

TWL-System Archive Format Manual

Explanation of Archive Format

2008/05/30

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. 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.

Table of Contents

1	Introduction	6
2	Basic Structure of an Archive	7
3	Archive Header	8
3.1	Structure of the Archive Header	8
3.1.1	signature	8
3.1.2	byteOrder	8
3.1.3	version	8
3.1.4	fileSize	8
3.1.5	headerSize	9
3.1.6	dataBlocks	9
4	File Allocation Table Block	10
4.1	Structure of the File Allocation Table Block	10
4.1.1	kind	10
4.1.2	size	10
4.1.3	numFiles	10
4.1.4	allocationTable	10
5	Filename Table Block	12
5.1	Structure of the Filename Table Block	12
5.1.1	kind	12
5.1.2	size	12
5.1.3	Directory Table	12
5.1.4	Entry Name Table	13
5.1.5	Padding for Alignment	13
5.2	Example of a Filename Table	14
6	File Image Block	15
6.1	kind	15
6.1.1	size	15
6.1.2	fileImage	15

Tables

Table 3-1 Archive Header Structure	8
Table 4-1 File Allocation Table Block	10
Table 4-2 File Allocation Entries	11
Table 5-1 Filename Table Block	12
Table 5-2 Directory Table Entries	12
Table 5-3 File Entry Structure	13
Table 5-4 Directory Entry Structure	13
Table 6-1 File Image Block	15

Figures

Figure 2-1 Archive Structure	7
Figure 5-1 Filename Table	14

Revision History

Revision Date	Description
2008/05/30	Made revisions in line with the NITRO-System name change (from NITRO-System to TWL-System).
2008/04/08	Changed the format of the Revision History. Added support for the TWL-SDK.
2007/11/26	Added an explanation for when the file alignment is specified.
2005/01/05	Changed an instance of "NITRO" to "Nintendo DS."
2004/06/10	Initial version.

1 Introduction

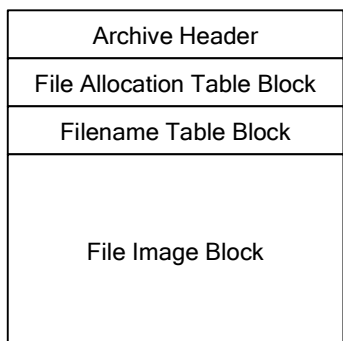
An archive is a file comprising a collection of files. An archive file can also contain hierarchical directory information, allowing you to access the individual files in the archive by specifying a file ID (an index value) or a path name.

2 Basic Structure of an Archive

Archives conform to TWL-System's binary file rules and have the structure shown in Figure 2-1. At the beginning of the archive file is the archive header, followed by data blocks for the file allocation table, the filename table, and the file image, in that order.

The file allocation table and filename table are similar in basic structure to that of the TWL-SDK's ROM file system.

Figure 2-1 Archive Structure



3 Archive Header

The archive header is always located at the start of the archive file. This header contains information pertaining to the overall archive file.

3.1 Structure of the Archive Header

The archive header has the structure shown in Table 3-1, below.

Table 3-1 Archive Header Structure

Type	Parameter Name	Description	Size
char[4]	signature	File signature. (N, A, R, C)	4 bytes
u16	byteOrder	Byte order marks (0xfeff)	2 bytes
u16	version	Archive format version number (0x0100)	2 bytes
u32	fileSize	Size of the archive file	4 bytes
u16	headerSize	Size of the archive header (16)	2 bytes
u16	dataBlocks	Number of data blocks (3)	2 bytes

3.1.1 signature

The `signature` parameter stores the file signature, which determines the binary file type. This file signature stores the four characters N, A, R, and C (always in that order), regardless of the endian method.

3.1.2 byteOrder

The `byteOrder` parameter stores the byte order marks (Zero-Width No-Break Space, 0xfeff), which are used to determine the endian method. The Nintendo DS and TWL both use the little-endian method, so `byteOrder` stores the byte order marks in the order 0xff, 0xfe. The TWL System archiver `nnsarc.exe` always creates archives using the little-endian method.

3.1.3 version

The `version` parameter stores the archive format's version number. The upper byte stores the major version (an integer value), and the lower byte stores the minor version (a decimal value). The current version is 1.0, so `version` stores the value 0x0100.

3.1.4 fileSize

The `fileSize` parameter stores a value that indicates the overall size of the archive. This value includes the size of the archive header.

3.1.5 headerSize

The `headerSize` parameter stores the value 16, which is the size of the archive header. In future versions, the size of the archive header might change, so do not assume the size of the header is 16.

3.1.6 dataBlocks

The `dataBlocks` parameter stores the number of data blocks contained in the archive. In the current version, there are always three blocks, so `dataBlocks` holds the value 3. However, new data blocks might be added in the future.

4 File Allocation Table Block

The File Allocation Table block stores information indicating the location of the contents of each file in the archive.

4.1 Structure of the File Allocation Table Block

The File Allocation Table block has the structure shown in Table 4-1, below. Each entry in this table is allocated a number called a file ID. The numbers allocated as file IDs begin at `0x0000` and increment to a maximum value of `0xffff`. The allocation table array is equal in size to the number of files, and the array subscripts are the same as these file IDs.

Table 4-1 File Allocation Table Block

Type	Parameter Name	Description	Size
u32	<code>kind</code>	Type of data block (FATB)	4 bytes
u32	<code>size</code>	Size of the data block	4 bytes
u16	<code>numFiles</code>	Number of files	2 bytes
u16	<code>reserved</code>	Reserved	2 bytes
	<code>allocationTable</code>	File allocation table (8 bytes per entry)	$8 \times n$ bytes

4.1.1 `kind`

The `kind` parameter stores a 4-byte code that defines the data block type. These 4 bytes store the code `FATB`. Because the archive uses the little-endian method, the characters are stored in reverse order.

4.1.2 `size`

The `size` parameter stores the size of the data block stored in the File Allocation Table.

4.1.3 `numFiles`

The `numFiles` parameter stores the number of files stored in the File Allocation Table. This value represents the number of files stored in the archive.

4.1.4 `allocationTable`

The `allocationTable` parameter is an array of file allocation entries whose size increases by 8 bytes for every file entry. A number called a file ID is allocated to every entry. The file IDs begin at `0x0000` and increment to a maximum of `0xffff`. The array is equal in size to the number of files, and the array subscripts are the same as these file IDs.

Table 4-2 File Allocation Entries

Type	Parameter Name	Description	Size
u32	fileTop	Offset from start of file.	4 bytes
u32	fileBottom	Offset +1 from end of file.	4 bytes

The offsets that indicate the positions at the start and the end of the file store values that assume 0 to be the location of the File Image block's `fileImage` parameter (eighth byte from the start of the File Image block).

To calculate the size of the file, use the following formula.

```
u32 fileSize = fileBottom - fileTop;
```

5 Filename Table Block

The Filename Table block stores information for use in getting file IDs from path names. It is composed of a Directory Table and an Entry Name Table, and it supports hierarchical directories.

5.1 Structure of the Filename Table Block

The Filename Table block has the structure shown in Table 5-1, below.

Table 5-1 Filename Table Block

Type	Parameter Name	Description	Size
u32	kind	Type of data block (FNTB)	4 bytes
u32	size	Size of the data block	4 bytes
	directoryTable	Directory Table	<i>n</i> bytes
	entryNameTable	Entry Name Table	<i>m</i> bytes
	padding	Padding for alignment	

5.1.1 kind

The `kind` parameter stores a 4-byte code that defines the data block type. These 4 bytes store the code FNTB. Because the archive uses the little-endian method, the letters are stored in reverse order.

5.1.2 size

The `size` parameter stores the size of the data block stored in the Filename Table.

5.1.3 Directory Table

The Directory Table is an array of data structures shown in Table 5-2. **Error! Reference source not found.** A number called a directory ID is allocated to each entry. These directory ID numbers increment in the order the entries are stored. Directory IDs take numbers from 0xf000 to 0xffff, so they can be distinguished from file IDs. Because of this specification, an archive can store up to 61,440 files and up to 4096 directories.

Table 5-2 Directory Table Entries

Type	Parameter Name	Description	Size
u32	dirEntryStart	Entry-name search location.	4 bytes
u16	dirEntryFileID	File ID of the entry at the start of the directory.	2 bytes
u16	dirParentID	ID of the parent directory (in the special case of the root directory, the number of directory entries)	2 bytes

The `dirEntryStart` parameter indicates the first entry (which could be either a file or a directory) in the directory. It stores an offset value that assumes 0 to be the start of the Directory Table (8th byte from the start of the Filename Table block).

The number of elements in the directory entry array is the same as the number of directories, and the array subscripts are equal to the directory IDs minus `0xf000`. The directory with the directory ID `0xf000` is the root directory. In the special case of the root directory, the `dirParentID` member stores a value that represents the total number of directory entries.

5.1.4 Entry Name Table

The Entry Name Table is an aggregate of two kinds of variable-length data. The structure differs depending on whether the entry is a file or a directory.

Table 5-3 File Entry Structure

Type	Parameter Name	Description	Size
u8	<code>entryNameLength</code>	Length of filename (upper 1 bit indicates entry type)	1 byte
char	<code>entryName[n]</code>	Filename ($n = \text{entryNameLength}$)	n bytes

Table 5-4 Directory Entry Structure

Type	Parameter Name	Description	Size
u8	<code>entryNameLength</code>	Length of filename (upper 1 bit indicates entry type)	1 byte
char	<code>entryName[n]</code>	Filename ($n = \text{entryNameLength}$)	n bytes
u16	<code>directoryID</code>	Directory ID	2 bytes

Entries in the same directory are positioned in a contiguous region and allocated consecutive file IDs. A file entry with an entry name length of 0 (`\0`) is placed after the last entry in the directory.

The entry name length is indicated by the lower 7 bits in `entryNameLength`, so the entry name can be a maximum of 127 characters (calculated on the basis that each character is 1 byte).

The highest-order bit of `entryNameLength` indicates the entry type. When the highest-order bit is 0, the entry is a file entry. When the highest-order bit is 1, it is a directory entry.

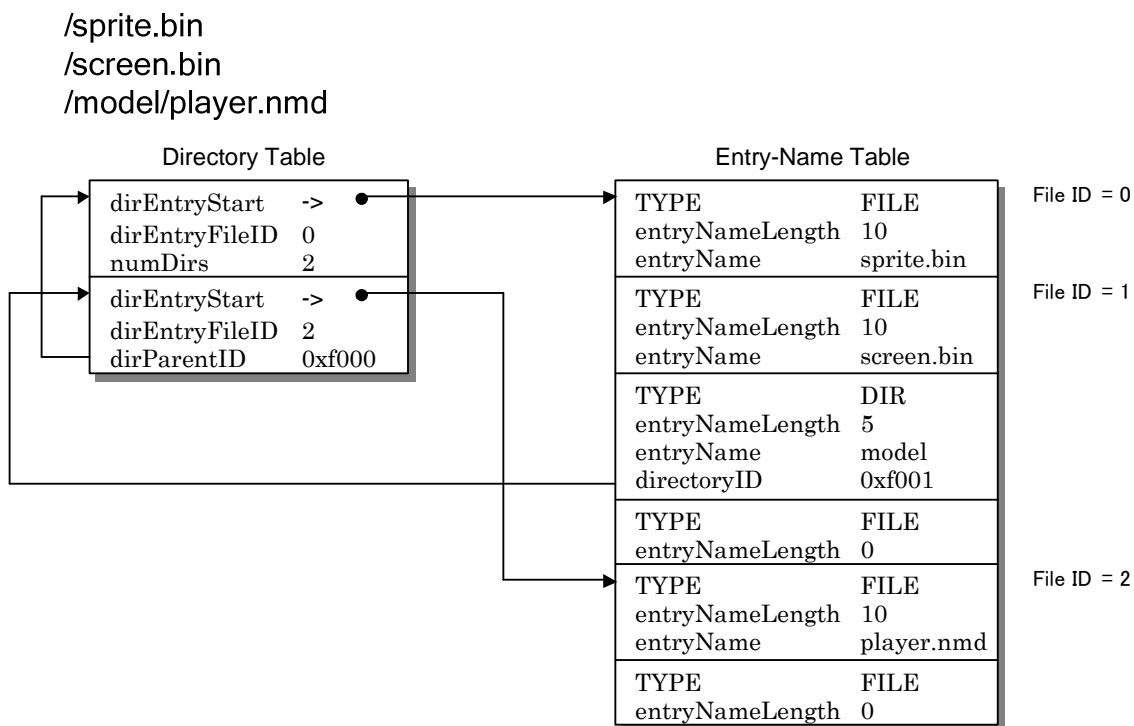
5.1.5 Padding for Alignment

At the end of each file name table block, padding is added so that the file stored at the start of the subsequent file image block is aligned to the specified boundary. The value `0xff` fills the padding.

5.2 Example of a Filename Table

Figure 5-1 depicts a Filename Table storing three files.

Figure 5-1 Filename Table



6 File Image Block

The File Image block stores images of the archived files. The starting location and ending location of each file image are indicated by the entries in the File Allocation Table.

Table 6-1 File Image Block

Type	Parameter Name	Description	Size
u32	kind	Type of data block (FIMG)	4 bytes
u32	size	Size of the data block	4 bytes
	fileImage	File images	<i>n</i> bytes

6.1 kind

The `kind` parameter stores a 4-byte code that defines the data block type. These 4 bytes store the code `FIMG`. Because the archive uses the little-endian method, the code is stored in reverse order.

6.1.1 size

The `size` parameter stores the size of the data block stored in the File Image.

6.1.2 fileImage

The `fileImage` parameter stores images of all the files in the archive file in a packed form. The stored file image is aligned to a 4-, 8-, 16-, or 32-byte boundary. To make the start of each file match an alignment boundary, padding values of `0xff` may be inserted between the files.

All company and product names in this document are the trademarks or registered trademarks of the respective companies.

© 2004-2009 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.