インターネットブラウザーの使用	対象外(システム側で制限します)	-
ニンテンドーeショップ等での商品やサービスの購入	ECDK を利用したコンテンツ購入など	11.2.9.6
3D映像の表示	対象外(システム側で制限します)	-
Miiverseの使用	Miiverse への投稿、Miiverse の閲覧	11.2.9.7
写真や画像・音声・動画・長文テキストの送受信	リッチ UGC の送受信	11.2.9.2
	(北米)リッチ UGC のアップロード	11.2.12
他のユーザーとのインターネット通信	ほかのユーザーとのデータ交換	11.2.9.4
他のユーザーとのすれちがい通信	すれちがい通信の使用	11.2.9.5
フレンドの登録	アプリケーション内でのフレンド登録	11.2.9.3
DSダウンロードプレイの使用	対象外	-
配信動画の視聴	通信を介して取得した動画の視聴	11.2.9.8

11.2.9.1. 暗証番号の入力による制限の一時的な解除

ペアレンタルコントロールを適用するタイミングで保護者が一時的に制限を解除することができるように、アプリケーションは暗証番号の入力を受け付け、入力された暗証番号と設定されている暗証番号とが合致した場合に、一時的に制限を解除することができます。このとき、入力中の暗証番号や設定されている暗証番号が画面には一切表示されないように注意してください。

設定されているペアレンタルコントロールの暗証番号は、nn::cfg::CheckParentalControlPinCode() で照合することができます。

コード 11-17. ペアレンタルコントロールの暗証番号の照合

```
bool nn::cfg::CheckParentalControlPinCode(const char *input);
```

input に渡した文字列と暗証番号(半角数字 4 桁)が合致した場合に true を返します。

ソフトウェアキーボードアプレットに「保護者による使用制限一時解除モード」を用意しています。このモードで起動したソフトウェアキーボードアプレットは専用のシーケンスで動作し、アプリケーションはアプレットからの返り値で制限を解除してよいかどうかを判断することができます。

11.2.9.2. 個人情報を含む可能性のあるデータの送受信に対する制限

ペアレンタルコントロールで写真交換(写真、画像、音声、動画、長文テキストなどの交換)が制限されているかどうかを、nn::cfg::IsRestrictPhotoExchange() で確認することができます。写真交換が制限されている場合、制限が一時的に解除されない限り、アプリケーションは撮影した写真などの画像をほかの本体と送受信してはいけません。なお、この制限は写真だけでなく、画像、音声、動画、長文テキストなど、個人情報を含む可能性のあるデータの送受信が対象となっています。

コード 11-18. 個人情報を含む可能性のあるデータの送受信に対する制限の確認

```
bool nn::cfg::IsRestrictPhotoExchange(void);
```

制限されているときに true が返されることに注意してください。

11.2.9.3. フレンド追加の制限

ペアレンタルコントロールでフレンドの追加が制限されているかどうかを、nn::cfg::IsRestrictAddFriend() で確認することができます。フレンドの追加が制限されている場合、アプリケーションからのフレンド登録はエラーとなります。原則的に、アプリケーションがこの制限を一時的に解除することは許可されていません。

コード 11-19. フレンド追加の制限の確認

```
bool nn::cfg::IsRestrictAddFriend(void);
```

制限されているときに true が返されることに注意してください。

11.2.9.4. ほかのユーザーとのインターネット通信の制限

ペアレンタルコントロールで、インターネット通信を経由したほかのユーザーとの対戦やデータのやり取りが制限されているかどうかを、nn::cfg::IsRestrictP2pInternet() で確認することができます。この制限が有効になっている場合、制限が一時的に解除されない限り、アプリケーションは対戦やユーザー作成のコンテンツをダウンロードするようなインターネット通信を行ってはいけません。

コード 11-20. ほかのユーザーとのインターネット通信の制限の確認

```
bool nn::cfg::IsRestrictP2pInternet(void);
```

制限されているときに true が返されることに注意してください。

11.2.9.5. すれちがい通信の制限

ペアレンタルコントロールで、すれちがい通信を経由したほかのユーザーとのデータのやり取りが制限されているかどうかを、nn::cfg::IsRestrictP2pCec() で確認することができます。この制限が有効になっている場合、すれちがい通信は一切行われず、すれちがいデータの登録時にエラーとなります。アプリケーションがこの制限を一時的に解除することはできません。

コード 11-21. すれちがい通信の制限の確認

```
bool nn::cfg::IsRestrictP2pCec(void);
```

制限されているときに true が返されることに注意してください。

11.2.9.6. ニンテンドーeショップ等の利用の制限

ペアレンタルコントロールで、ニンテンドーeショップなどでのクレジットカードの使用や、商品やサービスの購入が制限されているかを、nn::cfg::IsRestrictShopUse() で確認することができます。この制限が有効になっている場合、制限が一時的に解除されない限り、アプリケーションは残高の追加やコンテンツの購入などを行ってはいけません。

コード 11-22. ニンテンドーeショップ等の利用制限の確認

```
bool nn::cfg::IsRestrictShopUse(void);
```

制限されているときに true が返されることに注意してください。

11.2.9.7. Miiverseの使用の制限

ペアレンタルコントロールで、Miiverse の閲覧や Miiverse への投稿が制限されているかを確認する関数が用意されています。Miiverse の閲覧が制限されているかどうかは nn::cfg::IsRestrictMiiverseBrowse() で、Miiverse への投稿が制限されているかどうかは nn::cfg::IsRestrictMiiversePost() で確認することができます。

ペアレンタルコントロールの本体設定での項目名と、各関数が返す値の対応は以下の通りです。

表 11-5. Miiverse の使用の制限に対する本体設定での項目名と返り値の対応

本体設定での項目名	IsRestrictMiiverseBrowse()	IsRestrictMiiversePost()
「投稿・閲覧を制限する」	true	true
「投稿のみ制限する」	false	true
「制限しない」	false	false

これらの制限が一時的に解除されない限り、アプリケーションはほかのユーザーからの Miiverse への投稿を取得することや Miiverse への投稿を行ってはいけません。

11.2.9.8. 通信を介して取得した映像コンテンツの視聴の制限

ペアレンタルコントロールで、通信を介して取得した映像コンテンツの視聴が制限されているかを、nn::cfg::IsRestrictWatchVideo() で確認することができます。この制限が有効になっている場合、制限が一時的に解除されない限り、アプリケーションは通信を介して取得した動画の再生などを行ってはいけません。

コード 11-23. 通信を介して取得した映像コンテンツの視聴制限の確認

```
bool nn::cfg::IsRestrictWatchVideo(void);
```

制限されているときに true が返されることに注意してください。

11.2.10. EULA への同意の確認

ニンテンドーネットワークやすれちがい通信などを含む「ニンテンドー 3DS ネットワークサービス」をアプリケーションで利用するには、事前にユーザーによる EULA（利用規約）への同意が必要です。ユーザーが利用規約に同意しているかどうかを nn::cfg::IsAgreedEula() で確認し、同意していない場合はこれらの機能を使用してはいけません。

コード 11-24. EULA への同意の確認

```
bool nn::cfg::IsAgreedEula(void);
```

ユーザーが利用規約に同意しているときは true が返されます。この関数を呼び出すには、事前に FS ライブラリの初期化が行われていなければなりません。

利用規約への同意の確認が必要な機能については、ガイドラインを参照してください。

11.2.11. 本体固有 ID

本体の識別などに利用する本体固有 ID を取得することができます。

コード 11-25. 本体固有 ID の取得

```
bit64 nn::cfg::GetTransferableId(bit32 uniqueId);
```

uniqueId には、アプリケーションに割り当てられたユニーク ID(20 ビット)を指定します。

本体固有 ID は、ユニーク性がある程度保証された 64 ビット値です。また、本体の移動が可能で、買い替えなどで本体が変わったときに移行することができます。ただし、本体を紛失した場合や盗難、破壊された場合は、二度とその ID を復帰させることができません。本体の初期化を行ったときにも本体固有 ID が変更され、ID の復帰はできません。

本体固有 ID の利用目的として、以下のようなものを想定しています。

- アプリケーション内のパラメータの初期値を、本体ごとで異なる値にする
- 特定の本体からのみアクセス可能なデータを生成する
- ローカル通信などの際に識別 ID、またはそのシードとする

> **補足:** セーブデータの作成時に取得した本体固有 ID を保存し、以後はセーブデータに保存された本体固有 ID を使用することで修理などによる変更に対応できる場合があります。また、保存の際に現在時刻と合わせた 128 ビット値として保存することで、複数のカードに同じデータが保存されることを回避することができます。

特定の本体でのみアクセス可能にした場合は、本体の初期化、本体の紛失や盗難により二度とアクセスできなくなってしまうことに注意してください。

11.2.12. COPPACS による制限

COPPACS(COPPA Compliance System)とは、北米向け(本体リージョンが北米、かつ国設定がアメリカまたはカナダの場合)に作成されたアプリケーションが COPPA(Children's Online Privacy Protection Act)に対応するために任天堂が提供するシステムです。

> **補足:** どのようなアプリケーションが COPPACS による制限の対象となるかについては、ガイドラインの「UGC」を参照してください。COPPACS の詳細は、「システムアプリ・アプレット仕様書」に記載される予定です。

COPPACS による制限がかけられているかどうかを取得することができます。

コード 11-26. COPPACS による制限の取得

```
bool nn::cfg::IsCoppacsSupported();
nn::cfg::CfgCoppacsRestriction nn::cfg::GetCoppacsRestriction(void);
```

nn::cfg::GetCoppacsRestriction() を呼び出すことで、COPPACS への対応方法を確認することができます。

CFG_COPPACS_RESTRICTION_NONE が返された場合は制限がかけられていませんので、COPPACS への対応は必要ありません。

CFG_COPPACS_RESTRICTION_NEED_PARENTAL_PIN_CODE または CFG_COPPACS_RESTRICTION_NEED_PARENTAL_AUTHENTICATION が返された場合は制限がかけられていますので、COPPACS への対応が必要です。

前者はペアレンタルコントロールの暗証番号の入力で解除可能ですので、アプリケーション内で暗証番号の入力とチェック(「11.2.9.1. 暗証番号の入力による制限の一時的な解除」参照)を行い、正しい暗証番号が入力された場合に制限を一時的に解除することができます。後者はアプリケーション内で解除することができず、本体設定で COPPACS 認証の手続きを行う必要があります。

本体設定での認証手続きを行う場合、一旦アプリケーションを終了させなければならないことに注意してください。なお、本体設定からアプリケーションに戻ったことはアプリケーションの再起動時に確認することができます。ジャンプ前に行っていた処理を継続する場合は、必ず COPPACS への対応方法を再度確認してください。再確認時でも、本体設定での認証手続きが必要であると判断される可能性があります。

単に現在の本体設定が COPPACS の対象国であるかどうかの確認は、nn::cfg::IsCoppacsSupported() で行うことができます。本体設定が対象国であれば、COPPACS の制限が掛けられてなくても true が返されることに注意してください。

> **補足:** COPPACS 認証の手続きを行う画面(PARENTAL_CONTROLS_COPPACS)へジャンプする方法は、「5.3.7. 本体設定へのジャンプ」で説明しています。実装例については、サンプルデモ(cfg の coppacs)を参照してください。

11.3. 終了処理

CFG ライブラリの使用を終了するときは、nn::cfg::Finalize() を呼び出して終了処理を行ってください。

コード 11-27. CFG ライブラリの終了

```
void nn::cfg::Finalize(void);
```

初期化が行われていない状態で呼び出された場合は何も行いません。

ライブラリの初期化関数を呼び出した回数は記録されていますので、同じ回数の終了処理が呼び出されるまでライブラリの使用を終了したとは見なされません。

12. アプレット

この章では、3DS で提供されているアプレットを利用するための、ライブラリについて説明します。

各アプレットで提供されているライブラリにより、アプリケーションからそのアプレットを起動したり、機能を利用することができます。アプレットの起動に必要なメモリは基本的にシステム側から確保されますが、アプリケーションから作業メモリを渡さなければならないアプレットも存在します。また、アプレットの動作中は、呼び出したスレッドの動作を停止した上でアプリケーションと同じ CPU を使用しますので、実質的にアプリケーションは止まった状態になります。

12.1. ライブラリアプレット

3DS では、アプリケーションでよく使われる機能を、以下のライブラリアプレットとして提供しています。

- ソフトウェアキーボードアプレット
- 写真選択アプレット
- Mii 選択アプレット
- 音声選択アプレット
- エラー・EULA アプレット
- 拡張スライドパッド補正アプレット
- EC アプレット
- ログインアプレット

> **補足：** Mii 選択アプレットは CTR 似顔絵ライブラリパッケージで提供しています。

12.1.1. 各ライブラリアプレットに共通する情報

ライブラリアプレットを起動するために行わなければならない処理や、ライブラリアプレットから復帰したときの処理は、基本的に HOME メニューの起動と復帰で行っている処理と同じです。同様に、ライブラリアプレットから復帰するまで、キー入力の取得や描画などのデバイスの使用に制限がかかり、ライブラリアプレットを呼び出したスレッドのみが停止することにも注意が必要です。なるべく、不要なスレッドを動作中のままにしないようにしてください。ライブラリアプレットの起動中でも動作し続けるスレッドを作成する場合は、ライブラリアプレットが下表の優先度設定でスレッドを作成していることに注意し、それらのスレッドの処理に影響がないようにスレッドの優先度を設計してください。

表 12-1. ライブラリアプレットが作成するスレッドの優先度

ライブラリアプレット	優先度	備考
(すべてに共通)	15	スリープなどの通知に使用しています。
ソフトウェアキーボードアプレット	17～20	
写真選択アプレット	16, 18, 20, 21, 25	
Mii 選択アプレット	17	
音声選択アプレット	16, 18	
エラー・EULA アプレット	17, 20	
拡張スライドパッド補正アプレット	17～19	
EC アプレット	17, 22	通信などに使用しています。

ログインアプレット	17, 22	通信などに使用しています。

HOME メニューの起動で呼び出している nn::applet::WaitForStarting() から復帰するときのように、アプリケーション復帰後の nn::applet::IsExpectedToCloseApplication() でアプリケーションの終了に対応する必要があります。ただし、ライブラリアプレットの表示中に電源ボタンが押されたときは、描画の権限がない状態、かつ同時に nn::applet::IsExpectedToProcessPowerButton() が true を返す状態になります。そのため、そのまま終了要求への対処を行うと、終了処理を行っている間の画面更新が止まってしまいますので、先に電源ボタンへの対処を行うように実装してください。

12.1.1.1. ライブラリアプレットからの復帰

ライブラリアプレットの起動中に HOME ボタンや電源ボタン、ソフトウェアリセットとなる組み合わせのボタン(L + R + START)が押されたときは、すぐにアプリケーションに復帰し、以下のような挙動となります。

表 12-2. ライブラリアプレットからの復帰時の挙動

ボタン	挙動
HOME ボタン	リターンコードに HOME ボタンが押されたことが判別できる値を返し、復帰後の nn::applet::IsExpectedToProcessHomeButton() は true を返します。
電源ボタン	リターンコードに電源ボタンが押されたことが判別できる値を返し、復帰後の nn::applet::IsExpectedToProcessPowerButton() は true を返します。
ソフトウェアリセット	リターンコードにソフトウェアリセットされたことが判別できる値を返します。

12.1.1.2. プリロード

プリロードに対応しているライブラリアプレットは、事前にロードなどの処理を行うことができます。これにより、ライブラリアプレットを起動する関数を呼び出してから画面が表示されるまでの、描画が止まってしまう時間を短縮することができます。

プリロードのための関数名は各ライブラリアプレットで異なりますが、基本的に、プリロードを開始する Preload～()、プリロードの完了を待つ WaitForPreload～()、プリロードを取り消す CancelPreload～() に統一されています。

プリロードしたライブラリアプレットを起動するときは、必ずその完了を待ってから起動してください。また、完了を待つ関数は、プリロードを開始していない状態で呼び出すと制御を戻さなくなりますので注意してください。ただし、これは 1 つのスレッドで行った場合であり、完了を待っているスレッドと異なるスレッドでプリロードを開始させた場合は呼び出す順番が逆になっていても制御が戻ります。なお、nn::applet::ProcessHomeButton() により HOME メニューを表示するとプリロードがキャンセルされます。そのため、Preload～() と WaitForPreload～() の間で HOME メニューへ遷移すると WaitForPreload～() が制御を戻さなくなりますので注意してください。

同時に複数のライブラリアプレットをプリロードしておくことはできません。プリロードしていたライブラリアプレット以外のライブラリアプレットを起動する場合は、先にプリロードを取り消さなければなりません。ライブラリアプレットを起動した時点でプリロードは解除されますので、ライブラリアプレットから復帰したあとに、プリロードを取り消す必要はありません。

12.1.2. ソフトウェアキーボードアプレット

名前やパスワードなど、アプリケーションがユーザーからの文字入力を必要とする際に呼び出すライブラリアプレットです。ソフトウェアキーボードは下画面に表示され、パスワード入力用の補助機能や、入力文字数と入力可能文字の制限機能、NG ワードフィルタ機能などを備えています。上画面には、ソフトウェアキーボードを起動したときにアプリケーションが表示していた画面がそのまま、または暗くして表示されます。

このライブラリアプレットを利用するには、ヘッダファイル(nn/swkbd.h)のインクルードと、ライブラリファイル(

libnn_swkbd)の追加が必要です。

ライブラリアプレットの起動時に渡すパラメータは nn::swkbd::Parameter 構造体で定義されています。詳細な動作設定は、パラメータの config メンバ(nn::swkbd::Config 構造体)に対して行いますが、設定の前に初期化を行わなければなりません。

コード 12-1. ソフトウェアキーボードの動作設定の初期化

```
void nn::swkbd::InitializeConfig(nn::swkbd::Config* pConfig);
```

pConfig に渡された Config 構造体がデフォルト設定で初期化されます。Config 構造体のメンバに対して行うことのできる動作設定については、「アプレット仕様書」を参照してください。

このライブラリアプレットはアプリケーションで作業メモリを確保する必要があります。必要な作業メモリのサイズは動作設定によって異なり、nn::swkbd::GetSharedMemorySize() で取得することができます。

コード 12-2. ソフトウェアキーボードの作業メモリのサイズ取得

```
s32 nn::swkbd::GetSharedMemorySize(
                const nn::swkbd::Config* pConfig,
                const void* pInitialStatusData = NULL,
                const void* pInitialLearningData = NULL);
```

pConfig には、動作設定を行った Config 構造体へのポインタを渡します。

pInitialStatusData には、前回起動したソフトウェアキーボードの動作設定で最終状態を保存していた場合、そのデータの先頭アドレスを渡すことで前回起動時の最終状態を復元することができます。復元の必要がない場合や、最終状態を保存していない場合は NULL を指定してください。

pInitialLearningData には、前回起動したソフトウェアキーボードの動作設定で予測変換の学習データを保存していた場合、そのデータの先頭アドレスを渡すことで前回起動時の予測変換の学習状態を復元することができます。復元の必要がない場合や、学習状態を保存していない場合は NULL を指定してください。

作業メモリは、先頭アドレスが nn::swkbd::MEMORY_ALIGNMENT (4096 Byte) のアライメント、サイズが nn::swkbd::MEMORY_UNITSIZE (4096) の倍数で確保しなければなりません。また、**作業メモリにはデバイスメモリから確保したメモリ領域を指定しないでください。**

パラメータ設定と作業メモリの確保が完了したら、nn::swkbd::StartKeyboardApplet() でソフトウェアキーボードを起動することができます。

コード 12-3. ソフトウェアキーボードアプレットの起動

```
bool nn::swkbd::StartKeyboardApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::swkbd::Parameter* pParameter,
                void* pSharedMemoryAddr,
                size_t sharedMemorySize,
                const wchar_t* pInitialInputText = NULL,
                const nn::swkbd::UserWord* pUserWordArray = NULL,
                const void* pInitialStatusData = NULL,
                const void* pInitialLearningData = NULL
                nn::applet::AppTextCheckCallback callback = NULL);
```

pParameter には、動作設定を行った Config 構造体をメンバに持つ Parameter 構造体へのポインタを指定します。入力された文字列などの情報は、この引数で指定された構造体に格納されます。

pSharedMemoryAddr と *pSharedMemorySize* には、作業メモリの先頭アドレスとサイズを指定します。

pInitialInputText に、NULL ではなく UTF-16LE の文字列を渡すと、指定された文字列が入力欄に設定された状態でソフトウェアキーボードが起動します。

pUserWordArray には、ユーザー辞書に登録する単語の配列を指定します。

pInitialStatusData と *pInitialLearningData* には、作業メモリサイズの取得時と同じ値を指定してください。

callback には、入力された文字列をアプリケーションでチェックする場合に使用するコールバック関数を指定します。動作設定でアプリケーションによる入力チェックを行わない場合、この引数は無視されます。

返り値が false ならば起動に失敗しています。返り値が true ならば起動に成功しています。

ソフトウェアキーボードのスレッドは優先度 17～20 で動作していますので、ソフトウェアキーボードの起動中でも動作し続けるスレッドの優先度の設定には注意が必要です。

12.1.3. 写真選択アプレット

ニンテンドー 3DS カメラのアルバムに登録されている写真の中から、アプリケーションで使用するデータを 1 つ選択するためのライブラリアプレットです。起動時に撮影時間や種類などの抽出条件を指定することができ、一覧表示されるデータを絞り込むことができます。なお、アプレットで選択可能な写真は SD カードに保存されているものに限られています。

このライブラリアプレットを利用するには、ヘッダファイル(nn/phtsel.h)のインクルードと、ライブラリファイル(libnn_phtsel)の追加が必要です。

ライブラリアプレットの起動時に渡すパラメータは nn::phtsel::Parameter 構造体で定義されています。基本動作の設定は、パラメータの m_config メンバ(nn::phtsel::Config 構造体)に対して行い、抽出条件などの詳細な設定は m_input メンバ(nn::phtsel::PhtselInput 構造体)で行います。m_output メンバ(nn::phtsel::PhtselOutput 構造体)には実行結果が格納されます。設定項目などの詳細については、サンプルデモを参照してください。

アプレットの背景にアプリケーションのキャプチャ画像を表示する場合は、アプリケーションで作業メモリを確保する必要があります。必要な作業メモリのサイズは、nn::phtsel::GetWorkBufferSize() で取得することができます。

コード 12-4. 写真選択の作業メモリのサイズ取得

```
size_t nn::phtsel::GetWorkBufferSize();
```

作業メモリは、先頭アドレスが 4096 バイトのアライメント、サイズが 4096 の倍数で確保しなければなりません。また、**作業メモリにはデバイスメモリから確保したメモリ領域を指定しないでください。**

パラメータの設定と作業メモリの確保が完了したら、nn::phtsel::StartPhtsel() で写真選択のライブラリアプレットを起動することができます。nn::phtsel::StartPhtselNoCapture() はアプレットの背景にアプリケーションのキャプチャ画像を表示しない場合に呼び出します。

コード 12-5. 写真選択の起動

```
nn::applet::AppletWakeupState nn::phtsel::StartPhtsel(
                nn::phtsel::Parameter* pParameter, void* pWorkBuffer);
nn::applet::AppletWakeupState nn::phtsel::StartPhtselNoCapture(
                nn::phtsel::Parameter* pParameter);
```

pParameter には、設定を行った Parameter 構造体へのポインタを指定します。*pWorkBuffer* には、作業メモリの先頭アドレスを指定します。

アプリケーションに復帰したときには、*pParameter* の m_output メンバに実行結果が格納されています。

12.1.4. Mii 選択アプレット

アプリケーションで Mii を利用する際に、登録されている Mii から選択する際に呼び出すライブラリアプレットです。登録されている Mii とゲスト Mii(デフォルトの 6 体)から 1 体だけを選択することができます。ゲスト Mii を一覧に表示するかどうかや、操作画面を上画面または下画面のどちらに表示するかを選択することができます。

> **補足:** Mii 選択をアプリケーションで利用するには、CTR 似顔絵ライブラリ(ミドルウェア)が必要です。
> 使用方法は CTR 似顔絵ライブラリのドキュメントを参照してください。

12.1.5. 音声選択アプレット

アプリケーションの SE などに使用する音声を、ニンテンドー 3DS サウンドで録音した音声データから選択する際に呼び出すライブラリアプレットです。音声データは 1 つだけ選択することができます。音声の録音やデータの並べ替え、データの削除などはできません。なお、アプレットで選択可能な音声データは SD カードに保存されているものに限られています。

このライブラリアプレットを利用するには、ヘッダファイル(nn/voicesel.h)のインクルードと、ライブラリファイル(libnn_voicesel)の追加が必要です。

ライブラリアプレットの起動時に渡すパラメータは nn::voicesel::Parameter 構造体で定義されています。基本動作の設定は、パラメータの config メンバ(nn::voicesel::Config 構造体)に対して行い、抽出条件などの詳細な設定は input メンバ(nn::voicesel::Input 構造体)で行います。output メンバ(nn::voicesel::Output 構造体)には実行結果が格納されます。設定項目などの詳細については、サンプルデモを参照してください。

パラメータの設定が完了したら、nn::voicesel::StartVoiceSel() で音声選択のライブラリアプレットを起動することができます。

コード 12-6. 音声選択の起動

```
nn::applet::AppletWakeupState nn::voicesel::StartVoiceSel(
                nn::voicesel::Parameter* pParameter);
```

pParameter には、設定を行った Parameter 構造体へのポインタを指定します。

アプリケーションに復帰したときには、*pParameter* の output メンバに実行結果が格納されています。

12.1.6. エラー・EULA アプレット

すれちがい通信などの無線による通信機能を使用する際には、EULA(ニンテンドー3DS ネットワークサービスに関する利用規約)の確認と同意をユーザーから得ている必要があります。これらの通信機能を使用するアプリケーションは、ユーザーが最新の EULA に同意しているかどうかを確認し、同意を得ていない場合は、このライブラリアプレットで EULA の表示を行って同意を得ることができます。通信機能の使用に EULA への同意は必須ですが、アプリケーションの EULA 表示は任意です。EULA を表示しない場合は、通信時にエラーを表示して本体設定への誘導を行うなどの対応をしてください。

このライブラリアプレットは EULA の表示だけでなく、エラーメッセージの表示を行うことができます。インフラストラクチャ通信関連のライブラリ(AC や FRIENDS など)ならば、エラーコードを渡すことで対応するエラーメッセージを表示します。また、アプリケーション独自のエラーメッセージを表示させることもできます。

このライブラリアプレットを利用するには、ヘッダファイル(nn/erreula.h)のインクルードと、ライブラリファイル(libnn_erreula)の追加が必要です。

ライブラリアプレットの起動時に渡すパラメータは nn::erreula::Parameter 構造体で定義されています。エラーコードやエラーメッセージ、動作の設定は、パラメータの config メンバ(nn::erreula::Config 構造体)に対して行います。必ず、nn::erreula::InitializeConfig() で初期化を行ってから、設定を行ってください。設定項目などの詳細については、「アプレット仕様書」やサンプルデモを参照してください。

パラメータの設定が完了したら、nn::erreula::StartErrEulaApplet() でライブラリアプレットを起動することができます。

コード 12-7. エラー・EULA アプレットの起動

```
void nn::erreula::StartErrEulaApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::erreula::Parameter* pParameter);
```

pParameter には、設定を行った Parameter 構造体へのポインタを指定します。

設定により EULA 表示が行われた場合、EULA(ネットワークサービスの利用規約)の同意シーケンスを表示し、アプリケーションには EULA への同意・非同意の結果が返されます。

アプリケーションが関わるすべてのインフラ通信機能、およびすれちがい通信機能の関数は、アプリケーションが同意を必要とするバージョンの EULA にユーザーが同意していないと、EULA 非同意エラーを返します。NEX のログイン、ダウンロードタスクの登録、すれちがいボックスの作成などはすべて対象となります。同意を必要とする EULA バージョンは、自動的に ROM に埋め込まれます。必ず EULA 同意が必要かどうかを nn::cfg::IsAgreedEula() で確認し、EULA 未同意であれば本アプレットを呼び出すようにしてください。なお、EULA 未同意の状態で呼び出されたときのみ、正しく「EULA 同意」または「EULA 非同意」が返されます。

12.1.7. 拡張スライドパッド補正アプレット

このライブラリアプレットは、拡張スライドパッドに搭載されているスライドパッド(R)の操作感覚を補正するためのものです。拡張スライドパッドに対応するアプリケーションは、ユーザーがスライドパッド(R)の操作感覚を補正することができるように、このアプレットを起動するシーンを必ず用意しなければなりません。

> **補足:** SNAKE に搭載されている Cスティックは、常時接続されている拡張スライドパッドに搭載されているスライドパッド(R)として機能しています。SNAKE で動作しているアプリケーションが拡張スライドパッド補正アプレットを呼び出すと、Cスティックの補正方法を示すメッセージが表示されます。

このライブラリアプレットを利用するには、ヘッダファイル(nn/extrapad.h)のインクルードが必要です。また、CTR-SDK 11.3.x 以前を利用されている場合は、あわせてライブラリファイル(libnn_extrapad)の追加も必要となります。

ライブラリアプレットの起動時に渡すパラメータは nn::extrapad::Parameter 構造体で定義されています。動作の設定は、パラメータの config メンバ(nn::extrapad::Config 構造体)に対して行います。必ず、nn::extrapad::InitializeConfig() で初期化を行ってから、HOME ボタンやソフトウェアリセットへの対応を行うかを設定してください。

パラメータの設定が完了したら、nn::extrapad::StartExtraPadApplet() でライブラリアプレットを起動することができます。

コード 12-8. 拡張スライドパッド補正アプレットの起動

```
void nn::extrapad::StartExtraPadApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::extrapad::Parameter* pParameter);
```

pParameter には、設定を行った Parameter 構造体へのポインタを指定します。

12.1.8. EC アプレット

追加コンテンツやサービスアイテムの購入・管理を行うためのライブラリアプレットです。

> **補足:** EC 機能の詳細については、CTR-SDK の関数リファレンスを参照してください。

12.1.9. ログインアプレット

アカウントサーバーとの通信を行い、アカウント認証処理と各種サービストークンの取得処理などをアプリケーションの代わりに行うためのライブラリアプレットです。

> **補足:** ログインアプレットの詳細については、「3DS プログラミングマニュアル – 無線通信編」を参照してください。

12.2. システムアプレット

システムアプレットは、基本的に HOME メニューから起動されるアプレットですが、アプリケーションから起動や利用するためのライブラリを提供しています。アプリケーションが、システムアプレットを起動するために行わなければならない処理や、システムアプレットから復帰したときの処理は、基本的に HOME メニューの起動と復帰で行っている処理と同じです。

12.2.1. インターネットブラウザーアプレット

WEBBRS ライブラリを用いると、アプリケーションから URL を指定して、内蔵のインターネットブラウザーを起動することができます。起動方法はライブラリアプレットと類似していますが、起動したインターネットブラウザーを閉じるとアプリケーションが中断された状態で HOME メニューに戻る点が異なります。

> **注意:** 本体更新の状況によって、インターネットブラウザーがインストールされていない本体が存在します。そのため、本体にインターネットブラウザーがインストールされているかどうかを確認してから起動する必要があります。

> **補足:** WEBBRS ライブラリの詳細については、CTR-SDK の関数リファレンスを参照してください。

12.2.2. Miiverse アプリ、投稿アプリ

OLV ライブラリを用いると、アプリケーションで Miiverse を活用するための機能が利用できます。アプリケーションから Miiverse アプリや投稿アプリを起動したり、投稿データなどを受信することができます。これらは名称にアプリとついていますがシステムアプレットに含まれます。

> **補足:** OLV ライブラリの詳細については、CTR-SDK の関数リファレンスを参照してください。

12.2.2.1. Miiverse 投稿ページからのアプリケーション起動

Miiverse の投稿ページからアプリケーションを起動することができます。

投稿ページから起動できるようにするためには、アプリケーションに以下の設定が必要です。

- bsf ファイルに `EnableMiiverseJumpArgs:True` を指定
- 投稿時に `nn::olv::UploadPostDataByPostApp()` で、フラグ `FLAG_APP_STARTABLE` を指定

Miiverse アプリ上での操作

上記のフラグを設定された投稿を Miiverse アプリで閲覧すると、その投稿に起動ボタンが表示され、投稿をしたユーザーも、投稿をしていないユーザーでも対応するアプリケーションを持っていれば、画面の起動ボタンから起動できます。アプリケーションを持っていないユーザーは閲覧はできますが起動はできません。

図 12-1. 投稿ページからアプリケーション起動

13. 補助ライブラリ

この章では、電源や歩数計などの本体内蔵機能や、内蔵フォントなどの共有リソースを利用するための、補助的なライブラリについて説明します。

13.1. PTM ライブラリ

PTM ライブラリは、電源に関する情報の取得と RTC を利用したアラームを使用するためのライブラリです。

13.1.1. 初期化と終了

PTM ライブラリの初期化と終了は、nn::ptm::Initialize() と nn::ptm::Finalize() を呼び出して行います。

コード 13-1. PTM ライブラリの初期化と終了

```
nn::Result nn::ptm::Initialize();
nn::Result nn::ptm::Finalize();
```

どちらの関数も、ハードウェアが故障しているなどの状況でない限り、エラーを返すことはありません。

13.1.2. 電源に関する情報の取得

電源に関する情報として取得することができるのは、電源アダプタの接続状態、バッテリーの充電状態、バッテリーの残量レベルです。

コード 13-2. 電源に関する情報の取得

```
nn::ptm::AdapterState       nn::ptm::GetAdapterState();
nn::ptm::BatteryChargeState nn::ptm::GetBatteryChargeState();
nn::ptm::BatteryLevel       nn::ptm::GetBatteryLevel();
```

nn::ptm::GetAdapterState() を呼び出すことで、電源アダプタの接続状態を取得することができます。返り値には、以下の値が返されます。

表 13-1. nn::ptm::AdapterState 列挙子

値	説明
ADAPTERSTATE_NOCONNECTED	電源アダプタは接続されていません。
ADAPTERSTATE_CONNECTED	電源アダプタが接続されています。

nn::ptm::GetBatteryChargeState() を呼び出すことで、バッテリーの充電状態を取得することができます。返り値には、以下の値が返されます。

表 13-2. nn::ptm::BatteryChargeState 列挙子

値	説明
BATTERYCHARGESTATE_NOCHARGING	充電中ではありません。
BATTERYCHARGESTATE_CHARGING	充電中です。

nn::ptm::GetBatteryLevel() を呼び出すことで、バッテリーの残量をレベルで取得することができます。返り値には、以下の値が返されます。

表 13-3. nn::ptm::BatteryLevel 列挙子

値	説明
BATTERYLEVEL_0(BATTERYLEVEL_MIN)	バッテリーの残量は 0 % です。
BATTERYLEVEL_1	バッテリーの残量は 1 % ～ 5 % です。
BATTERYLEVEL_2	バッテリーの残量は 6 % ～ 10 % です。
BATTERYLEVEL_3	バッテリーの残量は 11 % ～ 30 % です。
BATTERYLEVEL_4	バッテリーの残量は 31 % ～ 60 % です。
BATTERYLEVEL_5(BATTERYLEVEL_MAX)	バッテリーの残量は 61 % ～ 100 % です。

13.1.3. RTC を利用したアラーム

この機能については、「8.5.2. RTC アラーム機能」を参照してください。

13.2. PL ライブラリ

PL ライブラリは、歩数計や内蔵(共有)フォントのように 3DS 本体に内蔵されている機能やリソースをアプリケーションで利用するためのライブラリです。PL ライブラリ自身には初期化や終了処理のための関数はありませんが、機能やリソースを利用するために、ほかのライブラリの初期化を必要とする場合があります。

13.2.1. 歩数計

歩数計の情報は最大で 120 ヶ月 (約 10 年)、1 時間ごとの歩数で記録されています。電源関係のモジュール内に歩数計の情報が記録されているため、歩数計の情報にアクセスするには、事前に PTM ライブラリの初期化を行っておく必要があります。PTM ライブラリの初期化については、「13.1.1. 初期化と終了」を参照してください。

歩数計に関する情報の取得には、以下の関数を使用します。

コード 13-3. 歩数計の情報を取得する関数

```
bool nn::pl::GetPedometerState();
u32 nn::pl::GetTotalStepCount();
s8 nn::pl::GetStepHistoryEntry(nn::pl::PedometerEntry* pEntry);
void nn::pl::GetStepHistory(u16 pStepCounts[], s32 numHours,
                            nn::fnd::DateTime start);
nn::Result nn::pl::GetStepHistoryAll(nn::pl::PedometerHistoryHeader& header,
                                     nn::pl::PedometerHistoryData& data);
```

nn::pl::GetPedometerState() は、加速度センサーが歩数計として動作しているかどうかを返します。歩数計として動作するのは電源がオンで蓋が閉じられている状態ですので、通常はスリープに移行しているため、アプリケーションが歩数計として動作しているかどうかを気にする必要はありません。

nn::pl::GetTotalStepCount() は、これまでの累計歩数を返します。

nn::pl::GetStepHistoryEntry() は、*pEntry* で指定された nn::pl::PedometerEntry の配列に歩数の記

録されているエントリの年と月を格納し、格納したエントリ数を返り値に返します。*pEntry* に指定する配列は、エントリ数の最大値(NUM_MONTHHISTORIES)が格納できるサイズで確保してください。

nn::pl::GetSetpHistory() は、*start* で指定された日時を含み、そこから *numHours* 時間分(未来方向のみ)の 1 時間ごとの歩数情報を *pStepCounts* に指定された配列に格納します。歩数情報のない時間には 0 が格納されます。

nn::pl::GetHistoryAll() は、すべての歩数情報を取得するための関数です。歩数情報はヘッダ情報とデータに分けて取得します。ヘッダ情報の順番とデータの順番は対応していますが、ヘッダ情報は必ずしも年月順に並んでいるとは限りません。何年何月の歩数情報であるかは、ヘッダ情報の monthInfo で確認することができます。その際、unusedCounter に INVALID_COUNTER が格納されているエントリは、歩数の記録されていない無効データであることを示します。

> **補足:** CTR-SDK には、歩数情報の閲覧と操作が可能な開発用ツール(PedometerChanger)が付属しています。

13.2.2. 内蔵フォント

アプリケーションからアクセスすることのできる共有リソースとして、内蔵フォントを用意しています。内蔵フォントは本体のリージョンによって、標準でロードされるフォントの種類が異なります。

利用することのできる内蔵フォントの種類は、nn::pl::SharedFontType 列挙子で以下のように定義されています。

表 13-4. nn::pl::SharedFontType 列挙子

値	説明
SHARED_FONT_TYPE_STD	日米欧フォント。日本、米州、欧州リージョンの標準です。
SHARED_FONT_TYPE_CN	中国フォント。中国リージョンの標準です。
SHARED_FONT_TYPE_KR	韓国フォント。韓国リージョンの標準です。
SHARED_FONT_TYPE_TW	台湾フォント。台湾リージョンの標準です。

13.2.2.1. 内蔵フォントのロード

内蔵フォントのロードは nn::pl::InitializeSharedFont() で行います。どのリージョンの本体でも、すべてのリージョン用のフォントを内蔵していますが、InitializeSharedFont() では本体のリージョンに対応するフォントのみがロードされます。

コード 13-4. 内蔵フォントのロード

```
nn::Result nn::pl::InitializeSharedFont();
nn::pl::SharedFontLoadState nn::pl::GetSharedFontLoadState();
```

nn::pl::InitializeSharedFont() の呼び出しで内蔵フォントのロードが開始されますが、関数の実行が完了した時点では、まだ内蔵フォントのロードが完了していない可能性があります。フォントのロードが完了したかどうかは nn::pl::GetSharedFontLoadState() の返り値(nn::pl::SharedFontLoadState)で確認することができます。

表 13-5. nn::pl::SharedFontLoadState 列挙子

値	説明
SHARED_FONT_LOAD_STATE_NULL	ロードは開始されていません。
SHARED_FONT_LOAD_STATE_LOADING	ロード中です。
SHARED_FONT_LOAD_STATE_LOADED	ロードは完了しています。
SHARED_FONT_LOAD_STATE_FAILED	ロードに失敗しました。

ロードされたフォントデータの先頭アドレスとサイズ、フォントの種類は、nn::pl::GetSharedFontAddress() と nn::pl::GetSharedFontSize()、nn::pl::GetSharedFontType() で、それぞれ取得することができます。

コード 13-5. ロードされたフォントデータの情報の取得

```
void*                   nn::pl::GetSharedFontAddress();
size_t                  nn::pl::GetSharedFontSize();
nn::pl::SharedFontType  nn::pl::GetSharedFontType();
```

ほかのリージョンの本体から送られたメッセージを表示するなど、本体のリージョン以外の内蔵フォントにしか含まれていない文字を表示したい場合は、nn::pl::MountSharedFont() で内蔵フォントのアーカイブをマウントし、対応する内蔵フォントのファイルをアプリケーションでロードしなければなりません。

コード 13-6. 内蔵フォントのロード（ほかのリージョン）

```
nn::Result nn::pl::MountSharedFont(const char* archiveName,
        nn::pl::SharedFontType sharedFontType, size_t maxFile,
        size_t maxDirectory, void* workingMemory, size_t workingMemorySize);
nn::Result nn::pl::UnmountSharedFont(const char* archiveName);
nn::Result nn::pl::GetSharedFontRequiredMemorySize(
        s32* pOut, nn::pl::SharedFontType sharedFontType, size_t maxFile,
        size_t maxDirectory);
```

archiveName には、フォントのアーカイブをマウントするアーカイブ名を指定します。

sharedFontType には、マウントする内蔵フォントの種類を指定します。内蔵フォントの種類によって、ロードするファイルが異なります。指定されたフォントが存在しない場合は、返り値に nn::pl::ResultSharedFontNotFound が返されます。

表 13-6. 内蔵フォントの種類と対応するファイル名（アーカイブ名に "font" を指定した場合）

内蔵フォントの種類	ファイル名
SHARED_FONT_TYPE_STD	font:/cbf_std.bcfnt.lz
SHARED_FONT_TYPE_CN	font:/cbf_zh-Hans-CN.bcfnt.lz
SHARED_FONT_TYPE_KR	font:/cbf_ko-Hang-KR.bcfnt.lz
SHARED_FONT_TYPE_TW	font:/cbf_zh-Hant-TW.bcfnt.lz

フォントファイルは LZ77 圧縮されていますので、フォントとして使用する前に CX ライブラリで解凍してください。

maxFile と *maxDirectory* には、同時に開くことができるファイルの数とディレクトリの数をそれぞれ指定します。*workingMemory* と *workingMemorySize* には、作業メモリとそのサイズを渡します。なお、必要となる作業メモリのサイズは `nn::pl::GetSharedFontRequiredMemorySize()` で取得してください。

フォントファイルのロードが完了したあとは、`nn::pl::UnmountSharedFont()` でアーカイブをアンマウントしてください。

13.2.2.2. フォントデータの使用

内蔵フォントのデータは、ctr_FontConverter で変換された bcfnt ファイルと同じフォーマットです。そのため、内蔵フォントを表示するには、FONT ライブラリの `nn::font::ResFont` クラスを利用しなければなりません。

FONT ライブラリによるフォントの表示方法については、関数リファレンスやサンプルデモを参照してください。

13.2.3. ゲームコイン

ユーザーが本体を持ち歩いた歩数で貯まるゲームコインを、アプリケーションで特典との交換など、自由に利用することができます。ゲームコインの所持枚数の取得や消費は PL ライブラリを介して行います。

ゲームコインを利用するには、ヘッダファイル(`nn/pl/CTR/pl_GameCoin.h`)のインクルードと、ライブラリファイル(`libnn_plCoin`)の追加が必要です。

> **補足:** CTR-SDK には、ゲームコインの所持枚数を設定する開発用ツール(PlayCoinSetter)が付属しています。

13.2.3.1. 初期化と終了

ライブラリの初期化と終了は `nn::pl::InitializeGameCoin()` と `nn::pl::FinalizeGameCoin()` の呼び出しで行います。

コード 13-7. ゲームコインライブラリの初期化と終了

```
void nn::pl::InitializeGameCoin();
void nn::pl::FinalizeGameCoin();
```

初期化の前に、FS ライブラリと PTM ライブラリを初期化しておく必要があります。多重呼び出しに対応していませんので、初期化関数が複数回呼び出された場合でも、1 回の `nn::pl::FinalizeGameCoin()` の呼び出しでライブラリの使用を終了します。

13.2.3.2. 所持枚数の取得

所持しているゲームコインの枚数は `nn::pl::GetGameCoinCount()` で取得することができます。

コード 13-8. 所持しているゲームコインの枚数の取得

```
nn::Result nn::pl::GetGameCoinCount(u16* pCount);
```

pCount には、現在所持しているゲームコインの枚数が格納されます。

この関数は負荷が高く、本体 NAND メモリへの書き込みを伴いますので、アプリケーションの起動時以外では、ゲームコインが増減する可能性のある、スリープや HOME メニュー、ライブラリアプレットからの復帰時などでのみ呼び出すようにしてください。

返り値に nn::pl::ResultGameCoinDataReset が返された場合は、データの破損などでゲームコインが初期化されたことを示しますが、成功(IsSuccess() が true を返す)として扱われます。ゲームコインが初期化されてしまったことをユーザーに知らせる場合は、この値が返されていないかをチェックしてください。

13.2.3.3. ゲームコインの消費

ゲームコインの消費は nn::pl::UseGameCoin() で行います。

コード 13-9. ゲームコインの消費

```
nn::Result nn::pl::UseGameCoin(u16* pCount, u16 useCount);
```

pCount には、処理に成功したときは消費後のゲームコインの枚数が格納されます。処理に失敗したときは不定値が格納されます。

useCount には、消費するゲームコインの枚数を指定します。所持している枚数よりも多くの枚数を指定した場合はゲームコインは消費されず、返り値にゲームコインの不足を示す nn::pl::ResultLackOfGameCoin が返されます。

この関数は負荷が高く、本体 NAND メモリへの書き込みを伴います。

14. 本体間赤外線通信

この章では、本体に搭載されている赤外線通信モジュールをアプリケーションで利用するために用意されている、IR ライブラリによる本体間通信について説明します。

補足: CTR-SDK には、赤外線通信による本体間通信の情報を確認する開発用ツール（IrCommunicatorChecker）が付属しています。

注意: SNAKE の拡張アプリで赤外線通信による本体間通信を行うアプリケーションは CTR - SNAKE 間の通信テストを行い、処理速度の差による通信障害が発生しないことを必ず確認してください。

注意: アプリケーションから本体間赤外線通信を行う際、赤外線通信を利用する他の機能を使用中ならば、その機能を先に終了させてください。

赤外線通信は以下の機能で利用されています。

- 拡張スライドパッド
- NFP（CTR のみ）

14.1. nn::ir::Communicator クラス

赤外線通信モジュールを利用した本体間通信を行う関数は、すべて `nn::ir::Communicator` クラスの static 関数として定義されています。

注意: `nn::ir::Communicator` クラスでは 1 対 1 での本体間通信のみをサポートしています。3DS 以外の機器との通信や複数の本体との通信に使用することはできません。

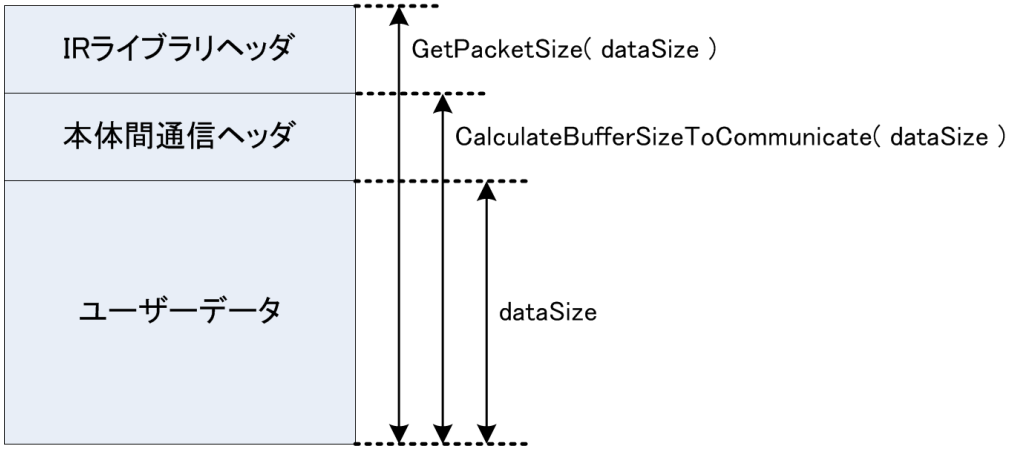
14.1.1. セキュリティ

送受信されるデータの暗号化と完全性検証はライブラリ内で行われます。また、送信のたびにインクリメントされるシーケンス番号を送信パケットに付与することで再送攻撃を防いでいます。

14.1.2. パケット

IR ライブラリが送受信するパケットには、通信制御用の IR ライブラリヘッダとセキュリティ情報などが含まれる本体間通信ヘッダをユーザーデータに付加したものが使用されます。

図 14-1. IR ライブラリのパケット



ヘッダ情報が付加されたときのユーザーデータのサイズは以下の関数で取得することができます。

コード 14-1. ヘッダ情報が付加されたユーザーデータのサイズ取得関数

```
size_t nn::ir::Communicator::GetPacketSize(size_t dataSize);
size_t nn::ir::Communicator::CalculateBufferSizeToCommunicate(size_t dataSize);
```

dataSize にはユーザーデータのサイズを指定します。

`GetPacketSize()` はパケット保存領域のサイズ計算に、`CalculateBufferSizeToCommunicate()` は送受信バッファのサイズ計算に利用します。

14.1.3. スリープへの対応

スリープに移行すると、接続が切断され、実行中の処理は強制的に終了されます。また、未送信のパケットや受信処理をしていないデータもすべて破棄されます。

スリープによる切断では相手の機器に切断要求は送信されません。相手に切断を通知する場合は、スリープ前に切断処理を実行し、その完了を確認してからスリープに移行してください。

14.1.4. 通信可能範囲

2 台の 3DS の本体背面にある赤外線受発光部同士が正対した状態で、以下の範囲にあれば赤外線通信が可能です。

表 14-1. 赤外線通信の通信可能範囲

項目	範囲
距離	下画面が水平な状態で、赤外線受発光部から 0～20 cm

14.2. 初期化

`nn::ir::Communicator` クラスの `Initialize()` を呼び出すことで、IR ライブラリの初期化が行われます。

コード 14-2. IR ライブラリの初期化関数

```
nn::Result nn::ir::Communicator::Initialize(
        void* pBuf, size_t bufSize,
        size_t receiveBufferDataSize, size_t receiveBufferManagementSize,
        size_t sendBufferDataSize, size_t sendBufferManagementSize);
```

pBuf と *bufSize* には、送受信されるパケットの管理などにライブラリが使用するバッファとそのサイズを指定します。ライブラリに渡すバッファはアプリケーションで事前に確保しますが、先頭アドレスが nn::ir::Communicator::BUFFER_ALIGNMENT (4096 Byte) のアライメント、サイズが nn::ir::Communicator::BUFFER_UNITSIZE (4096 Byte) の倍数でなければなりません。また、**デバイスメモリから確保したバッファは使用できません**。

ライブラリに渡されたバッファは複数の領域に分けて使用されます。各領域のサイズは、*receiveBufferDataSize*、*receiveBufferManagementSize*、*sendBufferDataSize*、*sendBufferManagementSize* で指定します。サイズ指定の詳細については、「14.2.1. ライブラリに与えるバッファ」を参照してください。

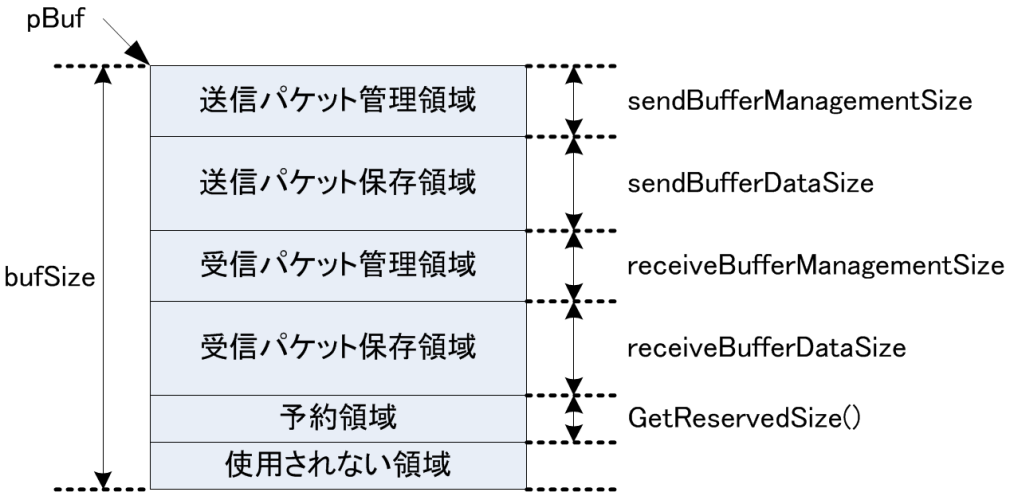
通信速度は 115200 bps 固定です。3DS の赤外線通信では、1 Byte のデータ送信にスタートビットとエンドビットの 2 ビットが付与されます。このため、実データの転送速度(bps)は、ボーレートの値に 8/10 を乗算した値となり、92160 bps(11520 Byte/s)となります。

14.2.1. ライブラリに与えるバッファ

IR ライブラリの送受信処理は非同期関数として実装されています。初期化関数でライブラリに与えるバッファは、主に送受信されるパケットを一時的に保存するために使用されます。

ライブラリに渡されたバッファは、以下の領域に区切られて使用されます。

図 14-2. IR ライブラリが使用するバッファ



各領域のサイズは、以下のように計算します。

14.2.1.1. 予約領域

ライブラリの動作に必要な固定サイズの領域です。

予約領域のサイズは、以下の関数で取得することができます。

コード 14-3. 予約領域のサイズを取得する関数

```
size_t nn::ir::Communicator::GetReservedSize();
```

14.2.1.2. 送信/受信パケット管理領域

送受信されるパケットの管理情報が格納される領域です。

同時に保持する最大のパケット数をもとにサイズを計算します。**接続処理のために、送信、受信ともに少なくとも GetManagementSize(1) で返されるサイズを指定しなければなりません。**

管理情報のサイズは、以下の関数で取得することができます。

コード 14-4. パケット管理情報のサイズを取得する関数

```
size_t nn::ir::Communicator::GetManagementSize(s32 dataNum);
```

dataNum に同時に保持する最大のパケット数を指定します。

14.2.1.3. 送信/受信パケット保存領域

送受信されるパケットが格納される領域です。**接続処理のために、送信、受信ともに少なくとも 32 バイト以上のサイズでなければなりません。**

データのサイズが可変の場合、領域のサイズには、最大サイズのデータで算出したパケットサイズ以上の値を指定しなければなりません。

データのサイズが固定の場合、領域のサイズには、1 データで算出したパケットのサイズに最大のパケット数を乗算した値を指定することができます。

以下の関数で、データのサイズからパケットサイズを計算することができます。

コード 14-5. パケットサイズを取得する関数

```
size_t nn::ir::Communicator::GetPacketSize(size_t dataSize);
```

dataSize にデータのサイズを指定します。

受信パケット保存領域の注意点

受信パケット保存領域には、ノイズによって発生する不正なパケットなどが保存されることがあります。そのため、そのあとに受信した正常なパケットが確実に保存されるように、**受信パケット保存領域は実際に受信するパケットよりも大きなサイズで指定することを推奨します**。例えば、各領域のサイズを決定したあと、使用されない領域のサイズを受信パケット保存領域のサイズに加算する方法があります。

以下のコード例では、サイズ固定のデータを送受信する場合に、受信パケット保存領域のサイズを最大限に確保しています。

コード 14-6. 領域サイズ計算のコード例

```
// バッファと送受信するデータのサイズ、保存するパケット数
static u8 buffer[4096] NN_ATRRIBUTE_ALIGN(4096);
size_t sendDataSize = 100;
size_t sendPacketNum = 10;
size_t recvDataSize = 50;
size_t recvPacketNum = 20;

// 必要なサイズを計算
```

```
size_t sendBufferSize =
        nn::ir::Communicator::GetPacketSize(sendDataSize) * sendPacketNum;
size_t sendManagementSize =
        nn::ir::Communicator::GetManagementSize(sendPacketNum);
size_t receiveBufferSize =
        nn::ir::Communicator::GetPacketSize(recvDataSize) * recvPacketNum;
size_t receiveManagementSize =
        nn::ir::Communicator::GetManagementSize(recvPacketNum);
size_t reservedSize = nn::ir::Communicator::GetReservedSize();

// 領域の総和がバッファサイズ以下であるかをチェック
NN_ASSERT((sendBufferSize + sendManagementSize + receiveBufferSize +
           receiveManagementSize + reservedSize) <= sizeof(buffer));

// 使用されない領域のサイズを受信パケット保存領域のサイズに加算
receiveBufferSize = sizeof(buffer) -
        (sendManagementSize + sendBufferSize +
         receiveManagementSize + reservedSize);
```

14.3. 接続

3DS 同士が赤外線通信を始めるには接続処理を行い、お互いを通信機器として認証しなければなりません。接続要求を待ち受ける側と要求する側で、呼び出す関数は明確に分かれています。待ち受ける側は WaitConnection() を、要求する側は RequireConnection() を呼び出してください。いずれの関数も非同期で処理を行いますので、接続処理が完了する前に制御が戻ります。

コード 14-7. 接続処理で使用する関数

```
static nn::Result nn::ir::Communicator::WaitConnection();
static nn::Result nn::ir::Communicator::WaitConnection( nn::fnd::TimeSpan
sendReplyDelay);
static nn::Result nn::ir::Communicator::RequireConnection();
static nn::ir::ConnectionStatus nn::ir::Communicator::GetConnectionStatus();
static nn::ir::TryingToConnectStatus
nn::ir::Communicator::GetTryingToConnectStatus();
```

非同期処理のため、接続処理が完了したかどうかは GetConnectionStatus() で得られるライブラリ全体の状態で確認します。接続処理が完了していれば、返り値に CONNECTION_STATUS_CONNECTED が返されます。また、CONNECTION_STATUS_TRYING_TO_CONNECT が返されたときは、GetTryingToConnectStatus() で詳細な処理状況を取得することができます。そのほかの返り値については「14.7. 接続状態の取得」を参照してください。

以下に、GetTryingToConnectStatus() の返り値の一覧を示します。

表 14-2. GetTryingToConnectStatus() の返り値

返り値	説明
TRYING_TO_CONNECT_STATUS_NONE	接続処理中ではありません。
TRYING_TO_CONNECT_STATUS_SENDING_REQUEST	接続要求を送信中です。
TRYING_TO_CONNECT_STATUS_WAITING_REPLY	接続要求への応答を待っています。
TRYING_TO_CONNECT_STATUS_WAITING_REQUEST	接続要求を待っています。

TRYING_TO_CONNECT_STATUS_SENDING_REPLY	接続要求への応答を送信中です。

14.3.1. 自動接続

同じメニューから赤外線通信を行うように、2 台の 3DS のどちらが接続要求を待ち受けるのかが明確ではない場合は自動接続を利用します。

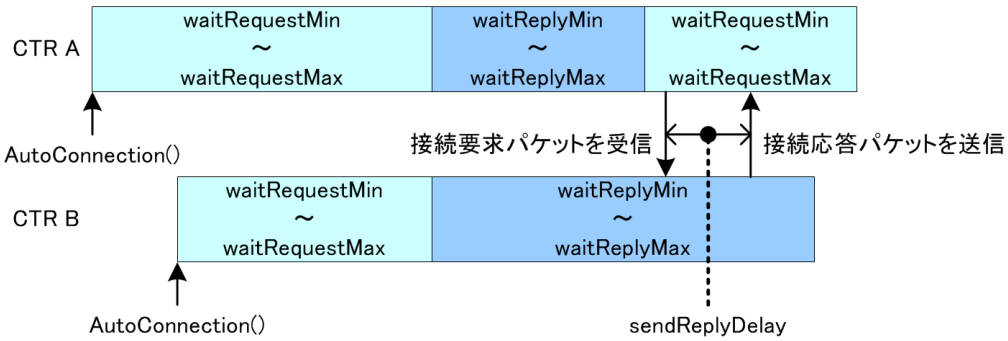
自動接続では、待ち受け側と要求側を自動的に切り替え、2 台の 3DS が待ち受け側と要求側に分かれたタイミングで接続処理が行われます。そのため、その時点での組み合わせによって接続関係が変化します。

コード 14-8. 自動接続で使用する関数

```
static nn::Result nn::ir::Communicator::AutoConnection();
static nn::Result nn::ir::Communicator::AutoConnection(
        nn::fnd::TimeSpan sendReplyDelay,
        nn::fnd::TimeSpan waitRequestMin, nn::fnd::TimeSpan waitRequestMax,
        nn::fnd::TimeSpan waitReplyMin, nn::fnd::TimeSpan waitReplyMax);
```

下図は、自動接続の動作を表した模式図です。

図 14-3. 自動接続の動作



引数なしで呼び出した場合は、初期化時のボーレートをもとにライブラリが計算した値が使用されます。アプリケーションで行っているほかの処理の影響を受け、引数なしで呼び出した自動接続では接続が行われない場合があります。そのような場合は、各引数を以下の点に注意しながらアプリケーションで指定してください。

sendReplyDelay は接続要求パケットを受信してから、接続応答パケットを送信するまでの時間です。デフォルトでは 3 ミリ秒が使用されています。少なくとも、相手の送受信が切り替わるまでの時間(3 ミリ秒)が必要です。

waitRequestMin と *waitRequestMax* は接続要求パケットの送信と接続応答パケットの受信待ちを行う最小時間と最大時間です。*waitRequestMin* は少なくとも、接続要求パケットの送信時間と接続応答パケットの受信時間、送受信の切り替え時間(3 ミリ秒)の合計よりも大きくなければなりません。

waitReplyMin と *waitReplyMax* は接続要求パケットの受信待ちと接続応答パケットの送信を行う最小時間と最大時間です。*waitReplyMin* は少なくとも、接続要求パケットの受信時間と接続応答パケットの送信時間、送受信の切り替え時間(3 ミリ秒)の合計よりも大きくなければなりません。

接続要求パケットの大きさは CONNECTION_REQUEST_PACKET_SIZE(8 バイト)で、接続応答パケットの大きさは CONNECTION_REPLY_PACKET_SIZE(5 バイト)で定義されています。送受信にかかる時間は実データの転送速度で計算します。パケットサイズの定義はバイト単位のため、ビット単位の転送速度から計算する場合は注意してください。

14.3.2. 接続関係

GetConnectionRole() で接続関係を取得することができます。接続関係によって、IR ライブラリの動作が異なることはありません。赤外線通信での役割を決定する指針として利用してください。

コード 14-9. 接続関係を取得する関数

```
static nn::ir::ConnectionRole nn::ir::Communicator::GetConnectionRole(void);
```

以下に、GetConnectionRole() の返り値の一覧を示します。

表 14-3. GetConnectionRole() の返り値

返り値	説明
CONNECTION_ROLE_NONE	接続処理前もしくは接続処理中のため、接続関係は決定していません。
CONNECTION_ROLE_TO_REQUIRE	接続を要求して接続状態になりました。 RequireConnection() または AutoConnection() で接続応答パケットを受信した側です。
CONNECTION_ROLE_TO_WAIT	接続要求を受け付けて接続状態になりました。 WaitConnection() または AutoConnection() で接続応答パケットを送信した側です。

14.4. 通信 ID の確認

接続処理が完了したあとは、異なるアプリケーションとの送受信を行わないように、事前にお互いの通信 ID を確認する必要があります。なお、通信 ID の確認が完了するまで、送信関数や受信関数を呼び出したときにエラーが返されます。

確認に使用する通信 ID は CreateCommunicationId() で取得します。

コード 14-10. 通信 ID を取得する関数

```
static bit32 nn::ir::Communicator::CreateCommunicationId(
        bit32 uniqueId, bool isDemo = false);
```

uniqueId には、弊社がタイトルごとに割り当てたユニーク ID を指定します。異なるタイトル間で通信を行う際は、どちらか片方の ID を指定してください。

isDemo には、ユニーク ID が共通となる製品版とダウンロードアプリ型体験版の間で赤外線通信を行いたくない場合に、体験版側のみ true を指定してください。体験版との通信を行うかどうかに関わらず、製品版では必ず false を指定してください。

> **補足:** 弊社からユニーク ID を割り当てられていないタイトルや実験プログラムで赤外線通信を使用する場合は、*uniqueId* にゲームソフト試作用コード(0xFF000～0xFF3FF)を指定してください。ただし、製品版では必ず弊社より割り当てられたユニーク ID を指定してください。

通信 ID の確認は、一方の 3DS で RequireToConfirmId() を、もう一方で WaitToConfirmId() を呼び出して行います。両方で指定された通信 ID と通信モード識別 ID、パスフレーズのすべてが一致する場合に両関数は成功し、送信関数や受信関数を使用することができるようになります。

コード 14-11. 通信 ID の確認を行う関数

```
static nn::Result nn::ir::Communicator::RequireToConfirmId(
        u8 subId, bit32 communicationId,
        const char passphrase[], size_t passphraseLength);
static nn::Result nn::ir::Communicator::WaitToConfirmId(
        u8 subId, bit32 communicationId,
        const char passphrase[], size_t passphraseLength,
        nn::fnd::TimeSpan timeout);
static bool nn::ir::Communicator::IsConfirmedId();
```

communicationId には、CreateCommunicationId() で生成された通信 ID を使用してください。

subId は、アプリケーションの複数のシーンで本体間通信を利用する場合に、異なるシーン間で通信が成立しないようにするための通信モード識別 ID です。シーンごとに異なる値を設定してください。

passphrase で指定するパスフレーズは、赤外線通信のパケットを暗号化するための鍵となりますので、容易に見破られるような文字列は避けて指定してください。パスフレーズはその長さが IR_PASSPHRASE_LENGTH_MIN 以上、IR_PASSPHRASE_LENGTH_MAX 以下の範囲で指定できます。

passphraseLength には、パスフレーズの長さを指定します。

RequireToConfirmedId() と WaitToConfirmedId() の両関数は、暗黙的にアプリケーションに割り当てられている nn::os::Event クラスを 1 つ消費することに注意してください。

通信 ID の確認が完了していても、切断もしくは再接続を行ったあとはそのたびに確認を行わなければなりません。

なお、すでに ID の確認が行われているかどうかは IsConfirmedId() で確認することができます。

14.5. 送信

通信 ID の確認が完了したあとであれば、Send() を呼び出すことで赤外線通信によるデータの送信を行うことができます。

コード 14-12. 送信処理で使用する関数

```
static nn::Result nn::ir::Communicator::Send(
        void *pBuffer, size_t dataSize, size_t bufferSize,
        bool restore = false);
static void nn::ir::Communicator::GetSendEvent(nn::os::Event* pEvent);
static nn::Result nn::ir::Communicator::GetLatestSendErrorResult(bool clear);
static void nn::ir::Communicator::GetSendSizeFreeAndUsed(
        size_t *pSizeFree, s32 *pCountFree,
        size_t *pSizeUsed, s32 *pCountUsed);
```

pBuffer と *bufferSize* には、送信バッファとそのサイズを指定します。ただし、送信バッファのサイズは送信するデータのサイズを CalculateBufferSizeToCommunicate() に渡して得たサイズ以上でなければなりません。また、先頭アドレスが nn::ir::Communicator::SEND_BUFFER_ALIGNMENT (4 Byte) のアライメントでなければなりません。

dataSize には、送信するデータのサイズを指定します。送信可能なデータサイズは 0 ～ 16316 Byte です。

送信されるデータは *pBuffer* の先頭から *dataSize* の範囲にあるデータです。送信が完了すると、バッファの内容は暗号化やヘッダの付加などが施された状態になります。送信後に元のデータが必要な場合は *restore* に true を指定してください。ただし、送信のみを行うよりも処理に時間がかかります。

実際の送信処理は非同期で行われ、Send() はライブラリに与えられたバッファに送信パケットを保存した時点で制御を戻します。実際に送信が行われたタイミングは、GetSendEvent() で取得できるイベントに通知されます。*pEvent* に渡され

たイベントは自動リセットイベントとして初期化されます。送信処理でエラーが発生したかどうかは、GetLatestSendErrorResult() で取得する Result クラスで判断することができます。引数 *clear* に true を渡して呼び出した場合は、ライブラリで保存されていた Result クラスのエラーがクリアされます。

ライブラリはバッファの状態を監視しており、できる限り早く送信パケットを赤外線通信モジュールから送信します。もし送信関数を短い期間で複数回実行した場合、ライブラリは一時的にバッファに複数のパケットをため、やがて先に保存されたパケットから順に取り出して送信します。

大きなパケットの送信や繰り返し送信することによって、保存したいパケットサイズの合計がバッファの容量を超えてしまった場合、ライブラリはそれ以上のパケットを保存せず破棄します。そのため、GetSendSizeFreeAndUsed() で送信パケットの管理領域と保存領域に空きがあるかを確認してから送信することを推奨します。特に、保存領域にはパケットとして保存されるため、GetPacketSize() で取得するパケットサイズ分の空きが必要であることに注意してください。

14.6. 受信

通信 ID の確認が完了したあとであれば、Receive() を呼び出すことで赤外線通信によるデータの受信を行うことができます。ライブラリは接続処理のあと、送信すべきパケットを保持していないときには常にデータを受信する状態になっているため、厳密にいうと Receive() は受信を行う関数ではなく、受信していたデータをバッファから取り出す関数です。受信したデータが正しいものであるかどうかは Receive() で取り出すまで判別できません。

コード 14-13. 受信処理で使用する関数

```
static nn::Result nn::ir::Communicator::Receive(
        void* pDst, size_t size, size_t *pReceiveSize,
        s32 *pRemainCount);
static void nn::ir::Communicator::GetReceiveEvent(nn::os::Event* pEvent);
static nn::Result nn::ir::Communicator::GetLatestReceiveErrorResult(
        bool clear);
static nn::Result nn::ir::Communicator::GetNextReceiveDataSize(size_t *pSize);
static nn::Result nn::ir::Communicator::DropNextReceiveData(s32 *pRemainCount);
static void nn::ir::Communicator::GetReceiveSizeFreeAndUsed(
        size_t *pSizeFree, s32 *pCountFree,
        size_t *pSizeUsed, s32 *pCountUsed);
```

pDst と *size* には、受信バッファとそのサイズを指定します。受信バッファの先頭アドレスは nn::ir::Communicator::RECEIVE_BUFFER_ALIGNMENT (4 Byte) のアライメントでなければなりません。また、受信バッファには暗号化やヘッダの付加などが施された状態のデータが一時的に保存されるため、そのサイズは実際に受信するデータのサイズを CalculateBufferSizeToCommunicate() に渡して得たサイズ以上でなければなりません。なお、受信に必要となるバッファのサイズは GetNextReceiveDataSize() であらかじめ取得することができます。

一度の呼び出しで 1 パケット分のデータが取り出されます。取り出されたデータのサイズは *pReceiveSize* に格納されます。また、データを格納すると同時に、*pRemainCount* には取り出すことのできる受信データの残数が格納されます。保存されているデータをすべて取り出したい場合は残数が 0 になるまで繰り返し呼び出してください。

GetReceiveEvent() で取得できるイベントには、受信パケットがバッファに保存されたタイミングが通知されますので、受信したデータをすぐに取り出したい場合に利用してください。*pEvent* に渡されたイベントは自動リセットイベントとして初期化されます。受信処理でエラーが発生したかどうかは、GetLatestReceiveErrorResult() で取得する Result クラスで判断することができます。引数 *clear* に true を渡して呼び出した場合は、ライブラリで保存されていた Result クラスのエラーがクリアされます。

GetNextReceiveDataSize() で取得したデータのサイズが想定する受信データのサイズとは明らかに異なるため、次

に受信する予定のデータが不要であると判断した場合は、1 パケット分のデータを DropNextReceiveData() で破棄することができます。

パケットの受信よりも受信データを取り出す頻度が低く、保存したいパケットサイズの合計がバッファの容量を超えてしまった場合、ライブラリはそれ以上のパケットを保存せず破棄します。GetReceiveSizeFreeAndUsed() で受信パケットの管理領域と保存領域に空きがあるかを確認することができますので、受信の管理に利用してください。

受信データが残っている状態で現在の接続が切断された場合、次の接続を開始するまでの間は、残っているデータを受信できます。

14.7. 接続状態の取得

GetConnectionStatus() で接続状態を取得することができます。また、接続状態が変化したタイミングは、GetConnectionStatusEvent() で取得できるイベントに通知されます。*pEvent* に渡されたイベントは自動リセットイベントとして初期化されます。

コード 14-14. 接続状態の取得に使用する関数

```
static void nn::ir::Communicator::GetConnectionStatusEvent(
        nn::os::Event* pEvent);
static nn::ir::ConnectionStatus nn::ir::Communicator::GetConnectionStatus();
```

以下に、GetConnectionStatus() の返り値の一覧を示します。

表 14-4. GetConnectionStatus() の返り値

返り値	説明
CONNECTION_STATUS_STOPPED	赤外線モジュールが停止している状態です。
CONNECTION_STATUS_TRYING_TO_CONNECT	接続処理中を示す状態です。
CONNECTION_STATUS_CONNECTED	接続状態です。
CONNECTION_STATUS_DISCONNECTING	切断処理中を示す状態です。
CONNECTION_STATUS_FATAL_ERROR	故障などの致命的なエラーが発生している状態です。

14.7.1. 故障時の対応

接続状態が CONNECTION_STATUS_FATAL_ERROR のときは、赤外線モジュールが故障した可能性があります。このとき、Finalize() を除く、Result を返す関数すべてが常に nn::ir::ResultFatalError() を返すようになります。以降の処理で赤外線通信を利用することはできなくなりますので、すぐに IR ライブラリを終了させてください。

赤外線モジュールが故障した可能性がある一方、この状態は電源を入れなおすことで復旧する可能性もあります。そのため、ユーザーに対して、故障の可能性と対応についてメッセージを表示することを推奨します。

14.8. 切断

赤外線通信の切断は Disconnect() の呼び出しで行われます。

コード 14-15. 切断処理で使用する関数

```
static nn::Result nn::ir::Communicator::Disconnect(void);
static nn::Result nn::ir::Communicator::ClearSendBuffer(void);
static nn::Result nn::ir::Communicator::ClearReceiveBuffer(void);
```

Disconnect() は、通信相手に切断要求を送信します。また、接続処理の途中であれば処理を中断します。切断処理は非同期で行われるため、処理が完了する前に制御が戻ります。

呼び出した時点で未処理の送信パケットがある場合、送信パケットを破棄して切断処理を行いますので、必要であれば、すべてのパケットの送信完了を確認してから本関数を実行してください。未送信のパケットすべてを明示的に破棄する場合は ClearSendBuffer() を呼び出してください。

切断される前に受信したパケットについては、切断処理のあとでも Receive() で取り出すことができます。しかし、次の接続および接続認証処理を開始した時点で破棄されることに注意してください。受信しているすべてのパケットを明示的に破棄する場合は ClearReceiveBuffer() を呼び出してください。

14.9. 終了

IR ライブラリの使用を終了するときは Finalize() を呼び出してください。

コード 14-16. 終了処理で使用する関数

```
static nn::Result nn::ir::Communicator::Finalize(void);
```

初期化がされていない状態で呼び出すと、nn::ir::ResultNotInitialized() のエラーが返されます。

15. TWL モードと TWL 本体の動作の違い

ここでは、3DS の TWLモード(DSi 互換モード)と TWL 本体(ニンテンドーDSi)では、同じアプリケーションでも一部動作に違いがあります。ここでは、その動作の違いを紹介します。これらの情報は、3DS 上でも動作するニンテンドーDS シリーズのアプリケーション(特にニンテンドーeショップでも販売される DSi ウェア)を開発する際の注意点として参照してください。

> **補足:** DSi ウェアをインポートする方法については ctr_makecia のリファレンスを参照してください。

15.1. 表示関連

15.1.1. LCD

LCD の設置方向の関係(「5.5. GX ライブラリの初期化」参照)で、液晶画面の走査方向が TWL とは異なります。また、エミュレーションのために約 1.3 フレーム遅れて描画されます。

15.2. 入力関連

15.2.1. タッチパネル

ドットバイドットモードで起動した場合、画面(表示域)の範囲外をタッチすることで、通常発生しにくい画面端のタッチを SDK(TWL/NITRO)の API が検出する可能性があります。

15.2.2. キー入力

スライドパッドによる十字ボタンのエミュレーション(「6.2.1. デジタルボタンとスライドパッド」参照)は、スライドパッドをスライドさせていない(入力と判定されない)ときはリアルタイムで十字ボタンの状態が更新されますが、スライドパッドをスライドさせているときは十字ボタンの状態が約 1 フレームに 1 度更新されます。そのためスライドパッドで操作している場合、更新タイミングによっては十字ボタンの入力が正確に反映されない可能性があります。

スライドパッドと十字ボタンの両方を使用して十字ボタンの上と下(もしくは左と右)を同時に入力したとしても、十字ボタンの入力は禁則処理(上下同時ならば上、左右同時ならば左が入力されたものとする)が行われた上でアプリケーションに通知されます。禁則入力が絶対に通知されないため、デバッグモードに入るなどのトリガとして禁則入力を使用しているアプリケーションは正しく動作しません。

15.2.3. 蓋閉じ

アプリケーションから見ると、HOME メニューが表示されている間は蓋閉じされた状態と同じです。通常の蓋閉じと同様に `PAD_DetectFold()` は `TRUE` を返しますが、キーやタッチパネルの入力はアプリケーションには通知されません。

15.2.4. サウンドボリューム

サウンドボリュームがスライドスイッチのため、ニンテンドーDSi では不可能なほど高速にサウンドボリュームが変更される可能性があります。

15.3. OS 関連

15.3.1. リセット、シャットダウン

OS_RebootSystem() の処理にかかる時間が TWL で動作するときに比べて非常に長くなります。

15.3.2. アプリジャンプ

システムメニューバージョンのファイル名と内容が TWL と異なるため、それらの情報をチェックするようなルーチンが組み込まれていると、TWL モードでのみ誤動作する可能性があります。

15.4. 本体設定関連

15.4.1. リージョンコード

3DS では豪州リージョンが欧州リージョンに含まれるようになったため、TWL のリージョンで豪州リージョンにあたるアプリケーションは、3DS では欧州リージョンの本体で動作させることができます。

表 15-1. TWL のリージョンと動作可能な 3DS 本体のリージョン

TWL リージョン	動作可能な 3DS 本体のリージョン
JP(日本)	JP(日本)
US(米州)	US(米州)
EU(欧州)	EU(欧州)
AU(豪州)	EU(欧州)
EU(欧州)、US(米州)	EU(欧州)または US(米州)
EU(欧州)、AU(豪州)	EU(欧州)
EU(欧州)、AU(豪州)、US(米州)	EU(欧州)または US(米州)
CN(中国)	CN(中国)
KR(韓国)	KR(韓国)

15.4.2. 国設定

3DS では設定可能ですが TWL では該当のリージョンにない国設定が行われている場合、OSOwnerInfoEx 構造体の country の値は 254(OTHER)となります。

15.4.3. 言語設定

3DS では設定可能ですが TWL では該当のリージョンにない言語設定が行われている場合、OSOwnerInfoEx 構造体の language の値は日本リージョンならば OS_LANGUAGE_JAPANESE(日本語)、そのほかのリージョンならば OS_LANGUAGE_ENGLISH(英語)となります。

16. 付録:拡張スライドパッドを利用する場合のフロー図

拡張スライドパッドには機器の状態を示すインジケータなどが搭載されていないため、アクティブであるかどうかや電池残量が減少しているか、CTR 本体と接続されているかどうかなどを外部から視認することができません。そのため、ユーザーにストレスなく拡張スライドパッドを使っていただくためには、拡張スライドパッドの検出や接続、再検出のフローに工夫が必要となります。

> **補足:** SNAKE では C スティック、ZL ボタン、ZR ボタンが本体に搭載されているため、拡張スライドパッドが常に接続されている状態とみなされます。また、電池残量が減少している状態にはなりません。
>
> そのため、アプリケーションが動作する本体が SNAKE であることを確認した上であれば、本章で示す利用フローの一部を省略することができます。

本章では、その一例として、弊社の推奨する利用フローをいくつか紹介しますので、実装時の参考にしてください。なお、フロー内で背景色が黄色になっている処理は表示するメッセージの例です。

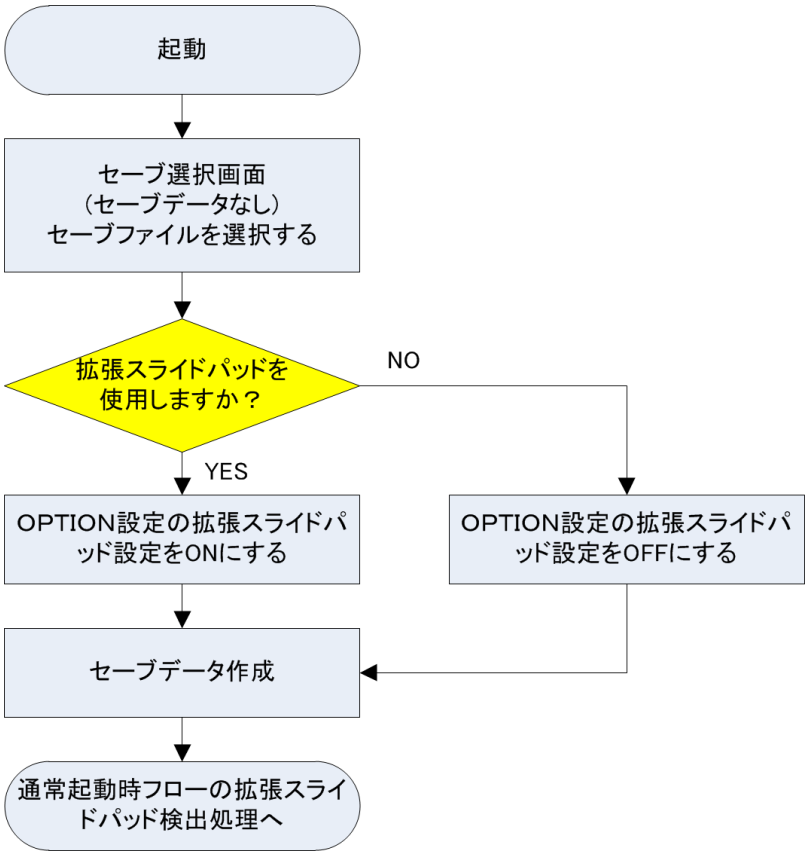
ユーザーに表示するメッセージに「拡張スライドパッド」と表記するか、「C スティック」と表記するかは以下を参考にしてください。

- 「拡張スライドパッド」と表記
  - CTR アプリケーション
  - SNAKE 対応タイトルで、CTR では拡張スライドパッドを、SNAKE では C スティックを使用する場合
- 「C スティック」と表記
  - SNAKE 専用タイトル
  - SNAKE 対応タイトルで、CTR では拡張スライドパッドを使用せず、SNAKE では C スティックのみを使用する場合

16.1. 初回起動時のフロー

ユーザーが拡張スライドパッドを使用しているかどうかをセーブデータに持たせる場合は、初回起動時に以下のようなフローでユーザーに確認することを推奨します。

図 16-1. 初回起動時のフロー

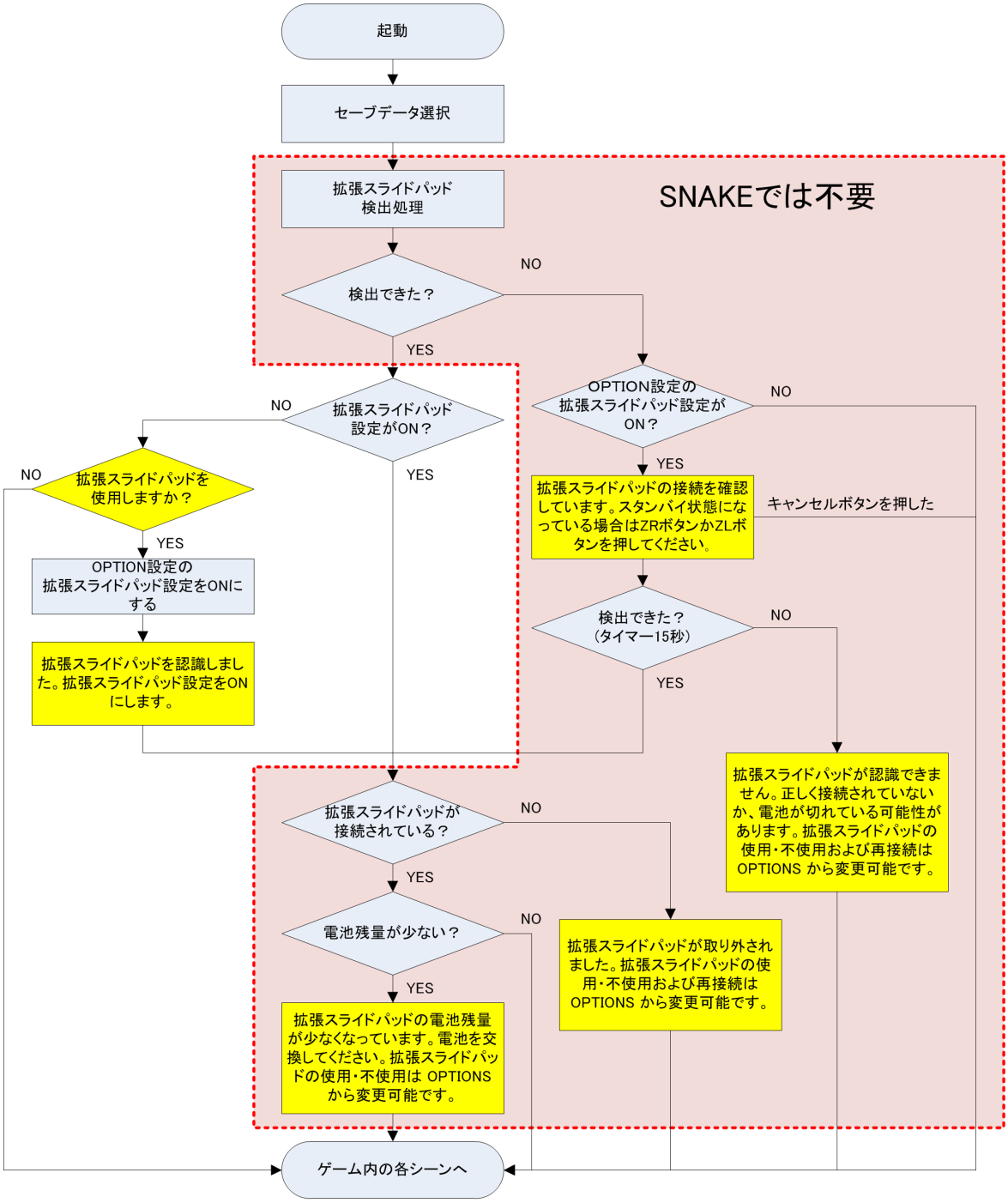


16.2. 通常起動時のフロー

「16.1. 初回起動時のフロー」と同じく、セーブデータに拡張スライドパッドの使用しているかどうかの情報を持たせている場合は、通常起動時に以下のようなフローでユーザーに確認することを推奨します。

なお、SNAKE では対応が不要なフローの範囲は図中に赤点線枠、薄赤背景で示されています。

図 16-2. 通常起動時のフロー

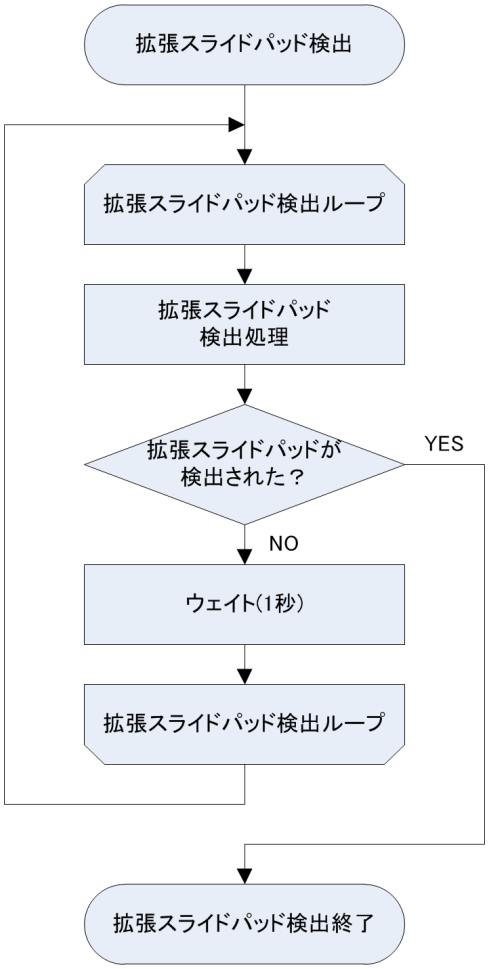


16.3. 拡張スライドパッド検出時のフロー

> **補足:** SNAKE では以下のフローを考慮する必要はありません。

起動後の初めての検出である場合を除き、拡張スライドパッドの検出時には以下のようなフローを推奨します。下図は、メインスレッドとは別のスレッドから自動で検出する場合を想定しています。明示的に検出する、または接続のオプションを設ける場合は「16.4. 検出オプションのフロー」を参照してください。

図 16-3. 拡張スライドパッド検出フロー



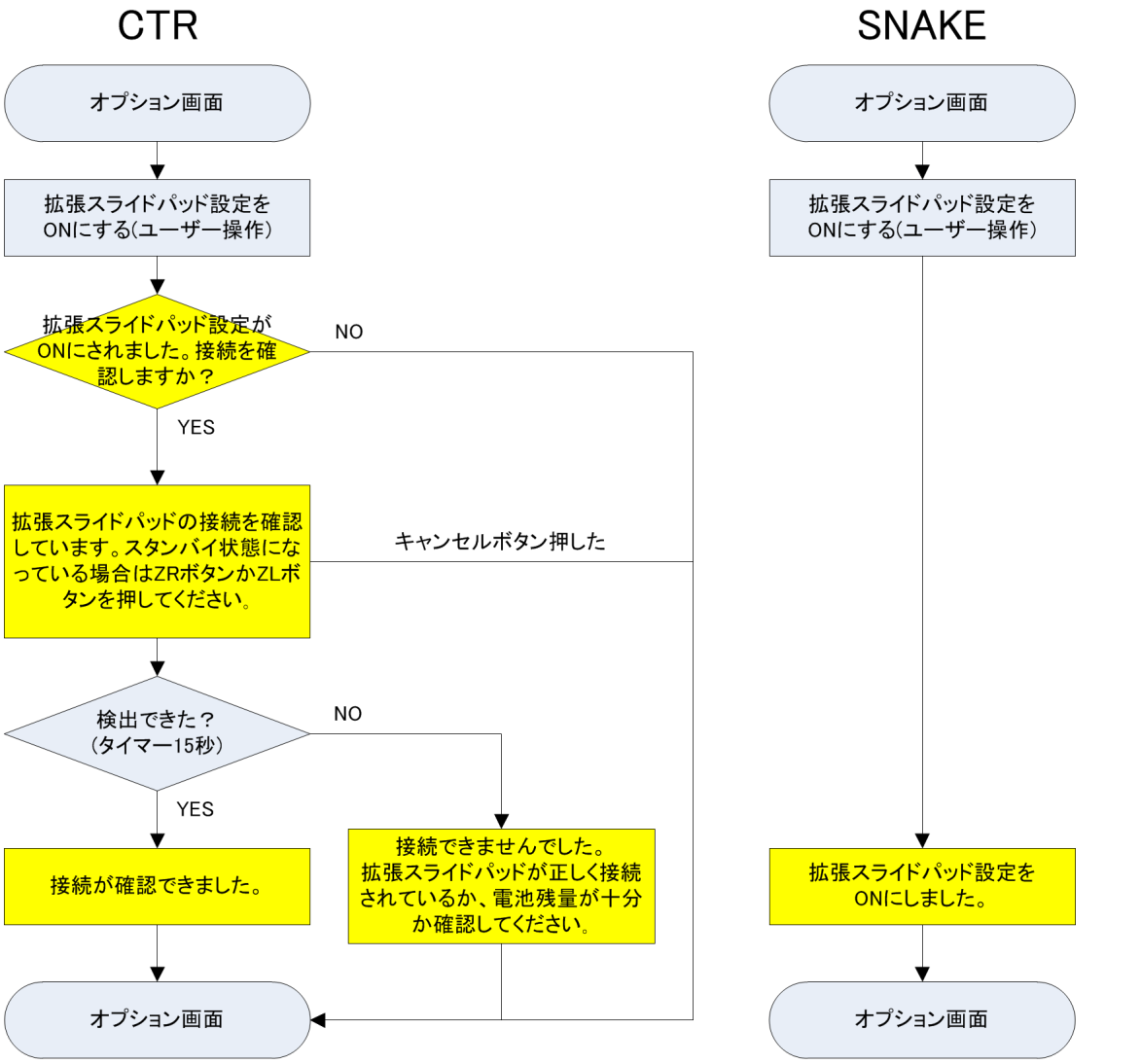
16.4. 検出オプションのフロー

オプション画面などに再接続フローへの遷移機能を用意している場合、拡張スライドパッドを使用しているときのフローは以下のようにすることを推奨します。

16.4.1. 拡張スライドパッドを使用するモードへ遷移する機能

拡張スライドパッドを使用する、または有効にするモードへ遷移するオプションを設ける場合は、以下のようなフローを推奨します。

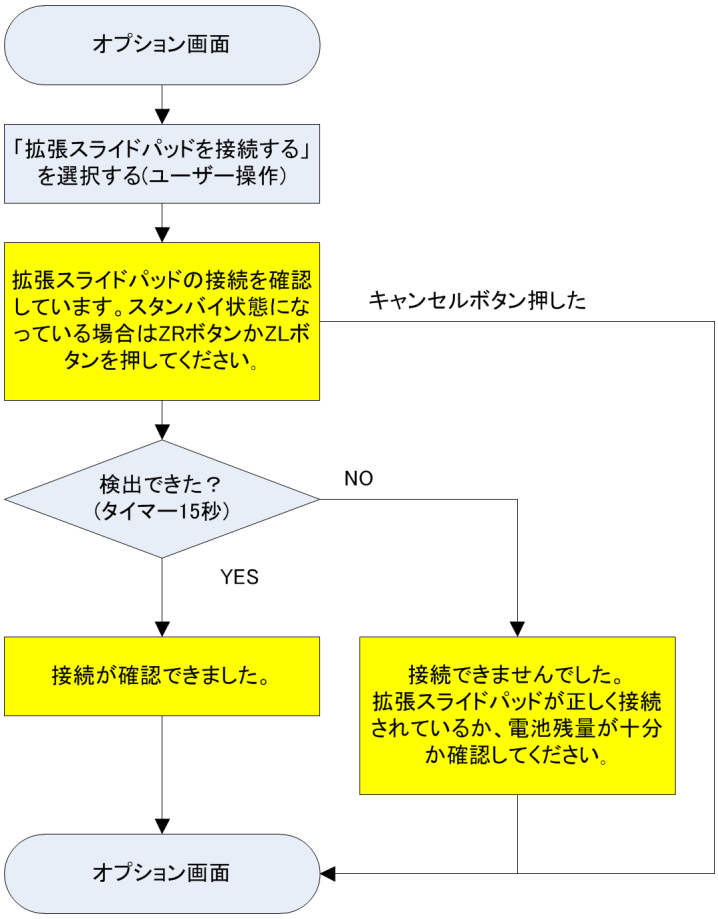
図 16-4. 拡張スライドパッド有効設定フロー



補足： SNAKE ではサンプリングを開始した時点で必ず接続状態になっていますので、以下のフローは不要になります。

単に拡張スライドパッドと接続するオプションを設ける場合は、以下のようなフローを推奨します。

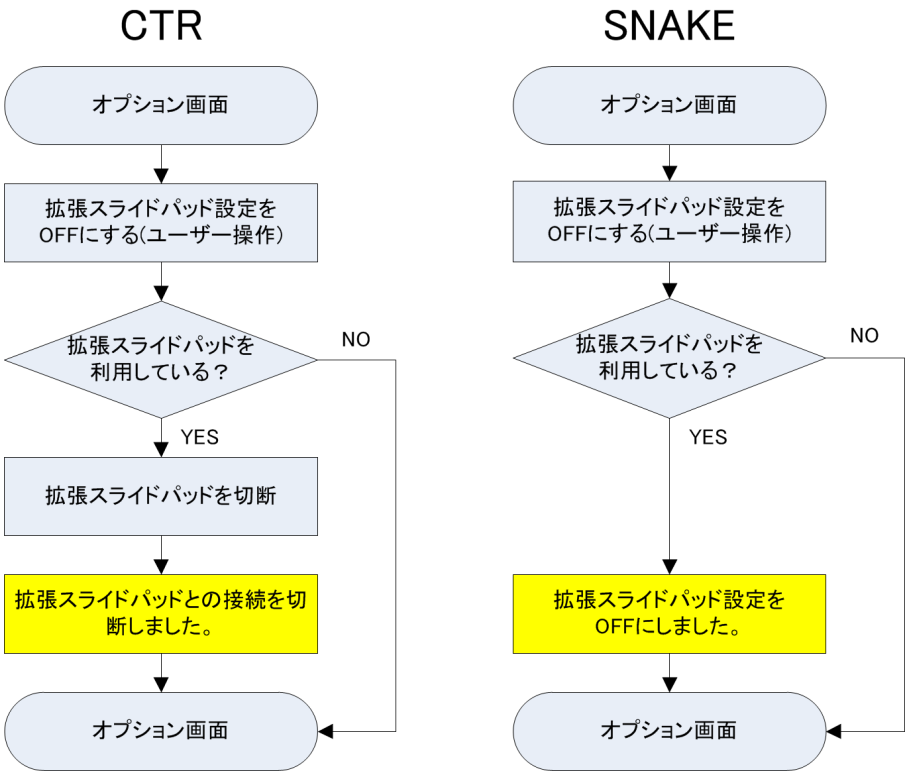
図 16-5. 拡張スライドパッド接続フロー



16.4.2. 拡張スライドパッド切断時のフロー

接続されている拡張スライドパッドとの接続を切断するオプションを設ける場合は、以下のようなフローを推奨します。

図 16-6. 拡張スライドパッド切断フロー



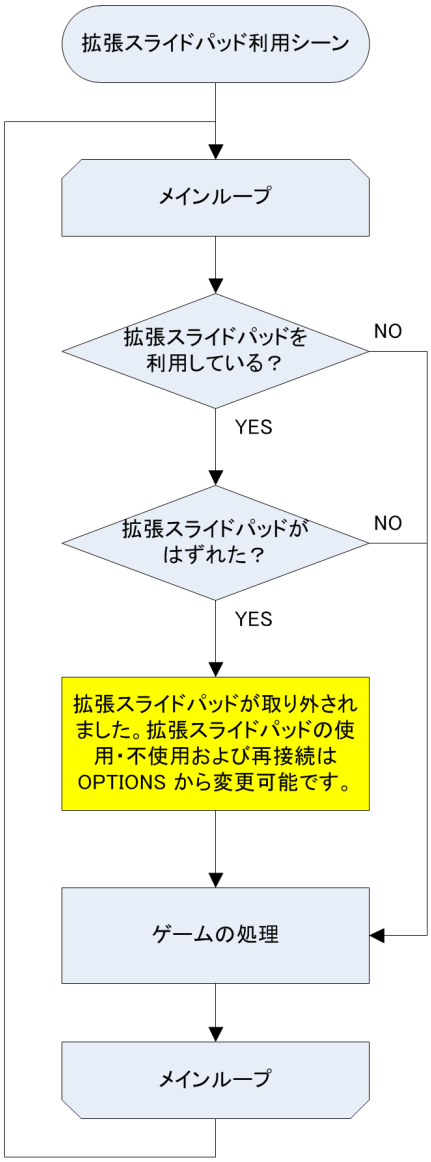
16.5. 通常の拡張スライドパッド使用時のフロー

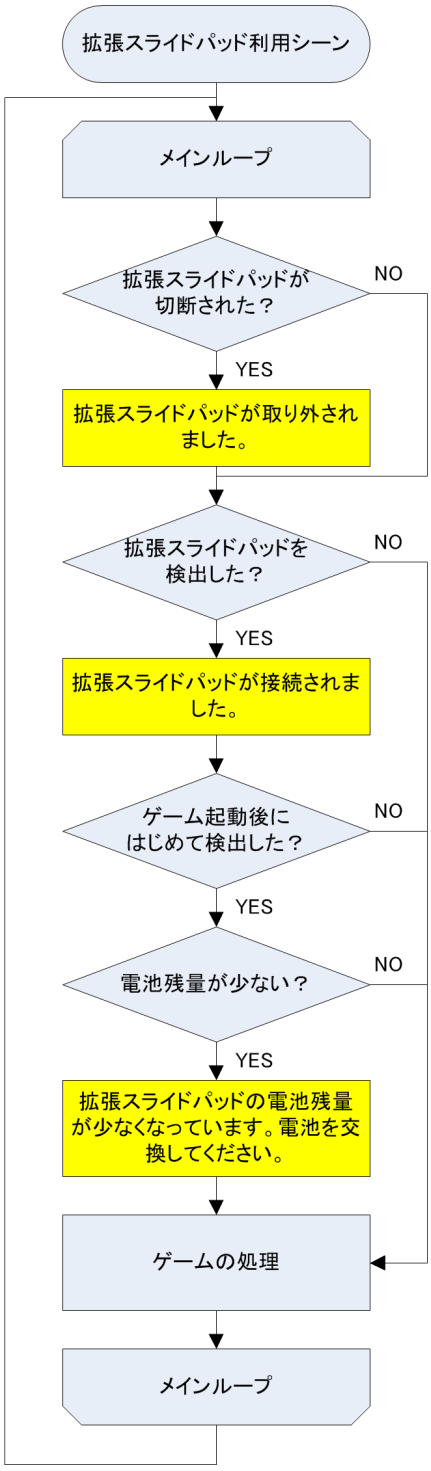
> **補足:** SNAKE では以下のフローを考慮する必要はありません。

拡張スライドパッドを利用している最中のフローは、以下を推奨します。

下図では、メインスレッドに組み込む場合に、アプリケーションに拡張スライドパッドの検出オプションを設けているか、常時検出しているかで処理を分けています。左のフローは拡張スライドパッドを再検出するオプションを設けている場合、右のフローは拡張スライドパッドを常時検出している場合です。なお、メインスレッドとは別のスレッドで常時検出を行う場合は「図 16-3. 拡張スライドパッド検出フロー」を参照してください。

図 16-7. 通常使用時のフロー





17. 付録:標準アプリの SNAKE での動作確認について

SNAKE には、3D ブレ防止機能・入力デバイスの追加・カメラの性能アップなど、新たな機能が搭載されていますが、これらの機能を実現するため、SNAKE は CTR とは一部非互換なハードウェアとなっており、システム処理も異なっています。
SNAKE の追加機能の詳細については、「3DS オーバービュー」をご参照ください。

標準アプリを CTR で動作させた場合と SNAKE で動作させた場合で挙動が変わる可能性がありますので、**標準アプリであったとしても CTR と SNAKE の両方で動作確認をお願いします**。

動作確認の際の注意点は以下の通りです。

パフォーマンスを追求しているシーン	SNAKE では CTR と比べ、メモリアクセスのレイテンシが増加するため、メモリアクセスが待たされ、アプリケーションの処理に時間がかかることがあります。アプリケーションの処理時間が最大 5 %ほど余分にかかっているというシーンも見つかっています。CTR でギリギリのチューニングを行っている場合、SNAKE では処理落ちが発生する可能性があります。 アプリケーション側の対処としては、処理を軽くする、もしくは、SNAKE 上では三倍速(804 MHz)動作する拡張アプリとしてビルドする方法があります。詳細は「4.2. 標準アプリ、拡張アプリ」をご参照ください。
ローカル通信	SNAKE と CTR では、ローカル通信のシステム処理が異なるため、両方が混在したローカル通信時の挙動が想定外になる場合があります。 ローカル通信の動作検証については SNAKE と CTR を混ぜる、それぞれを親機にするなどの確認も行ってください。
カメラの起動・終了	SNAKE の 3D ブレ防止機能はカメラからの入力を利用するため、アプリケーションがカメラを利用すると3Dブレ防止機能は無効になります。 カメラライブラリの初期化・終了の処理において、3D ブレ防止機能の有効・無効の切り替えも行いますので、SNAKE では CTR と比べてカメラライブラリの初期化・終了の処理時間が少し長くなることにご注意ください。
HOMEメニュー	SNAKE では、HOME メニューは拡張モードで動作し、標準アプリを起動する際に標準モードに切り替わります。 厳密には、標準モードに切り替わるのは、`nn::applet::Enable()` 呼び出し時であり、それまでは拡張モードで動作します。詳細は「4.2.4. 動作モードが切り替わるタイミング」をご参照ください。 HOME ボタン押下時には、中断中のアプリケーションも含めシステム全体が拡張モードで動作することにご注意ください。
入力インターフェースの違い	● SNAKE の IN カメラは CTR のカメラと比べて広角になっています。 ● SNAKE の IN/OUT カメラは CTR のカメラと比べノイズが低減されています。 ● SNAKE では液晶の自動輝度調整機能が動作します。(自動輝度調整機能はカメラからの入力を利用するため、アプリケーションがカメラを利用している間は無効になり、直前の画面輝度が保持されます。) ● SNAKE の C スティックは、拡張スライドパッドのスライドパッドとは使用感が異なります。 ● SNAKE の C スティックと拡張スライドパッドのスライドパッドでは、スリープ復帰時の状態変化が異なります。詳細は「6.2.7.2. 拡張スライドパッドとの相違点」をご参照ください。 ● SNAKE の NFC と NFC リーダー/ライター では使用感が異なります。

18. 付録:IS-SNAKE DevKit を使用した 3DS のアプリケーション開発の注意点

IS-SNAKE DevKit は SNAKE を対象としたアプリケーション開発支援ツールですが、CTR を対象とした標準アプリの開発にも使用することができます。

「4.2. 標準アプリ、拡張アプリ」で述べたとおり、標準アプリは CTR、SNAKE とも標準モードで動作しますが、拡張アプリは CTR と SNAKE で動作させる際、CPU の動作速度、使用できるメモリサイズが異なります。通常、IS-SNAKE DevKit では拡張アプリは拡張モードで動作しますが、強制 CTR 互換モードを使うことで、拡張アプリであっても標準モードで動作確認をすることができます。

ただし、「17. 付録:標準アプリの SNAKE での動作確認について」に記載の表のように、一部仕様に互換性がないため IS-SNAKE DevKit での動作は CTR 上での挙動と異なるものがあります。標準アプリ、拡張アプリいずれの場合でも必ず CTR 環境での動作確認を行ってください。

> **補足:** 「強制 CTR 互換モード」については IS-CTR-DEBUGGER のヘルプを参照してください。

更新履歴

Version 1.6 2016-06-24

変更

- 17. 付録:標準アプリの SNAKE での動作確認について
  - カメラ使用中は自動輝度調整機能が無効になることを追記しました。

Version 1.5 2016-05-10

追加

- 18. 付録:IS-SNAKE DevKit を使用した 3DS のアプリケーション開発の注意点

変更

- 3.1.2. デバイスメモリ
  - デバイスメモリをリサイズする場合について条件を追記しました。
- 3.1.4. ヒープメモリ
  - ヒープメモリをリサイズする場合について条件を追記しました。
- 17. 付録:標準アプリの SNAKE での動作確認について
  - IS-SNAKE DevKit での動作確認について別のページに移動しました。

Version 1.4 2015-11-05

追加

- 12.1.9. ログインアプレット

変更

- 全般
  - アライメント制約の定義名を追記しました
- 4.3.2. ライブラリ
  - ライブラリ一覧に aacdec、aacenc、act、nfp、qtm を追記しました。
- 6.2.6. 拡張スライドパッド
  - ほかに赤外線通信を使用中の場合はその機能を先に終了させることについて追記しました。
- 6.2.6.3. 初期化
  - 拡張スライドパッド初期化関数が指定するバッファのサイズが 12288 Byte に変更されました。
- 9.4. スケジューリング
  - Sleep()の過度な連続呼び出しはパフォーマンス低下要因になることを追記しました。
- 12.1.1. 各ライブラリアプレットに共通する情報
  - ログインアプレットについての情報を追記しました
- 12.1.7. 拡張スライドパッド補正アプレット
  - CTR-SDK 11.4 以降ではライブラリ使用時 libnn_extrapad のインクルードが不要になりました。
- 14. 本体間赤外線通信
  - ほかに赤外線通信を使用中の場合はその機能を先に終了させることについて追記しました。

Version 1.3 2015-04-28

変更

- 2.7.1. ゲームカードスロット
  - ゲームカードスロットに挿入できるカードの種類について修正しました。
- 2.9.1. スピーカー
  - 本体スピーカーの音圧周波数特性について情報を追加しました。
- 5.3.2. HOME ボタンへの対処
  - メニュー表示中のデバイスへの対処について、NFCを追記しました。
- 5.3.9. ニンテンドーeショップへのジャンプ
  - パッチページへのジャンプについての情報を追加しました。
- 7.1.8. レイテンシエミュレーション
  - Config ツールのメニュー階層を現行のツールに合わせて修正しました。
- 7.3. セーブデータ
  - セーブデータ領域のマウント時エラーハンドリングについて、対処すべきエラーを追記しました。
  - メディアによっては返さないエラーがあることについて情報を追記しました。
- 7.4. 拡張セーブデータ
  - 拡張セーブデータ領域のマウント時エラーについて情報を追記しました。
- 7.6.1. 3DS カード
  - セーブデータに関するエラーハンドリングについて記述を削除しました。
  - Config ツールのメニュー階層を現行のツールに合わせて修正しました。
- 17. 付録:CTR タイトルの SNAKE での動作確認について
  - IS-SNAKE DevKit の強制 CTR 互換モードについて注意事項を追記しました。
  - 入力インターフェースの違いについて情報を追記しました。

Version 1.2 2015-01-15

追加

- 12. アプレット
  - 別々の章に存在していたライブラリアプレットとシステムアプレットの内容を統合しました。
- 12.2. システムアプレット
- 12.2.2.1. Miiverse 投稿ページからのアプリケーション起動
- 17. 付録:CTR タイトルの SNAKE での動作確認について

変更

- 1. はじめに
  - アプレットページの追加に伴い、リンク先と説明を修正しました。
- 4.3.2. ライブラリ
  - OLVライブラリの説明を修正しました。
- 5.7.2. RO ライブラリの特徴と制限
  - 同時にロードできる動的モジュールの上限数を追記しました。
- 6.2.7.2. 拡張スライドパッドとの相違点
  - スリープ復帰後の挙動について追記しました。
- 12.1. ライブラリアプレット
  - アプレットの説明と重複する部分を削除しました。
  - 章を移動しました。
  - 不要なライブラリ情報を削除しました。
- 12.2.1. インターネットブラウザーアプレット
  - 章を移動しました。
  - WEBBRS ライブラリ(インターネットブラウザアプレット)から名称を変更しました。
  - ライブラリ名を追記しました。
- 12.2.2. Miiverse アプリ、投稿アプリ
  - OLV ライブラリ(Miiverse 投稿アプリ)から名称を変更しました。

- OLV ライブラリについての説明を修正しました。
- 章を移動しました。

- 13. 補助ライブラリ
  - システムアプレットについて章を移動しました。

削除

- プラットフォームの表記について
  - プラットフォーム表記を Readme に転記したため、ページを削除しました。

Version 1.1 2014-11-10

追加

- 5.3.10. 電子説明書へのジャンプ
- 12.2.2. Miiverse アプリ、投稿アプリ

変更

- 2.3.1. CPU
  - コアの構成と使用用途を追記しました。
- 4.2.6. SNAKE 専用タイトル
  - SNAKE 専用タイトルについて追記しました。
- 4.3.2. ライブラリ
  - "OLVライブラリの追加"に関する記述を追記しました。
- 9.6. スレッドローカルストレージ
  - スレッドローカルストレージのデストラクタ関数呼出し機能について、追記しました。

Version 1.0 2014-09-04

追加/変更

- 初版