Revolution SDK

Optical Disc Drive Library (DVD)

Version 1.01

The content of this document is highly confidential and should be handled accordingly.

RVL-06-0041-001-C
Released: March 27, 2009

Contents

Code Examples

Equations

Figures

Tables

Revision History

Version	Date Revised	Item	Description	Revised By
1.01	2009/02/17		Transferred content from the Optical Disc Guidelines and re-edited them.	
		2.1	Updated the total capacity of the optical disc to be the total capacity that the developer can use.	
		3	Added Chapter 3, Discs	
		5.3.1 5.4	Revised the text.	
		6.4	Added information regarding the disc check when the application initiates a restart.	
		7.1 7.2	Added an explanation about FST size restrictions.	
		9	Added “Multiple Disc Games (Reference Information).”	
1.00	2006/03/01	-	First release by Nintendo of America Inc.	-

1 Overview

The Wii console from Nintendo uses an optical disc drive for game media. Since this device is based on DVD technology, function names in the optical disc drive library have “DVD” as a prefix.

This document covers the following topics:

- "2 Main Features" on page 7 introduces you to the main features of the Wii optical disc drive. If you are not a programmer, you can get basic information on the optical disc drive by reading this chapter. If you are a programmer, this chapter is a good starting point for you.
- "3 Discs" on page 8 describes the disc structure and the unique ID for each disc.
- "4 Simple Demo" on page 10 is a tour of the Wii optical disc drive through the use of a simple code demo.
- "5 How to Use APIs" on page 12 describes the APIs and how to use them.
- "6 Error Handling" on page 18 describes APIs for error handling.
- "7 File System" on page 24 briefly describes the Wii file system.
- "8 Directory Access Support Functions" on page 26 describes how to use directory access support functions.
- "9 Multiple Disc Games (Reference Information)" on page 30 describes, as reference, games that use multiple discs.
- "10 Optical Disc Drive FAQs" on page 35 offers a FAQ list.

2 Main Features

Here are the main features of the Wii optical disc drive:

- Huge storage size
- Fast read speed
- Simple file system
- Easy error handling

2.1 Size

The total capacity available to a developer is 4,294,967,296 bytes.

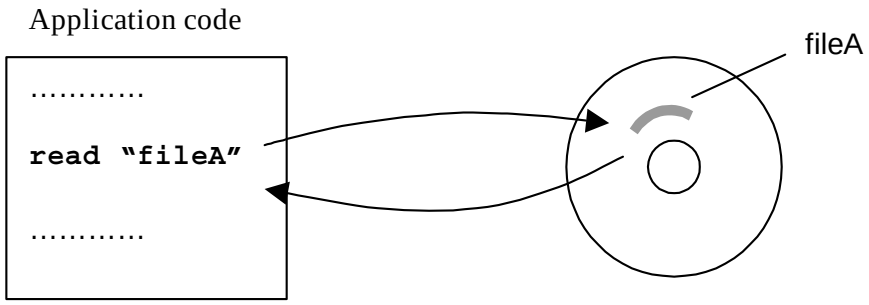
2.2 Read Speed

The read speed for a Wii disc is equivalent to the maximum DVD 6x. The Wii disc is always rotating uniformly, and data is recorded with the same density throughout the whole disc.

2.3 File System

The Wii disc can be accessed by symbolic string names, so you can read files by name. You don't have to specify the position of the file.

Figure 2–1 Applications Use Filenames to Retrieve Data From the Wii Disc



2.4 Error Handling

APIs automatically handle common user errors such as "no disc in drive," and so on, so you don't have to write complicated error-handling programs.

3 Discs

3.1 Disc Structure

The structural elements of a single optical disc can be classified into two categories.

- Program portion (`DOL` file)
- User file portion

The `DOL` file is the file converted from the `ELF` file during `ndrun` execution and is the game program. Because `ELF` files that can be used to build programs can have a slightly different format depending on the linker, the file is converted to a `DOL` file, which is a custom format similar to an `ELF` file, before being stored on the disc. `DOL` files can be loaded or run by the Wii menu.

The user file portion is visible as a file from the program portion. Any file application (data, relocatable module, etc.) is acceptable.

3.2 Disc ID

Each disc has a region to store a unique ID. The ID comprises the following four elements.

- Game Code
  A unique code assigned to each game. This is specified by Nintendo Business Division.
- Company Code
  A unique code assigned to each developer. This is also specified by the Nintendo Business Division.
- Disc Number
  This number is assigned for each disc for each game. The first disc is assigned `0`, and subsequent discs are assigned `1`, `2`, and so on.
- Game Version
  This is assigned for each version of a game. The version is assigned regardless of whether the game is released.

The Disc ID is used by the disc driver to determine whether the disc is appropriate.

4 Simple Demo

Here is some simple code to read a file named `testfile1.txt` from the optical disc into a prepared buffer. This demo is stored as `dvddemo/src/dvddemo1.c`. Please compile it and see how it works.

Code 4–1 Read File Demo

```
1:      // The name of the file we are going to read.
2:      char*   fileName = "testfile1.txt";
3:
4:
5:      void main(void)
6:      {
7:          DVDFileInfo fileInfo;
8:          u32         fileSize;
9:          u8*         buffer;         // pointer to the buffer
10:
11:         // Init the OS and heap
12:         MyOSInit();
13:
14:         DVDInit();
15:
16:         // We MUST open the file before accessing the file
17:         if (FALSE == DVDOpen(fileName, &fileInfo))
18:         {
19:             OSHalt("Cannot open file");
20:         }
21:
22:         // Get the size of the file
23:         fileSize = DVDGetLength(&fileInfo);
24:
25:         // Allocate a buffer to read the file.
26:         // NOTE: Pointers returned by OSAlloc are always 32byte aligned.
27:         buffer = (u8*)OSAlloc(OSRoundUp32B(fileSize));
28:
29:         // read the entire file here
30:         if (0 > DVDRead(&fileInfo, (void*)buffer, OSRoundUp32B(fileSize), 0))
31:         {
32:             OSHalt("Error occurred when reading file");
33:         }
34:
35:         // From here, we can use the file
36:         :
37:         // Close the file
38:         DVDClose(&fileInfo);
39:         :
40:         OSFree(buffer);
41:
42:         OSHalt("End of demo");
43:
44:         // NOT REACHED HERE
45:     }
```

The lines of code are referenced by number in the following explanations:

7: At this line, you declare a structure called `DVDFileInfo`. This structure holds the information for the target file, such as the start position and the length. The application does not have to fill the structure; `DVDOpen` handles this (see line 17).

12: Call a local function, `MyOSInit` (not shown here), to initialize the OS and heap. The heap will be required to use the `OSAlloc` function later. Refer to the *Revolution SDK Operating System* manual for an explanation of how to use the `OSAlloc` function.

14: You must call the `DVDInit` function before you issue any function related to the optical disc. This function handles initialization for optical disc access, such as setting the interrupt handler.

17: ***Open*** the file here. The `DVDOpen` function converts a filename into its file information. When debugging, we recommend checking the return value to know whether the open is successful.

23: The `DVDGetLength` function is a macro to get the size of the file, which is used so that this code knows how large a buffer to allocate for the file to be read.

27: Allocate a buffer to read the file.

Note: We have to round up the file size to a multiple of 32 due to drive hardware restrictions. The size of a read *must* be a multiple of 32.

30: This is the way to read files. With the `DVDRead` function, you can read a file synchronously, which means that the function returns when the read finishes. Since this function returns –1 if the read fails, you should check the return value. (For explanations of the arguments, as well as details on both synchronous and asynchronous reads, see "5.3 Read" on page 12.)

38: If you no longer need to access the file, you should ***close*** it.

Notes:

- There is no need to invalidate the data cache; the `DVDRead` function does that by default.
- In your real games, you should include code for error checking. This is simple to do; see "6 Error Handling" on page 18 for more details.

5 How to Use APIs

5.1 Initialization

Code 5–1 The DVDInit Function

```
#include<revolution/dvd.h>

void DVDInit(void);
```

You must call this function before you can access the disc drive. The `DVDInit` function handles all necessary initialization of the drive. You don't need to call this function more than once, but it doesn't cause any trouble if you do.

5.2 Open

Code 5–2 The DVDOpen Function

```
#include<revolution/dvd.h>

BOOL DVDOpen(const char* fileName, DVDFileInfo* fileInfo);
```

In order to access a file, you first have to get the `DVDFileInfo` structure of the file by calling the `DVDOpen` function. The information on the file *fileName* is retrieved and stored in *fileInfo*.

Filenames can take one of the following styles:

- `/texture/mario`
- `texture/mario`

If you use the former style, the optical disc drive device driver takes it as an absolute path name; in contrast, if you use the latter style, it's taken as a relative path name to the current directory by the device driver. There's no restriction on the length of the filename.

Like other file systems, you can use "`..`" (parent directory) and "`.`" (current directory).

To change the current directory, see section "5.6 Change Directory" on page 17.

5.3 Read

There are two types of optical disc read APIs: synchronous, meaning the function does not return until the read finishes; and asynchronous, meaning the function returns immediately while the read occurs in the background.

5.3.1 Synchronous Read

Code 5–3 The DVDRead Function

```
#include<revolution/dvd.h>

s32  DVDRead( DVDFileInfo* fileInfo, void* addr, s32 length, s32 offset);
```

The `DVDRead` function reads a portion of the file and stores it into the buffer specified by *addr*. *Length* and *offset* specify which part of the file to read, where *offset 0* means the top of the file.

The `DVDRead` function returns the number of bytes transferred if the read succeeded; it returns –1 if the read did not.

The programmer must prepare enough memory space for the buffer. To get the size of the file, use the `DVDGetLength` macro (see "5.5 Get Size" on page 17).

Because of drive hardware restrictions, *addr* should be 32-byte aligned, *length* should be a multiple of 32, and *offset* should be a multiple of 4.

As noted above, the `DVDRead` function does not return until the read finishes, hence the term "synchronous." Therefore, if the user ejects the disc, the `DVDRead` function will keep waiting in the function for the user to insert a disc. In the meantime, the CPU executes other runnable threads, if any. For more information on threads, refer to the *Operating System*.

The application does not need to invalidate the data cache because the `DVDRead` function does this by default. To change this behavior, call the `DVDSetAutoInvalidation` function (described in the SDK *Function Reference*). If the read data contains a program, however, you must invalidate the instruction cache regardless of whether you have changed this default behavior.

5.3.2 Asynchronous Read

Code 5–4 The DVDReadAsync Function

```
#include<revolution/dvd.h>

typedef void (*DVDCallback)(s32 result, DVDFileInfo* fileInfo);

BOOL DVDReadAsync(DVDFileInfo* fileInfo, void* addr, s32 length,
       s32 offset, DVDCallback callback );
```

`DVDReadAsync()` issues a command to the drive to read the file into the buffer specified by *addr* and then returns immediately. *Length* and *offset* specify which part of the file to read, where *offset 0* means the top of the file. The function specified by *callback* is invoked when the read finishes; however, you can set *callback* to `NULL` if you do not want any callback to be invoked.

As with `DVDRead()`, you must prepare enough memory space for the buffer. To get the size of the file, use the `DVDGetLength()` macro (see "5.5 Get Size" on page 17).

And again, *addr* should be 32-byte aligned, *length* should be a multiple of 32, and *offset* should be a multiple of 4.

`DVDReadAsync()` always returns `TRUE`.

The argument *result* shows whether the read succeeded or not. If the read was successful, the number of bytes transferred is set; the function returns –1 if the read was not successful.

Asynchronous means that the read goes on in the background. Therefore, you should call `DVDGetFileInfoStatus()` to display the appropriate message to the user while the transfer executes in the background. (See "6 Error Handling" on page 18 for more details about error handling.)

`DVDReadAsync()` invalidates the data cache by default. To change this behavior, call `DVDSetAutoInvalidation()` (described in the SDK *Function Reference*). If the read data contains a program, however, you must invalidate the instruction cache regardless of whether you have changed this default behavior.

You can call as many read commands as you want before any previously issued read finishes. In such cases, the argument *fileInfo* of *callback* shows which file transfer has just finished.

The following code sample reads a file using `DVDReadAsync()`:

Code 5–5 Asynchronous File Read

```
1:      // This variable should be declared "volatile" because this is shared
2:      // by two contexts.
3:      volatile s32     readDone = 0;
4:
5:      // The name of the file we are going to read.
6:      char*   fileName = "testfile1.txt";
7:
8:
9:      static void callback(s32 result, DVDFileInfo* fileInfo)
10:     {
11:         if (result == -1)
12:         {
13:             readDone = -1;
14:         }
15:         else
16:         {
17:             readDone = 1;
18:         }
19:         return;
20:     }
21:
22:     void main(void)
23:     {
24:         DVDFileInfo fileInfo;
25:         u32         fileSize;
26:         u8*         buffer;         // pointer to the buffer
27:
28:         MyOSInit();
29:
30:         DVDInit();
31:
32:         // We MUST open the file before accessing the file
33:         if (FALSE == DVDOpen(fileName, &fileInfo))
34:         {
35:             OSHalt("Cannot open file");
36:         }
37:
38:         // Get the size of the file
39:         fileSize = DVDGetLength(&fileInfo);
40:
41:         // Allocate a buffer to read the file.
42:         // NOTE: Pointers returned by OSAlloc are always 32-byte aligned.
43:         buffer = (u8*)OSAlloc(OSRoundUp32B(fileSize));
44:
45:         // This function only start reading the file.
46:         // The read keeps going in the background after the function returns
47:         if (FALSE == DVDReadAsync(&fileInfo, (void*)buffer,
48:                                   (s32)OSRoundUp32B(fileSize), 0, callback))
49:         {
50:             OSHalt("Error occurred when issuing read");
51:         }
52:
53:         while (1)
54:         {
55:             switch (readDone)
56:             {
57:               case 1:       // the read succeeded
58:                 goto exit;
59:                 // NOT REACHED HERE
60:               case -1:      // the read failed
```

```
61:                 OSHalt("Error occurred when reading file");
62:                 // NOT REACHED HERE
63:               case 0:       // processing...
64:                 // do something
65:                 break;
66:             }
67:         }
68:
69:       exit:
70:         // From here, we can use the file
71:         OSReport("read end\n");
72:
73:         // Close the file
74:         DVDClose(&fileInfo);
75:
76:         :
77:
78:         OSFree(buffer);
79:
80:         OSHalt("End of demo");
81:
82:         // NOT REACHED HERE
83:     }
```

You can do anything (for example, show graphics, play audio, and so forth) while reading the file in the background. If the read finishes, *callback* is invoked. You can write whatever you want in the callback function. In this example, we wrote it to set the flag *readDone* to verify whether the read finished in the main loop (lines 53-67).

Note: *Volatile* is necessary for the variable *readDone* because *readDone* is accessed by two contexts, `main` and *callback*. Without *volatile*, compilers might assign a register for the variable, which will cause inconsistency between the two contexts.

The above code is stored as "`dvddemo/src/dvddemo2.c`." Try it to see how it works.

Although we didn't document the code here, `dvddemo3` shows you how the system works when you issue a `DVDReadAsync()` without waiting for the previously issued read command to finish.

Note: You *cannot* issue `DVDReadAsync()` with a *fileInfo* that has already been used in a previously issued and/or unfinished `DVDReadAsync()` argument. For example, you might want to write something like the following code segment to read each half of the file separately, but this is illegal.

Code 5–6 Example of Illegal Code

```
DVDReadAsync(..., fileInfo, ...);
// Illegal!!
DVDReadAsync(..., fileInfo, ...);
```

In this next example, the second `DVDReadAsync()` fails and returns `FALSE`. To solve this problem, you have two options. Either you wait for the first read to finish, like this:

Code 5–7 Legal Code, Option 1

```
DVDReadAsync(..., fileInfo, ...);
:
// Wait to finish the first read
:
DVDReadAsync(..., fileInfo, ...);
```

Or you use a different *fileInfo* for each part, like this:

Code 5–8 Legal Code, Option 2

```
DVDOpen(..., fileInfo1);
DVDOpen(..., fileInfo2);

DVDReadAsync(..., fileInfo1, ...);
DVDReadAsync(..., fileInfo2, ...);
```

It is legal to open one file with multiple *fileInfo*s.

5.4 Close

Code 5–9 DVDClose()

```
#include<revolution/dvd.h>

BOOL DVDClose(DVDFileInfo* fileInfo);
```

If you no longer need to access the file, close the file with `DVDClose()`. If *fileInfo* is in use (i.e., the `DVDReadAsync()` function using *fileInfo* hasn't finished yet), call `DVDCancel()` internally and then close after the process cancels.

`DVDClose()` always returns `TRUE`.

5.5 Get Size

`DVDGetLength()` gets the size of the file specified by *fileInfo*. This is useful when you want to allocate memory to read the data dynamically.

Code 5–10 DVDGetLength()

```
#include<revolution/dvd.h>

u32 DVDGetLength(DVDFileInfo* fileInfo);
```

5.6 Change Directory

Code 5–11 DVDChangeDir()

```
#include<revolution/dvd.h>

BOOL DVDChangeDir(const char* dirName);
```

This function changes the current directory. As you can imagine, if you specify the directory beginning with a "`/`" (as in "`/texture`"), the function will try to change the current directory to"`/texture`" as an absolute path name. If you specify the directory beginning without a "`/`" (as in "`texture`"), the function treats the directory as relative to the current directory.

If the function succeeded in changing directory, it returns `TRUE`; otherwise, it returns `FALSE`.

6 Error Handling

The Wii optical disc API provides a very simple way to handle errors.

6.1 Disc Drive Device Driver Error Processing Policy

The policy for disc drive device drive error processing is *polling*. There is no need for complex error-processing routines. The game developer polls for the disc drive status, and then can only display an appropriate message on the screen as necessary.

For example, when a disc is inserted, the device driver confirms that the disc is appropriate, and then runs the requested command. Since these processes occur automatically, the game developer does not need to create a disc identification routine.

6.2 Error Checking

Use the following function to check the status for disc drive errors.

Code 6–1 Function to Acquire Disc Drive Status

```
#include<revolution/dvd.h>

s32 DVDGetFileInfoStatus(const DVDFileInfo* fileInfo);
s32 DVDGetCommandBlockStatus(const DVDCommandBlock* commandBlock);
s32 DVDGetDriveStatus(void);
```

`DVDGetFileInfoStatus()` returns the transfer status for the specified *fileInfo*, `DVDGetCommandlockStatus()` returns the status for the specified *commandBlock*, and `DVDGetDriveStatus()` returns the status of the current disc drive. These functions return one of the following values.

Table 6–1 Error Messages Returned by the Disc Drive Status Get Functions

Definition	Value	Meaning
DVD_STATE_FATAL_ERROR	–1	A fatal error occurred.
DVD_STATE_END	0	Transfer finished/no transfer requested.
DVD_STATE_BUSY	1	Transfer is being processed.
DVD_STATE_WAITING	2	Transfer is queued and waiting to be processed.
DVD_STATE_NO_DISK	4	No optical disc in the drive.
DVD_STATE_WRONG_DISK	6	Wrong optical disc in the drive.
DVD_STATE_MOTOR_STOPPED	7	Drive's motor stopped (use for multiple-disc games).
DVD_STATE_PAUSING	8	Paused by `DVDPause()`.
DVD_STATE_CANCELED	10	Transfer was canceled.
DVD_STATE_RETRY	11	A retry error was generated.

Note: The `DVD_STATE_COVER_OPEN`, `DVD_STATE_COVER_CLOSED`, and `DVD_STATE_IGNORED` errors messages that existed in Nintendo GameCube were deleted.

Note: Of these errors, the ones that must be processed as errors by the application are `DVD_STATE_FATAL_ERROR`, `DVD_STATE_NO_DISK`, `DVD_STATE_WRONG_DISK`, and `DVD_STATE_RETRY`.

Note: These errors are issued when the issued command does not process normally for some reason. If no command is issued, no error occurs. For example, if a disc is ejected when there is no command issued, then the No Disk error (`DVD_STATE_NO_DISK`) is not issued. In addition, it is not necessary for the developer to notify the game player of this type of *status change that is not reported as an error*. For example, this means that it is unnecessary to display a message on the screen that a disc has been ejected during a game until an attempt to acess the disc is made.

Here is some sample pseudo-code for how to use this function:

Code 6–2 Error Handling

```
:
DVDReadAsync(fileInfo);
:
while(1)
{
    :
    // Foreground process
    :
    switch (DVDGetFileInfoStatus(fileInfo))
    {
      case DVD_STATE_FATAL_ERROR:
        // Display “Fatal error occurred”
        // Stop the game
        break;
      case DVD_STATE_END:
        // File is read successfully
        break;
      case DVD_STATE_BUSY:
        // Display "Now loading..."
        break;
      case DVD_STATE_NO_DISK:
        // Display “Open the cover and put in the correct disc”
        break;
      case DVD_STATE_WRONG_DISK:
        // Display “This is not a disc of game XXX. Replace it with the correct disc”
        break;
      case DVD_STATE_RETRY:
        // Display “Cannot read this disk”
        break;
    }
}
```

Note: The `DVDRead()` and `DVDReadAsync()` functions process the following tasks internally.

- Wait for the user to open the cover and then close it (when no optical disc or wrong disc detected)
- Read a certain area, which is called the optical disc ID, to verify if the correct disc is in the drive
- Issue the read function again if the correct optical disc is verified to be in the drive

You need only to check the return value of the status getting functions and display the appropriate message to the user. No complicated error handling programming is needed.

6.2.1 No Disk Error (`DVD_STATE_NO_DISK`)

This error is issued when the drive cannot detect a Wii disc.

This error may also be generated even if a disc is inserted in the following circumstances.

- The Wii disc is inserted upside down.
- A disc that is not a Wii disc (such as a 12cm CD) is inserted.
- An extremely dirty Wii disc is inserted.

6.2.2 Wrong Disk Error (DVD_STATE_WRONG_DISK)

This error is issued when the wrong game disc is inserted in the drive.

6.2.3 Retry Error (DVD_STATE_RETRY)

This error is issued when a disc cannot be accessed because of dust or fingerprints on the disc. The device driver will automatically attempt a Retry when a disc is re-inserted after being ejected.

6.2.4 Fatal Error (DVD_STATE_FATAL)

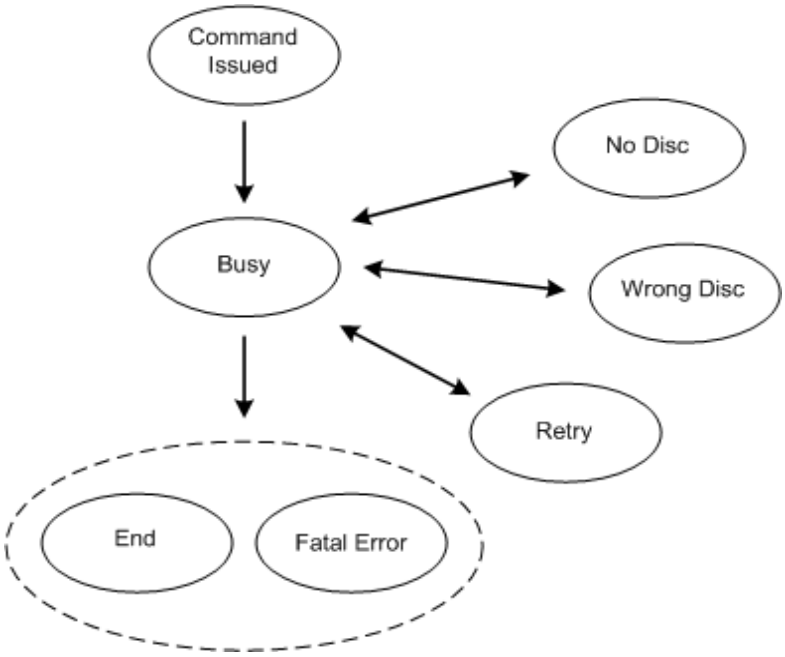
This error is issued when a problem that is irreparable is detected. Because the drive or disc may be damaged, the game must be stopped after the error message is displayed.

Note: Disable all controller operations and the RESET and POWER buttons while an error message for this error is being displayed. Display of a fatal error's error message takes precedence over anything else.

6.3 State Diagram

The following diagram indicates state (error type) transitions.

Figure 6–1 State Transition Diagram



The following describes the diagram.

- Changing from *No Disc*, *Wrong Disc*, or *Retry* to *Busy*
  This state change occurs while a disc is inserted. Be aware that the state does not change when the disc is ejected.
- Calibration time while a disc is inserted
  While a disc is inserted, be aware that more time is needed for the drive calibration process. For example, it takes time when a disc is replaced in the Wrong Disc state for the driver to check the Disc ID.

6.4 Disc Check by the Application during Restart

A *disc check* using the DVDCheckDiskAsync function must be performed when a restart is performed by the application, before beginning any other processing. If disc removal is detected during the reset process, the application should not restart. Instead, call the OSReturnToMenu function and return execution to the Wii Menu.

Code 6–3 The DVDCheckDiskAsync Function

```
#include<revolution/dvd.h>

typedef void (*DVDCBCallback)(s32 result, DVDCommandBlock* block);

BOOL DVDCheckDiskAsync(DVDCommandBlock* block, DVDCBCallback callback);
```

After it has been confirmed that the last inserted disc was the correct disc, the DVDCheckDiskAsync function checks whether the disc has been removed even once. TRUE is always returned. After reading completes, the function specified in *callback* is called.

The *result* parameter indicates whether the read was successful. If the correct disc is set, 1 is returned; if the correct disc was not inserted, 0 is set.

For example, while playing Game A, a player replaces the disc with Game B, and presses the reset button. At that time, the player probably expects Game B to start up. If Game A was created to *restart with the application* as the reset process and there is no disc check, then even when the reset button is pressed, Game A continues to run and the operation is counter to the player's intentions. Therefore, when performing a restart with the application, a disc check is required.

Even when the result of the disc check is that the disc has been removed at least once, it is possible that the correct disc was inserted thereafter. However, if the disc has been removed at least once, it will take a few seconds to determine whether the correct disc is inserted. For this reason, it is required that the disc not be identified and execution be returned to the Wii Menu with the OSReturnToMenu function. If disc identification is done and the result is that the wrong disc is inserted, then the system must be rebooted and several seconds will be needed for the disc to boot up.

Even if a disc is removed during a game, the device driver will automatically determine if the correct disc is inserted when the disc is next accessed (a process that requires several seconds). When the reset button is pressed after it has been verified that the correct disc is inserted as the result of that determination, the disc check will return *confirmed* so there is no need to reboot the system. The disc check is to verify whether a disc has been removed even once after the *final* determination, as in this case.

When restarting with the OSRestart or OSExec function, a disc check is performed in the API. When the result is anormal, a system reboot is performed automatically. For this reason, there is no need to do a disc check before running either the OSRestart or OSExec function.

When a disc has been removed, we do not recommend always rebooting the system. Rather, reboot the system when the reset button has been pressed and the disc has been confirmed. Please exercise caution.

6.5 Emulating Errors

The developer must verify that errors are handled appropriately by the program (for example, displaying an appropriate message on the TV). If NDEV is used, all error types, including the Wrong Disc, Retry, and Fatal errors, can be emulated.

NDEV:

Can emulate all errors. See the NDEV manual for details.

Table 6–2 Emulation with Each Development Device

	Retry Error	Fatal Error (Note 1)	Wrong Disc Error	No Disc Error
NDEV	ODEM (RETRY Button)	ODEM (FATAL Button)	ODEM (Select DLF file) ODEM (Eject) ODEM (Load)	ODEM (No Disc) ODEM (Eject) ODEM (Load)
RVT-R Reader	None	None	Exchange the disc (Note 2)	Remove disc

1. An error will occur when the button is continuously pressed until the next disc access.
2. The Disc ID of the new disc should be different than the Disc ID of the disc prior to the exchange.

7 File System

This chapter briefly describes how the Wii file system works and contains some tips for developers.

The Wii file system uses a conversion table called FST (for File Symbol Table). Every time the `DVDOpen` function is called, the Wii searches for the filename in the FST and retrieves the file information from it. In this chapter, we will mainly discuss the following:

- Size of the FST
- Location of the FST
- Speed of conversion
- Character set usable for file/directory names

7.1 FST Size

FST size can be calculated easily from the following equation:

Equation 7-1 Calculating FST File Size

$$12 + (\textit{length of filename} + 1)\ (\textit{bytes} / \textit{entry})$$

So, if we have a file named "`foo`," it consumes 12 + 3 + 1 = 16 bytes in the FST (the last +1 is for the termination `NULL`).

The same logic can be applied to directories. If we have a file named "`foo`" under "`/baa1/baa2/`", and there are no other files or directories on the optical disc, the size of the FST is going to be:

Equation 7-2 Calculating FST Directory Size

$$12[\textit{root directory}] + (12 + 4 + 1)\ [\textit{baa}1] + (12 + 4 + 1)\ [\textit{baa}2]\ (12 + 3 + 1)\ [\textit{foo}] = 62[\textit{bytes}]$$

FST size must not be more than 1,048,576 bytes.

7.2 FST Location

The boot program loads the FST at the highest memory point, higher than *arena hi*. Therefore, using Equation 7-2, which uses 62 bytes for FST, the FST is loaded to 0x817f_ffc0 (24MB – 64 bytes; note that it is 32-byte aligned) and *arena hi* is set to the same location.

Since the FST upper size limit is 1,048,576 bytes, the lower limit for the FST and *arena hi* location is 0x8170_0000. If the FST is placed under 0x8170_0000, the application will fail to run.

7.3 Tips to Reduce FST Size and Speed

In general, having fewer files improves the conversion speed. Changing the directory is also a good idea. Since the `DVDOpen` function starts searching for files in the current directory, changing the directory to that of the target file before you call the `DVDOpen` function improves the conversion speed if you need to open more than one file.

For example, say you need to open two files, "`/foo/bar/dummy1`" and "`/foo/bar/dummy2`". If you simply call the `DVDOpen` function twice, the `DVDOpen` function will need to search two separate times from the beginning of the optical disc. However, if you call the `DVDChangeDir` function to change the current directory to "`/foo/bar`", the two subsequent `DVDOpen` function calls using relative path names will be faster, even allowing for the time it takes the `DVDChangeDir` function to search the directory (it uses the same algorithm as the `DVDOpen` function).

7.4 Character Set Usable for File/Directory Names

You can use 7-bit ASCII characters for file/directory names, except the following.

- " * , / : ; < > ? \ |
- Control codes

In other words, you can use the following characters.

- Numerals
- Alphabets
- Spaces
- ! # $ % & ' ( ) + - . = [ ] ^ _ @ ` { } ~

Examples of *unusable* characters include Kanji, “Hankaku kana,” Japanese letters, and accents (such as the “é” in “Pokémon”).

Note: File and directory names are case-insensitive. For example, `foo.txt` and `FOO.TXT` and `FoO.tXt` would all be considered the same file. You can use *any* combination of lowercase and capital letters to open files and directories.

There is no restriction on the length of file and directory names. As far as the tool (or to be more precise, Windows) allows, you can use as long name as you want.

8 Directory Access Support Functions

Sometimes you may want to read all of the files in a certain directory without knowing the name of each file. Or you may want to create common functions which change their behavior according to the files within a directory. To facilitate directory access, the disc drive library provides BSD UNIX-like directory access functions.

In most cases, you can write code using three functions, open, read, and close, to access directories. So first we'll discuss how to use the three functions, and then introduce a simple code sample that uses these three functions. After that, we'll look at other useful functions.

8.1 Open Directory

Code 8–1 The DVDOpenDir Function

```
#include<revolution/dvd.h>

typedef struct
{
    u32         entryNum;
    u32         location;
    u32         next;
} DVDDir;

BOOL DVDOpenDir(const char* dirName, DVDDir* dir);
```

The `DVDOpenDir` function opens a directory specified by *dirName* and associates *dir* with that directory. It returns `TRUE` if it found the directory; otherwise, it returns `FALSE`.

You can use both relative and absolute path names for *dirName*. If you use a relative path name (i.e., *dirName* starts without a "`/`"), it will mean relative to the current directory.

8.2 Read Directory

Code 8–2 The DVDReadDir Function

```
#include<revolution/dvd.h>

typedef struct
{
    u32         entryNum;
    BOOL        isDir;
    char*       name;
} DVDDirEntry;

BOOL DVDReadDir(DVDDir* dir, DVDDirEntry* dirent);
```

The `DVDReadDir` function gets information on the next directory *entry*. The directory entry is either a directory or a file. If it's a directory, `dirent->isDir` is `TRUE`. You can call the `DVDOpenDir` function to open the subdirectory using `dirent->name`. If it's a file, `dirent->isDir` is `FALSE`. You can call the `DVDOpen` function to open the file using `dirent->name` as well.

The function returns `FALSE` upon reaching the end of the directory or detecting an invalid location was set by the `DVDSeekDir` function.

8.3 Close Directory

Code 8–3 The DVDCloseDir Function

```
#include<revolution/dvd.h>

BOOL DVDCloseDir(DVDDir* dir);
```

`DVDCloseDir()` closes the specified directory structure. It returns `TRUE` on success; otherwise, it returns `FALSE`.

8.4 Sample Code for Accessing a Directory

This is a very simple sample code to show how to access a directory. This code opens a directory "`foo/baa`" (relative to the current directory) and show all the files and directories under there.

Code 8–4 Sample Code for Accessing a Directory

```
    DVDDir          dirFooBaa;
    DVDDirEntry     dirent;

    if (FALSE == DVDOpenDir("foo/baa", &dirFooBaa))
    {
        // print "can't open the directory" and exit
    }

    while(TRUE == DVDReadDir(&dirFooBaa, &dirent))
    {
        OSReport("name: %s, type: %s\n", dirent.name,
                 (dirent.isDir)? dir : file);
    }

    if (FALSE == DVDCloseDir(&dirFooBaa))
    {
        // print "can't close the directory" and exit
    }
```

8.5 Tell Directory

Code 8–5 The DVDTellDir Function

```
#include<revolution/dvd.h>

u32 DVDTellDir(DVDDir* dir);
```

The `DVDTellDir` function is a macro that returns the current location of the specified directory structure.

This function should be used with the `DVDSeekDir` function to provide *save* and *restore* features for directory access. You can use the `DVDTellDir` function to get the current location of the directory structure, then restore it using the `DVDSeekDir` function to read the directory from the previous location again.

8.6 Seek Directory

Code 8–6 The DVDSeekDir Function

```
#include<revolution/dvd.h>

void DVDSeekDir(DVDDir* dir, u32 loc);
```

The `DVDSeekDir` function is a macro that sets the position of the next `DVDReadDir` function on the directory structure.

This function should be used with the `DVDTellDir` function to provide *save* and *restore* features for directory access. It's dangerous to use a number for *loc* that is not returned by the `DVDTellDir` function. It's also dangerous to use a value for *loc* that is returned by the `DVDTellDir` function but for the other directory structure.

Note: This function is a macro and performs no argument checking, but if you happened to specify an invalid value for *loc*, the `DVDReadDir` function can detect it and return `FALSE`.

8.7 Rewind Directory

Code 8–7 The DVDRewindDir Function

```
#include<revolution/dvd.h>

void DVDRewindDir(DVDDir* dir);
```

The `DVDRewindDir` function resets the position of the specified directory structure to the beginning of the directory. A call to the `DVDReadDir` function after this function behaves as if the directory had just been opened.

9 Multiple Disc Games (Reference Information)

When multiple discs are required for a single game, the discs must be exchanged at an appropriate place within the game. Perform this by specifying the Disc ID (see Chapter 3.2) of the next disc to insert. This section broadly describes the different methods for multiple disc games, how to specify discs, and a general procedure for exchanging discs. An example of instructing the user to exchange the disc is provided.

9.1 Disc Exchange Processing Pattern

When exchanging discs for a multiple disc game, two major methods exist depending on whether the `DOL` program portion (Chapter 3.1) of the new disc is loaded and executed. The following examples assume that Disc 2 is being exchanged for Disc 1.

A When the `DOL` file of Disc 2 is loaded and executed
After Disc 1 is replaced with Disc 2, perform the restart process. The `DOL` file on Disc 2 is booted when restart is performed.

B When the `DOL` file of Disc 2 is not loaded or executed
After Disc 1 is replaced with Disc 2, the user file (Chapter 3.1) on Disc 2 can be accessed. You do not need to do anything after the exchange. For this case, you may either request an independent boot of Disc 2 or not request such a boot.

B1 When an independent boot of Disc 2 is requested
This case can be when the program for Disc 1 and Disc 2 are the same and only the user file portion is different. Depending on the progress of the game in the Wii console NAND memory, there may be times when the user is instructed to return to Disc 1.

B2 When an independent boot of Disc 2 is not requested
This case can be when Disc 2 is a data-dedicated file. Even in this case, when performing an independent boot, be sure to provide appropriate instructions to the user.

We do not recommend B2 because disc exchange may occur frequently during game play. Unless there is a specific reason why a data-dedicated Disc is needed, it is more convenient for game play to have the same `DOL` file on both first and second discs, and allow the second disc to be independently booted.

Compared to Method A, B1 does not load the `DOL` program portion of Disc 2 and can complete the transition to Disc 2 a little faster. If the `DOL` program potion for the first and second discs can be the same, B1 is the best option.

In either case, we recommend that data be saved before the disc exchange. The reasons for this are listed below.

- To be able to avoid data loss when a disc read error occurs after the exchange.
- To simplify debugging after transitioning to the disc after exchange (For Method A only: by limiting the passing of data from Disc 1 to Disc 2 to go through the Wii console NAND memory, when a bug occurs during the exchange, Disc 2 can be booted for debugging).

Note: Because there is old information in the structure of the file opened before the exchange, do not use it after the exchange. Even if the file has the same name in Discs 1 and 2, be sure to reopen the file. We also recommend closing all files that were opened before the disc exchange.

9.2 Disc Specification during Exchange

Use the Disc ID (see Chapter 3.2) when specifying a disc for exchange. For details, see the `DVDChangeDisk*` function references.

A feature of the `DVDChangeDisk*` functions allows you to specify wildcards in the Disc ID element. For example, this feature allows a disc to be used for disc exchange of any version by setting the Game Version to `0xff` as long as the other three elements (Game Code, Company Code, and Disc Number) match.

The following sections indicate precautions regarding the use of wildcards.

9.2.1 Wildcard Use Restriction (excluding Game Version)

Do not use wildcards for elements other than the game version. These elements are the Game Code, Company Code, and Disc Number. Specify these explicitly. If wildcards are used for these elements, operation check combinations become enormous and difficult.

If you are considering using a wildcard in these elements, please contact Nintendo first.

9.2.2 Wildcard Use Restriction (Game Version)

There are advantages and disadvantages for using or not using (explicitly specifying) wildcards for the Game Version. The following lists those advantages and disadvantages; make your selection after considering them carefully.

A When the Game Version is explicitly specified

Advantages

- Because supported disc versions have a one-to-one correspondence, operations need to be verified only for those combinations.

Disadvantages

- When the version is upgraded in the future, both disc versions must be upgraded. In other words, if the Disc 2 version is specified as 0 in Disc 1, then when Disc 2 becomes the corrected version 1 after sales, after correcting Disc 1, both disc versions must be upgraded and resubmitted.
- If friends each own the same game but have different versions of that game, if the discs get mixed up, the discs will not work and will hang at the check when the discs are exchanged.
- A mark to distinguish different versions at the level are necessary to identify the version.

B When wildcards are used in the Game Version

Advantages

- Future upgrades of disc versions can occur independently.
- Because operations can occur in combination with different versions of the disc, there is no problem if the disc is mixed up with a friend’s disc.

Disadvantages

- Caution is required during future version upgrades. In other words, when Disc 2 is upgraded, it must be guaranteed that disc exchange is possible with all previously sold Disc 1 versions.

9.3 Disc Exchange Procedure

The following is the procedure for the player to exchange discs and then to access the newly inserted disc.

1. Stop the disc drive motor.
2. Verify that the disc drive motor has stopped. Indicate to the player to exchange the disc (see Chapter 9.4).
3. Eject the disc, then wait until a disc is inserted.
4. Determine whether the ID of the inserted disc is appropriate.
5. Load the FST (File Symbol Table) of the disc. The disc files are now accessible.

With the exception of displaying the message in Step (2), all of these steps are processed automatically by the `DVDChangeDisk` function (see Chapter 9.6).

The display time for the message in Step (2) can be found by performing a device driver status check using the `DVDGetDriveStatus` function. The method for checking the device driver status is the same as for error handling. See Chapter 6.2. When the disc drive motor stops, the device driver status is `DVD_STATE_MOTOR_STOPPED`, so display the message in Step (2) only when in this status.

If the inserted disc is not appropriate or if there is no disc inserted, display an appropriate message as in Chapter 9.4. The disc exchange can also be cancelled by having `DVDChangeDisk*()` call `DVDCancel()`.

9.4 Error Handling and Messages During Disc Exchange

Display messages to the player during disc exchange.

- When instructing to exchange the disc
  While the device driver state is `DVD_STATE_MOTOR_STOPPED`, display a message to the player instructing the player to insert an appropriate disc. Specifically, clearly indicate which of the multiple discs to insert.

In addition, perform the same error handled when running the `DVDChangeDisk function` as for the `DVDRead` function. See Chapter 6.2 for details (when a disc that was not specified is inserted, a Wrong Disc error occurs).

9.5 Multiple Disc Emulation

Currently, emulation of multiple discs is not supported.

9.6 Multiple Disc Functions

The following functions are for multiple discs. See the function reference for more information on these functions.

- `DVDChangeDisk*` (`DVDChangeDisk` or `DVDChangeDiskAsync`)
- `DVDCompareDiskID`
- `DVDGenerateDiskID`
- `DVDGetCurrentDiskID`

9.7 Cautions Regarding Multiple Disc Games

A File Symbol Table (FST) is overwritten by that of a disc inserted later when the correct disc is inserted. At this time, there is no problem if the size of the FST of the new disc is smaller than the disc before the exchange. However, if the size is larger, then the disc cannot be loaded, and the system hangs after giving an ASSERT message.

For multiple disc games, be sure to specify enough memory that is secured for FST.

See the development environment manuals for more detail.

10 Optical Disc Drive FAQs

1. Is the data recorded in one spiral, similar to CDs?

Yes.

2. How fast are the optical disc drive's random-access capabilities?

Compared to ROM cartridges, which have very fast seek times, we must accept that the optical disc drive will have a poorer random-access capability. To minimize any negative impact, you should place data as often as possible in the order that the game will need it. You can also solve this problem by duplicating data many times on the optical disc, but of course this trades data capacity for speed.

3. How many files can I open at one time?

There is no limit; you can open as many files as you want.

4. Can I call DVD*() from within a callback?

Yes. For example, you can call the `DVDClose` function when the read of the file finishes, like this:

Code 10–1 FAQ Code Sample

```
void callback(s32 result, DVDFileInfo* fileInfo)
{
    if (result == DVDGetLength(fileInfo))
    {
        DVDClose(fileInfo);
    }
    ...
}
```

As another example, you can call the `DVDReadAsync` function from within an audio callback routine while accessing audio data.