Revolution SDK

Operating System

Version 1.01

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

RVL-06-0042-001-D
Released: October 17, 2008

Contents

Revision History 14
1 Overview 15
2 Initializing the OS 16
2.1 OS Initialization 16
2.2 Getting the Console Type 16
3 Memory 18
3.1 System Memory Map 18
3.2 Getting Memory 19
3.2.1 Arena Management 20
3.3 Managing Memory 20
3.3.1 Allocation Alignment 21
3.3.2 One Heap 21
3.3.3 Multiple Heaps 23
3.3.4 Miscellaneous Details 24
3.4 Memory Management in C++ Code 24
3.5 Restrictions on Running Command Codes in the MEM2 Region 26
4 Error Handling and Notification 27
4.1 Error Display 27
4.2 Memory Protection 28
5 Cache Control 34
5.1 Cache Description 34
5.2 Cache Incoherence 35
5.3 Basic Cache Management 36
5.3.1 Efficiency 38
5.4 Locked Cache Operation 39
5.4.1 Locked Cache API Overview 39
5.4.2 DMA Engine 40
5.4.3 Basic Locked Cache API and Demos 40
5.4.4 Low-Level Locked Cache API and Demos 42
5.4.5 Additional Locked Cache Functions 44
5.4.6 Enabling and Disabling the Locked Cache 44
5.4.7 Dangers in Using the Locked Cache 45
6 Time 46
6.1 Real-Time Clock 46
6.2 Alarm 47
7 Critical Sections 49
7.1 Programming Model 49
8 Using CPU Idle Time 50
9 Threads 52
9.1 Initialization 52
9.2 Scheduling 52
9.2.1 Interaction with Interrupts 52
9.3 Thread Creation 53
9.4 Synchronization 56
9.4.1 Library Access 56
9.4.2 Synchronizing by Disabling the Scheduler 56
9.4.3 Synchronizing by Sleeping and Waking 57
9.4.4 Synchronizing with Messages 57
9.4.5 Synchronizing with Mutexes 59
9.4.6 Synchronizing with Conditional Variables 62
9.5 Context Switching 64
9.6 Checking the Active Threads 64
9.7 Threads and Callbacks 65

10 Fast Float/Integer Casting 66
10.1 Initializing the Fast Cast API 66
10.2 Fast Casting Routines 66
11 Fonts 68
11.1 Loading Fonts 68
11.2 Character Width 68
11.3 Font Header 68
12 Relocatable Module System (REL) 71
12.1 Relocatable Modules 71
13 Profiling 73
13.1 Stopwatches 73
13.2 Performance Monitors 74
14 Reset and Shutdown Processing 77
14.1 Reset and Shutdown Notes 77
14.1.1 Requirements for Conditions That Cause Fatal Errors, Such as System Lockups. 77
14.1.2 Recommendations for Non-Fatal Conditions that Should Be Handled, Such as Possible Delayed Resets77
14.1.3 Miscellaneous Notes 78

Code Examples

Code 2–1 OSGetConsoleType 16
Code 3–1 OSGetArena API 19
Code 3–2 Setting the Arena 20
Code 3–3 Heap Allocation 21
Code 3–4 OSAlloc APIs 22
Code 3–5 Memory Allocation 23
Code 3–6 Memory Management for Multiple Heaps 24
Code 3–7 Overriding New and Delete Operators 25
Code 3–8 OSEnableCodeExecOnMEM* API 26
Code 4–1 Debugging Functions 27
Code 4–2 OSReport Function 27
Code 4–3 OSPanic Function 28
Code 4–4 OSProtectRange Function 29
Code 4–5 OSSetErrorHandler Function 29
Code 5–1 Cached and Uncached Access to Memory 35
Code 5–2 Address Translation 36
Code 5–3 Storing and Invalidating Cache Lines 37
Code 5–4 Store and Invalidate Output Sample 37
Code 5–5 Data and Instruction Cache APIs 38
Code 5–6 Basic Locked Cache Demo 40
Code 5–7 Basic Locked Cache Functions 42
Code 5–8 Low-Level Locked Cache Demo 42
Code 5–9 Low-Level Locked Cache Load/Store Functions 43
Code 5–10 Additional Locked Cache Functions 44
Code 5–11 Dangerous Loop in the Locked Cache 45
Code 6–1 Time APIs 46
Code 6–2 Timer Program 47
Code 6–3 Timer APIs 48
Code 7–1 Interrupt Reception API 49
Code 7–2 Critical Section 49
Code 8–1 Idle Function (Background Task) Example 50
Code 8–2 Idle Function (Background Task) API 51
Code 9–1 Thread Creation Example 53

Code 9–2 Basic Thread Creation APIs 54
Code 9–3 Using OSJoinThread 55
Code 9–4 OSJoinThread API 56
Code 9–5 Disabling and Enabling the Scheduler Must Be Done with Interrupts Disabled 57
Code 9–6 Scheduler Control APIs 57
Code 9–7 Thread Sleep and Wakeup APIs 57
Code 9–8 Message API Example 58
Code 9–9 Message APIs 59
Code 9–10 Mutex and Yielding Example 60
Code 9–11 Code 38 Mutex and Yielding APIs 61
Code 9–12 Solving the Bounded Buffer Problem with Condition Variables 62
Code 9–13 Conditional Variable APIs 64
Code 9–14 OSCheckActiveThreads 64
Code 10–1 OSInitFastCast 66
Code 10–2 Fast Cast APIs 67
Code 12–1 _prolog and _epilog Functions 72
Code 13–1 Stopwatch Code Example 73
Code 13–2 Stopwatch Program Output 74
Code 13–3 Stopwatch APIs 74
Code 13–4 Stopwatch Intervals 74
Code 13–5 Basic Performance Monitor Counter Accessor Functions 75
Code 13–6 Performance Monitor Counter Macros 75
Code 13–7 Using Performance Monitor Counter Macros 76
Code 14–8 Reset and Shutdown Functions 77

Figures

Figure 3–1 Game Code Access to the Physical Address Space 18
Figure 3–2 Alternate Representation of Virtual and Physical Address Spaces 19
Figure 5–1 Normal Cache Lookup 35
Figure 5–2 L1 Data Cache Configured in Locked Cache Mode 39

Tables

Table 2–1 Return Values from OSGetConsoleType 16
Table 4–1 Error Types 30
Table 4–2 fpscr Bits 31
Table 4–3 __OSFpscrEnableBits Bits 31
Table 10–1 Quantization Register Values 66
Table 11–1 Font Header Members Accessible by Applications 69

Revision History

Version	Date Revised	Description
1.02	2008/07/11	Revised content to apply to the Wii console.
		Noted that since SDK 3.2, it is prohibited to run instruction codes in the MEM2 region.
1.01	2007/12/01	Applied a standardized format.
	2006/10/25	Added description of interrupt handler.
1.00	2006/03/01	First release by Nintendo of America Inc.

1 Overview

This document provides important information for developers using the Wii Operating System (Revolution OS). Using this information, developers can write real code as soon as possible.

Main features of the Revolution OS:

- Interrupt- and callback-driven programming model
- Simple default memory manager
- Cache management
- Basic timer support
- Basic profiling support

2 Initializing the OS

2.1 OS Initialization

The `OSInit` function, called from the C start-up routine, will initialize the OS. This function is called before all of the global C++ constructors. The application does not need to explicitly call the `OSInit` function.

This initialization prepares the low-level operating system layers to deal with the machine by:

- Determining the amount of memory available to the application
- Initializing the exception table
- Initializing interrupt handlers
- Initializing the thread interfaces
- Performing floating-point operations in non-IEEE mode
- Configuring the MEM1 arena
- Configuring the MEM2 arena
- Initializing the alarm system
- Initializing REL
- Initializing (enabling) the L1 and L2 cache
- Clearing the arenas
- Reading the system settings

2.2 Getting the Console Type

The `OSGetConsoleType` function returns information about the current system. If your application requires a particular console type, use this function.

Code 2–1 OSGetConsoleType

```
u32     OSGetConsoleType ( void );
```

The `OSGetConsoleType` function checks the console type. The left-most (most significant) 4 bits are used to distinguish development systems from retail production systems. In a development system, the left-most 4 bits are set to `0x1`; in a retail production system, these bits are set to `0x0`. The remaining 28 bits show the minor revision number of the console.

Table 2–1 Return Values from OSGetConsoleType

Defined Name	Value	Description
OS_CONSOLE_RVL_NDEV1_0	0x10000010	NDEV 1.0
OS_CONSOLE_RVL_NDEV1_1	0x10000011	NDEV 1.1
OS_CONSOLE_XXXXXX	0x????????	NDEV 2.0
OS_CONSOLE_XXXXXX	0x????????	Production model
OS_CONSOLE_RETAIL4	0x00000004	(for backward compatibility)
OS_CONSOLE_RETAIL3	0x00000003	(for backward compatibility)

Table 2–1 Return Values from OSGetConsoleType

Defined Name	Value	Description
OS_CONSOLE_RETAIL2	0x00000002	(for backward compatibility)
OS_CONSOLE_DEVHW4	0x10000007	(for backward compatibility)
OS_CONSOLE_DEVHW3	0x10000006	(for backward compatibility)
OS_CONSOLE_DEVHW2	0x10000005	(for backward compatibility)
OS_CONSOLE_TDEVHW4	0x20000007	(for backward compatibility)
OS_CONSOLE_TDEVHW3	0x20000006	(for backward compatibility)
OS_CONSOLE_TDEVHW2	0x20000005	(for backward compatibility)
OS_CONSOLE_EMULATOR	0x10000000	(for backward compatibility)

3 Memory

The first and most important resource to be managed is memory. This section presents the Wii memory map and some simple ways to manage it.

3.1 System Memory Map

The Revolution OS employs a memory map similar to MIPS. Application code can access physical memory one of two ways—cached or uncached. For every physical address, there are two corresponding virtual addresses: one which provides cached access, and one which provides uncached access to the same location.

The physical address space is 256 MB (all system memory and memory-mapped devices exist within a 256 MB address space). Cached accesses to the physical address space take the form `0x8xxxxxxx` for MEM1 and `0x9xxxxxxx` for MEM2, while uncached accesses take the form `0xCxxxxxxx` for MEM1 and `0xDxxxxxxx` for MEM2, as shown in the following table. Only the first 256 MB of each of these segments are accessible; accessing any other addresses will cause a memory access fault.

Figure 3–1 Game Code Access to the Physical Address Space

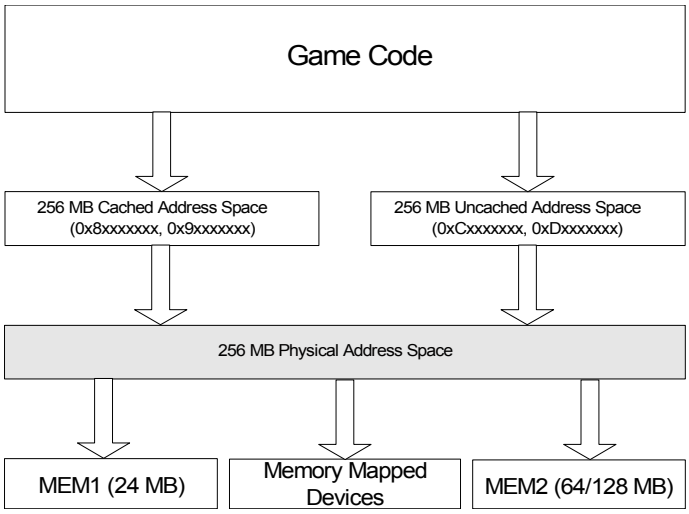
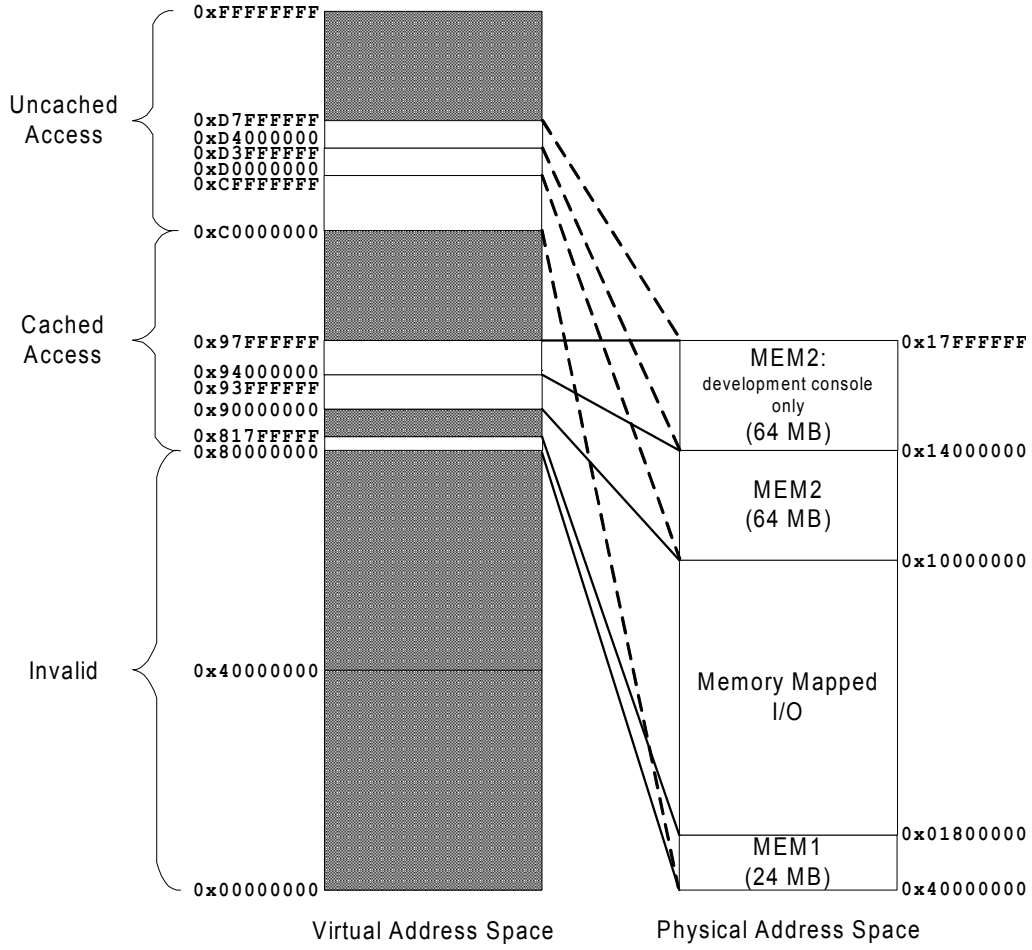


Figure 3–2 Alternate Representation of Virtual and Physical Address Spaces



3.2 Getting Memory

After loading the game program into the system, the pool of remaining memory (called the *arena*) is available to the game. The MEM1 and MEM2 arenas are configured when the OS is initialized. You can get and change the region for each arena using the following functions.

Code 3–1 OSGetArena API

```
void* OSGetMEM1ArenaHi( void );
void* OSGetMEM1ArenaLo( void );
void  OSSetMEM1ArenaHi( void* newHi );
void  OSSetMEM1ArenaLo( void* newLo );
void* OSGetMEM2ArenaHi( void );
void* OSGetMEM2ArenaLo( void );
void  OSSetMEM2ArenaHi( void* newHi );
void  OSSetMEM2ArenaLo( void* newLo );
```

Therefore, the first few lines of any program should look like this:

Code 3–2 Setting the Arena

```
#include <revolution.h>

#define MY_FIRST_MEMORY_AREA_SIZE       1024
#define MY_SECOND_MEMORY_AREA_SIZE      2048

void * MyFirstMemoryArea;
void * MySecondMemoryArea;

void main ()
{
    u8* arenaLo;
    u8* arenaHi;

    arenaLo             = (u8*)OSGetMEM1ArenaLo();
    arenaHi             = (u8*)OSGetMEM1ArenaHi();

    MyFirstMemoryArea   = arenaLo;
    arenaLo += MY_FIRST_MEMORY_AREA_SIZE;
    OSSetMEM1ArenaLo(arenaLo);

    MySecondMemoryArea  = arenaLo;
    arenaLo += MY_SECOND_MEMORY_AREA_SIZE;
    OSSetMEM1ArenaLo(arenaLo);
    :
```

This program grabs some memory from the bottom of the arena. Initially, the arena boundaries will be at least word-aligned.

3.2.1 Arena Management

Typically, the only entity manipulating the arena after the program starts (i.e., after `main()`), is the application program. It is only necessary to get the arena boundaries once without changing them. However, well-modularized code will not want to share global arena variables across the entire application. In that case, manipulating the arena through `OSGetMEM*Arena*` and `OSSetMEM*Arena*` is the proper way to manage the arena.

3.3 Managing Memory

Setting the boundaries of the arena manually may not be the most effective means to manage memory. Therefore, we have provided a simple memory manager that allows you to allocate and free arbitrary-sized blocks of memory from any number of heaps.

Note: The memory management functions (`OSAlloc`, `OSFree`, and others) provided by the Revolution OS still remain for GameCube compatibility. The Wii console has two main memories, MEM1 and MEM2, so its arena is split in two. However, the memory management functions provided by the OS library do not support multiple arenas. As a result, we do not recommend using these functions for Wii application development.

The Revolution SDK provides the MEM library, which supports memory management for multiple arenas. We recommend using the MEM library for Wii application development. Refer to the Memory Management (MEM library) section in the *Revolution Function Reference Manual* (HTML) for information on using the MEM library. Furthermore, some libraries require a memory allocator to be specified during initialization.

3.3.1 Allocation Alignment

Many of the Wii subsystems require memory to be aligned to 32-byte boundaries (the graphics FIFO, indexed data, etc.). The allocator always allocates to 32-byte alignment and always rounds allocation sizes up to 32-byte boundaries. It also rounds out the boundaries of any heaps created so that they are 32-byte aligned. To be safe, however, the following demos align heap boundaries appropriately.

Note: To allocate many small objects efficiently, pack them into arrays to avoid allocator overhead and internal fragmentation.

3.3.2 One Heap

The following program uses one heap to encompass the entire arena:

Code 3–3 Heap Allocation

```
#include <revolution.h>

OSHeapHandle                TheHeap;

#define INT_ARRAY_ENTRIES   1024

void main ()
{
    void*   arenaLo;
    void*   arenaHi;
    u32*    intArray;

    arenaLo = OSGetMEM1ArenaLo();
    arenaHi = OSGetMEM1ArenaHi();

    // OSInitAlloc should only ever be invoked once.
    arenaLo = OSInitAlloc(arenaLo, arenaHi, 1); // 1 heap
    OSSetMEM1ArenaLo(arenaLo);

    // The boundaries given to OSCreateHeap should be 32B aligned
    TheHeap = OSCreateHeap((void*)OSRoundUp32B(arenaLo),
                           (void*)OSRoundDown32B(arenaHi));
    OSSetCurrentHeap(TheHeap);
    // From here on out, OSAlloc and OSFree behave like malloc and free
    // respectively

    OSSetMEM1ArenaLo(arenaLo = arenaHi);

    intArray = (u32*)OSAlloc(sizeof(u32) * INT_ARRAY_ENTRIES);
    :
    // some interesting code using intArray would go here
    :
    OSFree(intArray);
    :
```

In the preceding example, you can use `OSAlloc`/`OSFree` as if it were a standard `malloc/free`.

Note: Its behavior changes slightly in the presence of multiple heaps.

The APIs introduced in this code sequence are as follows:

Code 3–4 OSAlloc APIs

```
typedef int     OSHeapHandle;

void*           OSInitAlloc          ( void* arenaStart, void* arenaEnd, int maxHeaps );

OSHeapHandle    OSCreateHeap         ( void* start, void* end );
void            OSSetCurrentHeap     ( OSHeapHandle heap );
void*           OSAlloc              ( u32 size );
void            OSFree               ( void* ptr );

#define         OSRoundUp32B(x)                 …
#define         OSRoundDown32B(x)               …
```

`OSHeapHandle` effectively functions as a pointer to a heap. As shown in "3.3.3 Multiple Heaps" on page 23, there can be any number of heaps.

`OSInitAlloc` requires some memory for bookkeeping (linear with `maxHeaps`). As a result, the lower boundary of the arena should be modified. When calling `OSInitAlloc`, ensure that the return value is assigned to your local copy of `arenaLo`. `OSInitAlloc` should be called only once per application run.

`OSCreateHeap` returns a handle to a new heap. Since there is only one heap in this example, we simply set the created heap to be the current heap with `OSSetCurrentHeap` and forget about the heap handle. `OSCreateHeap` will align the *start*/*end* arguments to the appropriate 32-byte boundaries; the start boundary is rounded up, while the end boundary is rounded down.

Note: The example program explicitly aligns arena boundaries to 32 bytes before calling `OSCreateHeap`. This approach ensures any boundary changes are visible to the application.

`OSRoundUp32B` and `OSRoundDown32B` are utility macros that round an address up or down to the closest 32-byte boundary.

3.3.3 Multiple Heaps

More complicated games may want to manage memory in separate heaps for several purposes (for example, to delineate memory usage clearly between discrete modules or to increase memory locality for different sections of code). The Revolution OS memory allocator supports this functionality, as demonstrated in the following code sequence:

Code 3–5 Memory Allocation

```
#include <revolution.h>

// Heap sizes MUST be multiples of 32
#define HEAP1_SIZE      65536
#define HEAP2_SIZE      4096

#define OBJ1SIZE        1024
#define OBJ2SIZE        2048

OSHeapHandle            Heap1, Heap2;

void main ()
{
    u8*     arenaLo;
    u8*     arenaHi;
    void*   fromHeap1;
    void*   fromHeap2;

    arenaLo = (u8*)OSGetMEM1ArenaLo();
    arenaHi = (u8*)OSGetMEM1ArenaHi();
    arenaLo = (u8*)OSInitAlloc(arenaLo, arenaHi, 2); // 2 heaps
    OSSetMEM1ArenaLo(arenaLo);

    // Ensure boundary is 32B aligned
    arenaLo = (void*)OSRoundUp32B(arenaLo);

    Heap1 = OSCreateHeap(arenaLo, arenaLo + HEAP1_SIZE);
    arenaLo += HEAP1_SIZE;
    Heap2 = OSCreateHeap(arenaLo, arenaLo + HEAP2_SIZE);
    arenaLo += HEAP2_SIZE;

    OSSetMEM1ArenaLo(arenaLo);

    OSSetCurrentHeap(Heap1);
    fromHeap1 = OSAlloc(OBJ1SIZE);

    // Some code allocating from heap1 goes here
    // OSFree will free to heap1 as well

    OSSetCurrentHeap(Heap2);
    fromHeap2 = OSAlloc(OBJ2SIZE);

    // Some code allocating from heap2 goes here
    // OSFree will free to heap2

    OSFreeToHeap(Heap1, fromHeap1);
    OSFreeToHeap(Heap2, fromHeap2);

    // example of allocation overriding the current heap
    fromHeap1 = OSAllocFromHeap(Heap1, OBJ2SIZE);

    OSHalt("Demo complete");
}
```

In the sequence outlined in "Code 3–5 Memory Allocation" on page 23, the following routines help to manage memory with multiple heaps:

Code 3–6 Memory Management for Multiple Heaps

```
void            OSFreeToHeap        ( OSHeapHandle heap, void* ptr );
void*           OSAllocFromHeap     ( OSHeapHandle heap, u32 size );
```

`OSFreeToHeap` returns a block of memory to a specific heap. An allocated block of memory must be returned to the original heap from which the block was allocated.

Note: `OSFree` can still be used, but it simply returns a block of memory to the current heap, set by `OSSetCurrentHeap`.

`OSAllocFromHeap` allows the program to allocate a block from a specific heap, overriding the default set by `OSSetCurrentHeap`.

3.3.4 Miscellaneous Details

For a few additional APIs that allow heap destruction, memory allocation at specific locations, or discontinuous heap creation, refer to the *Revolution Function Reference Manual* (HTML).

Overhead

The memory overhead of the allocator is as follows:

- Heap descriptor array: 24 bytes per heap for the Debug version of the library, and 12 bytes per heap for other versions of the library (subtracted from the arena after `OSInitAlloc`)
- Object headers: 32 bytes per object
- Object padding: object sizes are 32-byte aligned

Efficiency

The allocator uses a basic, double-linked free list that coalesces objects together whenever an object is freed. The computational efficiency of the allocator, in both allocation and de-allocation, is in a worst-case scenario *O(n)* in number of free fragments in the heap. In the expected case (i.e., where fragmentation does not occur too often, and objects are allocated together and de-allocated together), allocation and de-allocation should run in close to constant time.

3.4 Memory Management in C++ Code

The `new` and `delete` operators must be handled with caution when C++ code is used on the Wii console.

The `new` and `delete` operators provided by the Metrowerks Standard C++ Library (MSL C++), included with Metrowerks CodeWarrior, will be used by default. On the first invocation of the `new` operator after the application has started, a heap will be created for the `new` and `delete` operators to use. The entire MEM1 arena region will be assigned as the heap for `new` and `delete`. These operators are also implemented to call the `OSAlloc` and `OSFree` functions internally (see `$CW_FOLDER/PowerPC_EABI_Support/Runtime/Src/GCN_mem_alloc.c`).

As a result, you will run into the following complications if you use the default `new` and `delete` operators.

- The `new` and `delete` operators will only be able to use a heap allocated from the MEM1 arena region.
- It will not be possible to allocate a heap if the `new` operator is invoked without a MEM1 arena region.
- It will not be possible to allocate arena regions from MEM1 after the `new` operator has been invoked when a MEM1 arena region exists.

To avoid these problems when writing C++ code for the Wii console, you must override the `new` and `delete` operators in your application. You can use the memory management functions provided by the OS and MEM libraries within these operators.

The following demo code shows how to override the `new` and `delete` operators using the memory management functions provided by the OS library.

Code 3–7 Overriding New and Delete Operators

```
#define HEAP_ID 0

static BOOL IsHeapInitialized = FALSE;

/*---------------------------------------------------------------------------*
  Name:         CPPInit

  Description:  Initializes the Nintendo GameCube OS memory allocator and ensures that
                new and delete will work properly.

  Arguments:    None.

  Returns:      None.
 *---------------------------------------------------------------------------*/
static void CPPInit()
{
    void*    arenaLo;
    void*    arenaHi;

    if (IsHeapInitialized)
    {
        return;
    }

    arenaLo = OSGetMEM1ArenaLo();
    arenaHi = OSGetMEM1ArenaHi();

    // Create a heap
    // OSInitAlloc should only ever be invoked once.
    arenaLo = OSInitAlloc(arenaLo, arenaHi, 1); // 1 heap
    OSSetMEM1ArenaLo(arenaLo);

    // Ensure boundaries are 32B aligned
    arenaLo = (void*)OSRoundUp32B(arenaLo);
    arenaHi = (void*)OSRoundDown32B(arenaHi);

    // The boundaries given to OSCreateHeap should be 32B aligned
    OSSetCurrentHeap(OSCreateHeap(arenaLo, arenaHi));
    // From here on out, OSAlloc and OSFree behave like malloc and free
    // respectively
    OSSetMEM1ArenaLo(arenaLo=arenaHi);
    IsHeapInitialized = TRUE;

}

inline void* operator new       ( u32 blocksize )
{
    if (!IsHeapInitialized)
    {
        CPPInit();
    }
    return OSAllocFromHeap(HEAP_ID, blocksize);
}
```

```
inline void* operator new[]     ( u32 blocksize )
{
    if (!IsHeapInitialized)
    {
        CPPInit();
    }
    return OSAllocFromHeap(HEAP_ID, blocksize);
}


inline void operator delete     ( void* block )
{
    OSFreeToHeap(HEAP_ID, block);
}


inline void operator delete[]   ( void* block )
{
    OSFreeToHeap(HEAP_ID, block);
}
```

Note: The new definitions of the `new` and `delete` operators must occur before the Metrowerks Standard Libraries are included (they must be earlier on the link line than the MSL libraries). Otherwise, the Metrowerks `new` and `delete` operators will be used throughout your program.

3.5 Restrictions on Running Command Codes in the MEM2 Region

By default, it is prohibited to run command codes in the MEM2 region. You must use the following functions to allow command codes to be placed and run in the MEM2 region with the RSO library.

Code 3–8 `OSEnableCodeExecOnMEM*` API

```
void OSENableCodeExecOnMEM2Lo8MB( void );
void OSENableCodeExecOnMEM2Lo16MB( void);
```

However, even if these functions are used, only the first 8 or 16 MB of the MEM2 region will be available.

Note: These restrictions were added in version 3.2 of the SDK.

4 Error Handling and Notification

4.1 Error Display

One of the most common ways to handle errors or debug with the C standard library is to use `printf` and `assert`. The Revolution OS provides some functions that are similar to these C standard library functions.

Code 4–1 Debugging Functions

```
void    OSReport          ( const char* msg, … );
void    OSVReport         ( const char* msg, va_list list );
void    ASSERT            ( int expression );
void    ASSERTMSG         ( int expression, char* msg );
void    OSHalt            ( char* msg );
```

The `OSReport` function is the Revolution OS equivalent of the standard C `printf` function. Its output is sent to the serial port on the development hardware. The `OSVReport` function accepts an additional variable-argument list, but is otherwise the same as the `OSReport` function. The `OSReport` function in the OS library may be replaced with a separate implementation during Wii application development. The `OSReport` function is weakly defined by the OS library, so the definition for the separate implementation will take precedence over it. The following is the source code for the `OSReport` function.

Code 4–2 `OSReport` Function

```
/*---------------------------------------------------------------------------*
  Name:         OSReport()

  Description:  Outputs a formatted message into the output port

  Arguments:    msg  pointer to a null-terminated string that contains
                     the format specifications
                ...  optional arguments

  Returns:      None.
 *---------------------------------------------------------------------------*/
__declspec(weak) void OSReport(const char* msg, ...)
{
   va_list marker;

   va_start(marker, msg);
   vprintf(msg, marker);
   va_end(marker);
}

__declspec(weak) void OSVReport(const char* msg, va_list list)
{
   vprintf(msg, list);
}
```

`ASSERT` is the equivalent of the standard C `assert` macro. When the `_DEBUG` macro has been defined, `ASSERT` will display a message indicating that an assertion failed if *expression* evaluates to `FALSE`. A message that includes the text in *expression* will be sent to the output port along with the filename and line number at which `ASSERT` was invoked, and program execution will halt. `ASSERT` will do nothing if the `_DEBUG` macro has not been defined. By default, the assertion failure message will be sent to the serial port on the development hardware. When an assertion has failed, the following `OSPanic` function will be called. The `OSPanic` function in the OS library may be replaced with a separate implementation during WIi application development. The `OSPanic` function is weakly defined by the OS library, so the definition for the separate implementation will take precedence over it.

Code 4–3 `OSPanic` Function

```
/*---------------------------------------------------------------------------*
  Name:         OSPanic()

  Description:  Outputs a formatted message into the output port with the
                filename and the line number. And then halts.

  Arguments:    file  should be __FILE__
                line  should be __LINE__
                msg   pointer to a null-terminated string that contains
                      the format specifications
                ...   optional arguments

  Returns:      None.
 *---------------------------------------------------------------------------*/
__declspec(weak) void OSPanic(const char* file, int line, const char* msg, ...)
{
   va_list marker;
   u32     i;
   u32*    p;

   OSDisableInterrupts();
   va_start(marker, msg);
   vprintf(msg, marker);
   va_end(marker);
   OSReport(" in \"%s\" on line %d.\n", file, line);

   // Stack crawl
   OSReport("\nAddress: Back Chain LR Save\n");
   for (i = 0, p = (u32*) OSGetStackPointer(); // get current sp
        p && (u32) p != 0xffffffff && i++ < 16;
        p = (u32*) *p) // get caller sp
   {
      OSReport("0x%08x: 0x%08x 0x%08x\n", p, p[0], p[1]);
   }
   PPCHalt();
}
```

`ASSERTMSG` is nearly identical to `ASSERT`, but it will display the message specified by *msg* if *expression* evaluates to `FALSE` when the `_DEBUG` macro has been defined. The message will be displayed in the debugger log window along with the text in *expression* and both the filename and line number at which `ASSERTMSG` was invoked, and program execution will halt. `ASSERTMSG` will do nothing if the `_DEBUG` macro has not been defined.

`OSHalt` simply halts the program, outputting the message given by *msg*, along with the filename and line number at which `OSHalt` was invoked.

Note: These debugging functions will not output their messages on retail hardware. You can use terminal emulation software (such as TeraTerm) to confirm debugging message output from the serial port on the development console to the serial port on a PC.

Note: When a device has been connected to Memory Card slot A, output may be cancelled and the specified message will not be displayed to debugging output.

4.2 Memory Protection

The `OSProtectRange` function is used to configure access protection to some regions in internal main memory (the MEM1 region).

Code 4–4 OSProtectRange Function

```
// Capability bits
#define OS_PROTECT_CONTROL_NONE  0x00
#define OS_PROTECT_CONTROL_READ  0x01 // OK to read [addr, addr + nBytes)
#define OS_PROTECT_CONTROL_WRITE 0x02 // OK to write [addr, addr + nBytes)
#define OS_PROTECT_CONTROL_RDWR  (OS_PROTECT_CONTROL_READ | OS_PROTECT_CONTROL_WRITE)

// Error type for OSErrorHandler().
#define OS_ERROR_PROTECTION      15

// dsisr bits for memory protection error handler, which tells
// from which region the error was reported
#define OS_PROTECT0_BIT          0x00000001 // by OS_PROTECT_CHAN0 range
#define OS_PROTECT1_BIT          0x00000002 // by OS_PROTECT_CHAN1 range
#define OS_PROTECT2_BIT          0x00000004 // by OS_PROTECT_CHAN2 range
#define OS_PROTECT3_BIT          0x00000008 // by OS_PROTECT_CHAN3 range
#define OS_PROTECT_ADDRERR_BIT   0x00000010 // by [24M or 48M, 64M)

void OSProtectRange( u32 chan, void* addr, u32 nBytes, u32 control );
```

The *addr* argument will be truncated to the closest 1024-byte boundary, while the end address (*addr* + *nBytes*) will be rounded up to the nearest 1024-byte boundary. Any memory access (from the Broadway processor, the graphics processor, VI, DI, and so on) that violates the specified memory controls (`OS_PROTECT_CONTROL_*`) will cause an external interrupt from the Hollywood to the Broadway processor. When an exception occurs, the `OSProtectRange` function will call the error handler specified by the `OSSetErrorHandler` function.

Note: The `OSProtectRange` function can only be set for internal main memory (the MEM1 region) and memory-mapped I/O. This function cannot be set for external main memory (the MEM2 region).

Note: The `OSProtectRange` function flushes and invalidates the corresponding Broadway cache blocks. As a result, the Broadway processor will first perform cache line reads for load and store instructions directed at the specified region after the `OSProtectRange` function was called. These cache line reads will cause an `OS_ERROR_PROTECTION` error if the code is run with `OS_PROTECT_CONTROL_WRITE` or `OS_PROTECT_CONTROL_NONE` specified.

Code 4–5 OSSetErrorHandler Function

```
extern u32 __OSFpscrEnableBits;  // for OS_ERROR_FPE. OR-ed FPSCR_*E bits
typedef void (*OSErrorHandler)(OSError error, OSContext* context, ...);
OSErrorHandler OSSetErrorHandler(OSError error, OSErrorHandler handler);
```

The `OSSetErrorHandler` function registers a handler for the specified error type (see Table 4–1 on page 30). The error handler will catch exceptions generated by the Broadway processor.

Table 4–1 Error Types

Number	Defined Name	Description	Note
0	OS_ERROR_SYSTEM_RESET	System reset exception	
1	OS_ERROR_MACHINE_CHECK	Machine check exception	
2	OS_ERROR_DSI	DSI exception	
3	OS_ERROR_ISI	ISI exception	
4	OS_ERROR_EXTERNAL_INTERRUPT	External interrupt exception	(a)
5	OS_ERROR_ALIGNMENT	Alignment exception	
6	OS_ERROR_PROGRAM	Program exception	(b)
7	OS_ERROR_FLOATING_POINT	Unusable floating-point exception	(a)
8	OS_ERROR_DECREMENTER	Decrementer exception	(a)
9	OS_ERROR_SYSTEM_CALL	System call exception	(a)
10	OS_ERROR_TRACE	Trace exception	(b)
11	OS_ERROR_PERFORMANCE_MONITOR	Performance monitor exception	
12	OS_ERROR_BREAKPOINT	Instruction address break point exception	(b)
13	OS_ERROR_SYSTEM_INTERRUPT	System management interrupt exception	
14	OS_ERROR_THERMAL_INTERRUPT	Temperature interrupt exception	
15	OS_ERROR_PROTECTION	Memory protection error	
16	OS_ERROR_FPE	Floating-point exception	

All errors from `OS_ERROR_SYSTEM_RESET` to `OS_ERROR_SYSTEM_INTERRUPT` are caused by exceptions generated by the Broadway processor. For these errors, `OSErrorHandler` takes as its third and fourth arguments *dsisr* and *dar*, which are of type `u32`.

The *error* argument will indicate the error number that was generated. The *context* argument will contain the Broadway register values, except for the FPU register context, at the time the error occurred. The FPU registers are managed by the operating system to reduce context switch overhead, and are therefore not manipulated by the error handler. The *dsisr* and *dar* arguments will contain the Broadway register values when the error occurred.

Note (a): Do not set error handlers for `OS_ERROR_EXTERNAL_INTERRUPT`, `OS_ERROR_FLOATING_POINT`, `OS_ERROR_DECREMENTER`, and `OS_ERROR_SYSTEM_CALL`. These are used by the operating system.

Note (b): Do not set error handlers for `OS_ERROR_PROGRAM`, `OS_ERROR_TRACE`, and `OS_ERROR_BREAKPOINT` when using the debugger. These are used by the debugging kernel.

An `OS_ERROR_PROTECTION` error will occur if the Broadway processor manipulates the memory range between the last main memory address and 64 MB, or if the Hollywood chip detects a memory access that violates the settings specified by the `OSProtectRange` function. The *error* argument will indicate the error number that was generated (`OS_ERROR_PROTECTION`). The *context* argument will contain the Broadway register values, except for the FPU register context, at the time that the protection error notification was sent. An `OS_ERROR_PROTECTION` notification is sent asynchronously from the Hollywood to the Broadway processor, so *context* will not contain the register context that existed when the memory protection

exception occurred. The FPU registers are managed by the operating system to reduce context switch overhead and are therefore not manipulated by the error handler. The *dar* argument will contain the physical memory address at which the memory protection violation occurred. The *dsisr* argument will contain either OS_PROTECT_ADDRERR_BIT or one of the OS_PROTECT* values.

The OS_ERROR_FPE error will only occur when an error handler has been set by OSSetErrorHandler. Setting an error handler for OS_ERROR_FPE will allow floating-point exceptions, such as a division by zero, to be caught at runtime. The *error* argument will indicate the error number that was generated (OS_ERROR_FPE). The *context* argument will contain the Broadway register values, including the FPU register context, at the time that the floating-point exception notification was sent. The error handler can manipulate the FPU registers; when it is finished, the FPU registers will be applied to the thread that raised the exception. The *dsisr* and *dar* arguments will contain the Broadway register values when the error occurred. The *context->fpscr* bits will indicate the following types of floating-point exceptions.

Table 4–2 fpscr Bits

fpscr Bits	Description
FPSCR_VX	Invalid operation exception
FPSCR_OX	Overflow exception
FPSCR_UX	Underflow exception
FPSCR_ZX	Divide-by-zero exception
FPSCR_XX	Inaccuracy exception

To enable precise floating-point exceptions in the Broadway processor, the OSSetErrorHandler function will set the MSR[FE0] and MSR[FE1] bits in every thread, including threads that are created later, when the error handler is configured. Each thread's *fpscr* bits will be initialized, as well, by taking the bitwise OR of the __OSFpscrEnableBits global variable. By default, __OSFpscrEnableBits will set the VE, OE, UE, ZE, and XE bits (all bits will be enabled). You can set __OSFpscrEnableBits to a bitwise OR of the following valid bits to control the types of floating-point exceptions that will cause an OS_ERROR_FPE error.

Table 4–3 __OSFpscrEnableBits Bits

__OSFpscrEnableBits Bit	Description
FPSCR_VE	Enables invalid operation exceptions
FPSCR_OE	Enables overflow exceptions
FPSCR_UE	Enables underflow exceptions
FPSCR_ZE	Enables divide-by-zero exceptions
FPSCR_XE	Enables innacuracy exceptions

Note: Once the OS_ERROR_FPE error handler has been set, do not change the value of __OSFpscrEnableBits until the OS_ERROR_FPE handler has been removed.

You can find a demo program that shows how to use the OS_ERROR_FPE handler at the following location.

```
$REVOLUTION_SDK_ROOT/build/demos/osdemo/src/fpe.c
```

When the OS is initialized, a default error handler is registered for all error types that send an error message to serial output. As shown in the following example, the default error handler will send the cause of the exception, the exception number, registers, and a stack trace to serial output. Games can register an error handler to overwrite the default handler, as necessary, except for items mentioned in notes (a) and (b) from Table 4–1 on page 30.

```
Unhandled Exception 15(Protection error)
------------------------- Context 0x80182c98 -------------------------
r0 = 0x8001671c ( -2147391716) r16 = 0x00000000 (          0)
r1 = 0x801f7a50 ( -2145420720) r17 = 0x00000000 (          0)
r2 = 0x801f0920 ( -2145449696) r18 = 0x00000000 (          0)
r3 = 0x80281670 ( -2144856464) r19 = 0x00000000 (          0)
r4 = 0x801f7c48 ( -2145420216) r20 = 0x00000000 (          0)
r5 = 0x80255208 ( -2145037816) r21 = 0x80255208 ( -2145037816)
r6 = 0x802551a0 ( -2145037920) r22 = 0x802551a0 ( -2145037920)
r7 = 0x00000001 (          1) r23 = 0x801f7c48 ( -2145420216)
r8 = 0x80140000 ( -2146172928) r24 = 0x00000000 (          0)
r9 = 0x00000018 (         24) r25 = 0x00000000 (          0)
r10 = 0x801f7bb0 ( -2145420368) r26 = 0x801f7c48 ( -2145420216)
r11 = 0x801f7b70 ( -2145420432) r27 = 0x80281c1c ( -2144855012)
r12 = 0x80018078 ( -2147385224) r28 = 0x00000001 (          1)
r13 = 0x801ef720 ( -2145454304) r29 = 0x80281670 ( -2144856464)
r14 = 0x00000000 (          0) r30 = 0x80281670 ( -2144856464)
r15 = 0x00000000 (          0) r31 = 0x801f7a50 ( -2145420720)
LR = 0x800180ac               CR = 0x84000088
SRR0 = 0x800180c8             SRR1 = 0x0000b032

GQRs----------
gqr0 = 0x00000000    gqr4 = 0x00060006
gqr1 = 0x00000000    gqr5 = 0x00070007
gqr2 = 0x00040004    gqr6 = 0x05070507
gqr3 = 0x00050005    gqr7 = 0x08070807

FPRs----------
fr0 = 255        fr1 = 0
fr2 = 0        fr3 = 0
fr4 = 255        fr5 = 0
fr6 = 0        fr7 = 0
fr8 = 0        fr9 = 0
fr10 = 0       fr11 = 0
fr12 = 0       fr13 = 0
fr14 = 0       fr15 = 0
fr16 = 0       fr17 = 0
fr18 = 0       fr19 = 0
fr20 = 0       fr21 = 0
fr22 = 0       fr23 = 0
fr24 = 0       fr25 = 0
fr26 = 0       fr27 = 0
fr28 = 0       fr29 = 0
fr30 = 0       fr31 = 0

PSFs----------
ps0 = 0xffffffff    ps1 = 0x0
ps2 = 0x0       ps3 = 0x0
ps4 = 0xffffffff    ps5 = 0x0
ps6 = 0x0       ps7 = 0x0
ps8 = 0x0       ps9 = 0x0
ps10 = 0x0       ps11 = 0x0
ps12 = 0x0       ps13 = 0x0
ps14 = 0x0       ps15 = 0x0
ps16 = 0x0       ps17 = 0x0
ps18 = 0x0       ps19 = 0x0
ps20 = 0x0       ps21 = 0x0
```

```
ps22 = 0x0       ps23 = 0x0
ps24 = 0x0       ps25 = 0x0
ps26 = 0x0       ps27 = 0x0
ps28 = 0x0       ps29 = 0x0
ps30 = 0x0       ps31 = 0x0

Address:     Back Chain   LR Save
0x801f7a50:  0x801f7bb0   0x80018bac
0x801f7bb0:  0x801f7c20   0x8001671c
0x801f7c20:  0x801f7ca0   0x80016278
0x801f7ca0:  0x801f7e90   0x800088d8
0x801f7e90:  0x801f7ec0   0x80013c60
0x801f7ec0:  0x801f95a0   0x800068a8
0x801f95a0:  0x801f95b0   0x8000fd88
0x801f95b0:  0xffffffff   0x800041bc

DSISR = 0x00000010               DAR = 0x03225000
TB = 0x0035b3f2f7531eed

AI DMA Address = 0x00000000
ARAM DMA Address = 0x01000020
DI DMA Address = 0x00000000

Last interrupt (25): SRR0 = 0x800180c8 TB = 0x0035b3f2f7531c91
```

The information in the error output above can be described as follows.

- `Unhandled Exception 15(Protection error)`: The exception that was sent as a notification from the protection unit.
- `Context 0x80182c98`: The address at which the context (`OSContext` structure) is stored.
- `r0–SRR1`: Status of the Broadway general-purpose registers.
- `gqr0–gqr7`: Status of the Broadway GQR registers.
- `fr0–fr31`: Status of the Broadway floating-point registers.
- `ps0–ps31`: Values of the Broadway paired-single registers.
- `Address:  Back Chain  LR Save`: Stack trace.
- `DSISR`: Cause of the exception (`0x00000010` is equal to `OS_PROTECT_ADDRERR_BIT`, and indicates that memory access exceeded 64 MB)
- `DAR`: Address at which data was accessed.
- `TB`: Value of the Time Base register (the value obtained from the `OSGetTime` function).
- `AI DMA Address`: AI DMA address.
- `ARAM DMA Address`: ARAM DMA address.
- `DI DMA Address`: DI DMA address.
- `Last interrupt`: The value of the MSR and Time Base register the last time an interrupt occurred.

5 Cache Control

5.1 Cache Description

The Broadway CPU has separate instruction and data L1 caches, each 32 KB, and one unified L2 cache of 256 KB. Applications do *not* need to pay attention to L2 cache coherency, which is preserved in hardware using bus snooping. However, applications *do* need to preserve L1 cache coherency. The following are situations in which applications must pay attention to cache coherency.

- DMA-related memory operations

  Exercise caution with the cache coherency of memory used for DMA operations within functions, such as those in the DVD, AI, and VI libraries.

- GX FIFO

  The GX FIFO is sent through the Broadway write-gather pipe. As a result, be careful when the CPU accesses memory allocated to the FIFO.

- External framebuffer (XFB) operations from the CPU

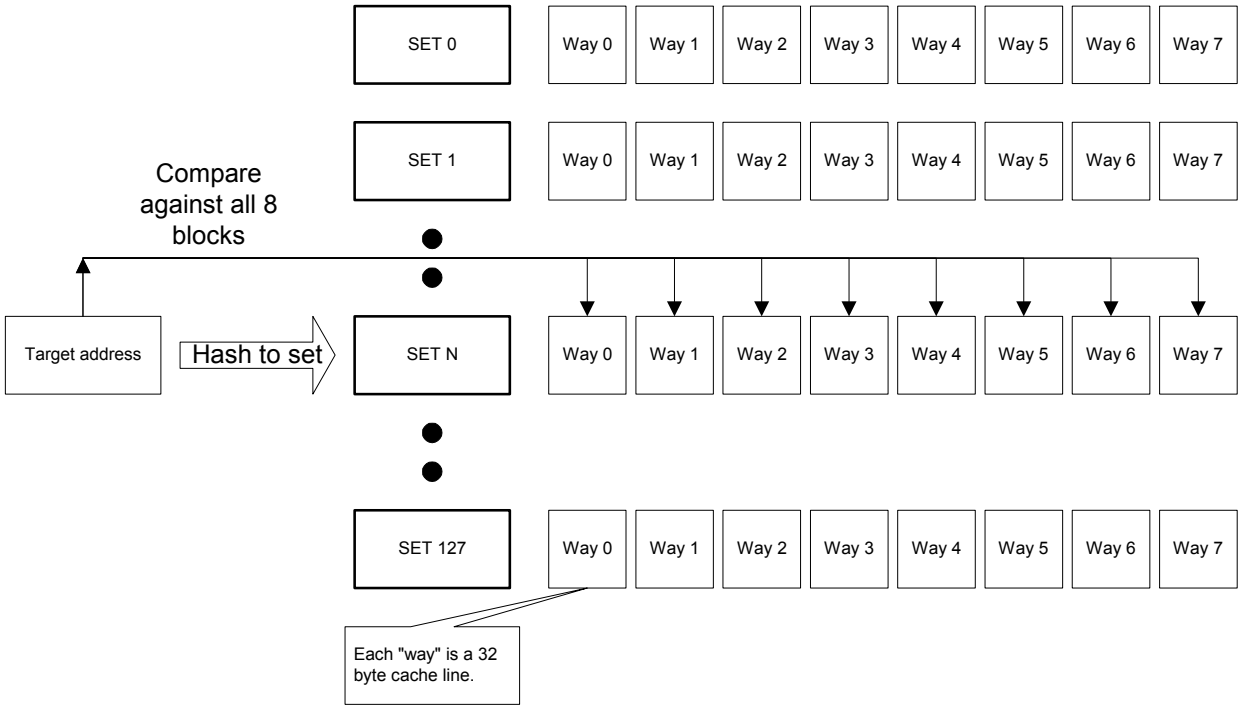
  The VI library accesses main memory directly, so the XFB must be treated with caution.

- Texture creation and manipulation from the CPU

The I/D L1 caches are *8-way set-associative*. There are 128 *sets*, each consisting of 8 *ways*. Each way, or cache line, is 32 bytes. This effectively means that up to 8 cache lines can hash to the same cache set before a cache line in the set is cast out.

After `OSInit`, the caches are initialized to be *enabled* and *write-back*. This means that data stores are held in the cache until the cache line is written out to physical memory, typically when the line is cast out. Thus, the information in the caches is not consistent with physical memory. Ordinarily, this is only an issue if an external device or processor needs the most up-to-date data. The APIs in this section help deal with cache coherency.

Figure 5–1 Normal Cache Lookup



Target address is hashed to a set, then compared with all 8 ways

5.2 Cache Incoherence

The following code sequence demonstrates how memory can be viewed differently by cached and uncached access:

Code 5–1 Cached and Uncached Access to Memory

```
#include <revolution.h>

void main (void)
{
u32* cachedAddress              = (u32*)OSGetMEM1ArenaLo();
u32* uncachedAddress            = OSCachedToUncached(cachedAddress);

        *cachedAddress          = 42;
        *uncachedAddress        = 24;
}
```

Even though both `cachedAddress` and `uncachedAddress` reference the same physical location in memory, they give different values. If you print each of the pointer values, the `cachedAddress` variable will give the copy that is in the cache, while the `uncachedAddress` variable will give the copy that is in physical memory.

The following APIs provide basic address translation between the cached and uncached address segments:

Code 5–2 Address Translation

```
void* OSCachedToUncached( const void* addr );
void* OSUncachedToCached( const void* addr );
```

5.3 Basic Cache Management

To ensure that all devices and processors are viewing the same data in physical memory, it may be necessary to force the cache to write back data to physical memory on demand, or perhaps to update its local copy of data. Three kinds of operations are available on ranges of memory that may be residing in the cache:

- **Store** – Performing a “store” operation on a range of memory forces the cache to write back any modified data in lines that correspond to that range.

  Note: The data in the cache will remain there.

- **Invalidate** – Performing an “invalidate” operation on a range of memory forces the cache to immediately discard any cache lines corresponding to that memory range. In effect, the data range no longer resides in the cache. This is a destructive operation—any modified data in the cache will be lost. Subsequent accesses to this memory range will reload the data from physical memory into the cache.
- **Flush** – Performing a “flush” operation is similar to a store + invalidate operation. The cache writes back any lines that correspond to that range (if modified) and invalidates the corresponding cache lines. Subsequent accesses to this memory range will fault in the cache lines.

The following example demonstrates the different effects of storing or invalidating cache lines:

Code 5–3 Storing and Invalidating Cache Lines

```
#include <revolution.h>

void main (void)
{
    u32* cachedAddress;
    u32* uncachedAddress;

    cachedAddress   = (u32*)OSGetMEM1ArenaLo();
    uncachedAddress = OSCachedToUncached(cachedAddress);

    OSReport("OS OVERVIEW - CACHE MANAGEMENT DEMO\n");
    OSReport("STORE EXAMPLE\n");
    *cachedAddress    = 0xFFFF;
    *uncachedAddress  = 0xAAAA;

    OSReport("Cache copy           = 0x%x\n", *cachedAddress);
    OSReport("Physical memory copy = 0x%x\n", *uncachedAddress);

    DCStoreRange(cachedAddress, sizeof(u32));

    OSReport("After STORE, Cache copy           = 0x%x\n",
             *cachedAddress);
    OSReport("After STORE, Physical memory copy = 0x%x\n",
             *uncachedAddress);

    OSReport("\nINVALIDATE EXAMPLE\n");
    *cachedAddress    = 0xFFFF;
    *uncachedAddress  = 0xAAAA;

    OSReport("Cache copy           = 0x%x\n", *cachedAddress);
    OSReport("Physical memory copy = 0x%x\n", *uncachedAddress);

    DCInvalidateRange(cachedAddress, sizeof(u32));

    OSReport("After INVALIDATE, Cache copy           = 0x%x\n",
             *cachedAddress);
    OSReport("After INVALIDATE, Physical memory copy = 0x%x\n",
             *uncachedAddress);

    OSHalt("Demo complete");
}
```

The "DC" prefix stands for "data cache." (An "IC" prefix in other APIs refers to "instruction cache"-related functions.) The output of the preceding program is:

Code 5–4 Store and Invalidate Output Sample

```
STORE EXAMPLE
Cache copy           = 0xFFFF
Physical memory copy = 0xAAAA
After STORE, Cache copy           = 0xFFFF
After STORE, Physical memory copy = 0xFFFF

INVALIDATE EXAMPLE
Cache copy           = 0xFFFF
Physical memory copy = 0xAAAA
After INVALIDATE, Cache copy           = 0xAAAA
After INVALIDATE, Physical memory copy = 0xAAAA
```

The relevant APIs for both instruction and data cache flushing are:

Code 5–5 Data and Instruction Cache APIs

```
void    DCFlushRange            ( void* startAddr, u32 nBytes );
void    DCFlushRangeNoSync      ( void* startAddr, u32 nBytes );
void    DCInvalidateRange       ( void* startAddr, u32 nBytes );
void    DCStoreRange            ( void* startAddr, u32 nBytes );
void    DCStoreRangeNoSync      ( void* startAddr, u32 nBytes );
void    DCZeroRange             ( void* startAddr, u32 nBytes);
void    ICInvalidateRange       ( void* startAddr, u32 nBytes );

void    PPCSync                 ( void );
```

In each of these routines, *startAddr* will be rounded down to the closest 32-byte boundary, while the end address (*startAddr* + *nBytes*) will be rounded up to the closest 32-byte boundary. This is to align the region to cache line boundaries.

`DCStoreRange` attempts to write back memory in *nBytes* starting from *startAddr*. Only modified data in the cache within that range will be written back.

Note: This function will perform a `PPCSync` operation. That is, it will stall the CPU until all data has been flushed to main memory. To avoid this overhead (for example, if you wish to store multiple, non-contiguous ranges), use `DCStoreRangeNoSync`.

`DCStoreRangeNoSync` behaves identically to `DCStoreRange`, except that it does not perform a `PPCSync` operation. Therefore, it is not guaranteed that any modified data has been sent to main memory until you perform a `PPCSync` (or call `DCStoreRange`).

`DCZeroRange` will clear the cache blocks associated with the *nBytes* of memory starting from *startAddr*. All bytes in the range will be zeroed. If the caches are disabled or if the addresses are marked uncached, an alignment exception will be generated.

`DCInvalidateRange` attempts to invalidate *nBytes* of memory, starting from *startAddr*. All lines within that range will be discarded.

`DCFlushRange` flushes *nBytes* of memory, starting from *startAddr*. This is effectively the same as calling `DCStoreRange`, followed by `DCInvalidateRange`, but it is more efficient. `DCFlushRange` performs a `PPCSync` operation, so it will not return until any modified data has been written to memory. To avoid this overhead (e.g. if you want to flush multiple, non-contiguous ranges), use `DCFlushRangeNoSync`.

`DCFlushRangeNoSync` behaves identically to `DCFlushRange`, except that it does not perform a `PPCSync` operation. Therefore, it is not guaranteed that any modified data has been sent to main memory until you perform a `PPCSync` (or call `DCFlushRange`).

`PPCSync` is not a cache control API, per se. It flushes all pending I/O traffic from the CPU to main memory, and is thus necessary whenever the application requires that memory be updated appropriately. For instance, after flushing data from the cache for use by the Graphics Processor (GP), `PPCSync` may be needed to ensure that the data is really in main memory before the GP accesses it. `PPCSync` can take a significant amount of time, so indiscriminate use of this function may adversely affect performance.

5.3.1 Efficiency

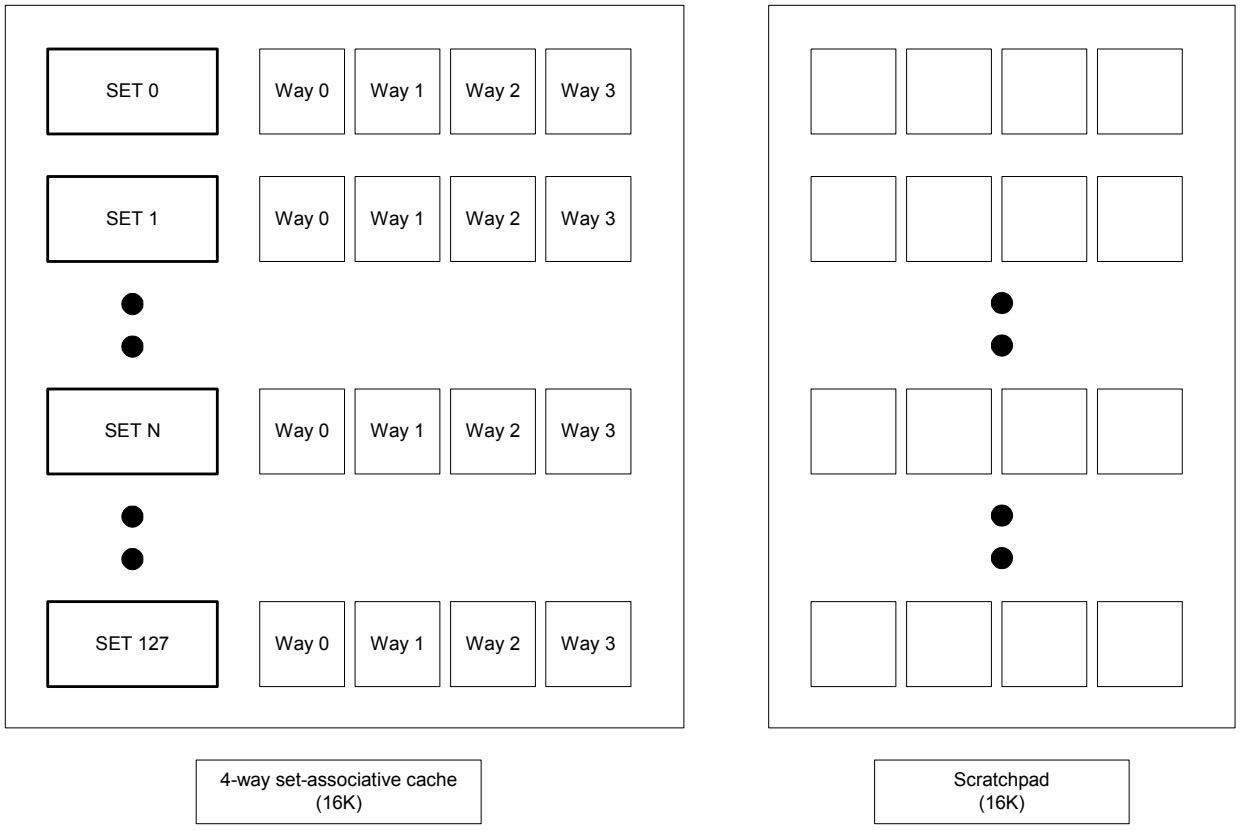
Cache operations map almost directly to the low-level instructions that manipulate the cache at the cache line level. Thus, all operations complete in *O(n)* time in the number of cache lines in the range.

5.4 Locked Cache Operation

One of the key features of the Broadway CPU is its ability to use the L1 data cache in one of two modes:

- **Normal** mode functions as a 32 KB 8-way set-associative cache.
- **Locked Cache** mode partitions into a 16 KB four-way set-associative cache, and a 16 KB scratchpad buffer with DMA engine.

Figure 5–2 L1 Data Cache Configured in Locked Cache Mode



The locked cache allows applications to manage the cache explicitly and efficiently as if it were a scratchpad buffer.

5.4.1 Locked Cache API Overview

The locked cache API presents the locked cache memory region as a piece of scratchpad memory. By default, the locked cache is disabled and no scratchpad region is set. A call to `LCEnable` enables the locked cache, and a pointer to the scratchpad region can be retrieved with the `LCGetBase` function. The OS maps the locked cache to a region outside of physical memory so that no real main memory is wasted.

The locked cache is particularly useful in places that require improved performance, but—because it takes time to enable and/or disable the locked cache—on the Wii console, we recommend leaving the locked cache enabled throughout the execution of game, although it is primarily useful where code must be as efficient as possible. The programming model for using the locked cache requires that the program process data using multiple buffers. As one buffer's contents are being computed, the other buffer(s) are DMA-ed in and out. This model is demonstrated in the locked cache demo programs.

LCEnable and LCDisable respectively enable or disable the locked cache at runtime. However, the costs for doing so are non-trivial and should be weighed carefully against the benefits of having a full 32KB L1 data cache for part of the game's execution. "5.4.6 Enabling and Disabling the Locked Cache" on page 44 provides an estimate of the cycle cost of this API.

5.4.2 DMA Engine

The Broadway CPU uses a DMA engine to move data into and out of the locked cache partition efficiently. The DMA engine has a queue of up to 15 outstanding transactions. Each transaction can transfer up to 128 cache blocks (4 KB).

The locked cache API has high-level "load data" and "store data" functions (LCLoadData and LCStoreData) that will break up a requested transfer size into the appropriate transactions. It also provides lower-level functions (LCLoadBlocks and LCStoreBlocks) that work on the granularity of a single DMA transaction. These latter functions have the advantage of lower overhead, but they impose restrictions on arguments and maximum transfer size, and perform no error checking.

The DMA transactions occur asynchronously, and there is no way to receive an interrupt or event from the chip, indicating that any transactions have completed. As a result, the program must poll the length of the transaction queue. We provide a simple and fast routine, LCQueueWait, to stall the CPU until the transaction queue reaches a certain length. As you will see in the locked cache demos, it is easy to analyze how many outstanding DMA transactions you must wait for—in the current paradigm, each buffer has, at most, one store request and one load request pending. Note, however, that the number of DMA transactions making up a store or load request depends on the size of the request.

5.4.3 Basic Locked Cache API and Demos

The following demo programs perform an operation, ProcessBuf, on blocks of a large array, and then commit those changes back to main memory. Two locked cache regions are used; while one is being processed, the other is storing older processed data to memory, and also loading to the next buffer.

Code 5–6 Basic Locked Cache Demo

```
// define 2 8k buffers in locked cache region
// NOTE: NUMBUFFERS * BUFFER_SIZE <= 16k
#define BUFFER_SIZE (8*1024)
#define NUM_BUFFERS (2)

#define DATA_ELEMENTS (10*1024*1024)
:

// real mem loc of Buffers[i] is at BufAddr[i]
u8* Buffers[NUM_BUFFERS];
u8* BufAddr[NUM_BUFFERS];
:

void main ()
{
    u8* data;
    u8* currDataPtr; // offset into data
    u32 i;
    void* arenaLo;
    void* arenaHi;
    u32 numTransactions;

    LCEnable();

    arenaLo = OSGetMEM1ArenaLo();
    arenaHi = OSGetMEM1ArenaHi();
    :
    OSReport("Splitting locked cache into %d buffers\n", NUM_BUFFERS);
```

```
    for (i = 0; i < NUM_BUFFERS; i++)
    {
        Buffers[i] = (u8*) ((u32)LCGetBase() + BUFFER_SIZE*i);
        OSReport("Locked Cache : Allocated %d bytes at 0x%x\n",
                 BUFFER_SIZE,
                 Buffers[i]);
    }

    // Initialize source data
    data = (u8*)OSAlloc(DATA_ELEMENTS * sizeof(u8));
    :
    DCFlushRange(data, DATA_ELEMENTS);

    OSReport(" Test 1 : using high level interface for DMA load/store \n");

    for (i = 0; i < NUM_BUFFERS; i++)
    {
        BufAddr[i] = data + BUFFER_SIZE*i;
        numTransactions = LCLoadData(Buffers[i], BufAddr[i], BUFFER_SIZE);
    }

    currDataPtr = data + BUFFER_SIZE * NUM_BUFFERS;

    LCQueueWait((NUM_BUFFERS-1) * 4);

    while (currDataPtr <= data+DATA_ELEMENTS)
    {
        for (i = 0; i < NUM_BUFFERS; i++)
        {
            LCQueueWait((NUM_BUFFERS-1)*numTransactions); // prevstore + prevload, each takes 2
            ProcessBuf(Buffers[i]);
            LCStoreData(BufAddr[i], Buffers[i], BUFFER_SIZE);
            BufAddr[i] = currDataPtr; // move to next unprocessed buffer
            LCLoadData(Buffers[i], BufAddr[i], BUFFER_SIZE);
            // advance the next block to be read
            currDataPtr += BUFFER_SIZE;
        }
    }
    LCQueueWait(numTransactions); // don't care about last dma's
    :

    OSHalt("Test complete");
}
```

The code uses *currDataPtr* to keep track of the next real data that should be processed. The `Buffers` array points to the locked cache buffers, and the `BufAddr` array points to the actual physical buffers in main memory that are being mirrored. Thus, the buffer in `Buffers[1]` should be stored out to `BufAddr[1]`.

Because we are using 8KB buffers, one single DMA transaction is insufficient to perform a full load or store. Therefore, we remember the number of DMA transactions created by each request in *numTransactions*, and we know exactly how many entries can be in the DMA queue at any time. In this multi-buffer case, we know that we are ready to process the next buffer whenever there is at most one load and one store pending for all the other buffers.

The locked cache API calls used in this demo are listed here:

Code 5–7 Basic Locked Cache Functions

```
void  LCEnable          ( void );
void* LCGetBase         ( void );
u32   LCLoadData        ( void* destAddr, void* srcAddr, u32 nBytes );
u32   LCStoreData       ( void* destAddr, void* srcAddr, u32 nBytes );
void  LCQueueWait       ( u32 len );
```

`LCEnable` turns on the locked cache facility. By default, the L1 data cache is configured as a 32KB eight-way set-associative cache. See "5.4.6 Enabling and Disabling the Locked Cache" on page 44 for a more detailed description of this function.

`LCGetBase` returns the base address of the locked cache region. Addresses from `LCGetBase()` to `LCGetBase()+16KB` will hit in the locked cache region.

`LCLoadData` queues the DMA transactions needed to load data into the locked cache at *destAddr* from main memory at *srcAddr*. The number of issued transactions is returned. All arguments should be 32-byte aligned. If the memory region specified by *destAddr* is not in the locked cache, a machine check will occur when the DMA engine processes the transaction request.

`LCStoreData` queues the DMA transactions needed to send data from the locked cache at *srcAddr* to main memory at *destAddr*. The number of issued transactions is returned. All arguments should be 32-byte aligned. If the memory region specified by *srcAddr* is not in the locked cache, a machine check will occur when the DMA engine processes the transaction request.

`LCQueueWait` polls the DMA queue length until it is less than or equal to *len*.

5.4.4 Low-Level Locked Cache API and Demos

The next demo uses four 4KB buffers and demonstrates the use of the low-level locked cache API.

Code 5–8 Low-Level Locked Cache Demo

```
// define 4 4k buffers in locked cache region
// NOTE: NUMBUFFERS * BUFFER_SIZE <= 16k
#define BUFFER_SIZE (4*1024)
#define NUM_BUFFERS (4)

#define DATA_ELEMENTS (10*1024*1024)
:

// real mem loc of Buffers[i] is at BufAddr[i]
u8* Buffers[NUM_BUFFERS];
u8* BufAddr[NUM_BUFFERS];
:

void main ()
{
    u8* data;
    u8* currDataPtr; // offset into data
    u32 i;
    void* arenaLo;
    void* arenaHi;

    LCEnable();

    arenaLo = OSGetMEM1ArenaLo();
    arenaHi = OSGetMEM1ArenaHi();
    :

    OSReport("Splitting locked cache into %d buffers\n", NUM_BUFFERS);
```

```
    for (i = 0; i < NUM_BUFFERS; i++)
    {
        Buffers[i] = (u8*) ((u32)LCGetBase() + BUFFER_SIZE*i);
        OSReport("Locked Cache : Allocated %d bytes at 0x%x\n",
                    BUFFER_SIZE,
                    Buffers[i]);
    }

    // Initialize source data
    data = (u8*)OSAlloc(DATA_ELEMENTS * sizeof(u8));
    :
    DCFlushRange(data, DATA_ELEMENTS);

    OSReport(" Test 2 : using low level interface for DMA load/store \n");

    for (i = 0; i < NUM_BUFFERS; i++)
    {
        BufAddr[i] = data + BUFFER_SIZE*i;
        LCLoadBlocks(Buffers[i], BufAddr[i], 0);
    }

    currDataPtr = data + BUFFER_SIZE * NUM_BUFFERS;

    LCQueueWait((NUM_BUFFERS-1));

    while (currDataPtr <= data+DATA_ELEMENTS)
    {
        for (i = 0; i < NUM_BUFFERS; i++)
        {
            LCQueueWait((NUM_BUFFERS-1)*2);
            ProcessBuf(Buffers[i]);
            LCStoreBlocks(BufAddr[i], Buffers[i], 0);
            LCLoadBlocks(Buffers[i], currDataPtr, 0);
            BufAddr[i] = currDataPtr; // move to next unprocessed buffer
            // advance the next block to be read
            currDataPtr += BUFFER_SIZE;
        }
    }
    LCQueueWait(NUM_BUFFERS); // don't care about last dma's
    :
    OSHalt("Test complete");
}
```

This example looks remarkably like the first locked cache demo, except that it uses four 4KB buffers because these fit in exactly one DMA transaction.

Note: The value used for `LCQueueWait` is *(NUM_BUFFERS-1)*2*. This means we know that there can be two transactions (a store and a load) for each buffer, and that we will want to proceed as soon as the two oldest transactions (i.e., the store and load for the buffer we want to use) have been committed.

The functions used in this code are detailed here:

Code 5–9 Low-Level Locked Cache Load/Store Functions

```
void LCLoadBlocks   ( void* destTag,  void* srcAddr, u32 numBlocks );
void LCStoreBlocks  ( void* destAddr, void* srcTag,  u32 numBlocks );
```

`LCLoadBlocks` queues a single DMA transaction to load data into the locked cache at *destTag* from main memory at *srcAddr*. This function imposes a maximum transaction limit of 128 cache blocks (4KB), and performs no error checking. Addresses are assumed to be 32-byte aligned, and *numBlocks* is assumed to be between 0-127.

Note: A value of 0 for numBlocks implies a transaction size of 128 blocks. There is no error checking to ensure that the arguments conform to these requirements.

`LCStoreBlocks` queues a single DMA transaction for moving data in the locked cache to main memory. This function has the same argument restrictions as `LCLoadBlocks`.

5.4.5 Additional Locked Cache Functions

There are a few more locked cache functions not used in these demos:

Code 5–10 Additional Locked Cache Functions

```
void LCDisable      ( void );
u32  LCQueueLength  ( void );
void LCFlushQueue   ( void );
```

`LCEnable` and `LCDisable` respectively enable or disable the locked cache mode. By default, the locked cache is disabled at boot time. See "5.4.6 Enabling and Disabling the Locked Cache" on page 44 for more details.

`LCQueueLength` returns the current length of the DMA queue. It performs a `sync` instruction that flushes all current memory transactions and the execution queue to ensure that the queue length value is accurate. It is more efficient to poll the queue length with `LCQueueWait`.

`LCFlushQueue` simply flushes the DMA queue and issues a `sync` instruction to wait until all active DMA transactions are committed.

5.4.6 Enabling and Disabling the Locked Cache

The locked cache is disabled by default. Enabling the locked cache is a relatively lengthy process. First, the cache must be flushed entirely, as any modified data trapped in the locked cache will be lost. Since there are no instructions that do this automatically, `LCEnable` touches and stores a 32KB region of the address space. Touching forces existing lines back to memory, while storing ensures that any modified lines that were already in the cache are flushed back.

After that, the locked cache can be enabled, and cache tags must be allocated for the scratchpad partition. `LCEnable` will disable interrupts for this entire sequence to ensure that the cache is not contaminated. Our current cycle counts indicate that all of this work can take between 15,000 to 19,510 cycles (i.e., 20.5 to 26.8 microseconds), depending on how much modified data in the cache must be flushed to main memory.

Note: While enabling the locked cache, `LCEnable` will also use `DBAT3` (data block address translation register) to map in the addresses that it assigns to the locked cache.

Disabling the locked cache involves invalidating all of the scratchpad memory addresses to prevent them from being cast out, as they are not backed by physical memory. This can take approximately 2000 cycles (2.8 microseconds).

Our measurements of the end-to-end overhead of enabling and disabling the locked cache show this to cost between 17,000 and 22,000 cycles (23.3 – 30.2 microseconds).

Note: There are second-order effects not reflected in these measurements. For instance, after `LCEnable` has been called, the entire L1 data cache will have been flushed and subsequent memory accesses will miss. In addition, interrupts are disabled during `LCEnable`, which can delay interrupt handling.

In any case, make sure that the benefits of having a full 32KB L1 data cache for part of a game's execution outweigh the costs of disabling and enabling the locked cache.

5.4.7 Dangers in Using the Locked Cache

Because the Revolution OS maps the locked cache addresses outside of the physical memory space (i.e., addresses are not backed by physical memory), there are certain dangers that go with using the locked cache.

First, if your application oversteps the boundaries of the locked cache region, you will receive an interrupt from the processor interface (PI) indicating that you have attempted to access invalid physical memory. Because your only notification is an asynchronous interrupt, there is no way to know exactly which part of your code caused the errant access, although the error message will alert you to the bad address you attempted to access.

Second, if you write extremely tight loops, the branch prediction hardware on the CPU may cause parts of a loop to be executed beyond the boundaries that you have set. For instance, consider the following simple loop:

Code 5–11 Dangerous Loop in the Locked Cache

```
void ProcessBuf(u8* buffer)
{
    u32 i;

    for (i = 0; i < BUFFER_SIZE; i++)
    {
        buffer[i] = (u8)(buffer[i] + (u8)0xA);
    }
}
```

There is a chance that the CPU will execute the loop one more time than you expect. The results will not actually be committed because the CPU will realize that it predicted the branch incorrectly. However, this loop may cause the CPU to fetch the data at `buffer[BUFFER_SIZE]`, even though the loop says to stop when *i* reaches `BUFFER_SIZE`. In cases where the `buffer` array extends to the end of the locked cache, you will receive an interrupt from the PI because `buffer[BUFFER_SIZE]` is 1 byte outside the locked cache region.

Note: Because the processor can execute instructions on a predicted branch only speculatively, it will not actually execute any stores or update any registers.

There are two potential solutions to this problem:

- Always pad the ends of your buffers in the locked cache (i.e., never use the last byte of the locked cache). This way, spurious loads at the ends of your loops will be handled safely.
- Pad the beginning of your loop with instructions that do not load from the locked cache. Although your code will probably look like that already, it is important to be aware of this problem.

6 Time

The Revolution OS has several time-related features.

- **Battery-backed clock**: Records the current date and time. The battery-backed clock will not be managed by the application.
- **Real-time clock**: Based on the Broadway's 64-bit time base register, it increments about every 16.5 nanoseconds (12 CPU cycles)
- **Alarm**: 64-bit one-short/periodic alarm with 16-nanosecond accuracy

6.1 Real-Time Clock

The Broadway CPU has a 64-bit time base register. This time base register increments every 12 CPU cycles, or about 16.5 nanoseconds. At boot time, the OS initializes this time base register to the number of ticks since 0:00 AM January 1, 2000.

The Revolution OS only provides APIs to read the time base register. The time base register must not be modified by an application program, since several system libraries, including OS and audio, reference the time base register for implementing time-critical code.

The Revolution OS provides two time types. The basic units of time are `OSTime` and `OSTick`. An `OSTime` value is a signed 64-bit value, while an `OSTick` value is an unsigned 32-bit value.

The APIs to get the time base register value are as follows:

Code 6–1 Time APIs

```
OSTick OSGetTick            ( void );
OSTime OSGetTime            ( void );

#define OSDiffTick(tick1, tick0)        …

// the following macros can actually act on OSTime (64-bit) values as well
#define OSTicksToSeconds( ticks )       …
#define OSTicksToMilliseconds( ticks )  …
#define OSTicksToMicroseconds( ticks )  …
#define OSSecondsToTicks( sec )         …
#define OSMillisecondsToTicks( msec )   …
#define OSMicrosecondsToTicks( usec )

OSTime OSCalendarTimeToTicks( OSCalendarTime* ct );
void   OSTicksToCalendarTime( OSTime ticks, OSCalendarTime* ct );  …
```

`OSGetTick` returns the lower 32 bits of the Broadway time base register.

`OSGetTime` allows the application to read the full 64-bit value of the time base.

`OSDiffTick` computes the differences between two ticks: `tick1` – `tick0`. If two ticks measured by `OSGetTick` are within a range of 35 seconds, `OSDiffTick` returns the correct number of ticks as a *signed* 32-bit value (even if `tick1`, measured later, is smaller than `tick0` as an unsigned 32-bit value).

In addition, several translation functions and macros are available to ease the transition from CPU ticks to conventional time metrics and vice versa.

Note: These macro functions work on `OSTime` values as well.

6.2 Alarm

The Revolution OS provides high-resolution alarms that can be used to schedule events in the future. These alarms have the same accuracy as the real-time clock. They use 64-bit counters and can thus trigger events at virtually anytime in the future.

These alarms use the Broadway decrementer register. The application will set the time for an alarm and call the alarm function. When the specified time has elapsed, a decrementer exception will be raised, and the handler for that exception will invoke the alarm handler. The following program demonstrates how to set up and handle an alarm:

Code 6–2 Timer Program

```
#include <revolution.h>

#define PERIOD  5   // sec

OSAlarm Alarm;

static void AlarmHandler(OSAlarm* alarm, OSContext* context)
{
    #pragma unused( alarm, context )
    OSTime t;

    t = OSGetTime();
    OSReport("Alarm at %lld.%03lld [sec]\n",
             OSTicksToSeconds(t),
             OSTicksToMilliseconds(t) % 1000);
}

void main(void)
{
    OSTime now;

    now = OSGetTime();
    OSReport("The time now is %lld.%03lld [sec]\n",
             OSTicksToSeconds(now),
             OSTicksToMilliseconds(now) % 1000);
    OSReport("Initializing period to %d [sec]\n", PERIOD);

    OSSetPeriodicAlarm(
        &Alarm,                     // pointer to alarm
        now,                        // start counting immediately
        OSSecondsToTicks(PERIOD),   // set 5 sec period
        AlarmHandler);              // alarm handler to be called at every
                                    // PERIOD sec

    for (;;)
    {

    }
}
```

The alarm-related APIs are:

Code 6–3 Timer APIs

```
typedef struct OSAlarm  OSAlarm;
typedef void          (*OSAlarmHandler)(OSAlarm* alarm, OSContext* context);

struct OSAlarm
{
    OSAlarmHandler  handler;
    u32             tag;
    OSTime          fire;
    OSAlarm*        prev;
    OSAlarm*        next;

    // Periodic alarm
    OSTime          period;
    OSTime          start;
    void*           queue;
};

void OSSetAlarm         ( OSAlarm* alarm, OSTime tick, OSAlarmHandler handler );
void OSSetPeriodicAlarm ( OSAlarm* alarm, OSTime start, OSTime period,
                          OSAlarmHandler handler );
void OSCancelAlarm      ( OSAlarm* alarm );
```

The alarm can be in two modes:

- `OSSetPeriodicAlarm`: The alarm will fire every *period* starting from *start*.
- `OSSetAlarm`: The alarm will fire just once, *tick* number of ticks in the future.

The values of *tick*, *start*, and *period* are in ticks. Both functions install the alarm handler for each alarm. The handler has two arguments: a pointer to the *alarm* that is fired, and a pointer to *context,* which holds the CPU register context when the decrement exception is taken.

You can cancel both one-shot and periodic alarms with `OSCancelAlarm`.

The alarm handlers run at high priority with interrupts disabled. It is expected that the application will perform little work in these handlers. Like other device callbacks in the system, alarm handlers can touch FPU registers.

7 Critical Sections

Because the Revolution OS does not require the use of threads, the default synchronization primitive is the capacity to enable and disable interrupts that include:

- Interrupts from devices
- Interrupts from the graphics chip
- The timer alarm

Although the application does not see these interrupts directly, it can be notified of their occurrence via callbacks from the device driver layers. These callbacks can be thought of as high-level interrupt handlers. As such, they run with interrupts *disabled* on the current thread's stack.

The following API allows the application to control whether interrupts are received:

Code 7–1 Interrupt Reception API

```
BOOL OSEnableInterrupts ( void );
BOOL OSDisableInterrupts( void );
BOOL OSRestoreInterrupts( BOOL enable );
```

`OSEnableInterrupts` and `OSDisableInterrupts` are fairly self-explanatory. They each return the previous state of the interrupts before they are enabled or disabled: non-zero if interrupts were enabled, zero if interrupts were disabled. `OSRestoreInterrupts` sets the state of interrupts to the value of *enable*.

Thus, a typical critical section should look like this:

Code 7–2 Critical Section

```
BOOL enabled;

enabled = OSDisableInterrupts();      // Saves original interrupt level
//
// Critical section code comes here
//
:
OSRestoreInterruptLevel(enabled);     // Restores original interrupt level
```

If an application disables interrupts for too long, the Revolution OS may malfunction. The longest an application can safely disable interrupts is 100µs.

7.1 Programming Model

When any application starts up, external interrupts are enabled by default.

Since callbacks are essentially part of the interrupt handling system, by default they run with interrupts *disabled*. This is important to note because callbacks should not perform any long-running computation that may block the occurrence of other key interrupts (such as the audio interrupt).

Efficiency

Although the time spent enabling and disabling interrupts is negligible, the Broadway CPU must flush the execution pipeline, which causes instruction latency to increase and instruction throughput to decrease. Therefore, constant enabling and disabling of interrupts may severely reduce performance.

8 Using CPU Idle Time

Occasionally, you may wish to call a function that takes a significant amount of time—such as to perform data decompression or dynamic linking—without slowing down the game frame rate. The Revolution OS supports a simple mechanism to run time-consuming functions in the background while, for example, the game main loop is idle. If more than one background task is necessary, however, the threads API (see "9 Threads" on page 52) may be more appropriate.

Code 8–1 Idle Function (Background Task) Example

```
u8  Stack[8192];
u64 Sum;

static void Func(void* param)
{
    #pragma unused( param )
    u64 n;

    //
    // Do background job
    //
    for (n = 1; n <= 1000000; n++)
    {
        Sum += n;
    }
}

int main(void)
{

    VIInit();

    OSSetIdleFunction(
        Func,                   // start function
        0,                      // initial parameter
        Stack + sizeof Stack,   // initial stack address
        sizeof Stack);          // stack size

    // Loop until the idle function completes. OSGetIdleFunction()
    // returns NULL after the idle function completes.
    do
    {
        OSReport("Sum of 1 to 1000000 >= %llu after %u V-syncs\n",
                 (volatile u64) Sum,
                 VIGetRetraceCount());
        VIWaitForRetrace();                 // Sleep till next V-sync
    } while (OSGetIdleFunction());

    OSReport("Sum of 1 to 1000000 == %llu after %u V-syncs\n",
             (volatile u64) Sum,
             VIGetRetraceCount());

    return 0;
}
```

"Code 8–1 Idle Function (Background Task) Example" on page 50 shows a simple background task that adds up all the numbers from 0 to 1,000,000. The APIs used are:

Code 8–2 Idle Function (Background Task) API

```
typedef void (*OSIdleFunction)(void* param);

OSThread* OSSetIdleFunction(
    OSIdleFunction idleFunction,
    void*          param,
    void*          stack,
    u32            stackSize
);

OSThread* OSGetIdleFunction(
    void
);
```

When `OSSetIdleFunction` registers a background task, this idle function creates a background thread that will run when the game's main loop is in an idle state. In other words, the operating system will call *idleFunction* while the game's main loop is waiting for the `VIWaitForRetrace` function to return.

You can use `OSGetIdleFunction`'s return value to find out if the idle function has finished. A value of `NULL` will be returned if the background thread created by `OSSetIdleFunction` has finished; otherwise, a non-`NULL` value will be returned. This function is useful for creating background tasks without worrying about thread management. Use `OSCreateThread`, instead, if you need two or more background tasks.

`OSGetIdleFunction` takes a single parameter, *param*, and will run with its own stack, specified by the *stack* argument. This stack must be large enough to run `OSGetIdleFunction`. The OS will write a "magic word" (`OS_THREAD_STACK_MAGIC`) into the last word of the stack. Because the stack grows downward, the magic word will be placed at the lowest address. This function will check if the magic word is intact when `OSCheckActiveThreads` is called.

Note: We have not determined what behavior may result if the game main loop and an idle function invoke a non-reentrant function concurrently. Many of the library functions provided with the development kit are not reentrant.

9 Threads

The Revolution OS provides transparent thread support; that is, if your application does not require threads, you do not need to know anything about threads.

The Revolution OS threads system provides all of the functionality of POSIX-style primitives. The system has the following features:

- Create, destroy, join, and yield threads
- Control when rescheduling can occur
- Send, jam, and/or receive messages
- Lock and unlock mutexes. (The mutex primitives use a basic priority inheritance scheme to prevent priority inversion and improve the ability to schedule the thread.)
- Handle condition variables

This chapter offers an overview of the basic use of threads. Refer to the *Revolution Function Reference Manual* (HTML) for more details.

9.1 Initialization

After `OSInit()` has been called, all of the thread functions may be invoked. `OSInit()` creates a default thread control block for the application. That is, the application automatically becomes a thread after `OSInit()` has been called. No idle thread is used—any idle time is spent at the scheduling points.

9.2 Scheduling

Each thread is assigned a base scheduling priority between 0 and 31. The highest priority is 0, the lowest is 31. The default thread created by `OSInit()` has a priority of 16. An idle task created by `OSSetIdleFunction()` (see "8 Using CPU Idle Time" on page 50) will be given a priority of 31.

Rescheduling occurs whenever a thread is suspended or made runnable; this includes interrupts. The scheduler attempts to run the next available thread with the highest priority.

Note: The scheduler is non-preemptive. After an interrupt, if the highest thread priority has not changed, the current thread will continue to run, even if there are other runnable threads at the same priority. However, when a thread is interrupted by a thread of higher priority, it is put at the end of the priority queue. Thus, round-robin scheduling will occur if a set of threads of equal priority are periodically interrupted by a higher priority thread.

9.2.1 Interaction with Interrupts

Interrupt and exception handlers both have higher priority than threads and will therefore block the execution of all threads.

Interrupts from devices will cause rescheduling once interrupt processing has completed. Interrupt processing includes any time spent performing callback operations. Rescheduling will never occur during interrupt processing, even if thread functions are invoked in a callback.

For example, suppose that an application wants to run a decompression thread once an optical disc file has been loaded.

1. The application initiates an asynchronous optical disc read.
2. When the optical disc has been completely read, the application-specified callback runs.
3. The application callback wakes the decompression thread and returns. Ordinarily, rescheduling would occur at this point, but since we are still processing an interrupt, rescheduling is deferred.
4. After the callback has returned, the scheduler is automatically invoked.

The callback can call as many thread functions as it wishes, but rescheduling will only occur after the callback has returned. This is because callbacks run with interrupts disabled and are considered part of interrupt processing. Thus, they should terminate as soon as possible.

The interrupt handler may use the stack of the current thread at the time the interrupt is generated. If an event such as a program crash occurs due to the execution of the interrupt handler, it may be possible to solve the problem by increasing the size of the thread stack.

9.3 Thread Creation

The following code example demonstrates how a thread is created.

Code 9–1 Thread Creation Example

```
OSThread      Thread;
u8            ThreadStack[8192];
u64           Sum;

static void* Func(void* param)
{
    #pragma unused( param )
    u64 n;

    // Do background job
    for (n = 1; n <= 1000000; n++)
    {
        Sum += n;
    }

    // Exits
    return 0;
}

int main(void)
{

    OSInit();
    VIInit();

    // Creates a new thread. The thread is suspended by default.
    OSCreateThread(
        &Thread,                            // pointer to the thread to initialize
        Func,                               // pointer to the start routine
        0,                                  // parameter passed to the start routine
        ThreadStack + sizeof ThreadStack,   // initial stack address
        sizeof ThreadStack,                 // stack size
        31,                                 // scheduling priority - lowest
        OS_THREAD_ATTR_DETACH);             // detached by default

    // Starts the thread
    OSResumeThread(&Thread);

    // Loop until the thread exits
    do
    {
        OSReport("Sum of 1 to 1000000 >= %llu after %u V-syncs\n",
                 (volatile u64) Sum,
                 VIGetRetraceCount());
        VIWaitForRetrace();                 // Sleep till next V-sync
    } while (!OSIsThreadTerminated(&Thread));
}
```

The key functions used in "Code 9–1 Thread Creation Example" on page 53 are:

Code 9–2 Basic Thread Creation APIs

```
BOOL OSCreateThread         ( OSThread*  thread,
                              void*    (*func)(void*),
                              void*      param,
                              void*      stackBase,
                              u32        stackSize,
                              OSPriority priority,
                              u16        attribute );
s32  OSResumeThread         ( OSThread*  thread );
BOOL OSIsThreadTerminated   ( OSThread*  thread );
```

`OSCreateThread` initializes a given thread structure. The thread is suspended initially, and it will not be scheduled until `OSResumeThread` has been invoked upon it. The thread will start as if *func(param)* had been called. The parameter *stackBase* is a pointer to the stack for this operation, and *priority* specifies the thread's scheduling priority. Remember that stacks grow downward; that is, the high address is at the bottom of the stack. The OS will write a magic word (`OS_THREAD_STACK_MAGIC`) into the last (lowest) word of the stack. You can check whether a stack overflow has occurred by making sure that this magic word stays intact throughout your program.

The *attribute* argument has two potential values: `OS_THREAD_ATTR_DETACH`, or 0. If a thread has the `OS_THREAD_ATTR_DETACH` attribute set, the thread control block will be released (removed from OS control) when the thread terminates. However, if the attribute is left at 0, the thread control block will be "joinable," meaning that it will remain under the control of the OS when the thread terminates (though it will not actually run), until a "joining" thread runs. This is best explained by the example in "Code 9–3 Using OSJoinThread" on page 55.

`OSIsThreadTerminated` simply returns `TRUE` if *thread* has terminated. In cases where a thread has been terminated and may be re-used to create a new thread, it is safer to communicate the termination of a thread with a message or global variable.

Code 9–3 Using OSJoinThread

```
OSThread      Thread;
u8            ThreadStack[8192];
u64           Sum;

static void* Func(void* param)
{
    #pragma unused( param )
    u64 n;

    // Do background job
    for (n = 1; n <= 1000000; n++)
    {
        Sum += n;
    }

    // Exits
    return 0;
}

void main(void)
{

    VIInit();

    // Creates a new thread. The thread is suspended by default.
    OSCreateThread(
        &Thread,                            // pointer to the thread to initialize
        Func,                               // pointer to the start routine
        0,                                  // parameter passed to the start routine
        ThreadStack + sizeof ThreadStack,   // initial stack address
        sizeof ThreadStack,                 // stack size
        31,                                 // scheduling priority
        0);                                 // joinable by default

    // Starts the thread
    OSResumeThread(&Thread);

    // Loop until the thread exits
    do
    {
        OSReport("Sum of 1 to 1000000 >= %llu after %u V-syncs\n",
                 (volatile u64) Sum,
                 VIGetRetraceCount());
        VIWaitForRetrace();                 // Sleep till next V-sync
    } while (!OSIsThreadTerminated(&Thread));

    // Wait till thread dies and release the thread from OS control
    OSJoinThread(&Thread, NULL);

    OSReport("Sum of 1 to 1000000 == %llu after %u V-syncs\n",
             (volatile u64) Sum,
             VIGetRetraceCount());

    OSHalt("Demo complete");
}
```

"Code 9–3 Using OSJoinThread" on page 55 is almost identical to "Code 9–1 Thread Creation Example" on page 53, except that the created thread has an attribute of 0. This makes it “joinable”—when the thread terminates, the thread control block remains in use until a “joining” thread (in this case, the main loop) has run. The main loop indicates it is joining the thread by calling `OSJoinThread`. After the main loop returns from `OSJoinThread`, the thread control block of the terminated thread may be safely re-used.

This is useful for several reasons:

- As a synchronization primitive, it allows a thread (ThreadA) to sleep until another thread (ThreadB) has terminated.
- It allows ThreadA to retrieve the return value of ThreadB.
- It allows you to examine the state of ThreadB in the debugger when ThreadB has terminated.

Code 9–4 OSJoinThread API

```
BOOL OSJoinThread( OSThread* thread, void** val );
```

`OSJoinThread` suspends the calling thread until *thread* has terminated. The return value of *thread* is placed in *val*. You may specify NULL for *val* if the return value of *thread* is not needed. Just before `OSJoinThread` returns, it will remove *thread* from OS control, effectively reclaiming its storage (i.e., you may safely use *thread* for other purposes).

`OSJoinThread` will return `TRUE` if *thread* was joinable and has terminated, and if the return value was properly retrieved. Several threads cannot wait for the same thread to terminate. Only one thread is guaranteed to return successfully, and the others will return `FALSE`. In this case, the return value of *thread* is invalid (since *thread*'s control block was already released, there is no way to know what happened to the memory).

If `OSJoinThread` is called when the thread has already terminated, it will return immediately with a return value of `TRUE`.

9.4 Synchronization

The Revolution OS provides messaging functionality and a few other synchronous primitives. This section briefly covers all of them. Unless your application uses synchronization very often in each frame, the performance differences between primitives is probably irrelevant, and the deciding factors will be familiarity and preference.

9.4.1 Library Access

Most of the Revolution OS libraries are not thread-safe. It is the responsibility of the developers to arbitrate access to shared resources (for example, GX function calls and `OSReport` output) with the thread synchronization primitives.

We have, however, made the optical disc drive (DVD) and Controller (PAD) libraries thread-safe.

9.4.2 Synchronizing by Disabling the Scheduler

The most basic synchronization primitive is the ability to disable scheduling and stop threads from switching context. This is different from disabling interrupts; while the scheduler is disabled, interrupts may still occur, but no other thread will be able to take control of the CPU.

These APIs are not atomic, so you must disable and re-enable interrupts when calling them.

Code 9–5 Disabling and Enabling the Scheduler Must Be Done with Interrupts Disabled

```
BOOL enabled;

enabled = OSDisableInterrupts();
OSDisableScheduler();
OSRestoreInterrupts(enabled);
//
// Critical region code comes here
:
enabled = OSDisableInterrupts();
OSEnableScheduler();
OSYieldThread();       // requests immediate context switch
OSRestoreInterrupts(enabled);
```

Code 9–6 Scheduler Control APIs

```
s32         OSDisableScheduler  ( void );
s32         OSEnableScheduler   ( void );
```

`OSDisableScheduler` is called when a thread wishes to defer thread scheduling. This means that higher priority threads will not run even if they are eligible for execution.

Note: The OS keeps a count of the number of times `OSDisableScheduler` has been called. While that count is positive, thread scheduling is disabled. The old value of this count is returned.

`OSEnableScheduler` reduces the `OSDisableScheduler` count. If the count is less than or equal to zero, thread rescheduling is enabled. The use of the count allows the application to nest calls to `OSDisableScheduler` without concern that one errant call to `OSEnableScheduler` will re-enable the scheduler. The original value of the count is returned.

9.4.3 Synchronizing by Sleeping and Waking

The Revolution OS provides another fast and primitive thread synchronization mechanism by which threads can be put to sleep and awakened from queues. These primitives are used by most of the other synchronization methods.

Code 9–7 Thread Sleep and Wakeup APIs

```
void        OSInitThreadQueue   ( OSThreadQueue* queue );
void        OSSleepThread       ( OSThreadQueue* queue );
void        OSWakeupThread      ( OSThreadQueue* queue );
```

`OSInitThreadQueue` initializes a thread queue structure.

`OSSleepThread` inserts the calling thread in the specified thread queue and makes that thread ineligible for execution until `OSWakeupThread` has been called on this queue. The OS will run the next available thread.

`OSWakeupThread` wakes *all* of the threads in the specified thread queue and makes them runnable. They will be run in priority order.

9.4.4 Synchronizing with Messages

The Revolution OS implementation of messages is quite fast and only slightly more expensive than simply creating thread queues and sleeping/awakening threads manually.

The following example creates a "`printf` server" thread that receives print requests from the main thread. Essentially, we are creating a thread to mediate access to a system resource (in this case, the `OSReport` channel).

Code 9–8 Message API Example

```
OSMessageQueue MessageQueue;
OSMessage      MessageArray[16];

OSThread       Thread;
u8             ThreadStack[8192];

// Print server thread function
static void* Printer(void* param)
{
    #pragma unused (param)
    OSMessage msg;

    for (;;)
    {
        OSReceiveMessage(&MessageQueue, &msg, OS_MESSAGE_BLOCK);
        OSReport("%s\n", msg);
    }

    return 0;
}

void main(void)
{
    int i;

    VIInit();

    // Initializes the message queue
    OSInitMessageQueue(
        &MessageQueue,                      // pointer to message queue
        MessageArray,                       // pointer to message boxes
        16);                                // # of message boxes

    // Creates a new thread. The thread is suspended by default.
    OSCreateThread(
        &Thread,                            // ptr to the thread to init
        Printer,                            // ptr to the start routine
        0,                                  // param passed to start routine
        ThreadStack + sizeof ThreadStack,   // initial stack address
        sizeof ThreadStack,                 // stack size
        8,                                  // scheduling priority
        OS_THREAD_ATTR_DETACH);             // detached since it will
                                            // never return

    // Starts the thread
    OSResumeThread(&Thread);

    // Main loop
    for (i = 0; i < 16; i++)
    {
        OSSendMessage(&MessageQueue, "Hello!", OS_MESSAGE_NOBLOCK);
        VIWaitForRetrace();                 // Sleep till next V-sync
    }

    OSHalt("Demo complete");
}
```

The message API is as follows.

Code 9–9 Message APIs

```
void OSInitMessageQueue( OSMessageQueue* mq, OSMessage* msgArray, s32 msgCount);
BOOL OSReceiveMessage  ( OSMessageQueue* mq, OSMessage* msg, s32 flags);
BOOL OSSendMessage     ( OSMessageQueue* mq, OSMessage  msg, s32 flags );
BOOL OSJamMessage      ( OSMessageQueue* mq, OSMessage  msg, s32 flags );
```

`OSInitMessageQueue` initializes the message queue structure `mq`. *msgArray* is memory that the caller must have already allocated, and *msgCount* indicates the number of entries in that array.

`OSReceiveMessage` retrieves a message from `mq`. It also wakes up any threads waiting to send a message to this queue. Sending threads will run in priority order. If *flags* is set to `OS_MESSAGE_BLOCK`, the calling thread will be suspended if the queue is empty. It will be resumed as soon as a message is sent to the queue.

Note: If there are other receiving threads of higher priority, those threads will run first and each retrieve a message. If the message queue is emptied by the time this thread runs, this thread will again be suspended until another message is sent to the queue.

If *flags* is set to `OS_MESSAGE_NOBLOCK`, the calling thread will return immediately. `TRUE` is returned if the queue was not empty, `FALSE` if the queue was empty.

`OSSendMessage` inserts the message at the tail of the specified message queue. It also wakes up the threads waiting on this message queue. The receiving threads will run in priority order. If *flags* is set to `OS_MESSAGE_BLOCK`, the calling thread will be suspended if the queue is full. It will be resumed as soon as a receiving thread has run and retrieved a message from the queue.

Note: If there are any other sending threads of higher priority, they will run first and potentially fill the message queue again. If this occurs, the thread will be suspended again until a receiving thread makes some room in the message queue.

If *flags* is set to `OS_MESSAGE_NOBLOCK`, the calling thread will return immediately. `TRUE` is returned if the queue was not full, `FALSE` if the queue was full.

`OSJamMessage` behaves exactly like `OSSendMessage`, except that the message is inserted at the head of the message queue, instead of the tail. This may be used to send high-priority messages.

9.4.5 Synchronizing with Mutexes

Mutexes (“mutual exclusions”) represent exclusive ownership of some system resource. These primitives are slightly more expensive to use than the basic sleeping and waking functions. The following example shows how a mutex can be used to protect the `OSReport` channel without using a separate thread.

Code 9–10 Mutex and Yielding Example

```
OSMutex         PrintMutex;
OSThread        Thread;
u8              ThreadStack[8192];

// Synchronous print
static void SyncPrint(char* msg)
{
    OSLockMutex(&PrintMutex);
    OSReport(msg);
    OSUnlockMutex(&PrintMutex);
}

static void* ThreadFunc(void * arg)
{
    #pragma unused(arg)
    u32 i;

    for (i = 0; i < 16; i++)
    {
        SyncPrint("<Thread1 says Hi!>\n");
        OSYieldThread();
    }

    return 0;
}

void main(void)
{
    int i;

    // Initializes the mutex
    OSInitMutex(&PrintMutex);

    // Initialize the thread
    OSCreateThread(
        &Thread,                            // pointer to the thread to initialize
        ThreadFunc,                         // pointer to the start routine
        0,                                  // parameter passed to the start routine
        ThreadStack + sizeof ThreadStack,   // initial stack address
        sizeof ThreadStack,                 // stack size
        16,                                 // scheduling priority
        0);                                 // joinable by default


    // Kick the thread off
    OSResumeThread(&Thread);

    // Main loop
    for (i = 0; i < 16; i++)
    {
        SyncPrint("<Main thread says Hi!>\n");
        OSYieldThread();
    }

    OSHalt("Demo Complete");
}
```

Because the main thread and *Thread1* are both at priority 16, the two threads will alternate control of the CPU at every scheduling point. In this case, the scheduling points only occur when `OSYieldThread()` is called, but receiving interrupts would have the same effect.

The new APIs used in this example are:

Code 9–11 Code 38 Mutex and Yielding APIs

```
void OSYieldThread   ( void );
void OSInitMutex     ( OSMutex* mutex );
void OSLockMutex     ( OSMutex* mutex );
void OSUnlockMutex   ( OSMutex* mutex );
```

`OSYieldThread` attempts to allow other threads of the same scheduling priority to run. Recall that if there are any other threads of higher priority available, they would already be running. If there are no other runnable threads at the calling thread's priority, this function returns immediately.

`OSInitMutex` initializes a given mutex structure.

Note: It is a programming error to initialize a mutex that is in use.

`OSLockMutex` attempts to acquire *mutex* for the calling thread. If *mutex* is held by a different thread, the calling thread suspends until *mutex* is released.

If *mutex* is already held by the current thread, each extra call to `OSLockMutex()` will return immediately.

Note: Each invocation of `OSUnlockMutex()` must match a call to `OSLockMutex()`; otherwise, the mutex will not be released. This allows a thread to nest multiple calls to `OSLockMutex()` and `OSUnlockMutex()` safely on the same mutex.

`OSUnlockMutex` releases the mutex.

Note: The calling thread must be the owner of the mutex. If the calling thread has locked this mutex *n* times, only the *n*th call to `OSUnlockMutex()` will actually release the mutex.

If the calling thread's priority was temporarily increased because a higher priority thread required this mutex, that priority will be recalculated. However, it may not be returned to its base priority, depending on what other mutexes it holds.

If the mutex is released, all threads blocked on this mutex will be made runnable and run in priority order.

9.4.5.1 Deadlock

The use of mutexes raises the possibility of deadlocks. Quite simply, any circular dependency of mutexes may result in a deadlock. For instance, suppose there are two threads, A and B, each holding mutex X and Y respectively. If A attempts to lock Y, but B also tries to lock X, both A and B will be blocked indefinitely.

One simple solution is to ensure that mutexes in the system are always locked in the same order. For instance, if X were always locked before Y, deadlock would not occur.

9.4.5.2 Priority Inversion

The use of mutexes also raises the possibility of priority inversion. For example, assume we have three threads, A0, B31, and C16, with priority 0, 31, and 16, respectively. If B31 holds a mutex that A0 depends on, A0's execution depends on B31 completing its computation and releasing the mutex in a timely fashion. However, if C16 is runnable, it will prevent B31 from running. This situation is called priority inversion because, in effect, A0's priority has been reduced. The Revolution OS avoids this situation by temporarily boosting the priority of B31 to that of A0 when it realizes that A0 is depending on B31. This prevents C16 from cutting in, and it increases scheduling predictability by ensuring that A0 will be run as soon as possible. As soon as B31 has released the mutex, its priority will be recalculated based on what other mutexes it still holds.

9.4.6 Synchronizing with Conditional Variables

If you find that you are creating queues to wait for certain resources to become available, or for certain conditions to exist, you may find condition variables to be a useful and compact primitive to use.

A condition variable encapsulates some information about a shared resource. If a thread is waiting for a certain condition to be true, it waits on the appropriate condition variable. When the condition is satisfied (usually by another thread), the condition variable is “signaled,” thus waking up the threads that were waiting for the condition to be true.

In the following example, commonly known as the bounded buffer problem, a producer thread and a consumer thread are using a shared buffer to transfer data. The producer thread creates data and places it into the buffer, while the consumer thread retrieves data from the buffer. The buffer has limited space, so when the buffer is full, the producer cannot insert more data. Similarly, if the buffer is empty, the consumer cannot proceed.

Two condition variables are used to encapsulate these two cases: `CondNotFull` and `CondNotEmpty`. They are used in the following manner:

- Whenever the producer thread inserts data into the buffer, it will signal `CondNotEmpty`, since it has satisfied that condition.
- Whenever the consumer thread retrieves data, it will signal `CondNotFull`, as it has made room for more data.
- Whenever the producer finds the buffer full, it will sleep on the `CondNotFull` condition.
- Whenever the consumer finds the buffer empty, it will sleep on the `CondNotEmpty` condition.

Code 9–12 Solving the Bounded Buffer Problem with Condition Variables

```
// Bounded buffer data structure
OSMutex  Mutex;
OSCond   CondNotFull;
OSCond   CondNotEmpty;
u32      Buffer[BUFFER_SIZE];
u32      Count;

OSThread Thread;
u8       ThreadStack[8192];

static u32 Get(void)
{
    u32 item;

    OSLockMutex(&Mutex);
    while (Count == 0)
    {
        OSWaitCond(&CondNotEmpty, &Mutex);
    }
    item = Buffer[0];
    --Count;
    memmove(&Buffer[0], &Buffer[1], sizeof(u32) * Count);
    OSUnlockMutex(&Mutex);
    OSSignalCond(&CondNotFull);
    return item;
}

static void Put(u32 item)
{
    OSLockMutex(&Mutex);
    while (BUFFER_SIZE <= Count)
    {
```

```
        OSWaitCond(&CondNotFull, &Mutex);
    }
    Buffer[Count] = item;
    ++Count;
    OSUnlockMutex(&Mutex);
    OSSignalCond(&CondNotEmpty);
}

static void* Func(void* param)
{
    #pragma unused (param)
    u32 item;

    for (item = 0; item < 16; item++)
    {
        Put(item);
    }

    return 0;
}

void main(void)
{
    u32 i;

    VIInit();

    // Initializes mutex and condition variables
    OSInitMutex(&Mutex);
    OSInitCond(&CondNotFull);
    OSInitCond(&CondNotEmpty);

    // Creates a new thread. The thread is suspended by default.
    OSCreateThread(
        &Thread,                            // pointer to the thread to initialize
        Func,                               // pointer to the start routine
        0,                                  // parameter passed to the start routine
        ThreadStack + sizeof ThreadStack,   // initial stack address
        sizeof ThreadStack,                 // stack size
        16,                                 // scheduling priority
        OS_THREAD_ATTR_DETACH);             // detached, and not joinable

    // Resumes the thread
    OSResumeThread(&Thread);

    // Main loop
    for (i = 0; i < 16; i++)
    {
        u32 item;

        item = Get();
        OSReport("%d\n", item);
        VIWaitForRetrace();                 // Sleep till next V-sync
    }

    OSHalt("Demo complete");
}
```

The APIs used in this example are:

Code 9–13 Conditional Variable APIs

```
void OSInitCond   ( OSCond* cond );
void OSWaitCond   ( OSCond* cond, OSMutex* mutex);
void OSSignalCond ( OSCond* cond );
```

`OSInitCond` simply initializes a condition variable structure.

`OSWaitCond` indicates that this thread wishes to be woken when the specified condition *cond* has been signaled. Since there is probably a mutex associated with the resource, the mutex is handed to `OSWaitCond`, which will atomically release the mutex (so that some other thread can use the resource and hopefully signal *cond*) and suspend the thread.

Note: When `OSWaitCond` returns, it will attempt to re-acquire the mutex.

`OSSignalCond` wakes up all threads waiting on a condition. They will run in priority order. Naturally, only one of them will re-acquire the mutex and proceed.

9.5 Context Switching

To help you better understand the behavior of this system, this section briefly describes how the Revolution OS saves and restores machine state.

Every execution context, whether a thread or interrupt handler, has an `OSContext` structure into which the OS can save machine state. The largest parts of the structure are the general purpose registers (GPRs) and floating point registers (FPRs). GPRs are saved whenever an interrupt or thread switch occurs.

FPRs, however, are exceptionally expensive to save and restore. In addition to their size (64 bits each instead of 32 bits), the Broadway CPU requires two passes over the FPRs for each save and restore—one pass to save them as if they were doubles, one pass to save them as paired-singles.

As a result, the Revolution OS tries as hard as possible to avoid the saving of floating point state. It saves floating point state only when it is clear that a thread or context is trampling on the registers of another thread or context using the FPRs. For example, if thread A uses FPRs and is interrupted by thread B (for example, by yielding the CPU), the OS saves only the GPRs, and the FPRs remain under the “ownership” of thread A. If thread B never touches the FPRs, the FPRs will never be saved. As soon as thread B touches an FPR, the OS will quickly save thread A’s FPR context, and FPR ownership will transfer to thread B.

The same mechanism is used during interrupt and exception processing.

9.6 Checking the Active Threads

The Revolution OS maintains a linked list of all the active threads that are runnable, running, suspended, or asleep. This linked list is called the *active thread queue*. In the Revolution OS, the memory space used for the thread management is not protected from the user program at all, so it may be damaged by a program error (such as accessing bad pointers or bad arrays).

Code 9–14 OSCheckActiveThreads

```
long OSCheckActiveThreads(void);
```

The OSCheckActiveThreads function disables interrupts and then sweeps the active thread queue, performing as many sanity checks as possible. If OSCheckActiveThreads finds a broken link or any other problems, it displays a failure message and halts the execution of the program. Calling OSCheckActiveThreads often should help you debug multithreaded programs by eliminating thread management as a potential source of problems, but may also slow down the application.

9.7 Threads and Callbacks

Essentially, callback functions are not threads. Instead, they are called directly from the operating system or a device driver. A callback function cannot call a function that will block the execution of the current thread. Since there may be situations in which there is no current thread, calling a callback function in this situation would instead halt the operating system. Doing so is a programming error.

The functions that can block the execution of the current thread are as follows:

- GXSet*() and GXLoad*() functions that generates output to the GXFifo
- GXWaitDrawDone() and GXDrawDone()
- OSJoinThread(), OSSleepThread(), and OSSuspendThread (OSGetCurrentThread())

  Note: OSSuspendThread() can also be used for threads other than the current thread.
- OSLockMutex() and OSWaitCond()
- Message queue functions in OS_MESSAGE_BLOCK mode
- OSWaitSemaphore()
- Synchronous I/O functions that have the corresponding asynchronous functions (for example, DVDRead() and CARDRead())
- VIWaitForRetrace()
- WPADGetInfo()
- AXQuit()

10 Fast Float/Integer Casting

The Broadway CPU's paired-single instruction set gives you the ability to cast single-precision floating-point numbers to 8 or 16 bit integer values very quickly with just the cost of a load and a store.

Since the compiler does not currently take advantage of this ability, we have provided a fast cast C API that executes the proper instructions to perform the fastest possible cast. The fast cast API uses the Broadway's paired-single load and store instructions, which include quantization instructions for casting from a floating-point number to an integer, and consumes exactly two instructions (one load and one store).

10.1 Initializing the Fast Cast API

Call the `OSInitFastCast` function to configure fast casting from floating-point numbers to integers.

Code 10–1 OSInitFastCast

```
void    OSInitFastCast( void );
```

This must be called before any other fast cast. Additionally, the `OSInitFastCast` function only affects the thread from which it was called.

`OSInitFastCast` initializes the Broadway's quantization registers (GQRs) to known values. GQRs control the way paired-single load/stores work. For instance, GQR0 is always set to 0, which means that the data representation in memory is a simple 32-bit floating-point value. GQR2 is set to assume that the data in memory is an unsigned 8-bit integer value, and so on. The fast cast routines use the following GQR setup:

Table 10–1 Quantization Register Values

Quantization Register	Initialization Value
GQR2	Load u8 / Store u8. No scaling.
GQR3	Load u16 / Store u16. No scaling.
GQR4	Load s8 / Store s8. No scaling.
GQR5	Load s16 / Store s16. No scaling.

You will probably find it necessary to implement an independent cast API for your application to apply scaling to the values. Developers can implement an independent cast API based on the code for these cast functions, which is provided by the Revolution OS. The API is defined in the `OSFastCast.h` header file using inlined functions.

Note: The fast cast API requires that the application not modify GQR2–GQR5 (the Broadway's quantization registers). GQR0 is already reserved for single precision floating-point use, and GQR1 is reserved for compiler use.

10.2 Fast Casting Routines

This section briefly describes the fast casting API.

Code 10–2 Fast Cast APIs

```
inline void OSu8tof32  (u8*  in, f32* out);
inline void OSu16tof32 (u16* in, f32* out);
inline void OSs8tof32  (s8*  in, f32* out);
inline void OSs16tof32 (s16* in, f32* out);

inline void OSf32tou8  (f32* out, u8*  out);
inline void OSf32tou16 (f32* out, u16* out);
inline void OSf32tos8  (f32* out, s8*  out);
inline void OSf32tos16 (f32* out, s16* out);
```

These functions perform a fast cast in two cycles (one load and one store). The name of each function describes the cast that it performs.

These fast cast functions must use a load and a store instruction, and therefore take an address to a typed variable as an argument. By getting the address as an argument, the compiler is guaranteed to allocate space on the stack, without forcing the programmer or inline function to allocate memory space. In this way, the compiler will allocate stack space for the variables, and by using an array for storing data, all of the local variables can be allocated at once. If these functions were to allocate memory for this purpose, they would take more than two cycles to perform each cast.

Note: You must call `OSInitFastCast` before these functions. Also, in a standard Debug build, inline expansion will not be used for these functions.

11 Fonts

Nintendo GameCube-compatible European and Kanji ROM fonts are installed on the Wii console.

The European ROM font is based on the ANSI (Windows Latin 1) character set. The European ROM font is used by North American, European, and other non-Japanese console specifications.

The Kanji ROM font is based on the Shift-JIS character encoding. The following character images are actually installed: the ASCII characters (`0x20`–`0x7f`), the half-width kana (`0xa0`–`0xdf`), and the double-byte characters from `0x8140` to `0x9872`, which includes the JIS Level 1 Kanji (`0x889f`–`0x9872`). The Kanji ROM font can be used on Japanese Wii consoles. Refer to the ROM Font item in the *Revolution Function Reference Manual* (HTML) for information on the installed font images.

Note: The Kanji ROM font does not include the JIS Level 2 Kanji.

11.1 Loading Fonts

You can use the `OSGetFontEncode` function to find the type of ROM font that can be used by the Wii console.

The ROM fonts are stored on the Wii console in a ROM/RTC compressed format. To use a compressed ROM font, it must first be expanded into main memory. The font must be maintained in either I2 (4-grade) or I4 (16-grade) format in main memory.

- I2 Format

  Call the `OSLoadFont` function to expand a font into main memory in I2 format. Although a font image in I2 format cannot be used unchanged as texture image data passed to the `GXTexObj` structure, the `OSGetFontTexel` function can be called to expand the font image for a specified character into any location on a given I4 texture image.

- I4 Format

  Call the `OSInitFont` function to expand a font into main memory in I4 format. The image data is expanded and split into multiple texture images (sheets) of the same size. An ANSI font is made up of a single texture image sheet, and a Kanji font is made up of nine texture image sheets. The texture image sheets can be used unchanged as the `GXTexObj` structure's texture image data. By calling the `OSGetFontTexture` function, you can get a pointer to the sheet that contains the specified character's font, and the position of the font on the sheet.

Note: The buffer specified to the `OSLoadFont` and `OSInitFont` functions must be 32-byte aligned.

11.2 Character Width

Wii ROM fonts are variable-pitch (proportional) fonts. The `OSGetFontTexel`, `OSGetFontTexture`, and `OSGetFontWidth` functions return the width of the specified character in texels.

Note: The ASCII characters in the Kanji ROM font are fixed-width. Use the full-width alphabet for proportional display.

11.3 Font Header

An `OSFontHeader` structure will be expanded into the start of the buffer specified as the first argument to the `OSLoadFont` and `OSInitFont` functions. Some of this structure's members are used only within the OS library, but the members shown below can also be accessed by applications.

Table 11–1 Font Header Members Accessible by Applications

Quantization Register	Description	Value
ascent	Character ascent (number of texels above the baseline)	24
descent	Character descent (number of texels below the baseline)	0
leading	Line spacing	28
width	Maximum character width, in texels	24
sheetWidth	Texture image sheet width	512
sheetHeight	Texture image sheet height	512
cellWidth	Cell width (shortened region for a character) in a sheet	24
cellHeight	Cell height (shortened region for a character) in a sheet	24

12 Relocatable Module System (REL)

The Revolution OS provides a relocatable module system. By using the relocatable module system, games can use main memory efficiently while dynamically loading and releasing program modules. Unlike dynamic link libraries in other operating systems, the Wii relocatable module system requires that games allocate and release main memory, as well as load modules from disc.

Note: The Revolution SDK provides the RSO library as a relocatable module system in addition to the REL system provided by the OS library. For details, refer to the RSO Library item in the *Revolution Function Reference Manual*.

12.1 Relocatable Modules

The relocatable module system is composed of a single static module (ELF file) and multiple relocatable modules (REL files). Once the boot ROM has loaded a program's static module, that module can control the placement of relocatable modules in memory. The static module is built as a normal ELF file. It can contain common functions and variables accessed by relocatable modules.

Relocatable modules can call functions and access variables in a static module. Relocatable modules can also call functions and access variables in other relocatable modules that have already been loaded into main memory. Inter-module references are resolved by directly modifying code and data sections in the modules when they are loaded. As a result, programs can run efficiently without excessive indirect references once inter-module references have been resolved. For example, only a single branch instruction (the Broadway `bl` instruction) will be used to call a function between modules, in the same manner as a statically linked function.

A relocatable module (REL) program can be written in the exact same way as a normal C or C++ program. However, a relocatable module is created from partially linked ELF (PLF) files. PLF files include unresolved external symbols and debugging information. The `makerel` tool, provided by the Revolution SDK, converts PLF files into Wii relocatable module files. To reduce module size and improve runtime efficiency, Wii relocatable module files include only normal program code and data sections, as well as a relocation instruction table. The relocation instruction table includes 8-byte relocation instructions for each part of code and data to modify at runtime. Each relocation instruction is made up of an offset to the location that will be modified, a relocation type, a target section number, and the value to add. Costly symbol table lookups are not performed at runtime (the symbol table will be removed by the `makerel` tool).

Refer to the Relocatable Module System item in the *Revolution Function Reference Manual* (HTML) for information on how to create relocable modules. There is also a demo program in the `/revolution/build/demos/reldemo` folder that uses relocatable modules. The makefile for this demo automatically creates a relocatable module and then runs the program.

Note: The Broadway branch instructions (`bx`) do not support jumps farther than 32 MB. As a result, modules loaded in internal main memory (MEM1) cannot call REL modules placed in external main memory (MEM2), and vice versa.

Note: Global C++ constructors and destructors must be called explicitly from the relocatable module's `_prolog` and `_epilog` sections, respectively. This is shown in the following code.

Code 12–1 `_prolog` and `_epilog` Functions

```
#ifdef __cplusplus
extern "C" {
#endif

typedef void (*voidfunctionptr) (void); /* ptr to function returning void */
__declspec(section ".init") extern voidfunctionptr _ctors[];
__declspec(section ".init") extern voidfunctionptr _dtors[];

void _prolog(void);
void _epilog(void);
void _unresolved(void);

#ifdef __cplusplus
}
#endif

void _prolog(void)
{
    voidfunctionptr *constructor;

    /*
     * call static initializers
     */
    for (constructor = _ctors; *constructor; constructor++) {
        (*constructor)();
    }
}

void _epilog(void)
{
    voidfunctionptr *destructor;

    /*
     * call destructors
     */
    for (destructor = _dtors; *destructor; destructor++) {
        (*destructor)();
    }
}
```

Link every relocatable module with `global_destructor_chain.c` (under `$(CWFOLDER)/PowerPC_EABI_Support/Runtime/Src)`. This ensures that each of the module's global destructors are called by the `_epilog` function. Otherwise, the global variables in each module will be linked to the static module's global destructor chain, and no destructors will be called if the static module does not exist. Note that `Runtime.PPCEABI.H.a` uses a small data section and contains unnecessary internal functions, so relocatable modules cannot link to it.

13 Profiling

The Revolution OS provides two key means of profiling: stopwatches and the Broadway CPU's performance monitors.

13.1 Stopwatches

Stopwatches allow the application to time multiple entries into a code segment. The paradigm is to start a stopwatch just before the profiled code, and to stop it afterwards. In this way, one can count the total time spent in the code, as well as the average time spent per entry. Any number of stopwatches can be created.

The stopwatches use the Broadway time base, and so will have 16.5 ns accuracy.

The following code demonstrates how to use the stopwatches. The program measures 5,000 matrix concatenations in groups of 100.

Code 13–1 Stopwatch Code Example

```
#include <revolution.h>

#define OUTER_ITERATIONS  50
#define INNER_ITERATIONS  100

OSStopwatch         MySW;

void main (void)
{
    u32             i, j;
    Mtx             a, b, ab;

    OSInitStopwatch(&MySW, "100 concat stopwatch");
    OSReport("Stopwatch demo program\nTimes %d matrix concatenations\n",
             OUTER_ITERATIONS*INNER_ITERATIONS);

    MTXIdentity(a);
    MTXIdentity(b);
    MTXIdentity(ab);

    for (i = 0; i < OUTER_ITERATIONS; i++)
    {
        OSStartStopwatch(&MySW);
        for (j = 0; j < INNER_ITERATIONS; j++)
        {
            MTXConcat(a, b, ab);
        }
        OSStopStopwatch(&MySW);
    }
    OSReport("\nEach hit is 100 matrix concats:\n");
    OSDumpStopwatch(&MySW);
    OSHalt("Stopwatch Demo complete");
}
```

The output generated by this program looks like:

Code 13–2 Stopwatch Program Output

```
Stopwatch demo program
Times 5000 matrix concatenations

Each hit is 100 matrix concats:
Stopwatch [100 concat stopwatch]    :
    Total= 1513 us
    Hits = 50
    Min  = 30 us
    Max  = 32 us
    Mean = 30 us
Stopwatch Demo complete in "stopwatchdemo.c" on line 54.
```

The preceding example uses these APIs:

Code 13–3 Stopwatch APIs

```
void            OSInitStopwatch         ( OSStopwatch* sw, char* name );
void            OSStartStopwatch        ( OSStopwatch* sw );
void            OSStopStopwatch         ( OSStopwatch* sw );

void            OSDumpStopwatch         ( OSStopwatch* sw );
```

The application must allocate stopwatches, either statically or dynamically.

`OSInitStopwatch` initializes a single stopwatch structure. It resets the stopwatch count to 0 and sets the stopwatch name to *name*. No stopwatch should be used without first calling `OSInitStopwatch`.

`OSStartStopwatch` is called upon entry to the code region to be measured. It records the current time.

`OSStopStopwatch` is called upon exit from the code region to be measured. The program compares the current time with the start time (i.e., the time at which `OSStartStopwatch` was called), then records the interval. A stopwatch can measure any number of intervals. The total time measured simply accumulates.

`OSDumpStopwatch` prints out the number of intervals (hits) measured by this stopwatch, the total time measured, the minimum and maximum latency measured, and the mean time spent per interval. All times are displayed in microseconds. For example:

Code 13–4 Stopwatch Intervals

```
Stopwatch [stopwatchname]    :
    Total= 2983746 us
    Hits = 1000
    Min  = 1160 us
    Max  = 277124 us
    Mean = 2983 us
```

13.2 Performance Monitors

The Broadway CPU has four performance monitor counters (PMCs), each of which can measure a subset of useful metrics on the chip (such as CPU cycles, load/store counts, and cache misses). We have concluded that a general-purpose API for these PMCs will either be too complex for general use or impose an unreasonable amount of overhead. Instead, this section presents a description of how we have used the PMCs in our code. The code outlined here is used in the locked cache demos.

The four PMCs are controlled by two control registers. These two control registers are known as the monitor mode control registers (MMCRs), and are named MMCR0 and MMCR1. MMCR0 controls PMCs 1 and 2, while MMCR1 controls PMCs 3 and 4. We use very basic functions from C code to set these registers:

Code 13–5 Basic Performance Monitor Counter Accessor Functions

```
void PPCMtmmcr0 ( u32 newMmcr0 );
void PPCMtmmcr1 ( u32 newMmcr1 );
u32  PPCMfmmcr0 ( void );
u32  PPCMfmmcr1 ( void );

void PPCMtpmc1  ( u32 newPmc1 );
void PPCMtpmc2  ( u32 newPmc2 );
void PPCMtpmc3  ( u32 newPmc3 );
void PPCMtpmc4  ( u32 newPmc4 );

u32  PPCMfpmc1  ( void );
u32  PPCMfpmc2  ( void );
u32  PPCMfpmc3  ( void );
u32  PPCMfpmc4  ( void );
```

This very basic API is used to “move values to” (`Mt` prefix) or “move values from” (`Mf` prefix) the various registers. We typically bundle these calls into useful macros such as the following:

Code 13–6 Performance Monitor Counter Macros

```
// STARTPMC sets both MMCRs (monitor control registers) going.
// PMC1 measures instruction count
// PMC2 measures # of loads and stores
// PMC3 measures # of cycles lost to L1 misses
// PMC4 measures cycle count
// Note : cycle counter is turned on last
#define STARTPMC            PPCMtmmcr0(MMCR0_PMC1_INSTRUCTION | \
                                       MMCR0_PMC2_LOAD_STORE);  \
                            PPCMtmmcr1(MMCR1_PMC3_L1_MISS_CYCLE | \
                                       MMCR1_PMC4_CYCLE); \


// STOPPMC pauses all performance counters by writing 0 to the MMCRs.
// NOTE: Cycle counter is turned off first.
#define STOPPMC             PPCMtmmcr1(0);  \
                            PPCMtmmcr0(0);
#define PRINTPMC            OSReport("<%d loadstores / %d miss cycles / %d cycles / %d \
Instructions>\n", \
                                     PPCMfpmc2(), PPCMfpmc3(), PPCMfpmc4(), PPCMfpmc1());
#define RESETPMC            PPCMtpmc1(0); \
                            PPCMtpmc2(0); \
                            PPCMtpmc3(0); \
                            PPCMtpmc4(0);
```

The constants used for the MMCR values can be found in `include/revolution/base/PPCArch.h`.

Note: The constant names are prefixed with the register for which they are appropriate. A detailed description of each event can be found in Chapter 11 of `BroadwayUserManual.pdf`.

The macros can be used in the following way to get basic measurements:

Code 13–7 Using Performance Monitor Counter Macros

```
RESETPMC
STARTPMC
    <code to be measured goes here>
STOPPMC
PRINTPMC
```

There are even tighter ways of measuring assembly code. See the *Optimization Primer* for more information.

14 Reset and Shutdown Processing

The Revolution OS provides several functions related to performing system state transitions, such as resetting and shutting down. An application must call these functions appropriately either to exit or when it detects that RESET or the Power Button was pressed.

Code 14–8 Reset and Shutdown Functions

```
void OSRebootSystem(void);
void OSShutdownSystem(void);
void OSRestart(u32 resetCode);
void OSReturnToMenu(void);
```

The four functions above have been provided to implement reset and shutdown processing. Run `OSRebootSystem` to reboot the system, `OSShutdownSystem` to shut down the system, `OSRestart` to restart the current application, and `OSReturnToMenu` to return to the system menu. Refer to the descriptions of each function in the *Revolution Function Reference Manual* (HTML) for more details. When called, these reset and shutdown functions will shut down each sub-system internally.

14.1 Reset and Shutdown Notes

Be aware of the following when calling the reset and shutdown functions.

14.1.1 Requirements for Conditions That Cause Fatal Errors, Such as System Lockups

- **Callbacks:** Do not call a reset or shutdown function when there is still a user callback or handler that may call GX functions or functions (such as `AXInit` and `AIInit`) that initialize audio-related libraries. The reset and shutdown functions shut down all sub-systems before disabling all interrupts and cancelling user alarms. This implies that any unfinished asynchronous functions or enabled alarms that exist may be invoked within a reset or shutdown function after it has shut down the sub-systems. If a callback or handler attempts to use a sub-system that has been shut down, the sub-system may cause a fatal error, such as a system lockup, to occur. If the system hangs after a reset or shutdown function is called, check for callbacks and handlers that use any combination of the GX API and functions for initializing audio-related libraries.

14.1.2 Recommendations for Non-Fatal Conditions that Should Be Handled, Such as Possible Delayed Resets

- **Audio:** We recommend shutting down all audio sub-systems before calling a reset or shutdown function. If an audio sub-system has not been shut down before a reset or shutdown function is called, the function may take some time to shut down the audio sub-system. Additionally, the reset and shutdown functions do not guarantee that audio will be stopped without any emitted noise.
- **Wii console NAND memory:** All processing in Wii console NAND memory must be completed before a reset or shutdown function is called. If processing has not completed, a comparatively long period of time may pass before a hot reset occurs.
- **VI:** We recommend that applications call the `VISetBlack(TRUE)`, `VIFlush`, and `VIWaitForRetrace` functions to flush the framebuffer before calling a reset or shutdown function. The reset and shutdown functions do not call `VISetBlack(TRUE)` internally. As a result, the framebuffer may change during reset and shutdown processing.
- **Optical disc drive:** We recommend that none of the DVD functions be called before a reset or shutdown function. The reset and shutdown functions cancel all optical disc drive requests. As a result, the drive may be reset twice if `DVDCancel` is called before a reset or shutdown function.

14.1.3 Miscellaneous Notes

The reset and shutdown functions temporarily stop the thread scheduler. Once a reset or shutdown function is called, user threads will no longer run.

The Revolution SDK does not use the Broadway's MMU for segment address conversion and will not access related registers, including SDR1 and SR*n*. When an application uses segment address conversion, it must restore the registers to their original state before performing a restart using `OS_RESET_RESTART`. Otherwise, unexpected application behavior may result.

No data in memory will be guaranteed after a system reboot.

Note: The reset and shutdown functions cannot be invoked from a callback function: callback behavior will not be guaranteed.