

# ファイルシステムライブラリ 解説

Ver 1.0.3

任天堂株式会社発行

このドキュメントの内容は、機密情報であるため、**厳重な取り扱い、管理を行ってください。**

## 目次

1	はじめに .....	6
1.1	概要 .....	6
1.2	ファイルシステムの使用法 .....	7
1.3	NitroROMフォーマット .....	8
2	ファイル/ディレクトリ インタフェース .....	9
2.1	用語の定義 .....	9
2.1.1	エントリ .....	9
2.1.2	ディレクトリ .....	9
2.1.3	ファイル .....	10
2.1.4	アーカイブ .....	10
2.1.5	パス .....	11
2.1.5.1	パス形式 .....	11
2.1.5.2	相対パス形式 .....	11
2.1.5.3	特殊パス表記 .....	11
2.1.6	ファイルID .....	12
2.1.6.1	ファイル / ファイルパス / ファイルIDの対応 .....	12
2.2	APIの解説 .....	13
2.2.1	共通操作 .....	13
2.2.1.1	FSライブラリの初期化 .....	13
2.2.1.2	FSFileオブジェクトの初期化 .....	13
2.2.1.3	パスの取得 .....	14
2.2.1.4	カレントディレクトリの操作 .....	14
2.2.2	ディレクトリ操作 .....	15
2.2.2.1	ディレクトリリストを取得する .....	15
2.2.2.2	ディレクトリリストからエントリを列挙する .....	15
2.2.2.3	さらに下層のディレクトリリストを検索する .....	16
2.2.3	ファイル操作 .....	17
2.2.3.1	ファイルを開く/閉じる .....	17
2.2.3.2	ファイルサイズを取得し、シーク位置を設定する .....	17
2.2.3.3	バイナリデータをリード/ライトする .....	18
3	アーカイブシステム .....	19
3.1	アーカイブシステムの目的 .....	19
3.2	アーカイブの構成 .....	19
3.2.1	固有アドレス空間とオフセット .....	19
3.2.2	コマンドとユーザプロシージャ .....	19
3.3	アーカイブの動作 .....	20
3.3.1	アーカイブの状態遷移 .....	20
3.3.1.1	設定状態の遷移 .....	20
3.3.1.2	動作状態の遷移 .....	21
3.3.2	コマンド処理シーケンス .....	22

3.4	アーカイブの設計 .....	23
3.4.1	標準的な仕様 .....	23
3.4.2	デフォルトプロシージャ .....	24
3.4.3	アーカイブの実装例 .....	25
3.4.3.1	ROMアーカイブ .....	25
3.4.3.2	独自フォーマットのメモリ上アーカイブ .....	26
3.4.3.3	無線通信経由のアーカイブ .....	27
3.4.3.4	その他のアーカイブ .....	28
3.5	APIの解説 .....	29
3.5.1	状態操作 .....	29
3.5.1.1	FSArchiveオブジェクトの初期化 .....	29
3.5.1.2	アーカイブ名の登録および解放 .....	29
3.5.1.3	アーカイブのロードおよびアンロード .....	30
3.5.1.4	アーカイブの停止および再開 .....	30
3.5.2	ユーザプロシージャ .....	31
3.5.3	非同期処理 .....	32
4	オーバーレイ インタフェース .....	33
4.1	スタティックセグメントとオーバーレイセグメント .....	33
4.2	オーバーレイの特性 .....	34
4.2.1	固有の寿命管理 .....	34
4.2.2	配置先領域の競合 .....	35
4.3	APIの解説 .....	36
4.3.1	.lsfファイルでの指定 .....	36
4.3.2	オーバーレイIDの宣言・定義 .....	36
4.3.3	オーバーレイのロード・アンロード .....	37
4.3.4	ロード処理の分割 .....	38

## コード

表 1	アプリケーションROMに含まれるFNTとFAT .....	8
表 2	ROMヘッダ構造体 .....	8
表 3	FSFileオブジェクトの初期化 .....	13
表 4	FSFileオブジェクトの初期化 .....	13
表 5	FSFileオブジェクトからパスを取得 .....	14
表 6	カレントディレクトリの変更 .....	14
表 7	ディレクトリリストの取得 .....	15
表 8	エントリの列挙 .....	15
表 9	再帰検索処理の例 .....	16
表 10	ファイルのオープン/クローズ .....	17
表 11	ファイルサイズおよびシーク位置の取得 .....	17
表 12	ファイルのリード/ライト .....	18
表 13	ファイルの非同期リード .....	18
表 14	FSArchiveオブジェクトの初期化 .....	29
表 15	アーカイブ名の登録 .....	29

表 16	アーカイブ名の解放 .....	29
表 17	アーカイブのロード .....	30
表 18	アーカイブのアンロード .....	30
表 19	アーカイブの停止・再開 .....	30
表 20	ユーザプロシージャの設定 .....	31
表 21	ユーザプロシージャの記述 .....	31
表 22	アクセスコールバックの非同期化 .....	32
表 23	ユーザプロシージャの非同期化 .....	32
表 24	.lsfファイルでのオーバーレイセグメント指定 .....	36
表 25	オーバーレイIDの宣言・定義 .....	36
表 26	オーバーレイのロード .....	37
表 27	オーバーレイのアンロード .....	37
表 28	ロード処理の分割 .....	38

## 図

図 1-1	ファイルシステム概念図 .....	6
図 2-1	エントリモード図 .....	9
図 2-2	ディレクトリモード図 .....	9
図 2-3	ファイルモード図 .....	10
図 2-4	アーカイブモード図 .....	10
図 2-5	ファイル / ファイルパス / ファイルIDの対応例 .....	12
図 3-1	アーカイブ設定状態 遷移図 .....	20
図 3-2	アーカイブ動作状態 遷移図 .....	21
図 3-3	コマンド処理 フロー .....	22
図 3-4	デフォルトプロシージャ .....	24
図 3-5	ROMアーカイブプロシージャ .....	25
図 3-6	独自フォーマットのメモリ上アーカイブプロシージャ .....	26
図 3-7	無線通信経由のアーカイブプロシージャ .....	27
図 4-1	セグメント 概念図 .....	33
図 4-2	スタティックセグメントとオーバーレイセグメント .....	33
図 4-3	オーバーレイセグメントの寿命 .....	34
図 4-4	オーバーレイセグメントの競合例 .....	35

## 改訂履歴

版	改訂日	改 訂 内 容	担当者
1.0.3	2009-04-13	1 項目追加 (1.3 NitroRom フォーマット)	吉崎
1.0.2	2008-09-26	TWL-SDK への収録にあたり、NITRO-SDK の表記の変更	吉崎
1.0.1	2005-08-19	2.2.1 項目追加 (2.2.1.1 FS ライブラリの初期化) 2.2.3.2 記述修正 (表内サンプルコード訂正)	吉崎
1.0.0	2005-01-11	初版	吉崎

# 1 はじめに

TWL-SDK には、NitroROM フォーマットに基づいて生成されたアプリケーションにおいてデータファイルやオーバーレイを扱い、さらに独自の拡張を行うことを可能とするファイルシステムライブラリが用意されています。

本ドキュメントは、ファイルシステムの基本的な構造と用法について解説するものです。

## 1.1 概要

TWL-SDK でビルドスイッチ `TWL_MAKEROM` を有効にしてビルドすると、`makerom` ツールによって NitroROM フォーマットの形式でアプリケーションが生成されます。(このビルドスイッチはデフォルトで有効であり、通常は有効のままビルドします)

生成されたアプリケーションの内部には1セットのディレクトリとそれに含まれるファイル情報、さらに指定があればオーバーレイと呼ばれる情報が格納されます。

アプリケーションからそれらにアクセスし操作するための仕組みを、総称して「ファイルシステム」と呼びます。

ファイルシステムは、おおよそ以下に示されるモジュールブロックで構成されています。

次章以降では、各々のモジュールブロックについて解説します。

- ・ファイル/ディレクトリ インタフェース : ファイルやディレクトリへの透過的なアクセス機構
- ・アーカイブ システム : ファイル/ディレクトリ インタフェースに対して互換性を持つ形式でファイルシステムに組み込まれるデータアクセス処理の集合
- ・ROM アーカイブ インタフェース : TWL カードへのアクセスを行う標準の内部定義アーカイブ
- ・オーバーレイ インタフェース : オーバーレイの操作全般

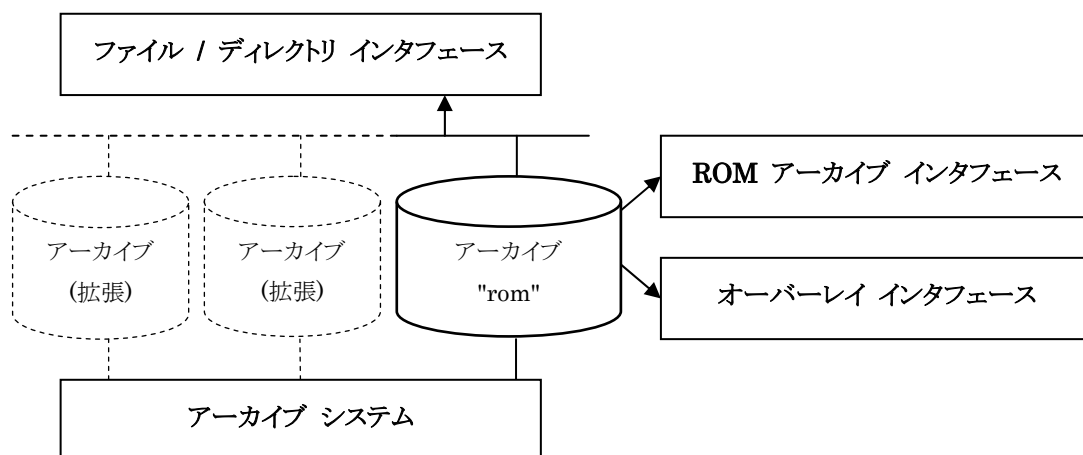


図 1-1 ファイルシステム概念図

## 1.2 ファイルシステムの使用方法

---

アプリケーションからファイルシステムを使用するには、以下の設定でアプリケーションをビルドする必要があります。  
(なお、これらの指定は TWL-SDK ライブラリ自体のビルド時には単に無視され、特に関係ありません。)

- アプリケーションでファイルシステムを有効にする場合、ビルドスイッチ **TWL\_MAKEROM** を有効にします。  
この指定は、**make** コマンドから **commondefs** の記述を通して **makerom** ツールを実行するために必要です。  
(ただし、このビルドスイッチはデフォルトで有効であり、通常は有効のままビルドします。)
- アプリケーションにディレクトリやファイルを含める場合、ビルドスイッチ **ROM\_SPEC** に **.rsf** ファイルを指定します。  
**makerom** ツールはこの **.rsf** ファイルの記述内容を反映してディレクトリやファイルの情報を格納します。  
**.rsf** ファイルの記法については「**TWL-SDK 関数リファレンスマニュアル**」- [ツール]- **[makerom]** 項を参照下さい。
- アプリケーションがオーバーレイを使用する場合、ビルドスイッチ **LCF\_FILE\_SPEC** に **.lsf** ファイルを指定し、さらにビルドスイッチ **SRC\_OVERLAY** にオーバーレイ対象のソースファイルを指定します。  
この指定は、**make** コマンドから **commondefs** の記述を通して **makelcf** ツールへ渡されます。  
**.lsf** ファイルの記法については「**TWL-SDK 関数リファレンスマニュアル**」- [ツール]- **[makelcf]** 項を参照下さい。
- アプリケーションが特殊な状態でオーバーレイを使用する場合、ビルドスイッチ **TWL\_DIGEST** を有効にします。  
ここでいう「特殊な状態」とは、自身のオーバーレイ情報が格納された **TWL** カードに直接アクセスできず無線その他の通信手段を用いて外部からそれらを間接的に取得する必要がある任意の状態を指します。  
このような状態で得られたオーバーレイ情報に対しては実行コードとしての正当性を保証する必要があり、**TWL-SDK** が内部でその判断を行うためにこのビルドスイッチを指定する必要があります。  
詳細については、「**DS ダウンロードプレイ解説**」を参照下さい。

## 1.3 NitroROMフォーマット

TWL-SDK のファイルシステムは NitroROM フォーマットに基づいて設計されており、前述のとおり `makerom` ツールでビルドされたアプリケーションは NitroROM フォーマット形式で構成されます。NitroROM フォーマットは管理情報として FNT と FAT を 1 個ずつ含み、これらを参照してファイル名やサイズなどの情報をルートディレクトリからたどることができます。ROM のおおまかな内部構造を下表1に示します。

ROM イメージの先頭領域(ROM ヘッド情報)を `CARDRomHeader` 構造体としてキャストすれば FNT と FAT の格納アドレスを知ることができます。NitroROM フォーマットの解析処理は FS ライブラリの内部で自動的に行われるので、通常では開発者がこの情報を直接取り扱う必要はありませんが、任意の ROM イメージをファイルシステムへ直接マウントして利用したい場合は `$TwlSDK/build/demos/fs/arc-1` サンプルが用意されていますのでご参考ください。

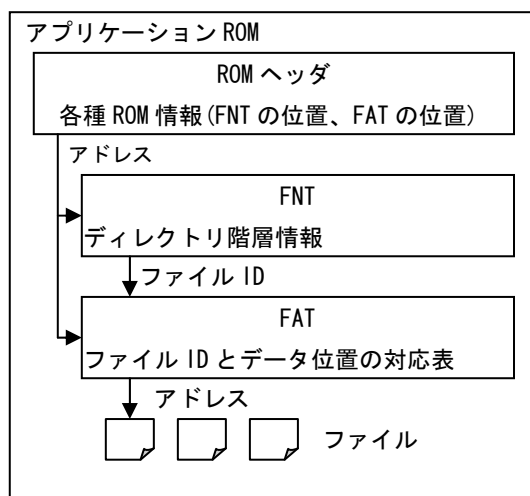


表1 アプリケーション ROM に含まれる FNT と FAT

```

// nitro/card/types.h:
typedef struct CARDRomHeader
{
    char    game_name[12];
    u32     game_code;
    ...

    // 0x040-0x050 [ファイルテーブル用パラメータ]
    CARDRomRegion fnt;           // ファイルネームテーブル
    CARDRomRegion fat;           // ファイルアロケーションテーブル
    ...
}
CARDRomHeader;

```

表2 ROM ヘッド情報構造体



## 2 ファイル/ディレクトリ インタフェース

ファイルシステムライブラリでは、ディレクトリやファイルを特定し操作する一連の基本的な機能が用意されています。  
この章ではこれらのインタフェースについて説明します。

### 2.1 用語の定義

ファイルシステムライブラリならびに本ドキュメントにて使用される「ファイル」「ディレクトリ」などという用語の概念は、一般にパーソナルコンピュータ上のオペレーティングシステム等に導入されているそれらとおおむね同様のものです。  
この節では、ファイルシステムライブラリにおけるそれらの用語についての厳密な定義を記述します。

#### 2.1.1 エントリ

エントリは、階層に含まれる要素です。

1つのファイルまたは1つのディレクトリいずれかを特定するための情報を持ちます。

同じ階層に含まれる他のどのエントリとも重複しない、ひとつの名前を持ちます。

名前は 127 文字以内の ASCII コードで構成されます。

ただし大文字/小文字は区別されず、次の文字群は除きます。【 ¥ / : ; \* ? " < > | 】

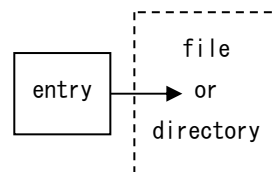


図 2-1 エントリ模式図

#### 2.1.2 ディレクトリ

ディレクトリは、1つの階層を表現する情報です。

0 個以上のエントリと、それらを特定するための情報を持ちます。

上の階層に位置するディレクトリ(親ディレクトリ)を特定するための情報を持ちます。

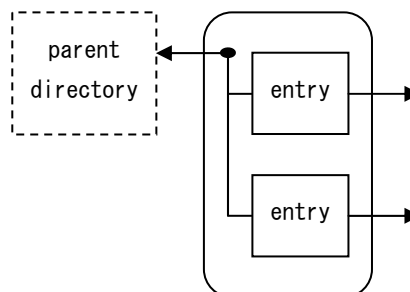


図 2-2 ディレクトリ模式図

### 2.1.3 ファイル

ファイルは、バイナリデータを有した一意のオブジェクトを参照するための情報です。

開く(オープンする)ことでオブジェクトの操作を開始し、閉じる(クローズする)ことで操作を終了します。

読み出し(リード)および書き込み(ライト)操作により、あたかもリニアなメモリと同様に振る舞います。

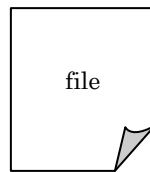


図 2-3 ファイル模式図

### 2.1.4 アーカイブ

アーカイブは、ファイル・ディレクトリ・エントリの情報およびそれらを制御する手段を備えたオブジェクトです。

ファイルシステムの内部において他のどのアーカイブとも重複しない、ひとつの名前を持ちます。

名前は英数字 3 文字以内で構成され、大文字/小文字は区別されません。

ひとつの階層関係を含み、階層の最上位には必ず無名ディレクトリ(ルートディレクトリ)があります。

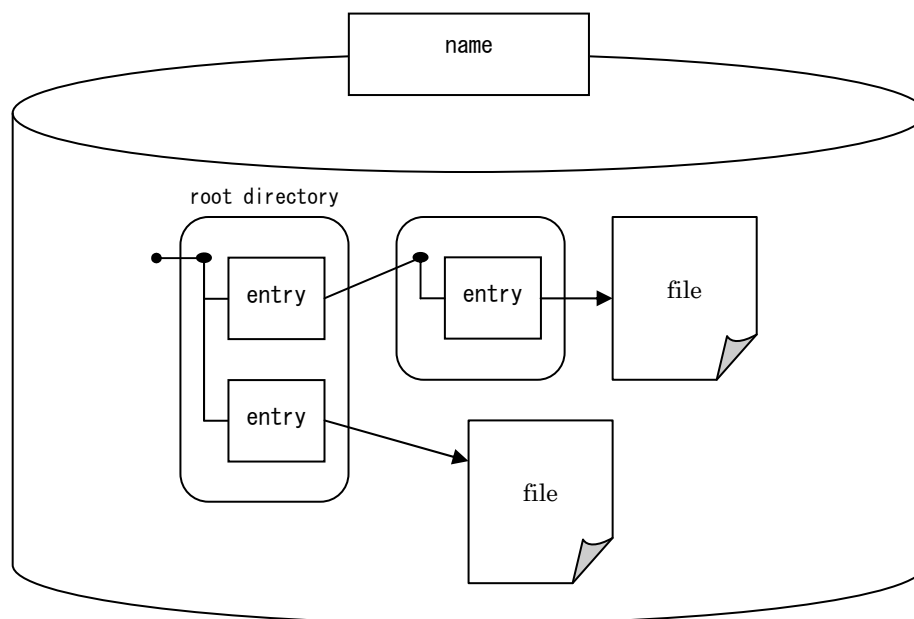


図 2-4 アーカイブ模式図

## 2.1.5 パス

ファイルシステム内には任意個数のアーカイブが並列に存在します。

ファイルシステム内において、アーカイブ名とルートディレクトリからの階層ごとのエントリ名の組み合わせで一意にエントリを識別することができます。この名前の組み合わせを「パス」（またはパス名、パス文字列）と呼びます。

エントリの内容がディレクトリを指す場合は、そのパスを特に「ディレクトリ(の)パス」と呼びます。

同様に、ファイルを指す場合はそのパスを特に「ファイル(の)パス」と呼びます。

### 2.1.5.1 パス形式

パスは以下のいずれかの形式で文字列として表現されます。

- 1) "(アーカイブ名) : / "
- 2) "(アーカイブ名) : / エントリ名 / エントリ名 / ... / エントリ名 / "
- 3) "(アーカイブ名) : / エントリ名 / エントリ名 / ... / エントリ名 "

終端でないエントリの内容はディレクトリを指していなければなりません。

" / " で終わるパスはディレクトリパスであることを明示します。

上記 1)、2) はいずれもディレクトリパスを表します。

アーカイブのルートディレクトリパスを表現できるのは 1) の形式のみです。

3) はファイルまたはディレクトリいずれかのパスを表します。終端のエントリがディレクトリを指す場合は2) と等価です。これは、ディレクトリパスについてはパス終端の " / " の有無が特に区別されないことを意味します。

### 2.1.5.2 相対パス形式

ファイルシステムにおいてパスは省略することができ、省略されたパスはファイルシステムに記憶されているディレクトリパスを基準として相対的に補完されます。このディレクトリパスを「カレントディレクトリ」と呼び、省略されたパスを「相対パス」と呼びます。また、通常の省略されていないパスは、相対パスに対して「絶対パス」と呼びます。

相対パスは以下の規則でカレントディレクトリから補完されます。

- 1) " / " で始まるパスはカレントディレクトリが属するアーカイブのルートディレクトリで補完されます。
- 2) そうでない場合は、カレントディレクトリパスを単純に先頭へ連結して補完されます。

例として、カレントディレクトリが " rom : / text / " である場合には、

相対パス " / snd / dat " は絶対パス " rom : / snd / dat " へ補完され、

相対パス " snd / dat " は絶対パス " rom : / text / snd / dat " へ補完されます。

### 2.1.5.3 特殊パス表記

絶対パス・相対パスとも、以下のエントリ名は特殊な名前として予約されています。

- 1) エントリ名 " . " はそのエントリ自身が含まれるディレクトリを指します。
- 2) エントリ名 " .. " はそのエントリ自身が含まれるディレクトリの1階層上のディレクトリを指します。

## 2.1.6 ファイルID

個々のアーカイブは、自身に属するファイルを識別する固有のインデックス値を持ちます。

ディレクトリ階層内のエントリは、このインデックス値によってファイルを特定します。

アーカイブとこのインデックス値との組み合わせから、ファイルシステム全体で一意にファイルが特定されます。

この組み合わせ情報を「ファイルID」と呼びます。

### 2.1.6.1 ファイル / ファイルパス / ファイルIDの対応

ファイルパス、ファイルID、およびファイル自体については、ファイルシステム関連の文書において文脈に応じて単に「ファイル」と呼称される場合があります。ここでは、これらの関係について前述の内容を基に解説します。

- ・「ファイル」はファイル自体を指し、あるアーカイブの内部にその実体が1つだけ存在します。
- ・任意のエントリがファイルを指す(具体的にはファイルのインデックス値を含む)場合、そのエントリのパスは「ファイルパス」ですが、文脈上「ファイルパスで指定されたファイル」の意味を以って単に「ファイル」と呼称される場合があります。
- ・「ファイルID」についても同様に、文脈上「ファイルIDで指定されたファイル」の意味を以って単に「ファイル」と呼称される場合があります。
- ・「ファイルパス」「ファイルID」が存在すればそれによって一意に特定される「ファイル」が存在しますが、その逆、すなわち任意の「ファイル」を特定する「ファイルパス」「ファイルID」が必ずしも存在するとは限りません。これは、アーカイブが全てのファイルに対してインデックス値やエントリを用意する必要が無いことを意味します。よって、特定する手段の無い(主に一時的な用途で生成される)ファイルをアーカイブが内包することも許されます。

下図は、ディレクトリ階層上にエントリのあるファイルが2個、インデックス値を用意されているファイルが3個、実際に存在するファイルが4個含まれるアーカイブにおける各ファイルのパスとIDの例を示します。

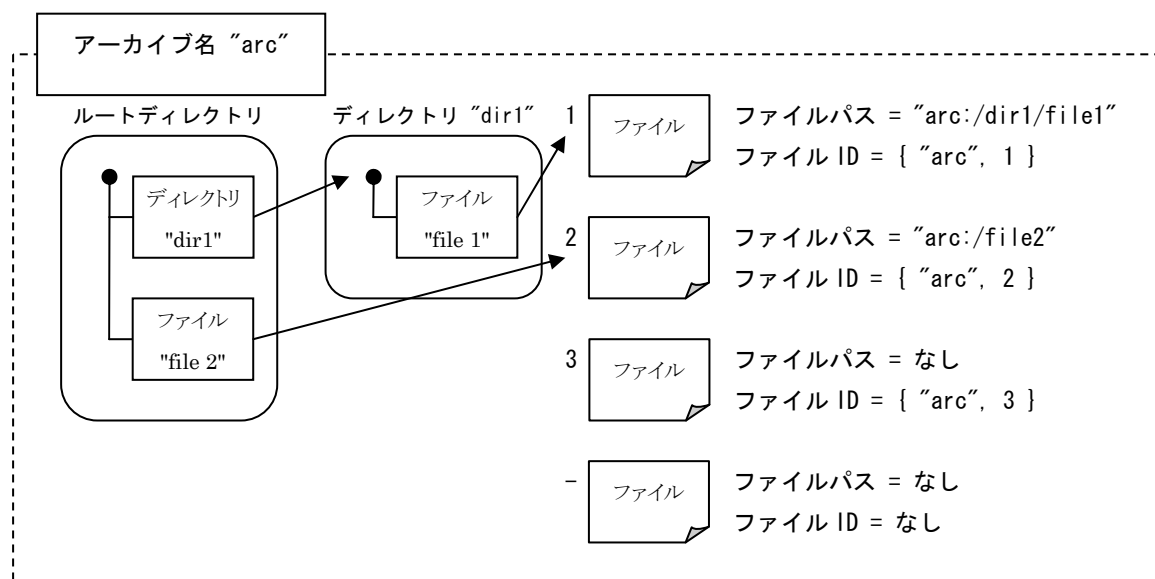


図 2-5 ファイル / ファイルパス / ファイルIDの対応例

## 2.2 APIの解説

前節までは、TWL-SDK におけるファイルシステムの各種定義を解説しました。

ここでは、これらの定義に基づいてファイルシステムライブラリのインタフェース関数(API)を使用して実際にアプリケーションからファイルやディレクトリを操作する手順を解説します。

### 2.2.1 共通操作

ファイル・ディレクトリインタフェースにおいては、関数呼び出しの際に **FSFile** 構造体オブジェクトを使用します。**FSFile** オブジェクトは処理に応じてその内部状態を更新し、ファイルやディレクトリに関する情報を保持します。

#### 2.2.1.1 FSライブラリの初期化

どの **FS** ライブラリ関数を使用するよりも前に、**FS\_Init ( )** 関数を使用して **FS** ライブラリを初期化する必要があります。この関数の呼び出しは 1 回だけで構いません。

初期化の際、**FS** ライブラリが内部でカードアクセスするために使用する **DMA** チャンネルを 1 つだけ割り当てる必要があります。この **DMA** チャンネルは **FS\_End 0** 関数で **FS** ライブラリを解放するまで内部で占有されることに注意してください。また、カードアクセスの転送元は **IO** レジスタなので **DMA** チャンネル **0** を指定することはできません。**FS** ライブラリへ特に **DMA** チャンネルを割り当てないのであれば特別な値として **FS\_DMA\_NOT\_USE** を明示的に指定することもでき、この場合は **CPU** 処理によるカードアクセスとなります。

```
/* FSライブラリを使用する前に初期化する */
#define DMA_CHANNEL_FOR_FS 2 /* FS に使用させる DMA */
FS_Init( DMA_CHANNEL_FOR_FS );
```

表 3 FSFile オブジェクトの初期化

#### 2.2.1.2 FSFileオブジェクトの初期化

ユーザは **FSFile** オブジェクトの各内部メンバについて直接操作する必要はありませんが、オブジェクトを使用する前に必ず **FS\_InitFile ( )** 関数を使用して内部状態を初期化する必要があります。

```
/* FSFileオブジェクトを最初に使用する前に必ず初期化する */
FSFile file;
FS_InitFile( &file );
```

表 4 FSFile オブジェクトの初期化

**FSFile** オブジェクトがファイルに関する情報を保持している場合、それを「ファイルハンドル」と呼ぶことがあります。同様に、ディレクトリに関する情報を保持している状態では、それを「ディレクトリリスト」と呼ぶことがあります。単一の **FSFile** オブジェクトが複数のファイルおよびディレクトリの情報を保持することはできません。

### 2.2.1.3 パスの取得

FSFile オブジェクトがファイルやディレクトリの情報を保持している場合、FS\_GetPathName ( ) 関数を使用してファイルパスまたはディレクトリパスを取得することができます。

```
/* FSFileオブジェクトが保持している内容のパス長を取得する */
const s32 len = FS_GetPathLength( &file );
/* ここで -1 が返される場合、指定したオブジェクトは
「2.1.6.1 ファイル/ファイルパス/ファイルIDの対応」
で挙げたようにエントリが存在しないファイルであるか、
または単に何の情報も保持していない状態にある */
if( len >= 0 )
{
    /* パス名を格納できる十分なメモリを用意する */
    char *buf = (char*)OS_Alloc( len );
    if( buf )
    {
        /* 実際にパス名を取得する */
        BOOL ret = FS_GetPathName( &file, buf, len );
        if( ret )
        {
            OS_Printf("path=%s¥n", buf);
        }
        OS_Free( buf );
    }
}
```

表 5 FSFile オブジェクトからパスを取得

### 2.2.1.4 カレントディレクトリの操作

FSFileオブジェクトがファイルやディレクトリの情報を取得するためのほとんどの関数はパスを要求します。

[「2.1.5 パス」](#) で述べたように、パスには絶対パスと相対パスの二種類が存在し、相対パスを補完するためのカレントディレクトリはファイルシステムが内部で1つだけ管理しています。

カレントディレクトリは、FSライブラリの初期化時にはデフォルトのROMアーカイブのルートディレクトリにあたる  
" rom : / " が設定されていますが、ユーザは FS\_ChangeDir ( ) 関数を使用して変更することが出来ます。

```
/* 現在のカレントディレクトリを "rom:/" とする */
BOOL ret;
/* 相対パスで指定した場合はカレントディレクトリで補完される */
ret = FS_ChangeDir( "dir_1" );
/* もし "rom:/dir_1/" がディレクトリとして存在すれば、
カレントディレクトリは変更され ret には TRUE が返る */

/* 絶対パスで指定した場合はカレントディレクトリを無視する */
ret = FS_ChangeDir( "arc:/" );
/* もし "arc" がアーカイブとして存在すれば、
カレントディレクトリはそのルートディレクトリに変更される */

...

/* アーカイブがファイルシステムから解放されるなどして
カレントディレクトリの指す対象が無効になった場合、
常に存在が保証される "rom:/" へ自動的に変更される */
FS_ReleaseArchiveName( FS_FindArchive( "arc", 3 ) );
```

表 6 カレントディレクトリの変更

## 2.2.2 ディレクトリ操作

実行時にアプリケーションからディレクトリ構成を検索する場合、FSFile オブジェクトをディレクトリリストとして使用し、エントリを列挙することで情報を取得します。

ディレクトリリストは FSFile オブジェクト内部で「ディレクトリ+列挙位置」の組み合わせ情報として保持されます。この組み合わせ情報は FSDirPos 構造体として表現され、「ディレクトリ位置」と呼ばれることがあります。

ディレクトリリストに対して行われる操作は、通常は以下のような手順になります。  
アプリケーションでの用途に応じて、これらの操作を適宜応用することになります。

### 2.2.2.1 ディレクトリリストを取得する

FSFile オブジェクトにディレクトリリストを取得するにはふたつの方法があります。

ひとつは、FS\_FindDir() 関数を使用してファイルシステム内の既知のパスを指定する方法です。  
この場合、取得したディレクトリリストの列挙位置は常に先頭のエントリを指した状態で初期化されます。  
もうひとつは、FS\_SeekDir() 関数を使用してディレクトリ位置を指定する方法です。  
この場合、取得したディレクトリリストは列挙位置も含めて指定のディレクトリ位置情報で初期化されます。  
ディレクトリ位置は、既得のディレクトリリストから FS\_TellDir() 関数で取得したり、  
後述の手順で FS\_ReadDir() 関数によって取得することができます。

```
BOOL    ret;
FSFile  dir;
FS_InitFile( &dir );
/* 既知のパスからディレクトリリストを取得 */
if( FS_FindDir( &dir, "rom:/" ) )
{
    /* 用意されたいくつかの手段でディレクトリ位置を取得・保存 */
    FSDirPos  pos;
    ret = FS_TellDir( &dir, &pos );
    SDK_ASSERT( ret );
    /* 既得のディレクトリ位置からディレクトリリストを取得 */
    ret = FS_SeekDir( &dir, &pos );
    SDK_ASSERT( ret );
}
```

表 7 ディレクトリリストの取得

### 2.2.2.2 ディレクトリリストからエントリを列挙する

FS\_ReadDir() 関数を使用して、現在の列挙位置からエントリ情報をひとつずつ取得することができます。  
エントリ情報は FSDirEntry 構造体の形式で取得され、列挙位置は次のエントリを指すように進められます。  
この処理は列挙位置が終端に達するまで繰り返すことができます。

```
FSDirEntry  entry;
/* 列挙位置が終端に達すると FS_ReadDir() は FALSE を返す */
while( FS_ReadDir( &dir, &entry ) )
{
    /* 得られたエントリが含む情報は、エントリ名と
       ファイルまたはディレクトリいずれかを特定する情報 */
    OS_Printf( "<?c>%s\n",
               entry.is_directory ? 'F' : 'D', entry.name );
}
```

表 8 エントリの列挙

### 2.2.2.3 さらに下層のディレクトリリストを検索する

前述の操作によって得られたエントリがディレクトリ情報を指している場合、処理によっては下層のディレクトリまで検索対象に含めたいことがあります。この場合は一般に再帰関数によって実装しますが、あらゆる再帰処理に共通する問題であるスタックオーバーフローについて注意する必要があります。

特に `FSDirEntry` オブジェクトは最大エントリ名サイズ分のバッファを含むためにスタック消費が大きく、また、検索用の `FSFile` オブジェクトも比較的サイズが大きいため、これらは階層ごとに持たないようにできれば良いでしょう。以下に、スタックメモリ消費を抑えた再帰的な検索処理の一例を示します。

```
/* 指定ディレクトリ位置からのエントリをダンプする再帰関数.
   引数の FSFile と FSDirEntry は使い回す */
void DumpDirEntriesSub(int tab,
    FSFile *p_dir, FSDirEntry *p_entry)
{
    /* ディレクトリの名前を出力 */
    OS_TPrintf( "%*s%s/%n", tab, "", p_entry->name );
    tab += 4;
    /* ディレクトリ内のエントリを列挙 */
    if( FS_SeekDir( p_dir, &p_entry->dir_id ) )
    {
        while( FS_ReadDir( p_dir, p_entry ) )
        {
            if( ( p_entry->is_directory == 1 ) )
            {
                /* 下層のサブディレクトリへ再帰し、その後に復帰.
                   FSFile と FSDirEntry の実体は使い回す */
                FSDirPos cur_pos;
                if( FS_TellDir( p_dir, &cur_pos ) )
                {
                    DumpDirEntriesSub( tab, p_dir, p_entry );
                    (void)FS_SeekDir( p_dir, &cur_pos );
                }
            }
            else
            {
                /* ファイルの名前を出力 */
                OS_TPrintf( "%*s%s/%n", tab, "",
                    p_entry->name );
            }
        }
    }
}

/* 再帰的ダンプ処理の起点となる関数 */
void DumpEntries(const char *dir_path)
{
    /* ここに再帰処理内で使用する唯一の実体を確保 */
    FSFile work_dir;
    FSDirEntry work_entry;
    FS_InitFile(&work_dir);
    if( FS_FindDir( &work_dir, dir_path ) &&
        FS_TellDir( &work_dir, &work_entry.dir_id ) )
    {
        work_entry.name[0] = '\0';
        DumpDirEntriesSub( 0, &work_dir, &work_entry );
    }
}
```

表 9 再帰検索処理の例



## 2.2.3 ファイル操作

アプリケーションの処理でファイルを扱う場合、FSFile オブジェクトをファイルハンドルとして使用し、アクセス用関数を呼び出すことでファイルデータを処理します。  
ファイルハンドルは FSFile オブジェクト内部で「バイナリデータの情報+シーク位置」として保持されます。  
(バイナリデータ本体は FSFile オブジェクト内部ではなく任意のアーカイブ内部に存在します)

ファイルハンドルに対して行われる操作は、おおむね以下のような手順になります。  
アプリケーションでの用途に応じて、これらの操作を適宜応用することになります。

### 2.2.3.1 ファイルを開く/閉じる

アプリケーションからファイルを特定するには、ファイルパスまたはファイルIDのいずれかが必要となります。

([「2.1.6.1 ファイル/ファイルパス/ファイルIDの対応」参照](#))

ファイルパスならFS\_OpenFile ( ) 関数を、ファイルIDならFS\_OpenFileFast ( ) 関数を使用することで、FSFileオブジェクトの内容はファイルハンドルになります。この操作がファイルのオープンに相当します。  
ファイルに対する全ての操作はこのファイルハンドルを使用して行われます。  
ファイルハンドルを使用した後はFS\_CloseFile ( ) 関数で解放します。この操作がファイルのクローズに相当します。  
これは、オープン可能なファイル総数などに制限を持つアーカイブが内部資源の管理を適切に行うために必要です。

```
FSFile    file;
FSFileID  file_id;
FS_InitFile( &file );
/* 既知のファイルパスからファイルを開き、閉じる */
if( FS_OpenFile( &file, "rom:/file1" ) )
    (void)FS_CloseFile( &file );
/* ファイルIDからファイルを開き、閉じる */
if( FS_ConvertPathToFileID( &file_id, "rom:/file2" ) )
{
    if( FS_OpenFileFast( &file, file_id ) )
        (void)FS_CloseFile( &file );
}
```

表 10 ファイルのオープンクローズ

### 2.2.3.2 ファイルサイズを取得し、シーク位置を設定する

ファイルに対する操作は基本的にリードとライトのみで、これらの操作は常に「シーク位置とサイズ」を必要とします。  
ファイル自体の全体サイズは FS\_GetLength ( ) 関数で取得することができます。  
ファイルハンドルが保持している現在のシーク位置は FS\_GetPosition ( ) 関数で取得することができ、FS\_SeekFile ( ) 関数で移動させることができます。

```
/* 全体サイズと現在位置から、残りのバイト数を計算 */
const u32 pos = FS_GetPosition( &file );
const u32 len = FS_GetLength( &file );
const u32 rest = (u32)(len - pos);
void *enough_buf = OS_Alloc( rest );
/* シーク位置を先頭に移動させる */
(void)FS_SeekFile( &file, 0, FS_SEEK_SET );
```

表 11 ファイルサイズおよびシーク位置の取得

### 2.2.3.3 バイナリデータをリード/ライトする

FS\_ReadFile ( ) 関数を使用してファイルの現在のシーク位置からバイナリデータを読み出すことができます。

また、FS\_WriteFile ( ) 関数を使用してファイルの現在のシーク位置にバイナリデータを書き込むことができます。

いずれの場合も、処理が完了した後でシーク位置は実際にアクセスしたデータサイズ分だけ移動します。

```
/* テキストファイルをリードしてデバッグ出力 */
char    string_buf[256 + 1];
string_buf[ sizeof(string_buf) - 1 ] = '¥0';
/* ファイルの終端に達すると読み出しサイズが 0 となる */
while( FS_ReadFile( &file, string_buf,
sizeof(string_buf) - 1 ) > 0 )
    OS_PutString( string_buf );
```

表 12 ファイルのリード/ライト

アーカイブの実装によってはリードおよびライトの操作がただちに完了せず、その処理中にプロセッサ自体が別の作業を行える場合があります。一般的に非同期処理と呼ばれるこのような制御については、アプリケーション側でスレッドを使用して実現することも可能ですが、この動作をアーカイブに対して期待する呼び出し形式も用意されています。

前述のリードおよびライト関数に対応して FS\_ReadFileAsync ( ) 関数、FS\_WriteFileAsync ( ) 関数を使用すると、アーカイブが非同期処理に適した実装であるならば処理の完了を待たずにただちに制御が返されます。

この処理が実際に完了したかどうかは FS\_IsBusy ( ) 関数で確認することができ、完了のタイミングまで待機するには FS\_WaitAsync ( ) 関数を使用します。

もしアーカイブが非同期処理を行わないなら、これらの非同期版関数は対応する同期版関数と同じ動作になります。

この場合は FS\_IsBusy ( ) が常に偽を返し FS\_WaitAsync ( ) 関数は何もせずただちに制御を返しますので、非同期処理がたちまち完了した場合と同様にみなすことができます。

```
/* 非同期のリード処理と並行して別の処理を実行する。
   ファイルデータに関連した逐次処理であればさらに効果的。 */
while( FS_ReadFileAsync( &file, string_buf,
sizeof(string_buf) - 1 ) > 0 )
{
    DrawScreen( );
    FS_WaitAsync( &file );
    OS_PutString( string_buf );
}
```

表 13 ファイルの非同期リード

## 3 アーカイブシステム

前章では、ファイル・ディレクトリインタフェースの概念と使用方法について解説しました。

それらのインタフェースに従って具体的な内部動作を実装するための枠組みがアーカイブシステムになります。

この章ではアーカイブシステムの構成と動作およびアーカイブのインタフェースについて説明します。

### 3.1 アーカイブシステムの目的

ファイルシステムライブラリにおけるアーカイブシステムの位置づけは [「1.1 概要」](#) で図示したとおり、アーカイブ実装に関する機能のみを提供するものです。ユーザアプリケーションがファイルシステムライブラリを使用する上で、

他のモジュールブロックを介さずにアーカイブシステム部分だけを直接必要とすることはありません。

アーカイブシステムは、主にアプリケーションのミドルウェアやユーティリティの実装者が利用します。

以下の用途にかなう場合、アーカイブシステムは有用なものになり得ます。

- ・新規導入モジュールと既存モジュールの間でプログラムコードの共通化・拡張・再利用をはかる
- ・複雑な制御を要するデータ格納媒体の内部実装をユーザから隠蔽する

### 3.2 アーカイブの構成

アーカイブは、いくつかの基本的なパラメータとコールバック関数の情報を保持するオブジェクトとして定義されます。

以下に、それらの具体的な用語を解説します。

#### 3.2.1 固有アドレス空間とオフセット

ファイルシステムは、アーカイブの保持する情報が NitroROM フォーマットに準拠したリニアなデータ構造であることを期待して設計されています。このためアーカイブは、その内部に 0 から始まる固有アドレス空間が存在するとみなしてそこに含まれる FNT、FAT、および各ファイルのデータイメージへアクセスする手段を提供する必要があります。

その手段は読み出し・書き込みの対になるコールバック関数で実現され、以降はこれらをそれぞれ

「リードコールバック」「ライトコールバック」と呼称し、この2つをあわせて「アクセスコールバック」と呼称します。

また、上記固有アドレス空間におけるアドレスは、CPU 本来のアドレスマップ上におけるそれと区別するために「オフセット」と呼称します。

#### 3.2.2 コマンドとユーザプロシージャ

アーカイブはファイルシステムの具体的な処理内容を把握せずとも、前述のアクセスコールバックおよび FNT と FAT のオフセットを正しく提供しさえすれば、ユーザの要求を透過的に満たすことができます。

しかし、固有アドレス空間の一部または全部を NitroROM フォーマットに準拠させることが不可能な場合についても、これを解決する方法が用意されています。ファイル・ディレクトリインタフェースからアーカイブへのアクセスは「コマンド」と呼ばれる定型処理でまとめられており、これらのコマンドの各々についてアクセスコールバックの前にアーカイブへ問い合わせるよう設定することができます。

アーカイブは、問い合わせられた各々のコマンドを「ユーザプロシージャ」と呼ばれるコールバック関数で処理し、独自の実装で直接置き換えることができます。これにより、NitroROM フォーマットへ厳密に準拠せずともファイルシステムの要求を満たすアーカイブを作成することが可能となっています。

また、ユーザプロシージャによって置き換えられずに実行される標準の処理を「デフォルトプロシージャ」と呼びます。

### 3.3 アーカイブの動作

アーカイブは、ファイルシステムからコールバック駆動されることにより自動的に処理を行います。

本節では、アーカイブがファイルシステム内においてどのように動作するかを解説します。

なお、解説内の図表に登場するおのおのの関数については、[「3.4 APIの解説」](#)で後述します。

#### 3.3.1 アーカイブの状態遷移

アーカイブの内部状態は、ファイルシステムにとっての設定状態とアーカイブ自身の動作状態に分けられます。

##### 3.3.1.1 設定状態の遷移

アーカイブは、ファイルシステムにとって以下の3つの状態を遷移します。

unregisterd	アーカイブはファイルシステムに何ら関連付けられていない状態。 初期化直後はこの状態から開始します。
registerd	ファイルシステム内で一意となる名前が登録された状態。 この状態ではファイルシステムに含まれていますが動作はしません。
loaded	アクセスコールバックとともにファイルシステムへロードされた状態。 この状態においてのみファイル・ディレクトリインタフェースからのコマンドが発生します。

また、これらの状態間の遷移は下図のようになります。

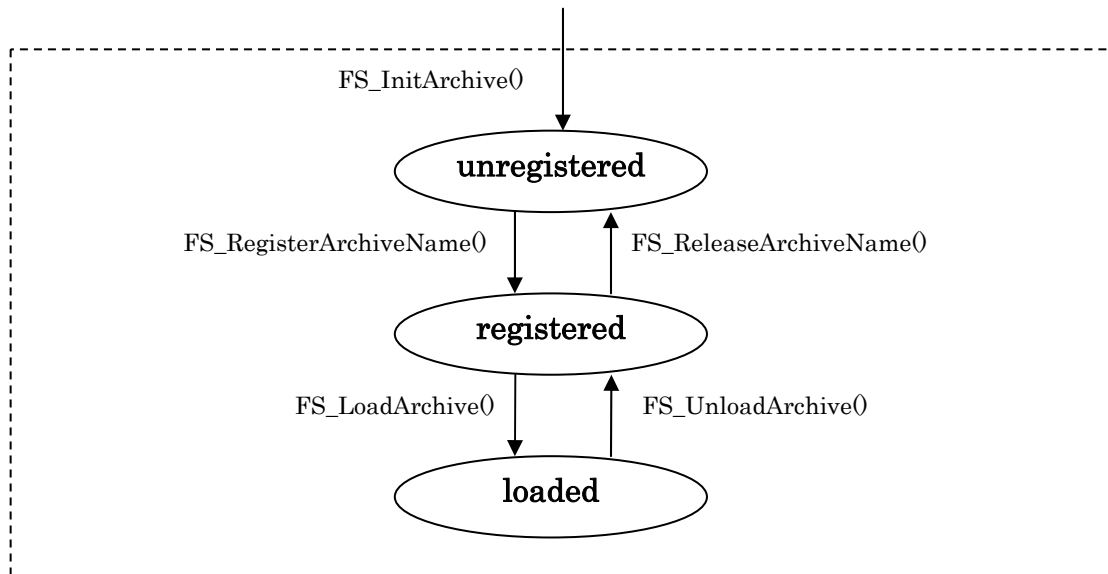


図 3-1 アーカイブ設定状態 遷移図

### 3.3.1.2 動作状態の遷移

アーカイブは、アーカイブ自身の動作に応じて以下の3つの状態を遷移します。

suspended	アーカイブとしての動作を停止されている状態。 ファイル・ディレクトリインタフェースからのコマンドは動作再開まで保留されます。
idle	アーカイブとして動作しているが何も未処理コマンドの無い状態。 最初のコマンドが発生するタイミング
busy	コマンドを処理している状態。 最初のコマンドが発生したタイミングからこの状態になります。

また、これらの状態間の遷移は下図のようになります。

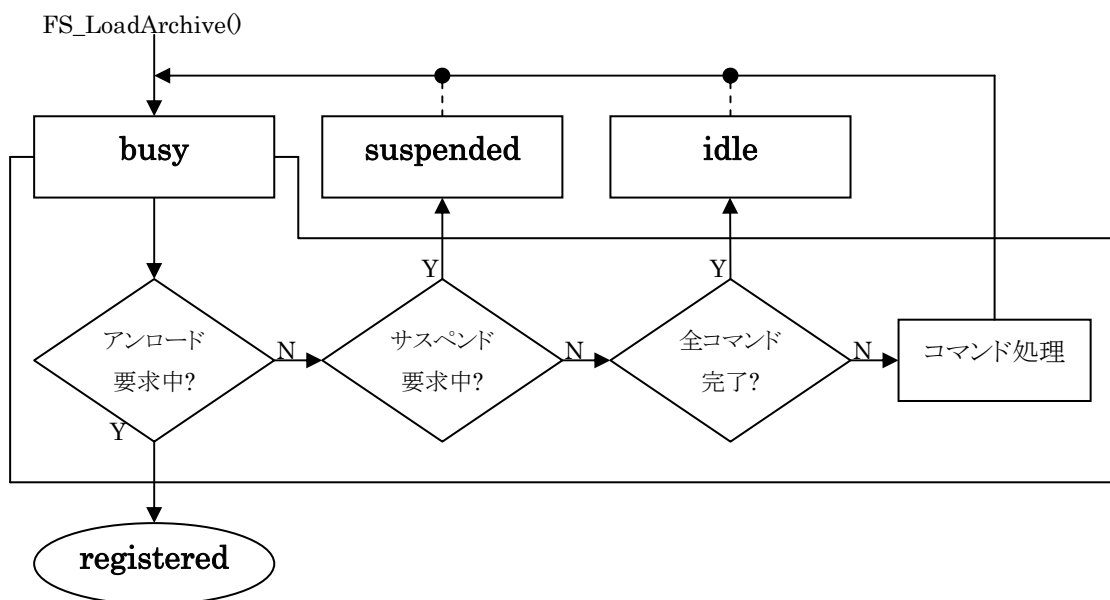


図 3-2 アーカイブ動作状態 遷移図

### 3.3.2 コマンド処理シーケンス

コマンドはファイルシステムからアーカイブへ逐次要求され、未処理のコマンドが蓄積すると先着順に保留されます。ファイルシステムは各々のアーカイブが常にひとつずつコマンドを処理するようにコールバックを駆動しますが、ファイルシステム内のアーカイブ同士は互いの状態へ干渉を受けずに複数が並列に動作することができます。

アーカイブが**busy**状態においてコマンドをひとつ処理する場合、[「3.2.2 コマンドとユーザプロシージャ」](#)で述べたようにユーザプロシージャによる置き換えかデフォルトプロシージャによる標準処理のいずれかによって実行されます。いずれのプロシージャも、処理を実行したあとは、定義されたいくつかの結果値のうちのひとつを返します。通常はここでコマンドが完了します。

[「2.2.3.3 バイナリデータをリード/ライトする」](#)で述べた非同期処理にアーカイブが対応していれば、プロシージャの結果値として「非同期処理中」を返すことがあります。この場合、アーカイブ自身が処理完了時にファイルシステムへその旨を通知する必要があり、ファイルシステムはその通知を受けるまで**busy**状態時の処理を中断します。ここで中断されるコマンドがファイルのリード/ライトなど非同期処理用APIの呼び出しから発生したものでない場合、ファイルシステムはその呼び出し内部で完了通知をブロックングすることになります。

以上のコマンド処理の流れを、下図に示します。

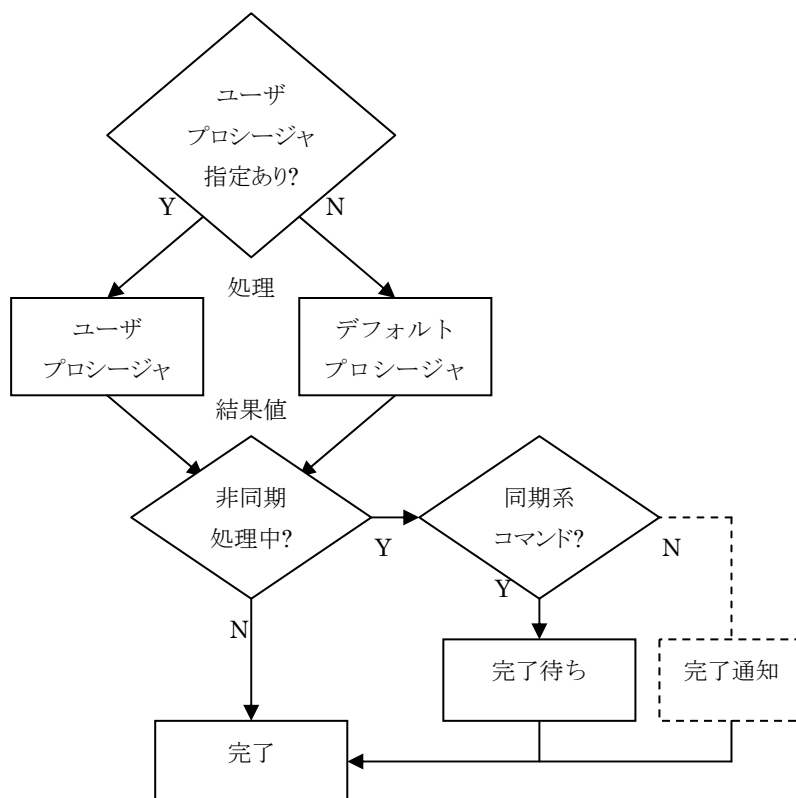


図 3-3 コマンド処理 フロー

## 3.4 アーカイブの設計

---

本節では、アーカイブを独自に実装する際に検討すべきおおまかな指針といくつかの実装例を解説します。

### 3.4.1 標準的な仕様

---

基本的にアーカイブの実装は、アクセスコールバックならびにユーザプロシージャの計3つのコールバック関数を適切に記述することだけです。ファイルシステムが期待する標準的な仕様に可能な限り近づけるべく、これらのコールバックにおいてターゲット固有の特性をラッピングすることが主な作業となります。

ここで、ファイルシステムがターゲットへ期待する標準的な仕様を以下に示します。

合致する条件の項目番号が小さいほど、アーカイブへの実装がおおよそ容易なターゲットと考えられます。

#### (1) 内部のデータ構造が NitroROM フォーマットに完全に準拠している

この場合、アクセスコールバックのみで全てのコマンドをデフォルトプロシージャが処理できるため実装は最も容易です。アーカイブの扱うデバイスが特殊なものでない限り、ユーザプロシージャも不要となります。

NitroROM フォーマットへの準拠が不完全、あるいはまったくの別フォーマットである場合には、FNT や FAT に関わる低位のコマンドやアクセスコールバックを適切に置き換える必要があります。

#### (2) ディレクトリ構成およびファイル情報が固定的である

この場合、少なくともユーザ側での使用に支障のない標準的なアーカイブを実装することが可能です。

ファイル・ディレクトリインタフェースは、ディレクトリが動的に変化したりファイルの情報が任意に変化するような環境を組み込むには不向きな特性を持つため、そのようなターゲットに対してはいくつかのコマンドに制約が生じたりサポート自体が不可能である場合があります。

#### (3) ディレクトリおよびファイルの概念がファイルシステムのそれと合致している

ターゲットがこの条件にも沿わない場合は、よほど特殊な用途でない限りファイルシステムのメリットは薄くなります。例えばネットワークにおける通信ソケットや URI のパスなどは、ファイル・ディレクトリインタフェースのいくつかの単体のコマンドについてのみ概念的によく合致します。

### 3.4.2 デフォルトプロシージャ

デフォルトプロシージャは、ファイルシステムに用意された全てのコマンドについて 1 つずつの標準処理を提供します。これらのうち低位なコマンドでは、アクセスコールバックおよび FNT と FAT を使用したり、あるいは何にも依存せず処理を実現していますが、高度なコマンドにおいてはその内部でさらに他の低位コマンドを使用するものもあります。

アーカイブ処理の基本となるデフォルトプロシージャの各コマンドにおける依存関係を、下図に示します。アクセスコールバックやユーザプロシージャの実装では、この依存関係の上位層を考慮する必要があります。各コマンドに要求される厳密な仕様や SDK での実際の対応については、関数リファレンスを参照ください。

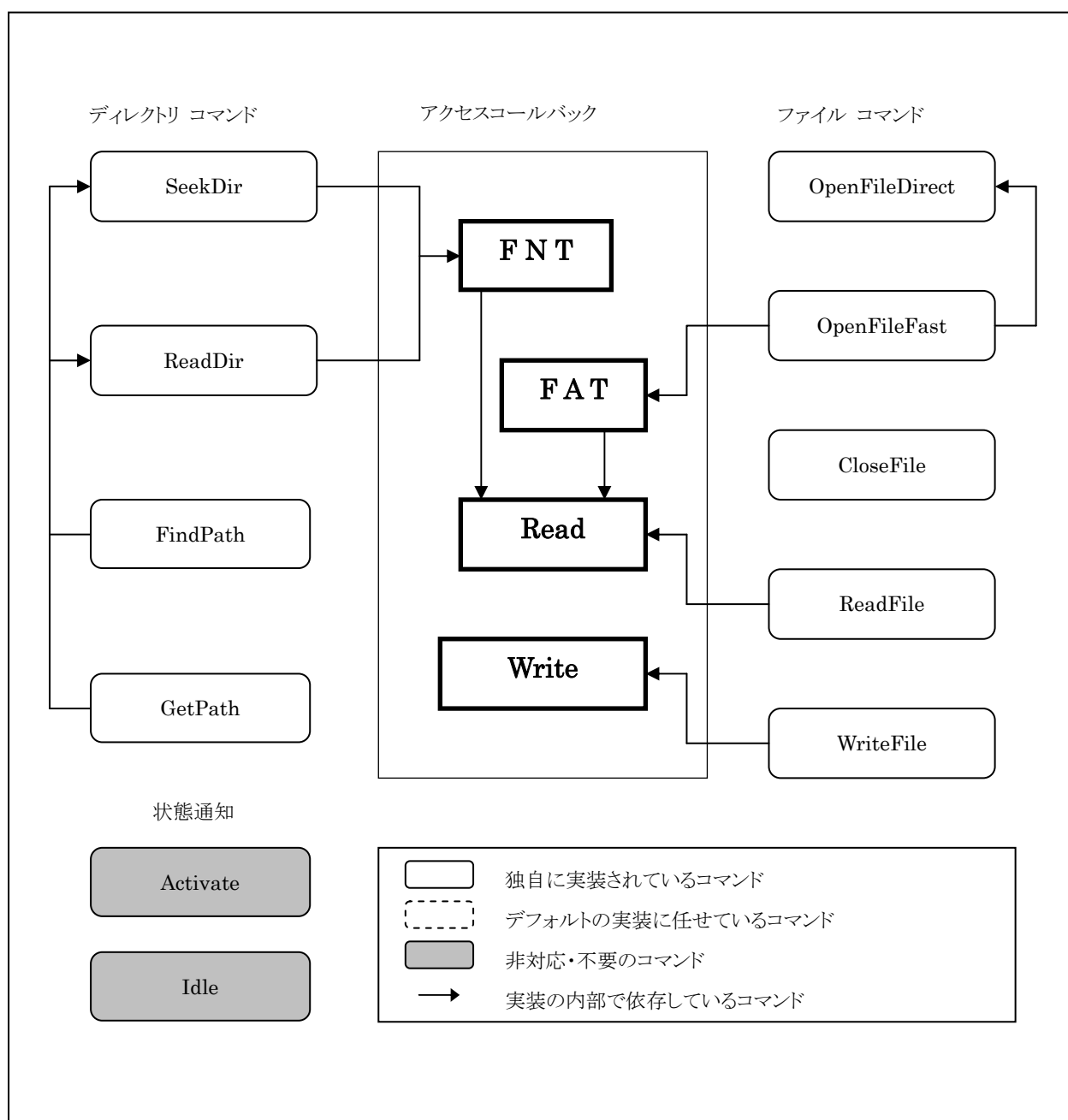


図 3-4 デフォルトプロシージャ



### 3.4.3 アーカイブの実装例

本節では、デフォルトプロシージャとの差分という形で各種アーカイブの実装例を解説します。

#### 3.4.3.1 ROMアーカイブ

ファイルシステムの初期化時から、"rom" という名前のシステム定義のアーカイブが標準でロードされています。このアーカイブは **makerom** ツールで構築され TWL カードの ROM 領域に格納されたファイル群へアクセスし、さらにオーバーレイ操作の一部を処理するためのものです。

このアーカイブが内部で置き換えている処理は、おおむね下図のように表せます。

媒体が ROM であるためファイルのライト処理だけを明示的に非対応とし、それ以外を全てデフォルトに任せています。

また、CARD バスのロックおよびアンロックのために状態通知を使用しています。

この実装の具体的なコードは、SDK サンプルデモ /build/demos/fs/arc-1 で紹介されています。

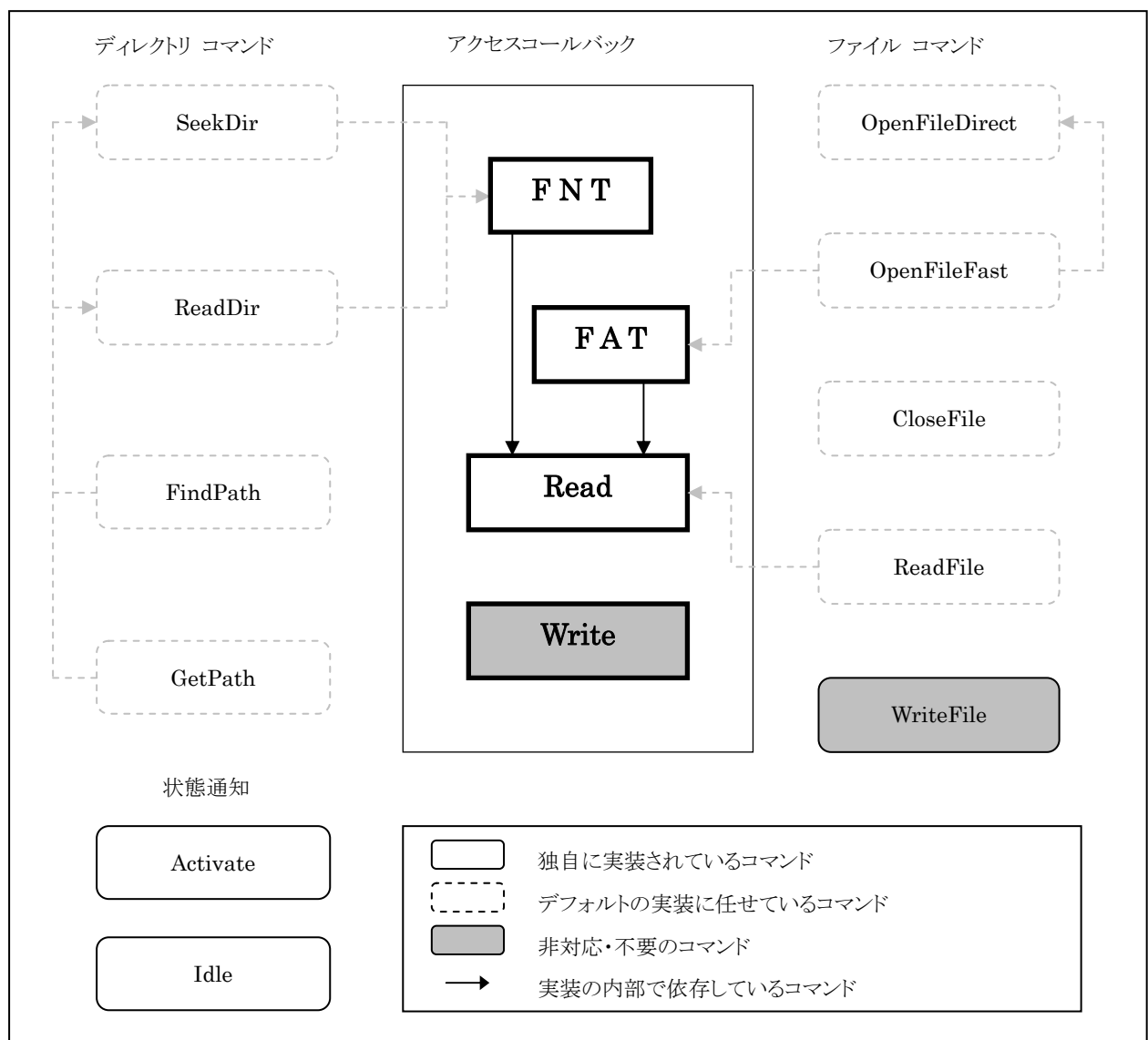


図 3-5 ROM アーカイブプロシージャ

### 3.4.3.2 独自フォーマットのメモリ上アーカイブ

NirtoROM フォーマットとは異なる独自のフォーマットを規定し、そのフォーマットに従った1つのディレクトリ構造がメインメモリ上に存在している場合に、それをアーカイブとして実装する例を下図に示します。

NirtoROM フォーマットと異なるため FNT および FAT は指定せずにそれらに依存していた4つのコマンドをユーザプロシージャで置き換えています。ただし置き換えたコマンドの動作は正しく仕様に従っているので、その上位のコマンドはデフォルトプロシージャを使用することができます。

アクセスコールバックは、ファイルのリードとライトのみで使用します。

この実装の具体的なコードは、SDK サンプルデモ /build/demos/fs/arc-2 で紹介されています。

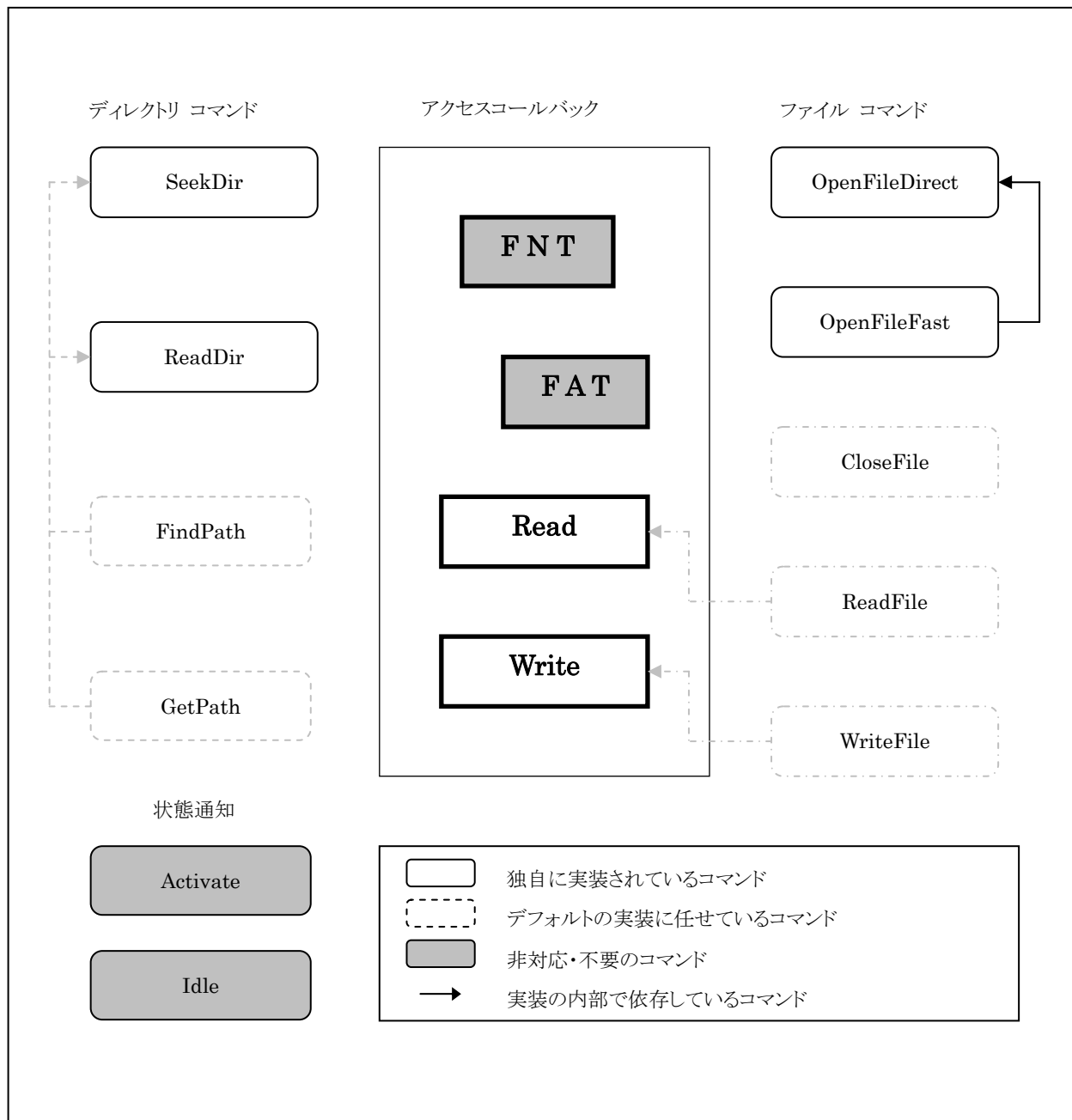


図 3-6 独自フォーマットのメモリ上アーカイブプロシージャ

### 3.4.3.3 無線通信経由のアーカイブ

ワイヤレスダウンロードで起動した子機プログラムが親機の TWL カード内のディレクトリ情報などを参照する、無線通信経由での動的データ取得を目的としたアーカイブの実装例を下図に示します。

データの取得先から前もって FNT および FAT だけは全て受信しておき、これをメモリ上に保持しておくことで、ファイルアクセス以外の全てのコマンドをデフォルトプロシージャに任せています。

ファイルのリードは、通信手段を介したリクエストによって受信完了までのあいだ非同期処理を実現します。

ファイルのライトは、用途にもよりますがこの例では非対応としています。

この実装の具体的なコードは、SDK サンプルデモ /build/wireless\_shared/wfs で紹介されています。

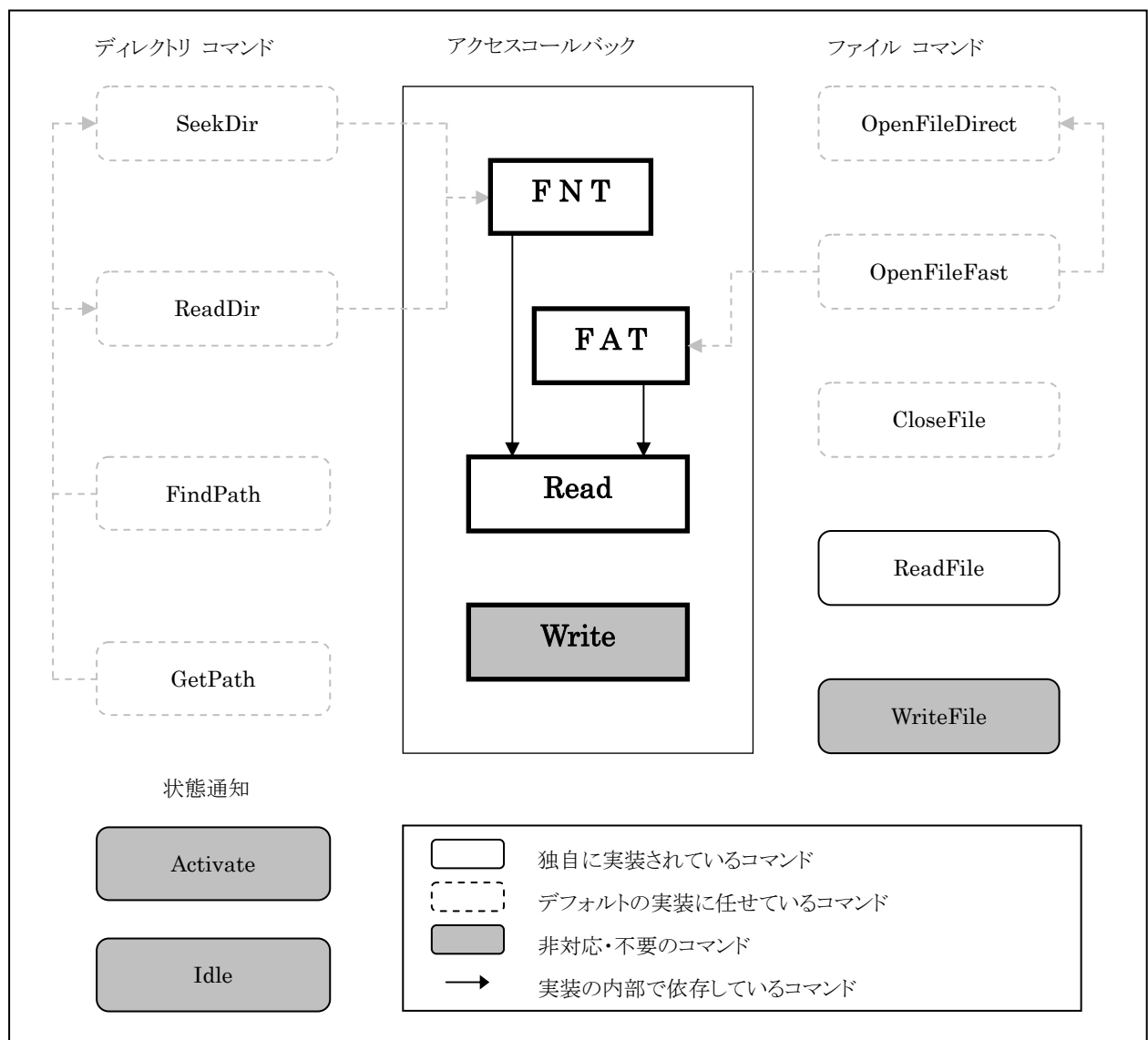


図 3-7 無線通信経由のアーカイブプロシージャ

#### 3.4.3.4 その他のアーカイブ

前述のサンプルデモ以外では、基本的に TWL-SDK がアーカイブの実装方法を直接提供することはありません。NitroROM フォーマットまたはそれ以外のアーカイブ向けフォーマットの形式でデータをパッケージ化するツールとそのパッケージに対してアプリケーションからアクセスするプログラムコードは、独自に用意する必要があります。

また、TWL-System にはアーカイブ用途として公開されている foundation ライブラリ(FND)が存在します。このアーカイブ用関数群と付属のツール `/TWLSystem/tools/bin/nnsarc.exe` とを組み合わせ使用し、標準的なメモリ上アーカイブを構築することが可能です。詳細は TWL-System での各種解説資料およびサンプルデモを参照ください。

## 3.5 APIの解説

前節までは、アーカイブシステム全般とアーカイブの動作について解説しました。

ここでは、これらの定義に基づいてファイルシステムライブラリのインタフェース関数(API)を使用して実際にアプリケーションでアーカイブを操作する手順を解説します。

### 3.5.1 状態操作

アーカイブの操作にあたっては、関数呼び出しの際にFSArchive構造体オブジェクトを使用します。

FSArchiveオブジェクトは各種のコールバックとパラメータを内部に保持します。

ここでは、[「3.3.1 アーカイブの状態遷移」](#)で説明したアーカイブの内部状態を遷移させる手順を解説します。

#### 3.5.1.1 FSArchiveオブジェクトの初期化

ユーザはFSArchive オブジェクトの各内部メンバについて直接操作する必要はありませんが、オブジェクトを使用する前に必ず FS\_InitArchive ( ) 関数を使用して内部状態を初期化する必要があります。初期化されたアーカイブは unregistered 状態になります。

```
/* FSArchiveオブジェクトを最初に使用する前に必ず初期化する */
FSArchive arc;
FS_InitArchive( &arc );
```

表 14 FSArchive オブジェクトの初期化

#### 3.5.1.2 アーカイブ名の登録および解放

アーカイブをファイルシステムにロードする前にアーカイブ名を登録します。

この名前はアーカイブ使用者が自由に指定して構いませんが、ファイルシステム内で一意である必要があります。

名前を登録したアーカイブはファイルシステムに管理され、registered 状態に移行します。

```
/* 指定の名前でアーカイブを登録 */
const char *name = "acl";
const int name_len = strlen( name );
const BOOL ret = FS_RegisterArchiveName(
    &arc, name, name_len );
/* 登録に失敗するのはアーカイブが unregistered 状態でないか
   名前が長すぎるかすでに同じ名前が登録済みの場合 */
SDK_ASSERT( ret );
```

表 15 アーカイブ名の登録

また、アーカイブをファイルシステムからアンロードした後に再び使用しないのであれば、アーカイブ名を解放します。

registered 状態の FSArchive オブジェクトはまだファイルシステムの管理のもとにあるので、

名前の解放をして unregistered 状態に移行しない限りオブジェクトを任意に破棄してはいけません。

```
/* アーカイブの名前を解放 */
FS_ReleaseArchiveName( &arc );
```

表 16 アーカイブ名の解放

### 3.5.1.3 アーカイブのロードおよびアンロード

名前を登録したFSArchiveオブジェクトはファイルシステムにロードすることができます。

[「3.2.1 固有アドレス空間とオフセット」](#) で触れたようにアーカイブはファイルシステムに対してアクセスコールバックとFNTおよびFATの情報を提供する必要があり、ロードの際に FS\_LoadArchive() 関数でそれらを指定します。呼び出しが成功すると、アーカイブは loaded 状態に移行します。

```
/* アーカイブのロード */
const BOOL ret = FS_LoadArchive(
    &arc,                /* アーカイブオブジェクト */
    base_offset,         /* ベースオフセット(ユーザ用) */
    fat_offset, fat_length, /* FAT情報 */
    fnt_offset, fnt_length, /* FNT情報 */
    ArcReadCallback,     /* リードコールバック */
    ArcWriteCallback     /* ライトコールバック */
);
/* ロードに失敗するのはアーカイブが registered 状態でない場合 */
SDK_ASSERT( ret );
```

表 17 アーカイブのロード

ロードされたアーカイブは任意のタイミングでファイルシステムからアンロードすることができます。

アンロードを要求した時点でアーカイブが busy 状態であれば、処理中のコマンドが完了するまでブロッキングされます。完了とともにアーカイブは registered 状態に移行します。

```
/* アーカイブをアンロード */
const BOOL ret = FS_UnloadArchive( &arc );
/* アンロードに失敗するのはアーカイブが loaded 状態でない場合 */
SDK_ASSERT( ret );
```

表 18 アーカイブのアンロード

### 3.5.1.4 アーカイブの停止および再開

アーカイブ動作の停止や再開は、アーカイブの初期化後であれば設定状態によらずいつでも変更することができます。

特に、アーカイブを停止状態のもとに開始したい場合などは、ロード処理の前に停止しておく必要があります。

停止中からの再開処理はただちに完了しますが、動作中からの停止処理は、現在のコマンドが完了するまでブロッキングされます。

```
/* アーカイブの停止 */
const BOOL bak_mode = FS_SuspendArchive( &arc );
/* アーカイブが停止していないと行えない処理を実行 */
...
/* 必要なら前回の動作状態に戻す */
if( bak_mode )
{
    (void)FS_ResumeArchive( &arc );
}
```

表 19 アーカイブの停止・再開

### 3.5.2 ユーザプロシージャ

アクセスコールバックとデフォルトプロシージャだけで全てのコマンドを適切に処理できない場合、`registered` 状態において `FS_SetArchiveProc()` 関数を呼び出してユーザプロシージャを設定する必要があります。

```
/* ユーザプロシージャの設定 */
FS_SetArchiveProc( &arc,      /* アーカイブオブジェクト */
                  ArcProc,    /* ユーザプロシージャ */
                  FS_ARCHIVE_PROC_WRITEFILE /* 問い合わせるコマンド */
                );
```

表 20 ユーザプロシージャの設定

設定したユーザプロシージャは、必要に応じてファイルシステムからコールバック呼び出しされます。ここで独自の処理を行い、適切な結果値を返す必要があります。

```
/* ユーザプロシージャの記述 */
FSResult  ArcProc( FSFile *p_file, FSCommandType cmd )
{
    /* 設定時に要求したコマンドだけが問い合わせられる */
    SDK_ASSERT( cmd == FS_COMMAND_WRITEFILE );
    (void)p_file;
    switch(cmd) {
        /* 特定のコマンドを非対応にすることも可能 */
        case FS_COMMAND_WRITEFILE:
            return FS_RESULT_UNSUPPORTED;
        /* あえて全ての問い合わせを有効にしつつ
           ユーザプロシージャ内で判断することも可能 */
        default:
            return FS_RESULT_PROC_UNKNOWN;
    }
}
```

表 21 ユーザプロシージャの記述

### 3.5.3 非同期処理

アーカイブが非同期処理をサポートする場合、各種のコールバックにおいてそれを実装する必要があります。  
基本的には、結果値を求められる場面で「非同期処理中」を返しつつ当該処理完了後に通知するという形式です。  
あらゆるアクセスが常に非同期処理になる場合、アクセスコールバックを変更します。

```
/* リードコールバック */
FSResult ArcReadCallback(
    FSArchive *p_arc, void *dst, u32 src, u32 len )
{
    /* 非同期の期待できる処理を実行 */
    CARD_ReadRomAsync(
        dma_no, (const void*)src, dst, len,
        OnCardReadDone, p_arc );
    /* 結果値として「非同期処理中」を返す。
       この return より早く完了通知が発生したとしても、
       システムはこれを正しく処理することが保証される。 */
    return FS_RESULT_PROC_ASYNC;
}

/* 非同期処理の完了コールバック */
void OnCardReadDone( void *p_arc )
{
    /* アーカイブへ完了を通知 */
    FS_NotifyArchiveAsyncEnd(
        (FSArchive*)p_arc, FS_RESULT_SUCCESS );
}
```

表 22 アクセスコールバックの非同期化

特定のコマンドにおいてのみ非同期処理になる場合、ユーザプロシーダを変更します。

```
/* ユーザプロシーダの記述 */
FSResult ArcProc( FSFile *p_file, FSCommandType cmd )
{
    switch(cmd) {
    case FS_COMMAND_READFILE:
        /* 特定のコマンドでのみ「非同期処理中」を返す */
        HostIO_Read(
            FS_GetFileImageTop( p_file ) +
            FS_GetPosition( p_file ),
            p_file->arg.readfile.dst,
            p_file->arg.readfile.len
        );
        return FS_RESULT_PROC_ASYNC;
    default:
        return FS_RESULT_PROC_UNKNOWN;
    }
}
```

表 23 ユーザプロシーダの非同期化



## 4 オーバーレイ インタフェース

NITRO および TWL アプリケーションでは、限られたメインメモリ上へ必要な実行コードだけを効率よく配置することのできるオーバーレイという機能を提供しています。

この章ではオーバーレイの動作原理およびインタフェースについて説明します。

### 4.1 スタティックセグメントとオーバーレイセグメント

NITRO および TWL アプリケーションの全てのプログラムコードは「セグメント」と呼ばれる単位にグループ化されます。セグメントは、実行コード、変数領域、定数領域、配置先アドレス、ならびに自身の初期化ルーチンから構成されます。

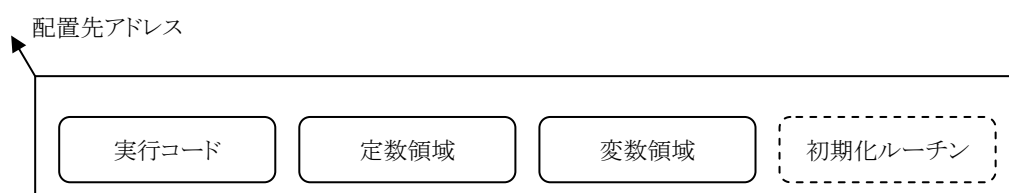


図 4-1 セグメント 概念図

通常のアプリケーションはその全てのコードが起動時にメインメモリ上へ読み込まれ、初期化ルーチン実行ののちにメインのエントリーポイント(NitroMain 関数)へ制御が渡されます。プログラムが実行しているあいだ静的に存在し続けるこれらの領域は「スタティックセグメント」と呼ばれ、ARM9 および ARM7 各プロセッサに 1 個ずつ用意されます。

スタティックセグメントは大規模なアプリケーションを作成するにつれてメインメモリ上の領域を多く占有するようになり、最悪の場合にはメインメモリのサイズを上回ってしまうこともあります。この問題を回避する有効な方法のひとつとして、全てのプログラムコードを不必要に常駐させるのではなく特定の場面や組み合わせでしか使用されないモジュールを必要な時にだけメモリ上へロードするという手段が求められます。この機能を「オーバーレイ」と呼び、このように分けられたモジュールを「オーバーレイセグメント」と呼びます。

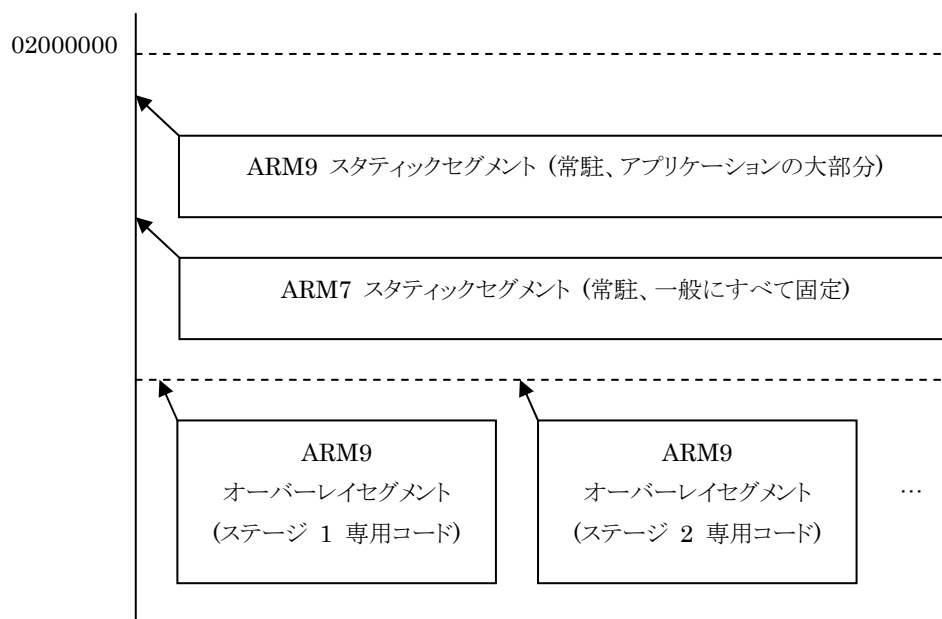


図 4-2 スタティックセグメントとオーバーレイセグメント

## 4.2 オーバーレイの特性

オーバーレイ導入にあたっては、それを使用しない場合とは根本的に異なる幾つかの特性を考慮する必要があります。以下に、それらのうち代表的なものについて述べます。

### 4.2.1 固有の寿命管理

オーバーレイはプログラムの任意の実行タイミングで動的に読みこまれ、また、任意の実行タイミングで解放されます。オーバーレイセグメント内で静的記憶期間を持つオブジェクトの寿命は、このタイミングに支配されることになります。

具体的には、オーバーレイセグメントが読み込まれ初期化ルーチンが実行されたタイミングからグローバルオブジェクトが存在し、オーバーレイセグメントが解放されるタイミングでグローバルオブジェクトが解体されます。C++言語におけるデストラクタはここで実行されます。

この動作は同一のオーバーレイに対して読み込みと破棄を繰り返すたびに生じ、オーバーレイセグメントの内部状態は各々の寿命の中で独立し、寿命を越えて持ち越されることはありません。

これらの働きはオーバーレイ固有でなくセグメントに共通の挙動ですが、スタティックセグメントについては本来の破棄タイミングである NitroMain 関数の終了そのものが通常ありえないため、その点でオーバーレイと異なっています。

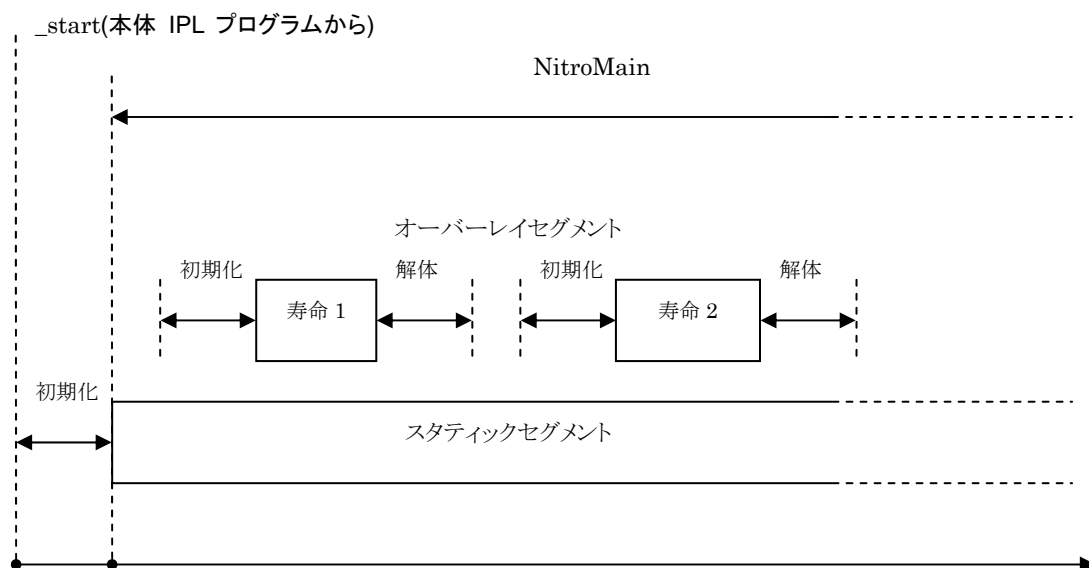
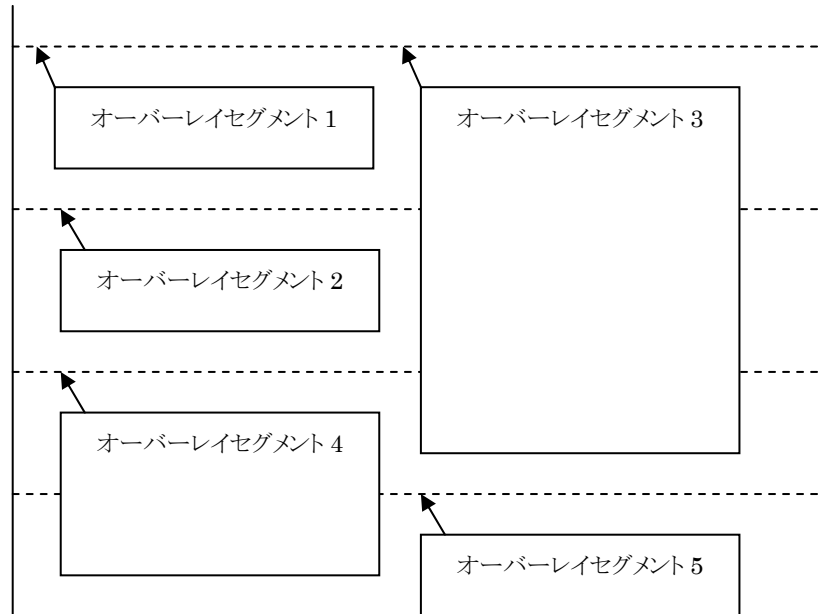


図 4-3 オーバーレイセグメントの寿命

### 4.2.2 配置先領域の競合

オーバーレイが提供する機能はセグメントの動的な読み込み処理だけであり、動的なリンク処理は行いません。オーバーレイセグメントの配置先アドレスおよび他のセグメント相互のシンボル参照は全て静的に解決されます。限られたメモリ上に全てのセグメントを配置していく上で複数のオーバーレイセグメントが領域を競合する場合、それらの複数を同時に使用することは出来ませんので、アプリケーションで適切に検討する必要があります。

競合するオーバーレイセグメントの例を、下図に示します。



オーバーレイ					
ロード側	同時使用				
	1	2	3	4	5
1		○	×	○	○
2	○		×	○	○
3	×	×		×	○
4	○	○	×		×
5	○	○	○	×	

図 4-4 オーバーレイセグメントの競合例

## 4.3 APIの解説

前節までは、オーバーレイセグメントの動作について解説しました。

ここでは、API 関数を使用して実際にアプリケーションでオーバーレイを操作する手順を解説します。

### 4.3.1 .lsfファイルでの指定

プログラムコードからはオーバーレイセグメントを名前で識別します。

オーバーレイの名前と内包モジュールの対応、および配置先の指定は.lsf ファイルに記述します。

.lsf ファイルの記法については makelcf ツールのリファレンスを参照ください。

```
Overlay main_overlay_1
{
    After          main
    Object          $(OBJDIR)/func_1.o
}

Overlay main_overlay_2
{
    After          main
    Object          $(OBJDIR)/func_2.o
}
...
```

表 24 .lsf ファイルでのオーバーレイセグメント指定

### 4.3.2 オーバーレイIDの宣言・定義

プログラムからは、オーバーレイセグメントを「オーバーレイ ID」という型によって指定し、操作します。

オーバーレイ ID はプログラムのリンク時に実体が解決され、プログラムコードからこれを使用するには

FS\_EXTERN\_OVERLAY マクロを使用して明示的に宣言する必要があります。

また、宣言したオーバーレイ ID の参照として FS\_OVERLAY\_ID マクロを使用します。

```
/* 使用するオーバーレイIDを宣言する */
FS_EXTERN_OVERLAY(main_overlay_1);
/* 宣言したオーバーレイIDは その参照を定義可能 */
FSOverlayID    ovl_id = FS_OVERLAY_ID(main_overlay_1);
```

表 25 オーバーレイ ID の宣言・定義

### 4.3.3 オーバーレイのロード・アンロード

オーバーレイは、プログラム実行中の任意のタイミングで読み込む(ロードする)ことができます。

ただし「[4.2.2 配置先領域の競合](#)」で述べたように領域を競合する他のオーバーレイセグメントが存在する場合、それらのいずれかが読み込まれている状態でオーバーレイをロードすることはできません。

また、すでに読み込まれているオーバーレイを解放せずに再度ロードすることもできません。

これらの正当性はライブラリ内部で判定できませんので、アプリケーション側で保証する必要があります。

最も簡単なロード手順は以下の通りです。

```
/* オーバーレイセグメントのロード.  
   関数から制御が返った段階でオーバーレイが使用可能 */  
BOOL ret = FS_LoadOverlay(  
    MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1) );  
/* 何らかの理由により指定されたオーバーレイが無ければ失敗する.  
   このようなことはmakerom生成の通常のプログラムでは起こらない */  
SDK_ASSERT( ret );
```

表 26 オーバーレイのロード

また、ロードされたオーバーレイセグメントを解放(アンロード)する手順は以下の通りです。

ロード時と同様、実際にはロードされていないオーバーレイをアンロードすることがないよう、アプリケーション側で保証する必要があります。

```
/* オーバーレイセグメントのアンロード.  
   関数を呼び出した段階でオーバーレイは使用不可 */  
BOOL ret = FS_UnloadOverlay(  
    MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1) );  
/* 何らかの理由により指定されたオーバーレイが無ければ失敗する.  
   このようなことはmakerom生成の通常のプログラムでは起こらない */  
SDK_ASSERT( ret );
```

表 27 オーバーレイのアンロード

### 4.3.4 ロード処理の分割

前節で解説した FS\_LoadOverlay 関数は、以下の処理を一括して内部で実行しています。

- (1) オーバーレイ ID からオーバーレイセグメントの詳細情報を取得
- (2) オーバーレイセグメントの詳細情報をもとに配置先アドレスへセグメントデータを読み込む
- (3) オーバーレイセグメントの初期化ルーチンを実行し、オーバーレイを有効にする

これらの各部分を分けて段階的に実行する必要がある場合は、各々の処理を行う単機能の関数を組み合わせて順次呼び出していきます。このような分割処理が必要とされるのは、主に上記 (2) のデータ読み込みで生じる処理時間に関する問題を回避する目的があります。

読み込みに 1 ピクチャーフレーム以上の時間を要する大きなオーバーレイを扱いつつゲームを進行する場合や、[「3.4.3.3 無線通信経由のアーカイブ」](#) のような構成のもとにセグメントデータを取得する場合、あるいは、将来においてより低速な CARD-ROM デバイスを使用したアプリケーションを実装する場合などにこの分割処理は有用となります。

分割されたロード手順は以下の通りです。

```
/* (1) オーバーレイ ID からオーバーレイ情報取得 */
FSOverlayInfo info;
if (FS_LoadOverlayInfo(&info,
    MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1)))
{
    /* (2) オーバーレイ情報を基にデータを読み込み */
    FSFile file;
    FS_InitFile(&file);
    (void)FS_LoadOverlayImageAsync(&info, &file);
    (void)FS_WaitAsync(&file);
    (void)FS_CloseFile(&file);
    /* (3) 初期化ルーチン実行 */
    FS_StartOverlay(&info);
}
```

表 28 ロード処理の分割

© 2005–2009 Nintendo

任天堂株式会社の許諾を得ることなく、本書に記載されている内容の一部あるいは全部を無断で複製・複写・転写・頒布・貸与することを禁じます。