Revolution

Operating System

Version 1.00

"Confidential"

Operating System

Version 1.00

Contents

Code Examples

Figures

Tables

Revision History

Revision No.	Date Revised	Items (Chapter)	Description	Revised By
1.00	3/1/2006	-	First release by Nintendo of America, Inc.	-

1 Overview

Note: The Revolution Operating System Programming Manual is still being written. This Nintendo GameCube Programming Manual is being provided on a temporary basis for reference. Please refer to the OS library function reference manual until the Programming Manual is available.

This document presents a high-level overview of those features of the Nintendo GameCube OS that are most important to developers. We present first the information most relevant for developers to write real code as soon as possible.

The key features of the OS are:

- No threads required (threads are supported, but their usage is transparent).
- 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 operating system will initialize before any application code runs. Technically, initialization occurs just before any global constructors are called for static C++ objects.

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.

Note: In development kits shipped before 9/29/00, your application must call `OSInit` from `main()` manually. `OSInit` can be called more than once; all successive calls to `OSInit` are simply ignored.

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 1 - OSGetConsoleType

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

The `OSGetConsoleType` function returns one of the console type values shown in the following table. We use the left-most bit (LMB—defined as `OS_CONSOLE_DEVELOPMENT`) to distinguish retail production systems from development systems. In a development system, the LMB is set to one; otherwise, the LMB is set to zero. The right-most 16 bits show the minor revision number of the console. The retail production systems will thus have console type values `0x00000001`, `0x00000002`, and so on.

Table 1 - Return values from OSGetConsoleType

Defined Name	Value	Description
`OS_CONSOLE_RETAIL2`	0x00000002	HW2 production board.
`OS_CONSOLE_RETAIL1`	0x00000001	HW1 production board.
`OS_CONSOLE_DEVHW2`	0x10000005	The second game development system (HW2).
`OS_CONSOLE_DEVHW1`	0x10000004	The first game development system (HW1).
`OS_CONSOLE_MINNOW`	0x10000003	A prototype system without graphics capability.
`OS_CONSOLE_ARTHUR`	0x10000002	PowerPC 750 evaluation board.
`OS_CONSOLE_PC_EMULATOR`	0x10000001	PC emulator.
`OS_CONSOLE_EMULATOR`	0x10000000	Mac emulator.

Note: `OS_CONSOLE_MINNOW` and `OS_CONSOLE_ARTHUR` are console types used only for system bring-up within Nintendo and its partner companies.

3 Memory

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

3.1 System memory map

The Nintendo GameCube 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 256MB (i.e., all system memory and memory-mapped devices live within a 256MB address space). Cached accesses to the physical address space takes the form 0x8xxxxxxx, while uncached accesses take the form 0xCxxxxxxx, as shown in the following table. Only the first 256MB of each of these segments are accessible; accessing any other addresses will cause a memory access fault.

Figure 1 - Game code access to the physical address space

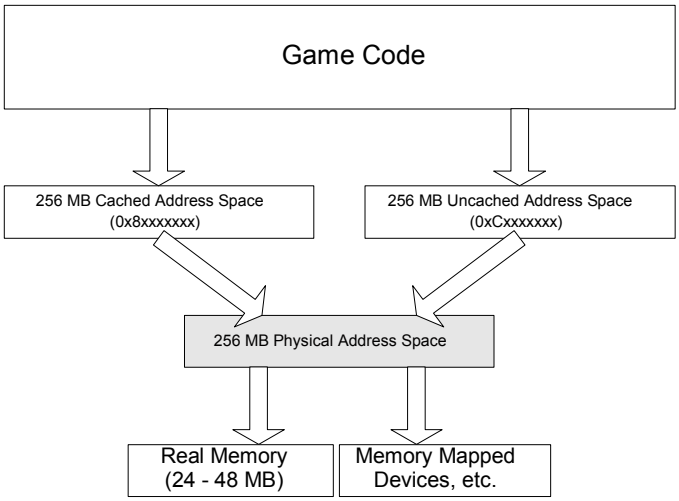
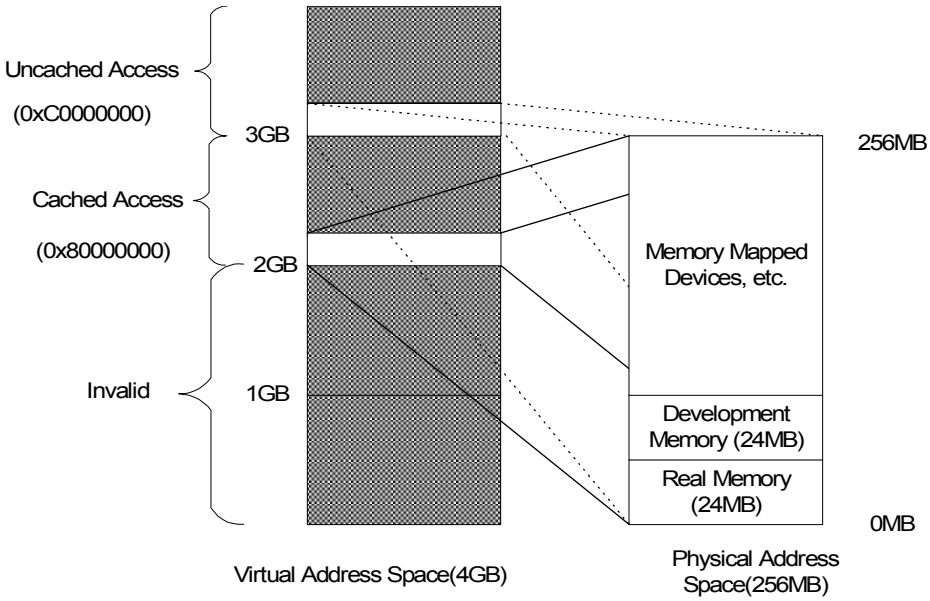


Figure 2 - Alternate representation of virtual and physical address spaces



Shaded regions indicate invalid addresses

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 game uses the following APIs to query the boundaries of this arena:

Code 2 - OSGetArena API

```
void* OSGetArenaHi( void );
voud* OSGetArenaLo( void );
void  OSSetArenaHi( void* newHi );
void  OSSetArenaLo( void* newLo );
```

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

Code 3 - Setting the arena

```
#include <dolphin.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             = OSGetArenaLo();
    arenaHi             = OSGetArenaHi();

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

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

Now the program can start to grab chunks of 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 `OSGetArena*` and `OSSetArena*` 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 arbitrarily-sized blocks of memory from any number of heaps.

3.3.1 Allocation alignment

Many of the Nintendo GameCube subsystems require memory to be aligned to 32-byte boundaries (e.g., 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: If you wish to allocate many small objects efficiently, you should 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 4 - Heap allocation

```
#include <dolphin.h>

OSHeapHandle                TheHeap;

#define INT_ARRAY_ENTRIES   1024

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

    OSInit();

    arenaLo = OSGetArenaLo();
    arenaHi = OSGetArenaHi();

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

    OSSetArenaLo(arenaLo = arenaHi);

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

In the preceding example, the developer 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:

Code 5 - OSAlloc APIs

```
typedef u32     OSHeapHandle;

void*           OSInitAlloc          ( void* arenaLo, void* arenaHi, u8 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 10, 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; i.e., 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 wish to manage memory in separate heaps for several purposes (e.g., to delineate memory usage clearly between discrete modules, or to increase memory locality for different sections of code). The Nintendo GameCube memory allocator supports this functionality, as demonstrated in the following code sequence:

Code 6 - Memory allocation

```
#include <dolphin.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;

    OSInit();

    arenaLo = OSGetArenaLo();
    arenaHi = OSGetArenaHi();
    arenaLo = OSInitAlloc(arenaLo, arenaHi, 2); // 2 heaps
    OSSetArenaLo(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;

    OSSetArenaLo(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 6 - Memory allocation" on page 10, the following routines help to manage memory with multiple heaps:

Code 7 - Memory management for multiple heaps

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

`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 *Dolphin Reference Manual* (HTML).

Overhead

The memory overhead of the allocator is as follows:

- Heap descriptor array: 24 bytes per heap (subtracted from arena after `OSInitAlloc`).
- Object headers: 32 bytes per object.
- Object padding: object sizes are rounded up to 32-byte alignment.

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 worst case *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

There are some issues of which you should be aware when using C++ code in the Nintendo GameCube OS.

If you want to use the `new` and `delete` operators with the Nintendo GameCube OS, then you must overload the library default `new` and `delete` operators. The overloaded operators can utilize `OSAlloc();` routines or your own memory manager.

The first invocation of your `new` operator should initialize the memory allocation system. This is because static C++ objects may require memory allocation before `main()` is reached. Only the first invocation of `new` should perform this initialization.

The following demo code shows how to do this with the Nintendo GameCube OS memory allocator:

Code 8 - 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 = OSGetArenaLo();
    arenaHi = OSGetArenaHi();

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

4 Error handling and notification

One of the most common ways to handle errors or debugging is to use `printf()` and `assert()`. The Nintendo GameCube OS provides some basic means to perform these functions.

Code 9 - Debugging functions

```
void    OSReport          ( char* msg, … );
void    ASSERT            ( int expression );
void    ASSERTMSG         ( int expression, char* msg );
void    OSHalt            ( char* msg );
#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)

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

typedef void (*OSErrorHandler)(OSError error, OSContext* context, ...);

OSErrorHandler   OSSetErrorHandler( OSError error, OSErrorHandler handler );
```

`OSReport` is the Nintendo GameCube OS equivalent of `printf` (except that on the final console, its output will always be directed to the serial port).

`ASSERT` is the Nintendo GameCube equivalent of the standard C `assert` macro. When the given expression evaluates to false, this implementation halts the machine while dumping a message indicating the file and line number of where the `assert` failed.

`ASSERTMSG` is identical to `ASSERT` except that it takes an additional message, *msg*, that is displayed when the expression fails to evaluate to true.

`OSHalt` simply stops the machine, displaying the given message *msg*, along with the file and line number of where the `OSHalt` was invoked.

`OSProtectRange` sets main memory access protection on up to four ranges. The *addr* argument will be rounded DOWN to the closest 1024-byte boundary, while the end address, i.e., *addr* + *nBytes*, will be rounded UP to closest 1024-byte boundary. Once a range is specified, any main memory access (from Gekko, TX, PE, VI, DI...) that violates the specified memory control setting (`OS_PROTECT_CONTROL_*`) will raise an `OS_ERROR_PROTECTION` error.

`OSSetErrorHandler` installs an error handler of the specified error type (shown in the following table). An error handler captures exceptions generated by the Gekko CPU. `OSInit` installs default error handlers for every error type. These error handlers print out error messages to the serial output line. A game program can overload the default error handlers if necessary.

Table 2 - Error types

Defined Name	Description
OS_ERROR_MACHINE_CHECK	Machine check
OS_ERROR_DSI	DSI
OS_ERROR_ISI	ISI
OS_ERROR_ALIGNMENT	Alignment
OS_ERROR_PERFORMACE_MONITOR	Performance monitor
OS_ERROR_PROGRAM	Program
OS_ERROR_TRACE	Trace
OS_ERROR_BREAKPOINT	Instruction address break point
OS_ERROR_PROTECTION	Memory protection error

For the errors shown above, `OSErrorHandler` takes as its third and fourth arguments *dsisr* and *dar*, which are of type `u32` as shown:

```
void (*OSErrorHandler)( OSError error, OSContext* context, u32 dsisr, u32 dar );
```

dsisr and *dar* contain the corresponding Gekko register values at the time the error occurred. The *context* parameter contains Gekko GPR and other register values (except FPR) from the time the error occurred. *context->srr0* contains the effective address of the instruction that caused the error. For DSI and ISI, the memory address that violates memory protection is held in *context->dar*. Refer to the *IBM Gekko RISC Microprocessor User's Manual* for more details.

Notes:

- Do not install error handlers for `OS_ERROR_PROGRAM`, `OS_ERROR_TRACE`, or `OS_ERROR_BREAKPOINT` when using the debugger. These are used by the debug monitor.
- `OS_ERROR_PROTECTION` will be triggered when Gekko touches memory range from the end of main memory address (24MB or 48MB as set by `setsmemsize` command) to 64MB, or when Flipper detects any memory access violation against the settings specified by `OSProtectRange`. dar contain physical memory address that violates the memory protection. dsisr will be one of `OS_PROTECT_ADDRERR_BIT` or `OS_PROTECT_`*n*. [September 2001 or later SDK only]

5 Cache control

5.1 Cache description

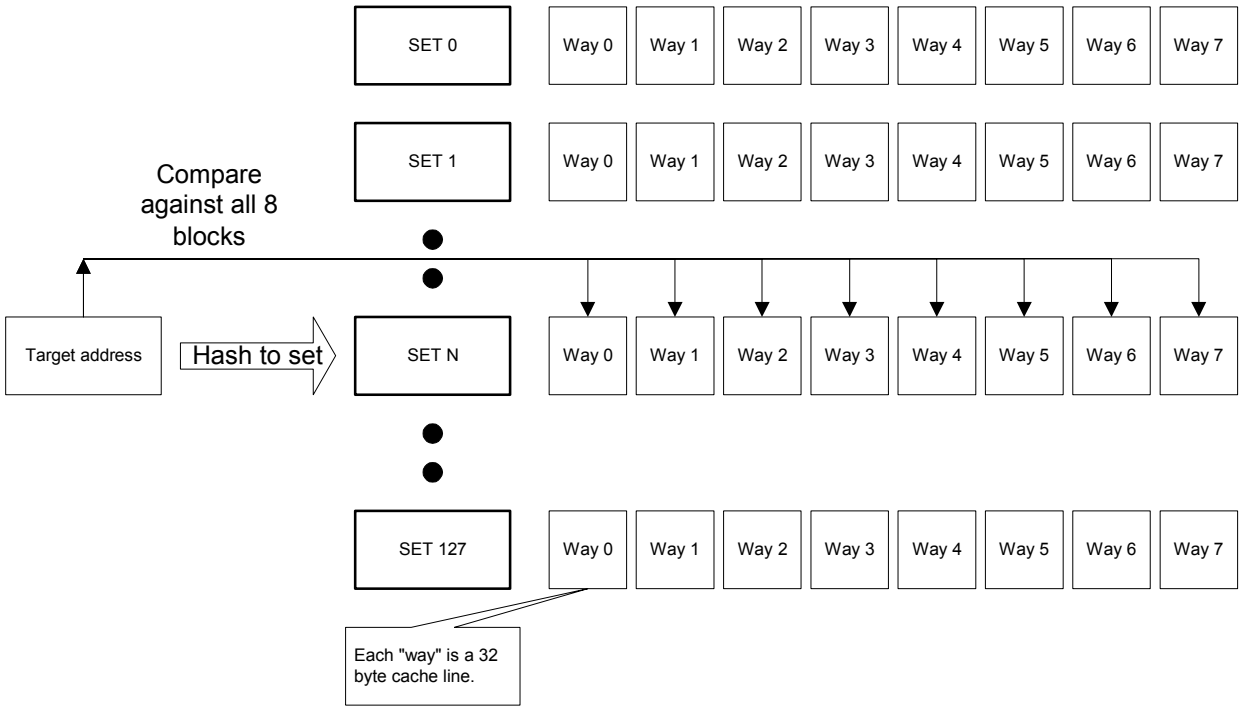
The Gekko CPU has separate instruction and data L1 caches, each 32KB, and one unified L2 cache of 256KB. The L2 cache is completely transparent to the application, and therefore will not be mentioned again in this section.

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.

The Nintendo 64 (N64) has a direct-mapped cache, which means that if any cache lines collide, the existing line will be cast out immediately. The associativity of the Gekko CPU's caches means that code placement is not as critical as it is for the N64.

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 coherent 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 3 - 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 10 - Cached and uncached access to memory

```
#include <dolphin.h>

void main (void)
{
u32* cachedAddress              = (u32*)OSGetArenaLo();
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 11 - Address translation

```
void* OSCachedToUncached(void* addr);
void* OSUncachedToCached(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 12 - Storing and invalidating cache lines

```
#include <dolphin.h>

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

    OSInit();

    cachedAddress   = (u32*)OSGetArenaLo();
    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 13 - 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 14 - 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 (e.g. 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 in the cache blocks associated with the *nBytes* value 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 wish 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.

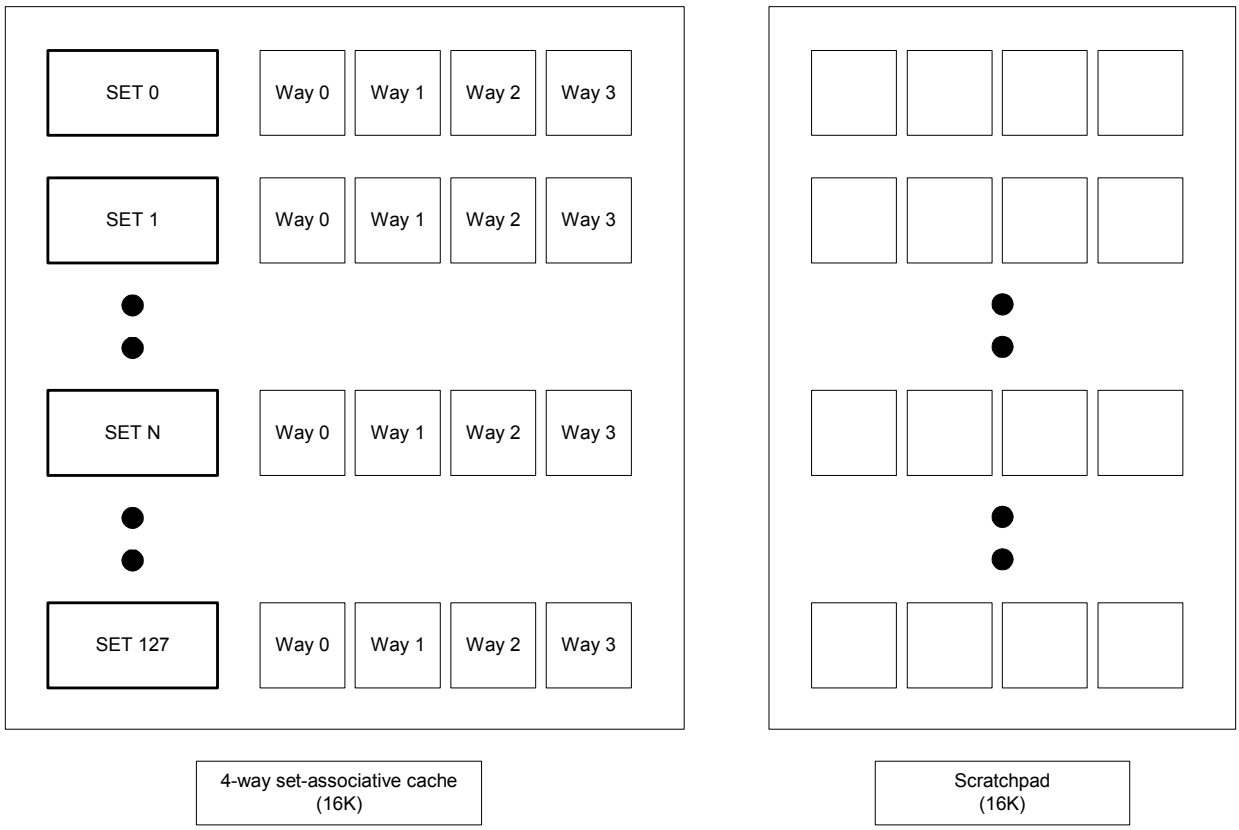
5.4 Locked cache operation

Note: The locked cache API is still undergoing development and profiling, and is thus apt to change. The basic functionality, however, will remain the same.

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

- **Normal** mode functions as a 32KB 8-way set-associative cache.
- **Locked Cache** mode partitions into a 16KB 4-way set-associative cache, and a 16KB scratchpad buffer with DMA engine.

Figure 4 - 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 scratch-pad 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.

We expect developers to leave the locked cache enabled throughout the execution of a game, although its usefulness exists primarily 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 nontrivial 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 27 provides an estimate of the cycle cost of this API.

5.4.2 DMA engine

The Gekko 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 (4KB).

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 in the next buffer.

Code 15 - 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;

    OSInit();
    LCEnable();

    arenaLo = OSGetArenaLo();
    arenaHi = OSGetArenaHi();
    :
    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 16 - 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 8-way set-associative cache. See "5.4.6 Enabling and disabling the locked cache" on page 27 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 Lower-level locked cache API and demos

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

Code 17 - 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;

    OSInit();
    LCEnable();

    arenaLo = OSGetArenaLo();
    arenaHi = OSGetArenaHi();
    :

    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 18 - Lower-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 19 - 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 27 for more details.

`LCQueueLength` returns the current length of the DMA queue.

Note: It performs a `sync` instruction which 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 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., 37.5 to 48.8 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 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 (5 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 (42.5 – 55 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, developers should 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 Nintendo GameCube OS maps the locked cache addresses out into 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 you have set. For instance, consider the following simple loop:

Code 20 - A 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 one 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 nonetheless.

6 Time

Nintendo GameCube 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 Gekko CPU's 64-bit time base register, it increments about every 24.7 nanoseconds (12 CPU cycles).
- **Alarm** – 64-bit one-short/periodic alarm with 24.7-nanosecond accuracy.

6.1 Real-time clock

The Gekko CPU has a 64-bit time base register. This time base register increments every twelve CPU cycles, or about 24.7 nanoseconds. At boot time, the OS initializes this time base register to the number of ticks since 0:00 AM January 1, 2000.

The 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, make references to the time base register for implementing time-critical code.

The Nintendo GameCube 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 21 - Time APIs

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

#define OSDiffTick(tick1, tick0)        …

// the following macros can actully 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 Gekko 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 42 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 Nintendo GameCube 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 any time in the future.

The application sets an alarm, and when the specified time elapses, the OS invokes an alarm handler. The following program demonstrates how to set up and handle an alarm:

Code 22 - Timer program

```
#include <dolphin.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;

    OSInit();

    // initialize the alarm module
    OSInitAlarm();

    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 23 - Timer APIs

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

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

    // Periodic alarm
    OSTime          period;
    OSTime          start;
};

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

The alarm module must be initialized first with `OSInitAlarm`. 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

Since the Nintendo GameCube OS does not require the use of threads, the default synchronization primitive is the ability to enable and disable interrupts. Interrupts 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, effectively. As such, they run with interrupts *disabled* on the current stack.

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

Code 24 - 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 25 - Critical section

```
BOOL enabled;

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

The Nintendo GameCube OS can malfunction if applications disable interrupts too long. The longest time applications can disable interrupts safely 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 Gekko 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 (e.g., data decompression, dynamic linking, etc.) without slowing down the game frame rate. The Nintendo GameCube OS supports a simple mechanism to run such time-consuming functions in the background (i.e., while the game main loop is idle). If more than one background task is necessary, however, the threads API (see "9 Threads" on page 37) may be more appropriate.

Code 26 - Idle function (background task) example

```
u8  Stack[4096];
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)
{

    OSInit();
    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 26 - Idle function (background task) example" on page 35 shows a simple background task that adds up all the numbers from 0 to 1,000,000. The APIs used are:

Code 27 - Idle function (background task) APIs

```
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 is processed while `VIWaitForRetrace` is in progress (i.e., while the game main loop is waiting for the next vertical retrace interrupt).

The idle function takes a single parameter *param*. The idle function will run with its own stack (separate from the main stack), specified by the *stack* parameter. You must make sure that this stack is big enough to complete the idle function. 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.

You can examine the return value of `OSGetIdleFunction`; it should be `NULL` if the idle function has completed execution.

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 Nintendo GameCube development kit are not reentrant.

9 Threads

The Nintendo GameCube OS provides transparent threads support; that is, if your application does not require threads, you do not need to know anything about threads. This is different from the N64, where an idle thread and scheduler always have to be created.

The Nintendo GameCube OS threads system provides all of the functionality of the N64 threads and messaging system, as well as 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 *Dolphin 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. Unlike N64 threads, 0 is the highest priority, while 31 is the lowest. 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 35) 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 read completes, the application-specified callback runs.
3. The application's callback wakes the decompression thread and returns. Ordinarily, rescheduling would occur now, 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.

9.3 Thread creation

The following code example demonstrates how a thread is created. The created thread has exactly the same semantics as N64 threads.

Code 28 - Thread creation example

```
OSThread      Thread;
u8            ThreadStack[4096];
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 28 - Thread creation example" on page 39 are:

Code 29 - 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, it behaves exactly like an N64 thread; i.e., 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 30 - Using OSJoinThread" on page 41.

`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 30 - Using OSJoinThread

```
OSThread      Thread;
u8            ThreadStack[4096];
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)
{

    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
        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 30 - Using OSJoinThread" on page 41 is almost identical to "Code 28 - Thread creation example" on page 39, 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 indicated it was 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 31 - 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 telling 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 Nintendo GameCube OS provides the same messaging functionality of the N64, along with a few extra 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 reentrance

Most of the Nintendo GameCube OS libraries are not thread-safe. It is the responsibility of the developers to arbitrate access to shared resources (e.g., GX function calls, `printf` output) with the thread synchronization primitives.

We have, however, made the Optical Disc (DVD) and Controller (PAD) libraries thread-safe.

9.4.2 Synchronizing by disabling the scheduler

The most basic synchronization primitive is the ability to stop threads from being context switched. 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 32 - 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 33 - 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 Nintendo GameCube OS provides another fast and primitive thread synchronization mechanism where threads can be put to sleep and awakened from queues. These primitives are used by most of the other synchronization methods.

Code 34 - 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

N64 developers are probably very familiar with messages. The Nintendo GameCube OS implementation is quite fast and is 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 35 - Message API example

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

OSThread       Thread;
u8             ThreadStack[4096];

// 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;

    OSInit();
    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 virtually identical to the N64 message API:

Code 36 - 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, this 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 37 - Mutex and yielding example

```
OSMutex         PrintMutex;
OSThread        Thread;
u8              ThreadStack[4096];

// 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)
{
    OSInit();

    // 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(&Thread1);

    // 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 38 - 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 a chance 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 forever.

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 Nintendo GameCube 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 condition 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 39 - 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[4096];

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;

    OSInit();
    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 40 - Condition 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 manage to re-acquire the mutex and proceed.

9.5 Context switching

In order to help you better understand the behavior of this system, this section briefly describes how the Nintendo GameCube OS saves and restores machine state.

Every execution context, whether it be 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 Gekko 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 Nintendo GameCube 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 (e.g., 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 Nintendo GameCube OS maintains a linked list of all the active threads that are runnable, running, waiting, or moribund. This linked list is called the *active thread queue.* In the Nintendo GameCube 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 41 - 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()`
- `sndInit` and `sndPushGroup` in MusyX

10 Fast float/integer casting

The Gekko 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 C language API that executes the proper instructions to perform the fastest possible cast. The API is implemented as inlined assembly functions, and each cast costs two cycles despite being called from C code.

Since the Gekko CPU must perform a load and a store, it is essential that the data addresses hit in the cache.

10.1 Initializing the fast cast API

Code 42 - OSInitFastCast

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

`OSInitFastCast` initializes the Gekko CPU's quantization registers (GQRs) to known values. GQRs control the way paired-single load/stores work. For instance, GQR 0 is always set to 0, which means that the data representation in memory is a simple 32-bit floating point value. GQR 2 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 3 - Quantization register values

Quantization Register	Initialization Value
GQR 2	Load u8 / Store u8. No scaling.
GQR 3	Load u16 / Store u16. No scaling.
GQR 4	Load s8 / Store s8. No scaling.
GQR 5	Load s16 / Store s16. No scaling.

Your application may prefer to apply scaling to the values. It should be easy for you to develop your own similar API based on our implementation. Since the API is inlined, you can see the source in the `OSFastCast.h` header file.

Note: In the future, the compiler will take over GQRs 0 and 1, so all quantization-related APIs must avoid modifying these two registers.

10.2 Fast casting routines

This section briefly describes the fast casting API:

Code 43 - 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);
```

The names of each API are self-explanatory.

Note: Each takes typed pointers to your variables. This is necessary since each must perform a load and a store. If the API itself had to allocate memory for this purpose, it would take more than two cycles to perform each cast. By letting the compiler allocate stack space for the variables instead, or by using your own arrays to store data, you can allocate space for all local variables at once.

11 Profiling

The Nintendo GameCube OS provides two key means of profiling: stopwatches, and the Gekko CPU's performance monitors.

11.1 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 Gekko CPU's time base, and so will have 20ns accuracy.

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

Code 44 - Stopwatch code example

```
#include <dolphin.h>

#define OUTER_ITERATIONS  50
#define INNER_ITERATIONS  100

OSStopwatch         MySW;

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

    OSInit();

    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 45 - Stopwatch program output

```
Stopwatch demo program
Times 5000 matrix concatenations

Each hit is 100 matrix concats:
Stopwatch [100 concat stopwatch]    :
    Total= 2611 us
    Hits = 50
    Min  = 52 us
    Max  = 56 us
    Mean = 52 us
Stopwatch Demo complete in "stopwatchdemo.c" on line 54.
```

The preceding example uses these APIs:

Code 46 - 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's 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 47 - Stopwatch intervals

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

11.2 Performance monitors

The Gekko 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 48 - 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 49 - 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 `dolphin/include/dolphin/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 the *IBM Gekko RISC Microprocessor User’s Manual*.

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

Code 50 - 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.