NITRO-SDK
File System Library Manual

Version 1.0.1

The contents in this document are highly
confidential and should be handled accordingly.

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo of America Inc. and/or Nintendo Company Ltd. and are protected by Federal copyright law. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Contents

Code

Figures

Revision History

Version	Revision Date	Description
1.0.1	8/19/2005	2.2.1 Added a section (2.2.1.1. Initializing the FS Library) 2.2.3.2. Revised code (revised sample code in the list)
1.0.0	1/11/2005	Initial release.

1   Introduction

The NITRO-SDK has a File System library to handle the files and overlays of applications created in the `NitroROM` format and to make your own extensions to these files and overlays.

This document explains the basic organization of the File System library and how to use the library.

To learn more about the `NitroROM` format, refer to the NITRO ROM File System Specifications. This document can be found in the NITRO-SDK source tree as the file:

    /NitroSDK/docs/TechnicalNotes/NitroRomFormat.rtf

1.1   Overview

With the NITRO-SDK, when the `NITRO_MAKEROM` build switch is enabled for building, the `makerom` tool generates the application in the `NitroROM` format. (This build switch is enabled by default, so applications are normally created in this format.) The generated application stores one set of directories, along with information on the files that are included in those directories and, if specified, overlay information as well.

The "File System" is the name used for the mechanism for accessing and manipulating this data from the application. In broad terms, this File System is composed of the module blocks listed below. The following chapters provide explanations of these blocks.

- File/Directory Interface   Mechanism for transparent access to files and directories
- Archive System   A collection of data-access processes built into the File System in a format compatible with the File/Directory Interface
- ROM Archive Interface   A standard internal definitions archive for accessing NITRO-CARD
- Overlay Interface   General operations for overlay
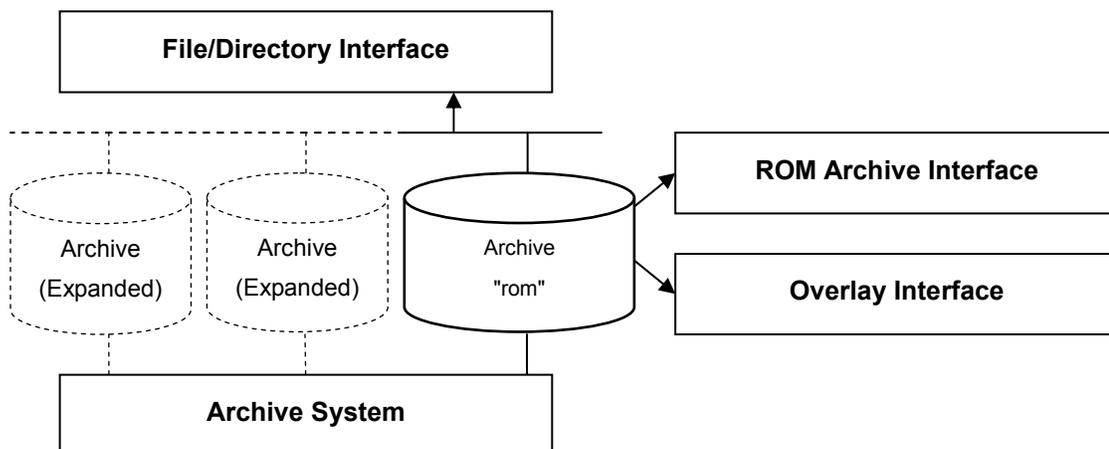
Figure 1-1   Schematic Overview of File System

1.2   How to Use the File System

In order to use the File System from your application, you need to build the application with the settings described below. (These specifications are simply ignored when the NITRO-SDK library itself gets built.)

- To enable the File System in the application, enable the `NITRO_MAKEROM` build switch. This specification is necessary in order to execute the `makerom` tool as described in `commondefs` from the `make` command. (Since this build switch is enabled by default, the application is normally built this way.)

- If directories and files are to be used in the application, specify a `.rsf` file in the `ROM_SPEC` build switch. The `makerom` tool will store the information on directories and files (as described in the `.rsf` file). To read more about `.rsf` files, see the `makerom` item of Tools in the NITRO-SDK Function Reference Manual.

- If the application uses overlays, specify `.lsf` files with the `LCFILE_SPEC` build switch, and specify the source file for the overlay in the `SRC_OVERLAY` build switch. These specifications get passed to the `makelcf` tool as described in `commondefs` from the `make` command. To read about the notation rules for `.lsf` files, see the `makelcf` item of Tools in the NITRO-SDK Function Reference Manual.

- If the application makes use of overlays, in special situations, enable the `NITRO_DIGEST` build switch. These are situations where the NITRO-CARD storing the overlay information cannot be accessed directly, so the information must be acquired indirectly via wireless communications or some other means of communications. For overlay information obtained under such circumstances, it is necessary to guarantee the correctness of execution code. This build switch must be specified so the NITRO-SDK can act internally to determine this correctness. (For details, see the DS Download Play Manual.)

2   The File/Directory Interface

A series of basic capabilities has been built into the File System Library to specify and manipulate directories and files. This chapter explains the interface to those capabilities.

2.1   Definitions of Terminology

Terms like "file" and "directory" that are adopted by the File System Library and appear in this document are generally used in the same way they are used by the operating system on a standard PC.

This section presents the strict definitions of these terms as they pertain to the File System Library.

2.1.1   Entry

An entry is a hierarchical element. It holds information for identifying a single specific file or a single specific directory. Each entry must have a name that does not duplicate the names of other entries at the same hierarchical level. The name can be composed of up to 127 characters of ASCII code. Uppercase and lowercase are not distinguished, and the following characters cannot be used: ¥ / : ; * ? " < > |
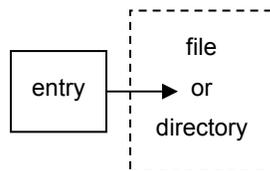
Figure 2-1   A Typical Entry

2.1.2   Directory

A directory expresses information for a single level in the hierarchy. It contains zero or more entries and information that identifies each entry. It also has information that identifies the directory at the top of the hierarchy (the parent directory).
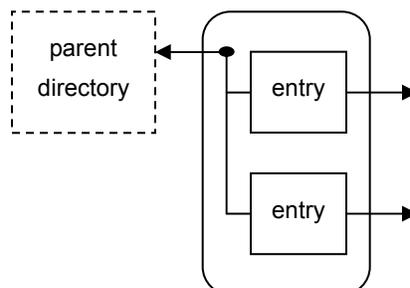
Figure 2-2   A Typical Directory

2.1.3   File

A file is the information for referencing a unique object possessing binary data. "Opening" the file commences operation of the object, and "closing" the file ends operations. The file behaves like linear memory when using the "read" and "write" operations.
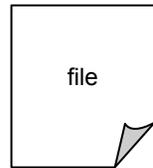
Figure 2-3   A Typical File

2.1.4   Archive

An archive is an object that has information for files, directories, and entries, as well as the means to control these files, directories, and entries.

Each archive has a single name that does not duplicate the name of any other archive inside the File System. This name is composed of up to 3 alphanumeric characters. Names are not case-sensitive.

The archive encompasses a single hierarchical relationship, with an unnamed directory at the highest level (the root directory).
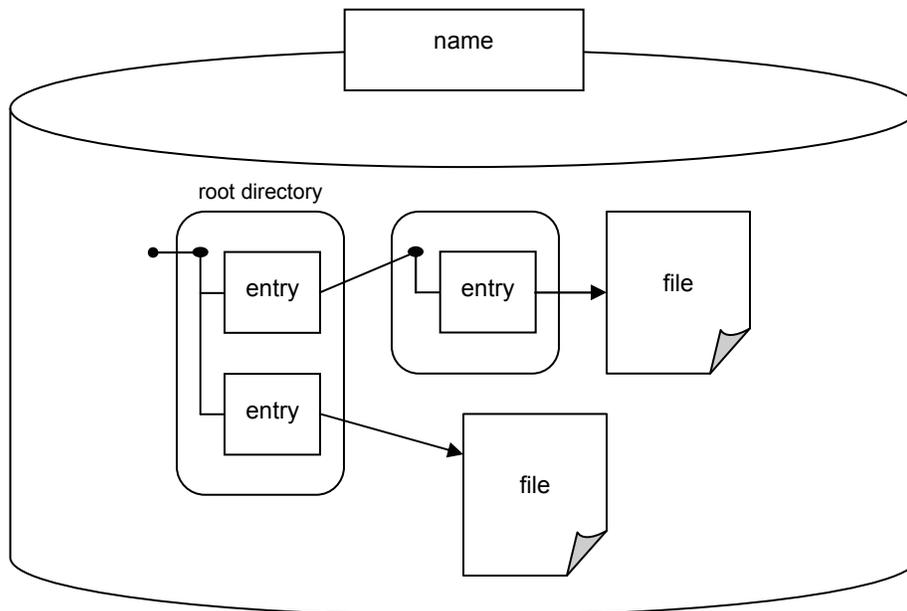
Figure 2-4   A Typical Archive

2.1.5   Path

An arbitrary number of archives can exist in parallel in the File System. Each entry can be uniquely identified by using a combination of the archive name and the entry name for each hierarchical level from the root directory. This combination of names is called the "path." (It is also sometimes called the path name or the path string.)

If the entry information indicates a directory, then the path is called a "directory path." Similarly, if the entry information indicates a file, then the path is called a "file path."

2.1.5.1   Path Format

A path is expressed as a character string, entered in any of the following formats:

1)   " (Archive name) : / "
2)   " (Archive name) : / Entry name / Entry name / ... / Entry name / "
3)   " (Archive name) : / Entry name / Entry name / ... / Entry name "

All entries that are not at the end of the path must be entries that indicate directories.

If there is a slash character ("/") at the end of the path, this means it is a directory path.

Paths 1) and 2) above are both examples of directory paths. The 1) format is the only format that can express the root directory path of an archive. Path 3) can be either a directory path or a file path. If the final entry in this path indicates a directory, then the path is equivalent to path 2). In other words, there is no distinction between directory paths with and without a slash ("/") at the path end.

2.1.5.2   Relative Path Format

The File System allows parts of the path to be omitted. When parts of the path are omitted, the File System uses the directory path in memory as the base from which to supplement the omitted parts. This path in memory is called the "current directory," and a path with omissions is called a "relative path." A normal path with nothing omitted is called an "absolute path."

The relative path is supplemented from the current directory by following these rules:

1) If the entry starts with a slash ("/"), then the path is supplemented with the root directory of the archive to which the current directory belongs.

2) If not, then the path is supplemented by simply attaching it to the end of the current directory path.

Thus, if the current directory is `rom:/text/` then the relative path `/snd/dat` gets changed to the absolute path `rom:/snd/dat`, whereas the relative path `snd/dat` gets changed to the absolute path `rom:/text/snd/dat`.

2.1.5.3   Notation of Special Paths

Two special entry names are reserved for use with both absolute paths and relative paths:

1) The entry name "`.`" indicates the directory in which this entry resides.
2) The entry name "`..`" indicates the directory one level above the directory in which this entry resides.

2.1.6   File ID

Each archive has unique index values that identify the files that belong to the archive. The entries in the directory hierarchy specify files using these index values. From the set of information about the archive and the index value, every file in the entire File System can be uniquely identified. This set of information is called the "file ID."

2.1.6.1   Correspondence Between File, File Path, and File ID

In the documentation relating to the File System, the term "file" may be used to refer to the file path, the file ID or the file itself, depending on the context of the sentence. The relationship between these three terms is as follows:

- "File" indicates the file itself, and only one such file exists in a given archive.
- When some entry indicates a file (including the index value), the path for that entry is the "file path," but sometimes this will be simply referred to as the "file," meaning "the File specified by the file path."
- The same goes for the term "file ID." Sometimes this will be simply referred to as the "file," meaning "the File specified by the file ID."
- If there is a "file path" and a "file ID," then a unique "File" exists. However, this does not mean that the file path or file ID that identifies an arbitrary file always exists.

This means that the archive does not require an index value and an entry for each file. Thus, the archive is permitted to contain files that cannot be pinpointed. (Such files are typically created for temporary use.) The following figure shows the example of each file path and ID of the archive that has two files with entries on the directory hierarchy, and three files for which index values have been provided, and four files that actually exist.
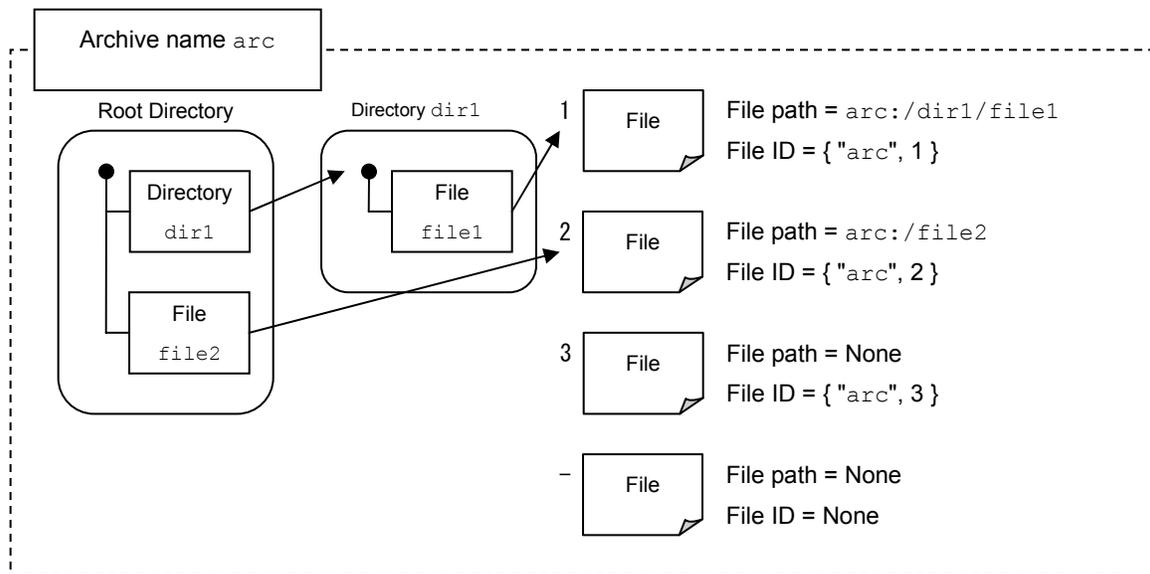
Figure 2-5   Example of Correspondence Between File, File Path, and File ID

2.2   Explanation of the API

The previous section discussed various definitions for the NITRO-SDK File System. This section uses those definitions to explain ways of using the File System Library's interface functions (API) to actually manipulate files and directories from the application.

2.2.1   Common Operations

`FSFile` structure objects are used when calling functions in the File/Directory Interface. The `FSFile` object saves information related to the file or directory, and the internal state of the `FSFile` object is updated in accordance with the current process.

2.2.1.1   Initializing the FS Library

Before using any function in the FS library, you must initialize the FS library with the `FS_Init()` function. Calling this function once is sufficient.

During initialization the FS library performs card accesses internally, so a single DMA channel must be allocated for this. Notice that this DMA channel will be used exclusively internally until the FS library is released by the `FS_End()` function. Also, because the IO register is the card access transfer source, DMA channel 0 cannot be used.

If you are not going to allocate a DMA channel to the FS library, you can explicitly specify `FS_DMA_NOT_USE` as a special value. In this case, the CPU will process card access.

```
/* Initialize before using FS library */
#define DMA_CHANNEL_FOR_FS  2 /* DMA to use with FS */
FS_Init( DMA_CHANNEL_FOR_FS );
```

Code 1 FSFile Object Initialization

2.2.1.2   Initializing the FSFile Object

The internal state of an `FSFile` object must be initialized with the `FS_InitFile()` function before the object is used. The user does not need to directly operate on any of the various internal members of the `FSFile` object.

```
/* Must initialize FSFile object before using it first time * /
  FSFile  file;
  FS_InitFile( &file );
```

Code 2 Initializing the FSFile Object

If the `FSFile` object stores file-related information, that object can also be called the "file handle." If the object stores directory-related information, that object can also be called the "directory list." A single `FSFile` object cannot store multiple sets of file or directory information.

2.2.1.3   Getting the Path

If the `FSFile` object stores file or directory information, you can use the `FS_GetPathName()` function to get the file path or the directory path.

```
/* Get path length for content held by FSFile object */
const s32  len = FS_GetPathLength( &file );
/* If -1 is returned here, either the specified object is a
file without an entry (as described in 2.1.6.1 Correspondence
between file, file path and file ID) or the FSFile object
holds no information. */
if( len >= 0 )
{
    /* Prepare enough memory to store path name */
    char  *buf = (char*)OS_Alloc( len );
    if( buf )
    {
        /* Actually get the path name */
        BOOL  ret = FS_GetPathName( &file, buf, len );
        if( ret )
        {
            OS_Printf("path=%s¥n", buf);
        }
        OS_Free( buf );
    }
}
```

Code 3 Getting the Path from the FSFile Object

2.2.1.4   Manipulating the Current Directory

Almost all functions that obtain file or directory information for an `FSFile` object require a path. As described in 2.1.5 Path, there are both absolute paths and relative paths, and the File System internally manages a single "current directory" that gets used to supplement a relative path.

When the FS Library is initialized, the current directory gets set to the ROM Archive's root directory `"rom:/"` by default. Users can change the setting using the `FS_ChangeDir()` function.

```
/* The current directory is "rom:/" */
BOOL  ret;
/* If a relative path has been specified, it gets
supplemented with the current directory */
ret = FS_ChangeDir( "dir_1" );

/* If a directory named "rom:/dir_1/" exists, the current
directory gets changed and TRUE is returned to ret. */

/* If an absolute path has been specified, the current
directory is ignored */
ret = FS_ChangeDir( "arc:/" );
/* If an archive named "arc" exists, the current directory
gets changed to be that archive's root directory.*/


...

/* If the archive has been released from the File System or
for any other reason the target indicated by the current
directory has become invalid, then the current directory
automatically changes to rom:/, which is always guaranteed
to exist. */
FS_ReleaseArchiveName( FS_FindArchive( "arc", 3 ) );
```

Code 4 Changing the Current Directory

2.2.2   Manipulating Directories

To search the directory structure from the application at the time of execution, use the `FSFile` object as a directory list to enumerate entries to obtain the information. The directory list is stored inside the `FSFile` object as the combined information that consists of directory and enumeration location. This combination of information is expressed by the `FSDirPos` structure. It also goes by the name of "directory position."

The directory list is normally manipulated by the procedures described below. Use these operations as you deem best for your application.

2.2.2.1   Getting the Directory List

There are two ways to get the directory list into the `FSFile` object. The first way is to use `FS_FindDir()` function to specify a known path in the File System. When this function is used, the obtained directory list is always initialized with the enumeration position pointing to the first entry in the list. The second way is to use to `FS_SeekDir()` function to specify the directory position. When this function is used, the obtained directory list is initialized with the specified directory-position information, which includes information about its position in the list. You can use the `FS_TellDir()` function to get this directory position from the already obtained directory list, or you can follow the procedure described below and get it using the `FS_ReadDir()` function.

```
    BOOL   ret;
    FSFile dir;
    FS_InitFile( &dir );
    /* Get directory list from known path  */
    if( FS_FindDir( &dir, "rom:/" ) )
    {
        /* Get and store directory location using several
    prepared procedures */
        FSDirPos   pos;
        ret = FS_TellDir( &dir, &pos );
        SDK_ASSERT( ret );
        /* Get directory list from an directory location
    already obtained */
        ret = FS_SeekDir( &dir, &pos );
        SDK_ASSERT( ret );
```

Code 5 Getting the Directory List

2.2.2.2   Enumerating Entries from the Directory List

Entry information can be obtained one set at a time from the current list position by using the `FS_ReadDir()` function. The entry information is obtained in the form of the `FSDirEntry` structure, and the list position then advances to point to the next entry. This process can be repeated until the end of the list is reached.

```
    FSDirEntry  entry;
    /* When end of list is reached, FS_ReadDir() returns FALSE */
    while( FS_ReadDir( &dir, &entry ) )
    {
        /* The information in the obtained entry includes the
    entry name and whether the entry is a file or a directory */
        OS_Printf( "<%c>%s¥n",
            entry.is_directory ? 'F' : 'D', entry.name );
    }
```

Code 6 Listing Entries

2.2.2.3   Searching in Lower-Level Directory Lists

There may be times when you want to include a directory's subdirectories in your target search. This is generally done by using recursive functions on obtained entries that prove to hold directory information, and you need to be careful about stack overflow, which is a problem that all sorts of recursive processes have in common. Note that the FSDirEntry object consumes a lot of stack memory because it includes a buffer that is the size of the largest entry name, and that the FSFile object used for searching is also relatively large. Because of this, it is best that you write you code so neither of these is maintained for every level.

Following is an example of a recursive kind of search process that does not consume a lot of stack memory.

```
/* Recursive function that dumps entries from specified
directory positions. Uses FSFile and FSDirEntry arguments */
void   DumpDirEntriesSub(int tab,
    FSFile *p_dir, FSDirEntry *p_entry)
{
    /* Output directory names */
    OS_TPrintf( "%*s%s/¥n", tab, "", p_entry->name );
    tab += 4;
   /* Enumerate the entries in the directory */
    if( FS_SeekDir( p_dir, &p_entry->dir_id ) )
    {
        while( FS_ReadDir( p_dir, p_entry ) )
        {
            if( ( p_entry->is_directory == 1 ) )
            {
                /* Recursion to lower subdirectory, then return.
                 Uses FSFile and FSDirEntry entities*/
                FSDirPos   cur_pos;
                if( FS_TellDir( p_dir, &cur_pos ) )
                {
                    DumpDirEntriesSub( tab, p_dir, p_entry );
                    (void)FS_SeekDir( p_dir, &cur_pos );
                }
            }
            else
            {
                /* Output file names */
                OS_TPrintf( "%*s%s¥n", tab, "",
                    p_entry->name );
            }
        }
    }
}

/* This function is the starting point for recursive dumping */
void   DumpEntries(const char *dir_path)
{
    /* Secure the only entity used inside recursive processes */
    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 );
    }
}
```

Code 7 Example of a Recursive Search Process

2.2.3   Manipulating Files

To handle files within your application, use the `FSFile` object as a file handle and call functions to access the file and its data. The file handle is kept as the combination of binary data information and the seek position inside the `FSFile` object. (The binary data itself is stored not inside the `FSFile` object but rather inside some archive.)

Operations done with the file handle are performed with the procedures broadly outlined below. Use these operations as you deem best for your application.

2.2.3.1   Opening and Closing Files

You need either a file path or a file ID to specify a file from the application.

Content of the `FSFile` object becomes a file handle when you specify a file path with the `FS_OpenFile()` function or a file ID with the `FS_OpenFileFast()` function. In either case, the operation is tantamount to opening the file. All manipulations on files are done using this file handle. After you are done with the file handle, use the `FS_CloseFile()` function to release it. This operation is tantamount to closing the file.

These operations are necessary for appropriate management of internal resources in archives, where there are restrictions on the total number of files that can be open.

```
FSFile   file;
FSFileID  file_id;
FS_InitFile( &file );
/* Open/close file from known file path */
if( FS_OpenFile( &file, "rom:/" ) )
    (void)FS_CloseFile( &file );
/* Open/close file from File ID */
if( FS_ConvertPathToFileID( &file_id, "rom:/" ) )
{
    if( FS_OpenFileFast( &file, file_id )
        (void)FS_CloseFile( &file );
}
```

Code 8 Opening and Closing Files

2.2.3.2   Getting File Size and Setting Seek Position

There are only two basic operations performed on files: reading and writing. For these operations you always need the "seek position" and the "size." Use the `FS_GetLength()` function to get the overall size of the file. Get the current seek position maintained by the file handle using the `FS_GetPosition()` function. Move around using the `FS_SeekFile()` function.

```
/* Compute remaining bytes from total size and current
position */
const u32  pos = FS_GetPosition( &file );
const u32  len = FS_GetLength( &file );
const u32  rest = (u32)(len - pos);
void  *enough_buf = OS_Alloc( rest );
/* Move seek position to the start */
(void)FS_SeekFile( &file, 0, FS_SEEK_SET );
```

Code 9 Getting the File's Size and Seek Position

2.2.3.3   Reading and Writing Binary Data

Use the FS_ReadFile() function to read binary data from the current seek position of the file. Use the
FS_WriteFile() function to write binary data from the current seek position of the file.

With either function, after the process ends the seek position moves by an amount equal to the size of the data
that was actually accessed.

```
/* Read text file and output for debugging */
char   string_buf[256 + 1];
string_buf[ sizeof(string_buf) - 1 ] = '¥0';
/* Read size becomes zero when end of file is reached */
while( FS_ReadFile( &file, string_buf,
sizeof(string_buf) - 1 ) > 0 )
    OS_PutString( string_buf );
```

Code 10 File Reading/Writing

Depending on how the archive is implemented the read/write process may not end immediately and the processor
itself may conduct some other task during the reading/writing. Typically, the application side uses threads to control
this kind of asynchronous process. But asynchronous versions of the read and write functions have been prepared
to perform these operations with respect to the archive. If the archive has been implemented to suit asynchronous
processes, you can use the FS_ReadFileAsync() and FS_WriteFileAsync() functions to return control
immediately without waiting for the process to end.

To check whether the process has actually ended, use the FS_IsBusy() function. To wait for the process to end,
use the FS_WaitAsync() function.

If the archive does not perform asynchronous process, these asynchronous functions will operate the same way
as the synchronous functions. In cases like this, the FS_IsBusy() function always returns FALSE, and the
FS_WaitAsync() function returns control without doing anything, so this can be considered the same as the
case where the asynchronous process completed immediately.

```
/* Execute other processes at the same time as the
asynchronous read process. This is more effective when
it is a serial processes relating to the file data. */
while( FS_ReadFileAsync( &file, string_buf,
sizeof(string_buf) - 1 ) > 0 )
 {
    DrawScreen( );
    FS_WaitAsync( &file );
    OS_PutString( string_buf );
}
```

Code 11 Asynchronous Read of File

3   Archive System

Chapter 2 talked about the File/Directory Interface and how it is used. This chapter introduces the Archive System, which is the framework for implementing internal operations by following the interface. This chapter explains the configuration and operations of the Archive System, as well as the archive interface.

3.1   The Purpose of the Archive System

As depicted by its position in the File System in 1.1 Overview, the Archive System only provides functions for implementing archives. In using the File System library, the user application has no need for the Archive System alone without the other module blocks.

The Archive System is primarily used by those who are implementing NITRO application middleware and utilities.

The Archive System may prove useful in the following applications:

- For the sharing, extension, or reuse of program code between existing modules and newly introduced modules
- To hide from users the internal implementation of a data-storage medium where complex controls are required

3.2   Archive Configuration

An archive is defined as an object that holds information for a number of basic parameters and callback functions. Explanations for some terms are presented below.

3.2.1   Unique Address Space and Offsets

The File System is designed with the expectation that the information stored in the archive has a linear data structure conforming to the `NitroROM` format. For this reason, there must be a unique address space that begins from 0 inside the archive, and there must be a means provided for accessing the data images of the FNT, FAT, and each files in that space.

This means is provided through a pair of callback functions for reading and writing. For the remainder of this document, these functions will be called the "read callback" and the "write callback." Together, the pair will be called the "access callbacks."

Addresses in the unique address space are called "offsets" in order to distinguish them from the address map in the CPU.

3.2.2   Commands and User Procedures

By providing the access callbacks and FNT and FAT offsets in the correct manner, the archive can transparently satisfy user requests even if the user does not have a firm handle on the actual processes of the File System.

But a method has also been prepared that can resolve issues when a part or all of the unique address space cannot be made to conform to the `NitroROM` format. There is a set of defined processes called "commands" that can be used to access the archive from the File/Directory Interface. Each of these commands can be set to query the archive before the access callback is executed. The archive can process these query-making commands using

callback functions called "user procedures" and directly replace the commands with an independent implementation. In this way, even an archive that does not strictly conform to the `NitroROM` format can be created that satisfies all requests from the File System.

There is also a set of standard processes called the "default procedure" that is executed without the replacement step of user procedures.

3.3   Archive Operations

Archive processes run automatically from the File System driven by callbacks. This section explains how the archive operates inside the File System. The functions shown in the figures and tables are explained in 3.5 The API.

3.3.1   Archive State Transitions

The archive's internal state has two components: its state set in the File System, and its own operating state.

3.3.1.1   Transitioning Through Archive States

The archive can transition through three states in the File System, as shown below.

Table 3-1   File System State Set

State in File System	Meaning
Unregistered	The archive does not have any association with the File System. The archive begins in this state immediately after initialization.
Registered	The archive has been registered with a unique name in the File System. In this state, the archive is included in the File System but it is not operating.
Loaded	Access callback has been executed and archive is loaded to the File System. Only in this state can commands be issued from the File/Directory Interface.

The transitions between these states are depicted in the following figure.
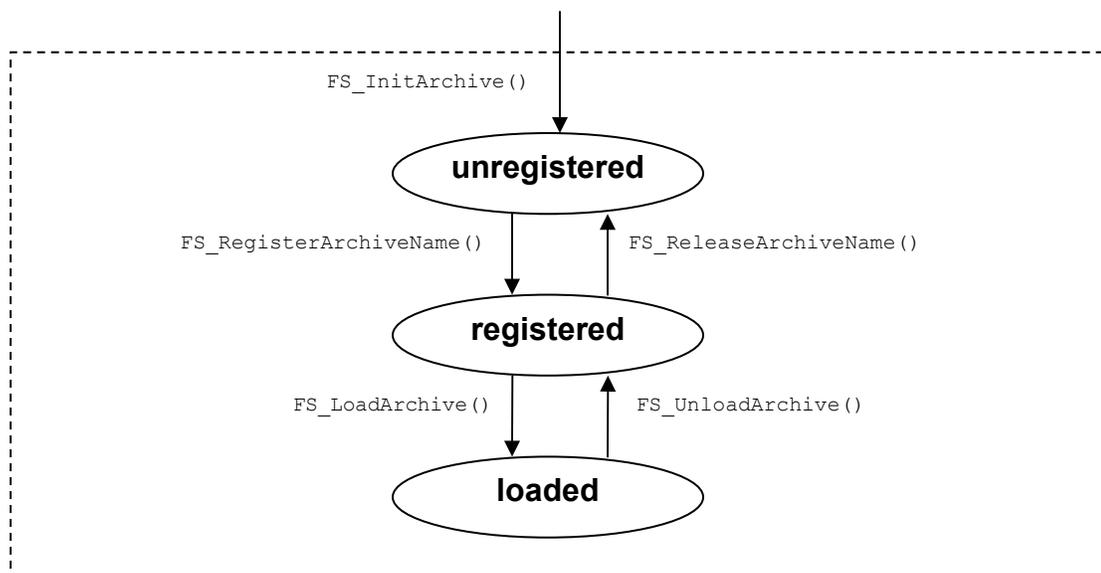
Figure 3-1   Transitioning Through Archive States

3.3.1.2   Transitioning Through Operating States

The archive itself transitions through three different operating states according to the operation of the archive itself.

Table 3-2   Transitions Between States

Operating State	Meaning
Suspended	Archive operations have been stopped. Commands from the File/Directory Interface are kept on hold until the archive begins operating again.
Idle	Archive is operating, but there are no unprocessed commands. This is the timing when the first command is generated.
busy	Command is being processed. The archive moves to this state after the first command is issued.

The transitions between these states are depicted in the following figure.
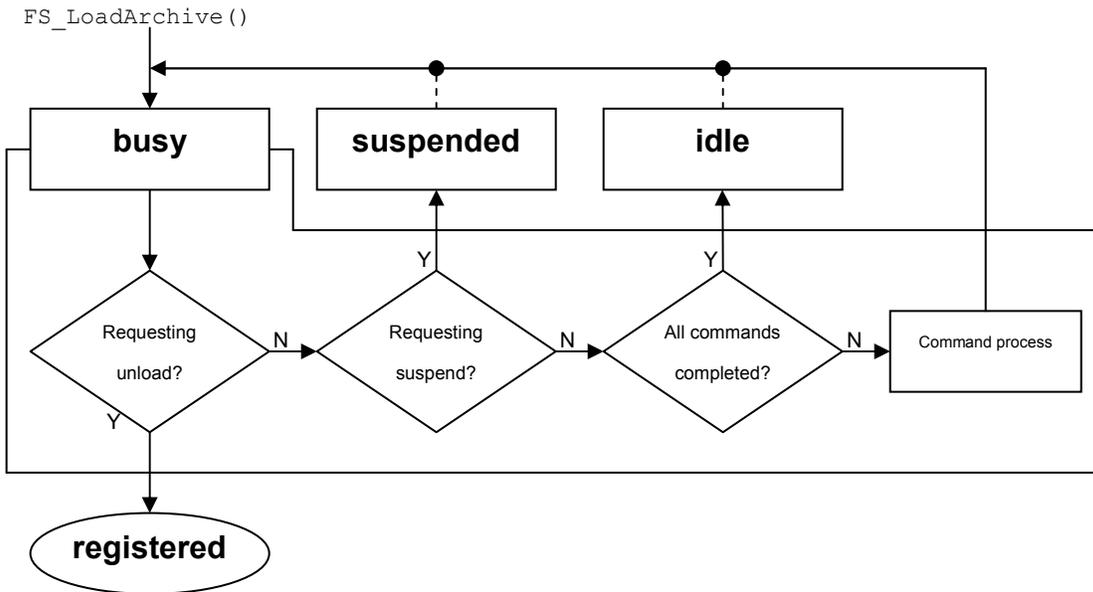
**Figure 3-2   Archive Operating-State Transitions**

3.3.2   Command Process Sequence

Command requests are sent in series from the File System to the archive, and if unprocessed commands pile up, they get reserved in first-come order. The File System drives callbacks so that every archive always processes commands one at a time, but it is nevertheless possible to operate multiple archives in parallel in the File System without the archives interfering with each others' states.

When an archive is in the *busy* state, the processing of single commands is executed with either user procedures or the default procedure, as described above in 3.2.2 Commands and user procedures. With either procedure, after the process is executed one of the result values gets returned. Normally the command ends at this point.

If the process in the archive is an asynchronous process (as mentioned in 2.2.3.3 Reading and writing binary data), then the procedure returns "asynchronous processing" as a result value. If this is the case, the archive itself will need to notify the File System of the result when the process has ended. Until the File System receives this notification it will suspend busy-state processes. If the command that gets suspended here is not a command that was issued from the call to an API for an asynchronous process like file reading or writing, then the File System will block process-end notifications inside that call.

The command process flow for this is as follows.
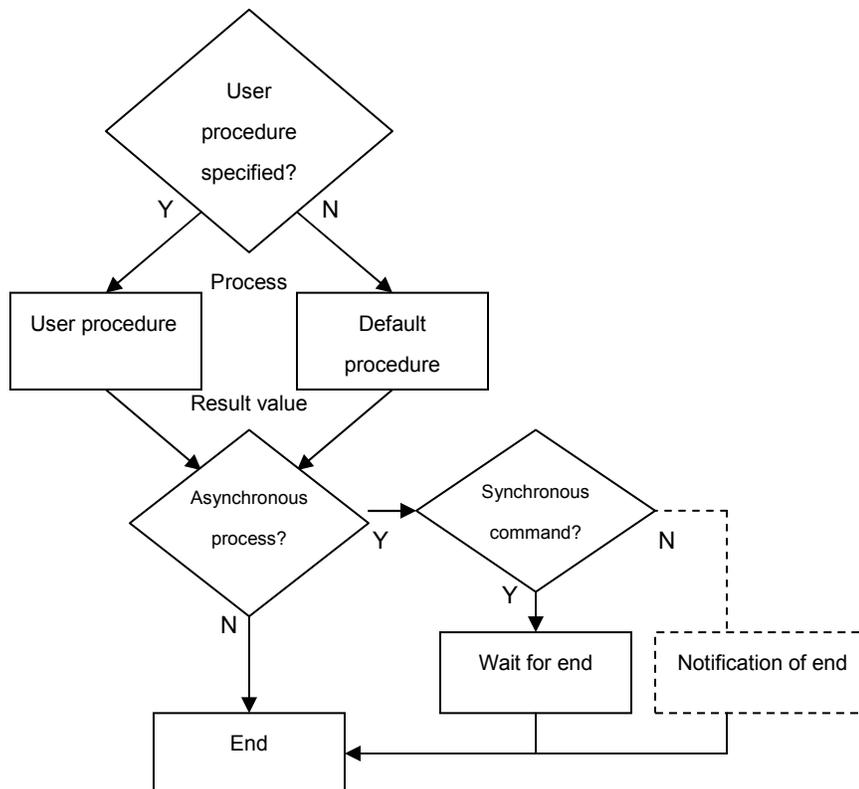
Figure 3-3   Command Process Flow

3.4   Archive Designs

This section covers the broad guidelines you should consider when implementing your own archives and presents several implementation examples.

3.4.1   Standard Specifications

Basically speaking, implementing an archive involves just properly describing three callback functions for the access callback and user procedures. The main task is to wrap the target-specific characteristics in these callbacks so that they are created as close as possible to the standard specifications expected by the File system.

The standard specifications that the File System expects of the target are shown below. The three sets of conditions are presented in order of appropriateness; a target that meets the first set of conditions has the easiest time being implemented as an archive.

(1) The internal data structure conforms entirely to the `NitroROM` format

In this case the implementation is easiest because all of the commands can be processed with the default procedure by using only the access callback. As long as no special device will be handled by the archive, there is no need for any user procedures.

If the format does not fully conform to the `NitroROM` format or is an entirely different format, you will need to appropriately replace the FNT and FAT-related low-order commands and access callback.

(2) The directory structure and the file information are fixed

In this case, you can implement a standard archive that can be used without a problem at least on the user side. However, the characteristics are such that the File/Directory Interface are not suitable for incorporating an environment where directories are dynamically changed and the information in files is freely altered. As a result, for targets like this there are a number of limitations on commands and some commands may not even be supported.

(3) Generally speaking, the concept of the directories and files conform to that of the File System

If the target does not even meet this third set of conditions, there is very little merit to using the File System except in the case of very special applications. One example would be for a network, where the communications socket and URI path were generally in agreement with a number of individual commands of the File/Directory Interface.

3.4.2   Default Procedure

The default procedure is a set of standard processes for each command available in the File System. Of these commands, the low-order ones make use of access callbacks as well as FNT and FAT, or implement processes that depend on nothing at all, and there are also some high-order commands that make use internally of other low-order commands.

The figure below shows the dependency relationships of the various commands that make up the default procedure for basic archive processing. The upper level of this dependency relationship should be taken into consideration for the implementation of access callbacks and user procedures. Read the function reference to learn about the strict specifications required of each command and how they are actually supported in the SDK.
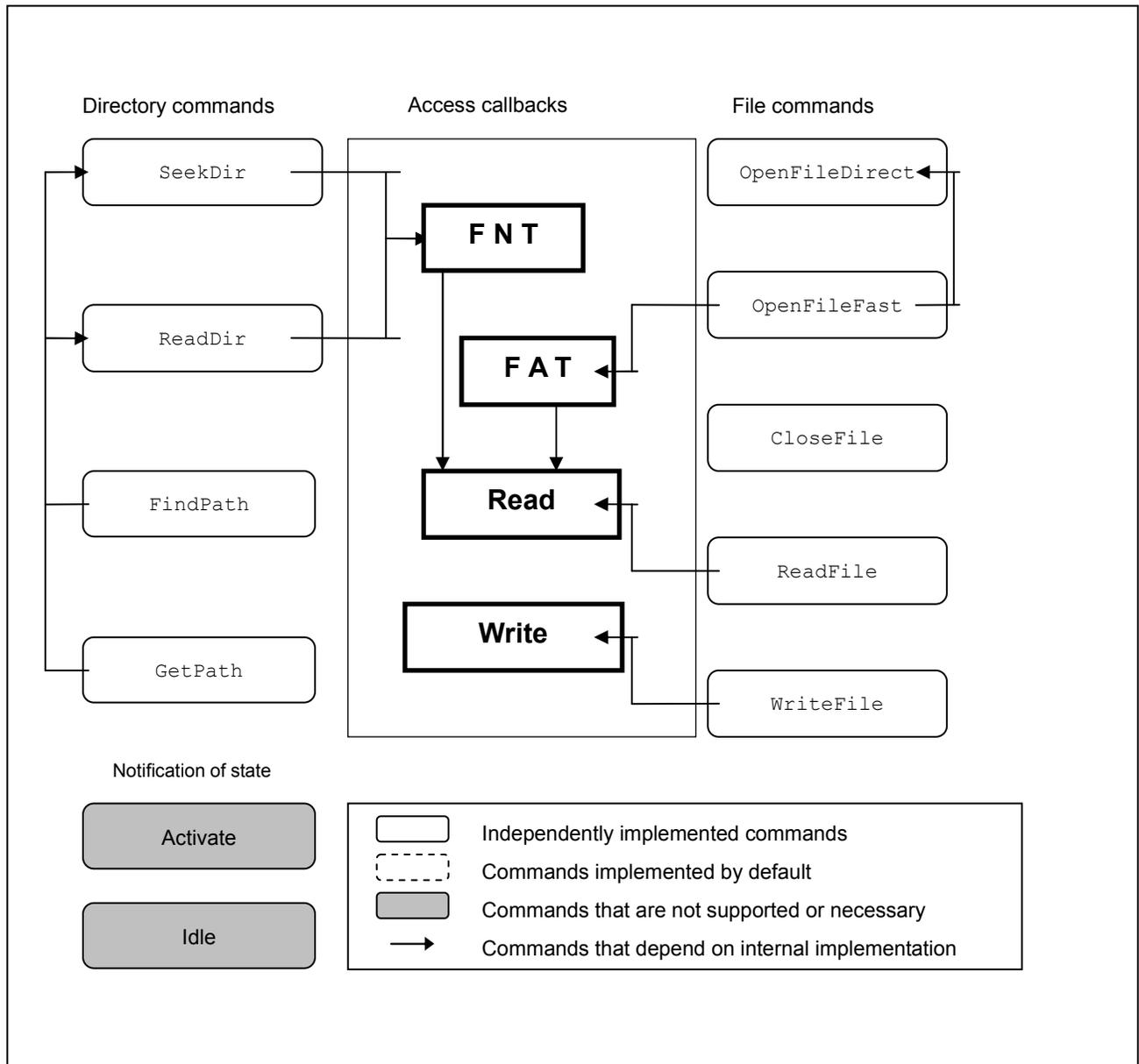
Figure 3-4   Default Procedure

3.4.3   Implementing Archives

This section explains the implementation of several types of archives by showing the difference from the default procedure.

3.4.3.1   ROM Archive

After the File System is initialized, the standard practice is to load the *rom* archive, which is the system definition archive. The *rom* archive is for accessing the file group stored in the ROM region that was created in NITRO-CARD by the `makerom` tool, and also for processing some of the Overlay operations.

The figure shows in broad terms the processes that get replaced internally in the case of the *rom* archive.

Because the medium is ROM, the process of writing to files is made explicitly not to be supported. All other processes are left to the defaults. State notifications are used to lock and unlock the CARD bus.

The actual code for this implementation is presented in the SDK sample demo `/build/demos/fs/arc-1`.
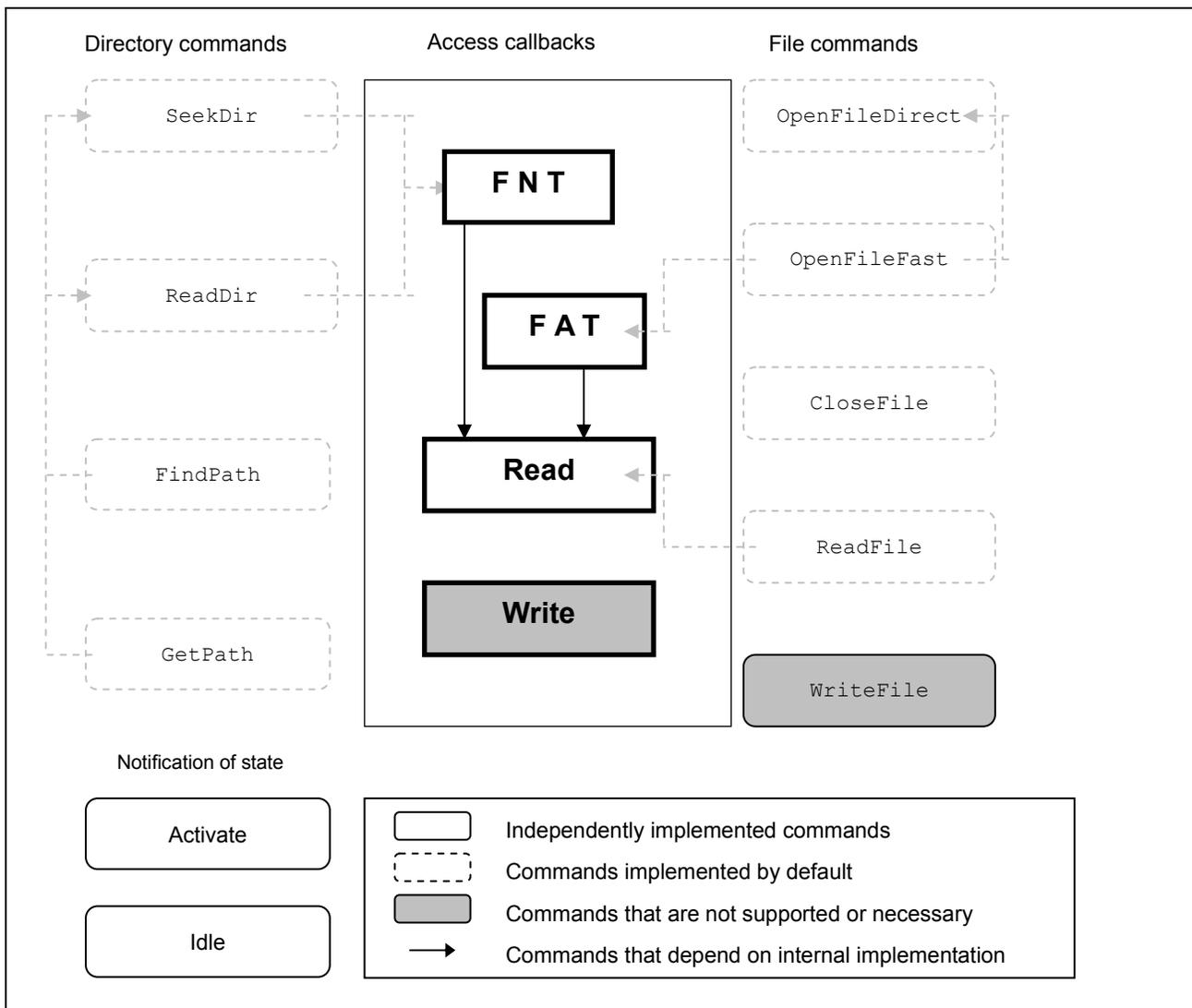
Figure 3-5   ROM Archive Procedure

3.4.3.2   Archive in Your Own Format in Memory

The following figure is an example of an archive that has been implemented by defining a format of your own that is different from the `NitroROM` format and then placing a directory structure that conforms to that format in memory.

Since the format differs from the `NitroROM` format, neither FNT nor FAT is specified. Instead, user procedures are used to replace these with four commands that are dependent on FNT and FAT. Because the substituted commands operate according to the correct specifications, higher-order commands can use the default procedure. Access callbacks are used only for file reading and writing.

The actual code for this implementation is presented in the SDK sample demo `/build/demos/fs/arc-2`.
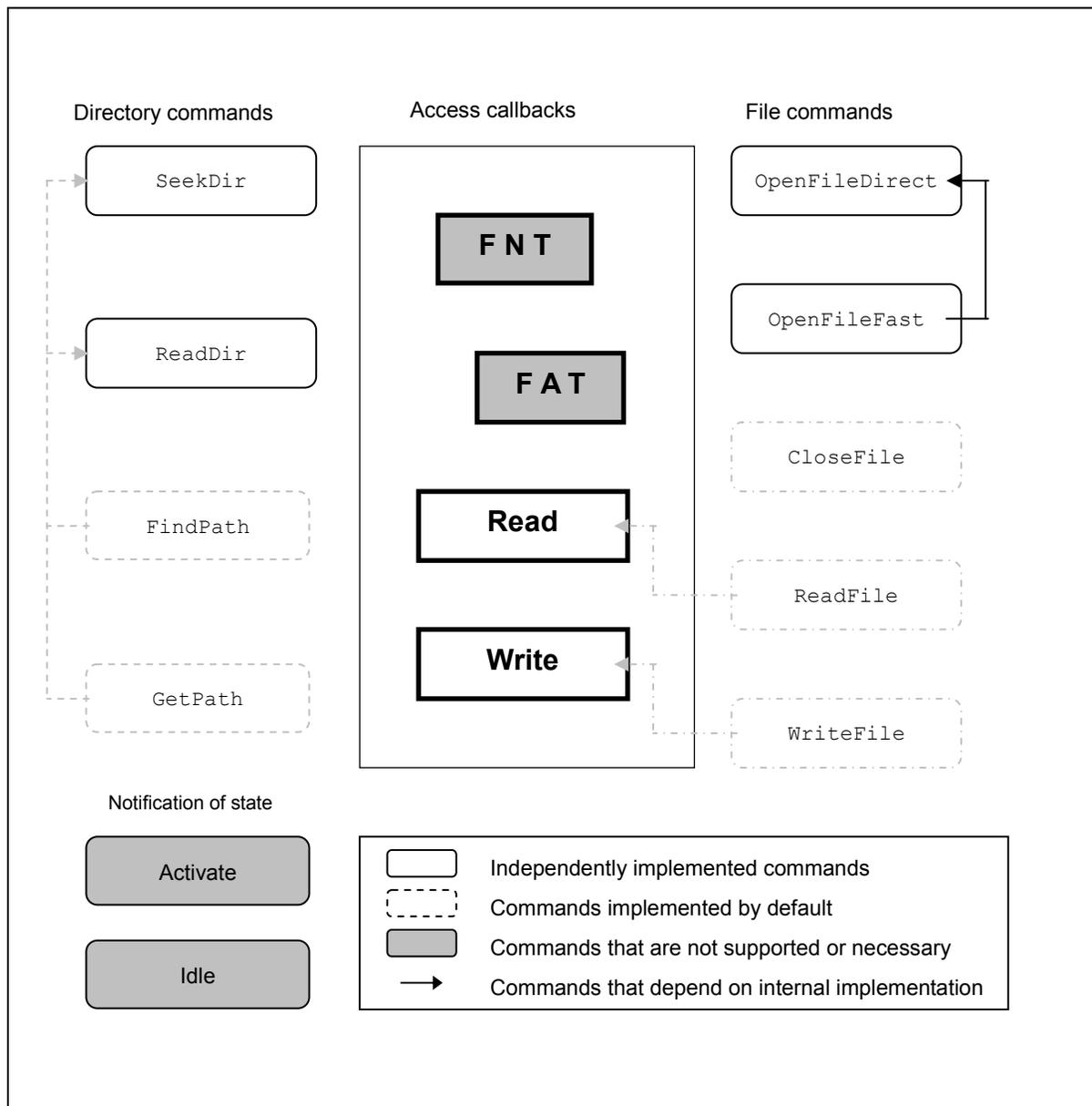
Figure 3-6   Procedure for Archive in Your Own Format in Memory

3.4.3.3   Archive via Wireless Communication

The figure below is an example of an archive that has been implemented so that a child program booted from a wireless download can use wireless communications to reference the directory information in the parent's NITRO-CARD and obtain dynamic data.

All of FNT and FAT is received and stored in memory before getting data, and all commands other than file access are left to the default procedure.

Reading of the file realizes asynchronous processes up to the time reception ends on a request via the communications protocol. This example does not support file writing, but there are applications where data might be written to a file.

The actual code for this implementation is presented in the SDK sample demo `/build/wireless_shared/wfs`.
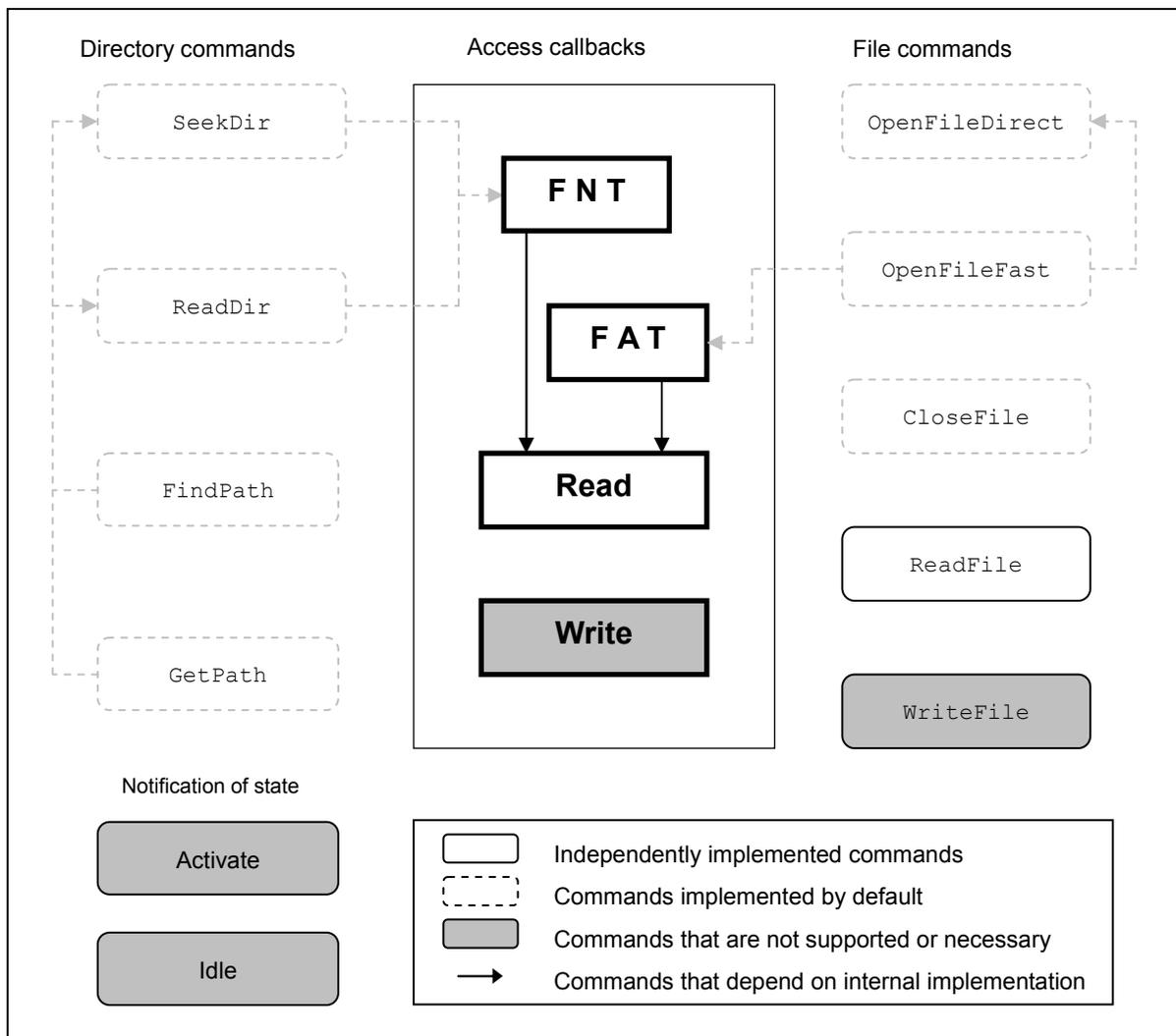
**Figure 3-7   Archive Procedure via Wireless Communication**

3.4.3.4　　Other Archives

Aside from the sample demos, the NITRO-SDK does not directly provide any way to implement archives. You will need to prepare the program code yourself so your application can access archives in the `NitroROM` format or in some other format created by a tool that packages data in archive form.

NITRO-System also has the Foundation library (FND), which has been released to be used for archives. You can build archives in standard memory by using these archive functions in combination with the included tool `/NitroSystem/tools/win/bin/nnsarc.exe`. To read more about this, see the NITRO-System manuals and sample demos.

3.5　　Explanation of the API

The previous section looked at the overall Archive System and explained archive operations. This section explains the procedures for using the File System Library's interface functions (the API) to actually manipulate archives from the application.

3.5.1　Manipulating the State

To manipulate archives, the `FSArchive` structure object is used when calling functions. FSArchive object holds various callbacks and parameters inside.

Here we explain how to change the archive's internal state. (See 3.3.1 Archive state transitions to read about archive internal states.)

3.5.1.1　　Initializing the FSArchive Object

The user does not need to directly manipulate the various internal members of the `FSArchive` object, but the internal state of the object must be initialized using the `FS_InitArchive()` function before the object is used. An initialized archive enters the *unregistered* state.

```
/* Must initialize FSArchive object before using first time */
FSArchive  arc;
FS_InitArchive( &arc );
```

Code 12 Initializing FSArchive Object

3.5.1.2   Registering and Releasing the Archive Name

The archive name must be registered before it can be loaded in the File System. The user of the archive is free to use any name, but it must be a name that is unique inside the File System. Once the archive name is registered, the archive becomes managed by the File System and moves into the *registered* state.

```
/* Register the archive with a specified name */
const char *name = "ac1";
const int name_len = strlen( name );
const BOOL ret = FS_RegisterArchiveName(
&arc, name, name_len );
/* Registration will fail for only one of three reasons:
The archive is not in the unregistered state, the name is
too long, or the same name has already been registered */
SDK_ASSERT( ret );
```

Code 13 Registering Archive Name

Once the archive has been unloaded from the File System, the name can be unregistered if the archive will not be used again. The `FSArchive` object is still being managed by the File System if it is in the *registered* state, so the object must not be arbitrarily destroyed until the name has been released and the object has moved to the *unregistered* state.

```
/* Release the archive name */
FS_ReleaseArchiveName( &arc );
```

Code 14 Releasing Archive Name

3.5.1.3   Loading and Unloading Archives

Once the `FSArchive` object has had its name registered it can be loaded to the File System. As mentioned in 3.2.1 Unique address space and offsets, the archive must provide access callbacks and FNT and FAT information for the File System. Specify them with the `FS_LoadArchive()` function when loading the object. If the call is successful, the archive moves to the *loaded* state.

```
/* Load archive */
const BOOL ret = FS_LoadArchive(
    &arc,                   /* Archive object */
    base_offset,            /* Base offset (for user) */
    fat_offset, fat_length, /* FAT information */
    fnt_offset, fnt_length, /* FNT information */
    ArcReadCallback,        /* Read callback */
    ArcWriteCallback        /* Write callback */
    );
/* Load will fail if archive is not in registered state */
SDK_ASSERT( ret );
```

Code 15 Loading Archive

A loaded archive can be unloaded from the File System at any time. If the archive is in the *busy* state when the unload request is made, the request will be blocked until the command that is processing is completed. When completed the archive will move to the *registered* state.

```
                          /* Unload archive */
                          const BOOL ret = FS_UnloadArchive( &arc );
                          /* Unload will fail if archive is not in loaded state */
                          SDK_ASSERT( ret );
```

Code 16 Unloading Archive

3.5.1.4   Suspending and Resuming Archives

Once the archive has been initialized, archive operations can be suspended and resumed at any time, regardless of the state of the archive.

If you want to start the archive in the suspended state, you must suspend the archive before you load it.

Resumption of processing from the suspended state is achieved immediately. However, if the archive is operating, the process to suspend operations will be blocked until the processing of the current command is completed.

```
        /* Suspend archive */
        const BOOL bak_mode = FS_SuspendArchive( &arc );
        /* Execute processes that must run while archive is not suspended */
        ...
        /* If necessary, return to the previous operating state */
        if( bak_mode )
        {
            (void)FS_ResumeArchive( &arc );
        }
```

Code 17 Suspending and Resuming Archive

3.5.2   User Procedures

If you cannot appropriately process all commands with just the access callbacks and the default procedure, then you will need to set the archive in the *registered* state and call the `FS_SetArchiveProc()` function to configure a user procedure.

```
/* Configure the user procedure */
FS_SetArchiveProc( &arc,          /* Archive object */
  ArcProc,                        /* User procedure */
  FS_ARCHIVE_PROC_WRITEFILE  /* Query command */
  );
```

Code 18 Configuring the User Procedure

The configured user procedure is called in a callback from the File System as needed. Here independent processes are conducted and the appropriate values must be returned.

```
/* Description of user procedure */
FSResult  ArcProc( FSFile *p_file, FSCommandType cmd )
{
   /* Only the commands requested at the time of
configuration can be queried */
   SDK_ASSERT( cmd == FS_COMMAND_WRITEFILE );
   (void)p_file;
   switch(cmd) {
   /* Certain commands can be unsupported */
   case FS_COMMAND_WRITEFILE:
      return  FS_RESULT_UNSUPPORTED;
   /* Can enable all queries and evaluate inside the user
      procedure */
   default:
      return  FS_RESULT_PROC_UNKNOWN;
   }
```

Code 19 Describing the User Procedure

3.5.3   Asynchronous Processes

In order for the archive to support asynchronous processes, the process must be implemented in various callbacks. Specifically, at places where a result value is requested, the callback returns an "asynchronous processing" and later sends notice after the pertinent process has ended.

Change the access callbacks if all types of access will always be done with asynchronous processes.

```
/* Read callback */
FSResult  ArcReadCallback(
   FSArchive *p_arc, void *dst, u32 src, u32 len )
{
   /* Execute process that can be expected to be
asynchronous  */
   CARD_ReadRomAsync(
       dma_no, (const void*)src, dst, len,
       OnCardReadDone, p_arc );
   /* Return "asynchronous processing" as result. Even if
notification of completion is generated sooner than this
return, the system can guarantee correct processing */
   return FS_RESULT_PROC_ASYNC;
}

/* Callback at completion of asynchronous process */
void OnCardReadDone( void *p_arc )
{
   /* Notify archive of completion */
   FS_NotifyArchiveAsyncEnd(
       (FSArchive*)p_arc, FS_RESULT_SUCCESS );
}
```

Code 20 Desynchronizing Access Callback

Change the user procedure if only certain commands will be done with asynchronous processes.

```
/* Description of user procedure */
FSResult  ArcProc( FSFile *p_file, FSCommandType cmd )
{
   switch(cmd) {
   case FS_COMMAND_READFILE:
      /* Only certain commands return
         "asynchronous processing" */
      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;
   }
   }
```

Code 21 Desynchronizing User Procedure

4   Overlay Interface

Overlay is a feature that helps NITRO applications improve efficiency by placing only necessary execution code in the limited amount of main memory that is available. This chapter explains the operating principles of the overlay feature and describes the Overlay Interface.

4.1   Starting Segment and Overlay Segments

All of the program code for a NITRO application is grouped in a unit called a "segment." The segment consists of executable code, a variable region, a constant region, a destination address and a routine for its own initialization.

Destination address

| Execution code | Constants region | Variables region | Initialization routine |
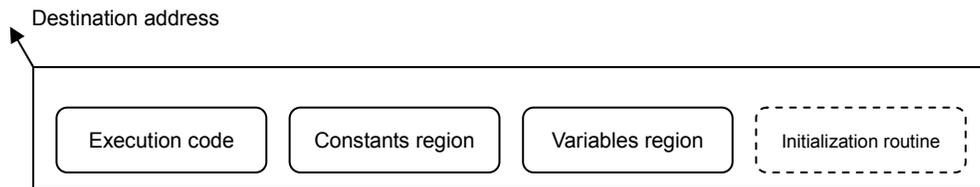
Figure 4-1   Segment Composition

In a normal application, all of this code is loaded into main memory when the application has been started, and after the initialization routine executes, control is passed to the main entry point (the `NitroMain` function). While the program is executing, these regions are stored statically and are collectively called the "static segment." One copy of the static segment is prepared for the ARM9 and one for the ARM7.

For a large application, the static segment may take up a large region of main memory, and in the worst-case scenario it may be larger than the size of main memory. One good way to avoid this kind of problem is to not make the entire program resident, but instead to load modules that are only used for specific scenes or combination as needed into main memory. The "overlay" feature is what is used to perform this process, and the divided up modules are called "overlay segments."

02000000

ARM9 static segment (Resident; most of the application)

ARM7 static segment (Resident; usually all fixed)

| ARM9 Overlay segment (Special code for stage 1) | ARM9 Overlay segment (Special code for stage 2) | … |
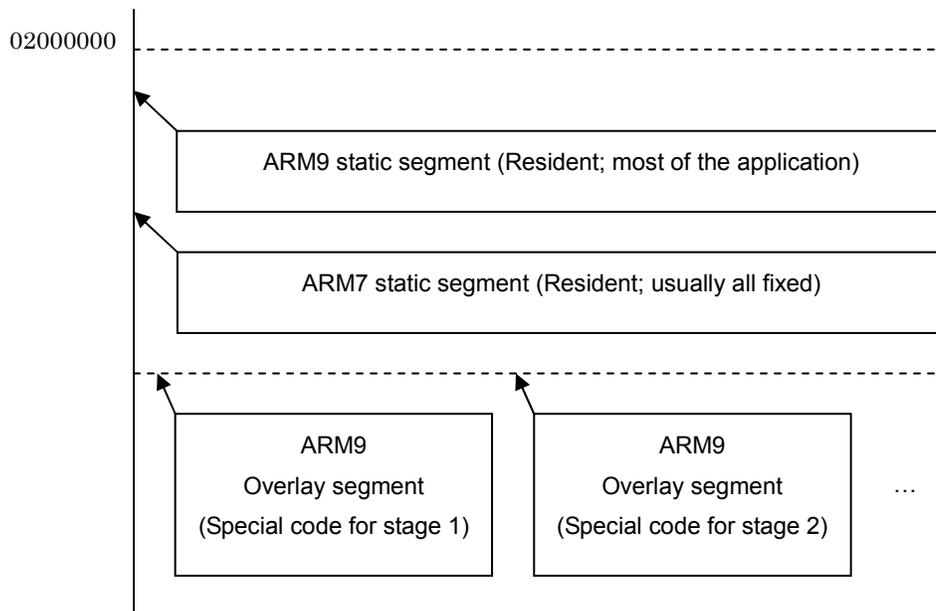
Figure 4-2   Static Segment and Overlay Segments

4.2   Characteristics of Overlays

It is important to consider the many characteristics that differ when overlays are used. The following sections look at several representative differences.

4.2.1   Idiosyncratic Life Management

Overlays can be loaded dynamically with arbitrary execution timing and released with arbitrary execution timing. This timing controls the life of objects with static storage periods inside overlay segments. Specifically, global objects exist from the time an overlay segment is loaded and the initialization routine is executed, and they are disassembled when the overlay segment is released. In the C++ language, this is where the destructor is executed.

For a given overlay, this action occurs every time the process of loading and deleting the overlay is repeated. The internal state of the overlay segment is independent inside each lifetime, and the lifetime is not extended nor deferred.

This action is not unique to overlays; it is a behavior common to segments. However, overlays differ in that they primarily get deleted when the NitroMain function ends, which for static segments is normally impossible.
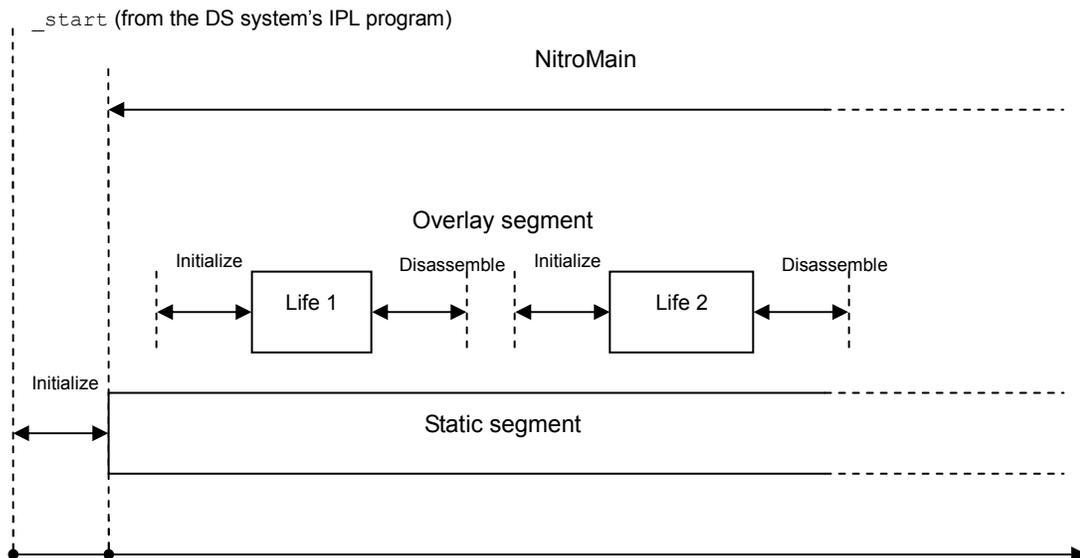

Figure 4-3   Life of an Overlay Segment

4.2.2   Competing for Position

The overlay feature is only for dynamic loading of segments and not for dynamic linking. The address where the overlay segment will be placed and the symbol references among other segments are all resolved statically.

If numerous overlay segments compete for regions for placing them in limited memory space, those overlay segments cannot be used at once. In that case you will need to examine it in your application.

Overlay					
	Simultaneous use				
Load side	1	2	3	4	5
1		○	×	○	○
2	○		×	○	○
3	×	×		×	○
4	○	○	×		×
5	○	○	○	×	

The figure below shows an example of competition among overlay segments.
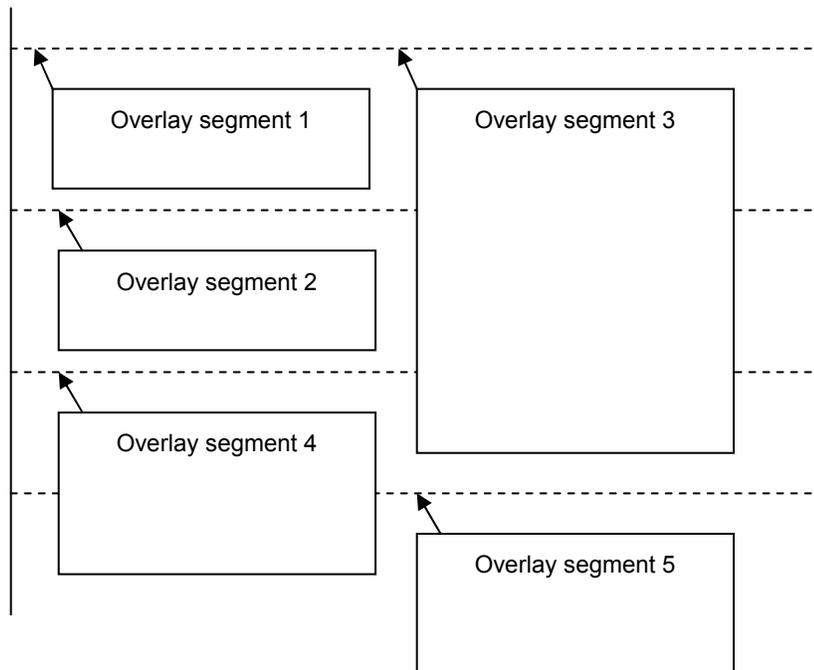
Figure 4-4   Competition Among Overlay Segments

4.3   Explanation of the API

The previous section explained the actions of overlay segments. This section explains how the application actually manipulates overlays using the API functions.

4.3.1   Specifying in the .lsf File

Overlay segments are identified from the program code by their names.

Overlay names, support for inclusion modules and specification of position placements are all described via a .lsf file. For information on the notation of .lsf files, see the reference for the makelcf tool.

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

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

Code 22 Specifying Overlay Segment with a `.lsf` File

4.3.2   Overlay ID Declaration and Definition

The overlay segment is specified and manipulated from the program in the form of an "overlay ID."

This overlay ID entity is resolved when the program is linked. To use this overlay ID from the program code requires its explicit declaration using the FS_EXTERN_OVERLAY macro.

The FS_OVERLAY_ID macro is then used to reference this declared overlay ID.

```
/* Declare the overlay ID to be used */
FS_EXTERN_OVERLAY(main_overlay_1);
/* A reference to the declared overlay ID can be defined */
FSOverlayID      ovl_id = FS_OVERLAY_ID(main_overlay_1);
```

Code 23 Overlay ID Declaration and Definition

4.3.3   Loading and Unloading Overlays

An overlay can be read (loaded) at any time while the program is executing. However, as mentioned in 4.2.2 Competing for position, if other overlay segments exist that are competing for the same region, the overlay cannot be loaded if one of these others is being read.

Further, an overlay that has already been loaded cannot be reloaded without first being released. The library cannot internally determine the correctness, so the application needs to guarantee the situation.

The following is the simplest procedure for loading an overlay:

```
/* Load overlay segment.
  Overlay can be used once control returns from function */
BOOL ret = FS_LoadOverlay(
   MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1) };
 /* Fail if for some reason the specified overlay does
   not exist. This will not arise on a normal program
  generated with makerom. */
SDK_ASSERT( ret );
```

Code 24 Loading an Overlay

The procedure for releasing (unloading) a loaded overlay is show below. An overlay that is not loaded cannot be unloaded, so the application needs to guarantee that the situation is correct, as mentioned above for loading.

```
/* Unload overlay segment.
   When function called, overlay cannot be used. */
BOOL ret = FS_UnloadOverlay(
   MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1) };
/* Fail if for some reason the specified overlay does
  not exist. This will not arise on a normal program
  generated with makerom. */
SDK_ASSERT( ret );
```

Code 25 Unloading an Overlay

4.3.4   Dividing the Load Process

The FS_LoadOverlay function executes the following set of processes internally in a batch:

    (1)   Gets detailed information about the overlay segment from the overlay ID.

    (2)   Loads the segment data into the placement position, based on the overlay segment's detailed information.

    (3)   Executes the overlay segment's initialization routine and enables the overlay.

If these tasks need to be divided out and executed in a stepwise fashion, the single-feature functions that perform each task can be assembled together and called in sequence. This may prove necessary in order to avoid problems related to the processing time involved in the data-reading task of step 2.

There are several situations in which dividing the process is beneficial. One is when advancing the game while dealing with a huge overlay that requires more time than one picture frame. Another is when getting segment data based on a configuration like that in 3.4.3.3 Archive for wireless access. A third is the future implementation of applications that use a low-speed CARD-ROM device.

The procedure for a divided-up load process looks like this:

```
/* (1) Get overlay information from overlay ID */
FSOverlayInfo  info;
if(FS_LoadOverlayInfo(&info,
   MI_PROCESSOR_ARM9, FS_OVERLAY_ID(main_overlay_1)))
{
    /* (2) Load data based on overlay information */
    FSFile  file;
    FS_InitFile(&file);
    (void)FS_LoadOverlayImageAsync(&info, &file);
    (void)FS_WaitAsync(&file);
    (void)FS_CloseFile(&file);
    /* (3) Execute the initialization routine */
    FS_StartOverlay(&info);
}
```

Code 26 Dividing up the Load Process