BIOS Emulation

Design Specification

Version: 1.2
Date: July 30, 2007

Authors	**Reviewers**	**Approvals**
Ramsey Harris	Karl Wechsler	Karl Wechsler
	Alan DeMars	Nick Lethaby
	Nitya Ramdas	
	Dave Russo	

Template Version: 2.0

IMPORTANT NOTICE

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

Contents

List of Figures

List of Tables

1 Introduction

This document details the design for emulating the BIOS RTOS on a High-Level Operating System, such as Windows and Linux.

1.1 Purpose & Scope

The purpose and scope of this document is to describe the feature design with sufficient detail to facilitate a comprehensive design review, and to provide sufficient detail for implementation.

1.2 Terms & Abbreviations

Term	Description
HLOS	High-Level Operating System, such as Microsoft Windows or Linux
Task	BIOS Task
Swi	BIOS Software Interrupt
Hwi	BIOS Hardware Interrupt

Table 1: Terms & Abbreviations

1.3 References

Ref #	Book/Document Title, Revision/Version, Date
1.	BIOS 6 Internals, Version 0.4, Nov 4, 2006
2.	Advanced Windows, Third Edition, 1997
3.	Inside Windows 2000, Third Edition, 2000
4.	

Table 2: References

1.4 Document History

Rev	Date	Description of Changes	Editor
1.0	22-Nov-2006	Initial version.	Ramsey Harris
1.0	13-Dec-2006	Submitted for formal review.	Ramsey Harris
1.1	29-Jul-2007	Applied review comments.	Ramsey Harris
1.2	30-Jul-2007	Describe design changes in Appendix A.	Ramsey Harris

Table 3: Document History

2 Architectural Design

The architectural design focuses on the high-level aspects of the design and how it integrates with the overall system.

2.1 Justification

This feature is motivated by customer needs. When developing a software module, such as a Codec, customers will typically write a “Golden C” version first. Once the software module is functioning properly, the golden C version is used as a baseline for porting the software to specific target platforms.

When developing the Golden C version, it is often preferable to do this work on a High-Level Operating System (HLOS), such as Windows or Linux. This allows the use of HLOS tool-chains for code profiling and validation. Providing a BIOS Emulation layer which runs on the HLOS, makes this effort more efficient for the customer. It allows them to write a native BIOS application which builds and runs as a native executable on the HLOS.

It should be noted that the emulation layer does not attempt to accurately model target cycles. This layer is not intended for such analysis. Nor is it designed for target specific performance analysis. It is designed to develop program logic using BIOS running on a HLOS.

[Need to justify the emulation of the BIOS scheduler.]

2.2 Requirements

R1 Reproduce the BIOS Scheduler behavior with respect to scheduling of BIOS Tasks, Swi’s, and Hwi’s.

R2 Platform independent application code shall not require any modifications in order to be used with the BIOS Emulation layer. All BIOS API’s, with the exception of Hwi, are fully supported in the Emulation Layer. The Hwi object emulates the C64x+ platform as closely as possible.

R3 On Windows, the final executable binary shall be a native Windows user level executable.

R4 No custom HLOS kernel mode support will be required. The final executable shall be a native HLOS user-mode binary.

R5 Any BIOS task shall be able to make Windows Win32 API calls with the understanding that any blocking call will block the entire emulation process. In addition, creating new threads is not supported.

R6 Will support all target independen APIs.

R7 Must be compatible with Windows VC 8.0 tool chain.

R8 Must be compatible with Windows TCP/IP stack.

R9 Must be compatible with Windows development tools, such as Purify and Bounds Checker.

R10 When using a memory profiling tool (such as Purify), a special Windows version of the memory manager is required to get accurate results.

2.3 Assumptions

1. The HLOS uses a priority based thread scheduler.
2. The HLOS will use round-robin scheduling for threads of the same priority.
3. Immediate thread switch to a higher priority thread when it becomes ready to run.

2.4 Constraints

1. A thread’s stack usage and context management is owned by the HLOS kernel which makes it inaccessible to user-level code. Therefore, these aspects of the BIOS kernel will not be emulated in the emulation layer.

2. Nested interrupts will not be supported. Each emulated interrupt will be allowed to complete before another interrupt is raised.

2.5 High Level Description

The high-level description provides sufficient detail to help you understand the functional behavior of the design but without the implementation details. See Section 3, Detailed Design, for a detailed description.

2.5.1 System Overview

BIOS emulation will be supported by a new “emulation” layer of software positioned between BIOS and the High-Level Operating System (HLOS). The BIOS Emulation layer provides a HLOS implementation of the platform proxy modules used in BIOS. These proxy modules provide interfaces for

- hardware interrupts
- thread context switching
- general purpose timers
- system clock tick counter

To implement these interfaces, some hardware functionality will also be emulated in the emulation layer because the HLOS does not allow any direct access to hardware. A block diagram comparing a C64x implementation with that of a HLOS implementation is shown in Figure 1.

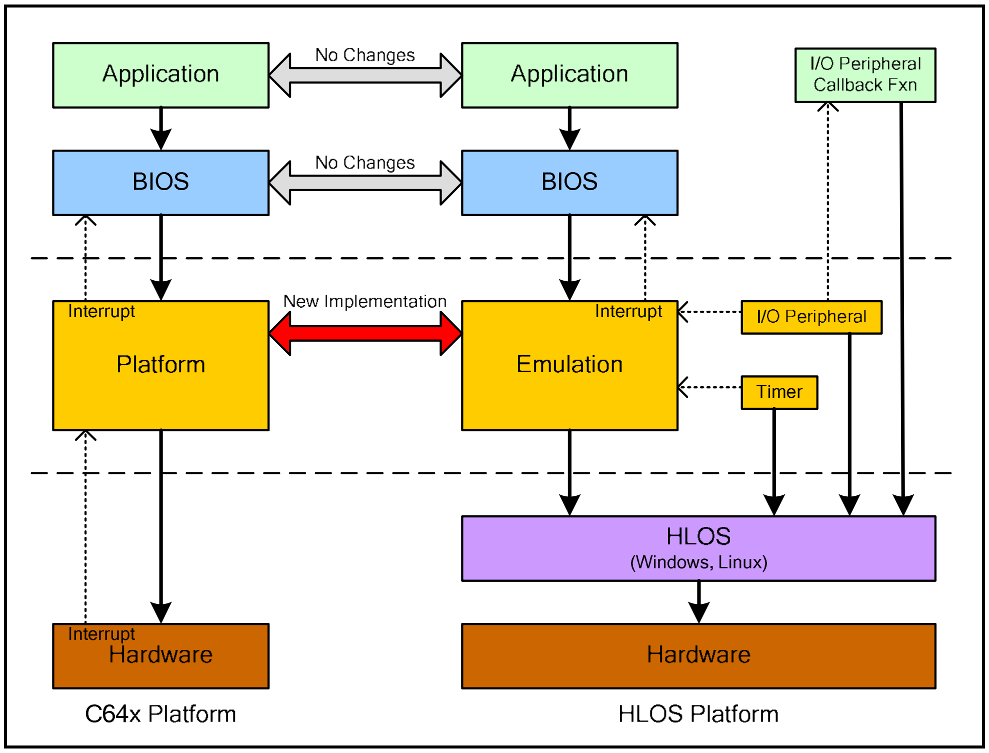

Figure 1: System Block Diagram

Application code written in C which makes only BIOS API calls should not require any changes. However, any code written for peripheral control will need to be replaced. In addition, when using the BIOS Emulation layer, the application must provide an I/O peripheral callback function. This callback function provides the hook needed to read/write data to disk for testing purposes. The callback function should not make any BIOS API calls; it should only call into the HLOS user-level libraries.The callback is invoked by the BIOS Emulation layer and it must return in order for the system to continue running. When the callback returns to the emulation layer, an interrupt will be automatically generated to simulate the peripheral behavior.

The BIOS Kernel does not require any changes. Through RTSC configuration, the kernel will bind with the appropriate proxy modules relevant to the target platform (i.e. the HLOS platform or a hardware platform).

When building the application, RTSC configuration is used to select the runtime platform. Through these configuration options, the application can bind with the appropriate modules which are hardware or emulation specific. This will most likely pertain to peripheral and/or test framework code.

On hardware platforms, peripheral devices typically raise interrupts to the CPU who then invokes the Hwi Dispatcher to service the interrupt. To emulate this behavior, the BIOS Emulation layer provides support for I/O and timer peripherals. These peripherals will simulate an interrupt which causes the emulation layer to call up into BIOS to invoke the Hwi Dispatcher. This is done asynchronously with respect to BIOS threads.

2.5.2 Emulation Objects

Central to the emulation layer, is the Emulation Scheduler. It manages the scheduling of all HLOS threads used by the emulation layer. Figure 2 contains the emulation object diagram.

The BIOS task thread context is emulated by two HLOS threads, a synchronizing object for each thread, and a flag to indicate the preemption state of the task. The task object has a context pointer which is used as an opaque pointer into the emulation layer task context structure. To faithfully reproduce the BIOS scheduler behavior, the Emulation Scheduler must manage the HLOS threads in such a way as to reproduce the same task scheduling behavior as the BIOS scheduler.

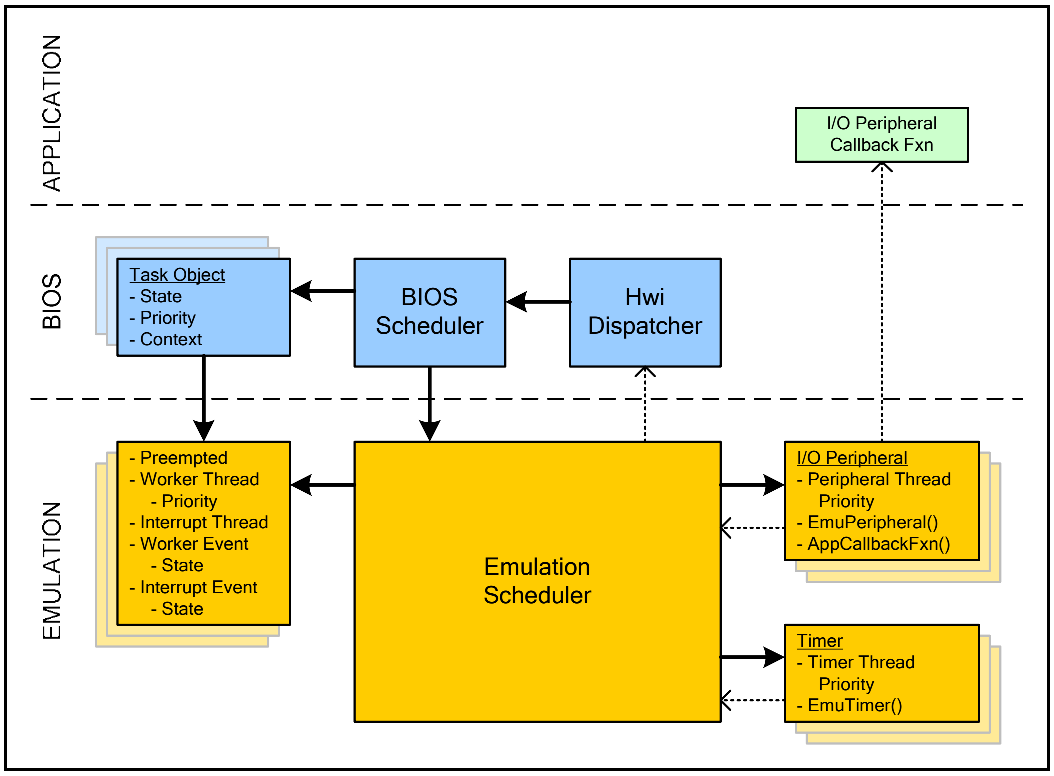

Figure 2: Emulation Object Diagram

Each BIOS task thread is emulated by two HLOS threads. Two threads are needed in order to support asynchronous task preemption. One thread, the worker thread, executes the task function. The second thread, the interrupt thread, is used to preempt the worker thread when emulating an asynchronous task switch. This is done by setting the interrupt thread priority higher than the worker thread, and then blocking the interrupt thread on an event object. When an interrupt is raised by a peripheral thread, it will signal the event object which releases the interrupt thread. Windows will preempt the worker thread in favor of the interrupt thread because of its higher priority. Thus, and asynchronous interrupt of the worker thread has been achieved.

Additional threads are created in the emulation layer to emulate I/O peripherals (such as a McBSP) and timer peripherals (such as a general purpose timer). The scheduling of the peripheral threads is also managed by the emulation scheduler.

2.5.3 Task Switching

Each BIOS task will be emulated by two HLOS threads; a worker thread and an interrupt thread. This is necessary to support the two fundamental types of task switching: 1) synchronous context switch (performed by the worker thread), and 2) asynchronous context switch (performed by the interrupt thread).

All synchronous task switching is initiated in the application code by making use of BIOS synchronizing objects, such as semaphores. Once the application calls into BIOS to either wait or signal a synchronizing object, the BIOS scheduler will determine if the calling thread is allowed to proceed or should yield the processor. If the BIOS scheduler determines a thread switch needs to take place, it will manage the thread switching directly and then call into a task-swap function to complete the context switch. It is at this point that the emulation layer is entered and the actual thread switch is performed. This is done by making use of the HLOS synchronizing objects. A timing diagram illustrating synchronous BIOS task switching along with the emulation thread switching is given in Figure 3.

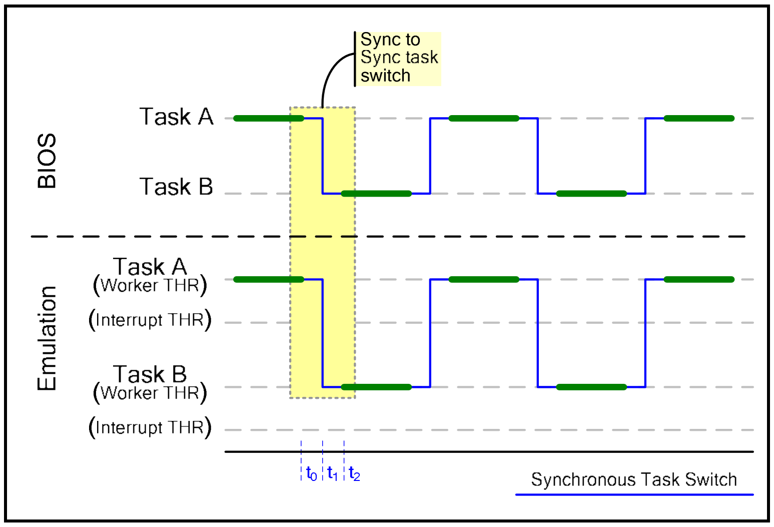

Figure 3: Synchronous Task Switch Timing Diagram

An asynchronous task switch is performed in response to servicing an interrupt which has altered the ready state of a task. When the task scheduler is invoked from the tail end of the HWI Dispatcher, it is done asynchronously with respect to the currently running task. To support this asynchronous behavior in the emulation layer two threads must be used. One thread is used to emulate the peripheral device which generates the interrupt. The second thread is used to emulate the asynchronous task switch of the currently running thread. This happens in the following manner. When the peripheral thread generates an interrupt, it calls into the emulation layer to raise an interrupt. The emulation layer scheduler knows which BIOS thread is currently running and will signal the appropriate interrupt thread which is currently blocked on a HLOS synchronizing object. The interrupt thread runs at a higher priority than the worker thread, and thus the HLOS will perform an immediate, asynchronous context switch to the interrupt thread. The BIOS Hwi Dispatcher runs in the context of this interrupt thread. After the Hwi Dispatcher has called the ISR, it will invoke the BIOS Task Scheduler who will perform a context switch to the new target task. At the point where the BIOS Task Scheduler calls the task swap function, the emulation layer performs the context switch to the target thread by signaling the target threads synchronizing object (if needed) and then blocking the currently running interrupt thread.

Figure 4 illustrates how the emulation layer performs an asynchronous task switch from Task A to Task B followed by a synchronous task switch from Task B back to Task A. Note that Task B runs at a higher priority than Task A (which never blocks).

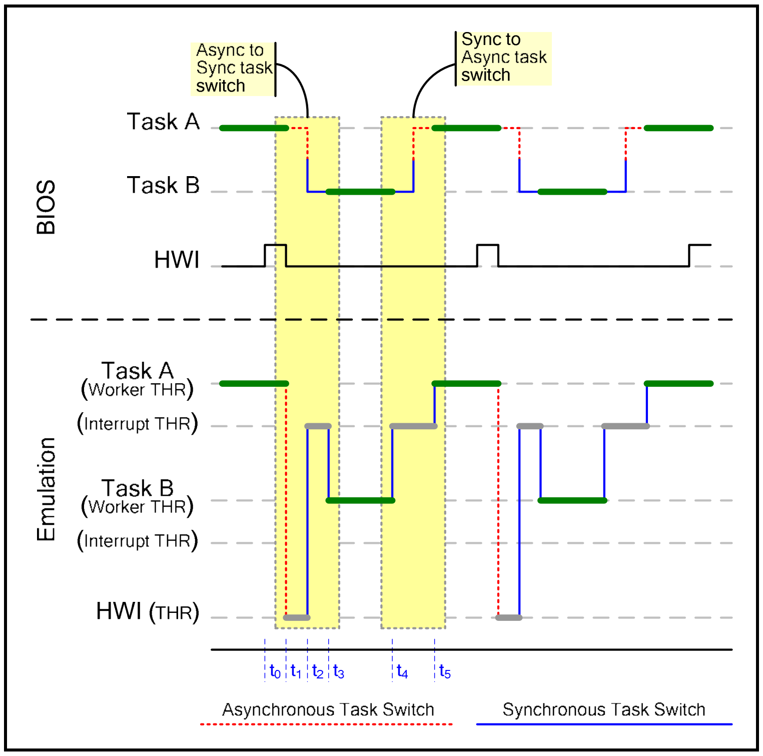

Figure 4: Synchronous & Asynchronous Task Switch Timing Diagram

Figure 5 illustrates the timing between two tasks which are preempting each other asynchronously. The top half of the diagram represents two tasks and a hardware interrupt source as managed by BIOS on a hardware platform. The bottom half illustrates how the emulation layer would support the same task switch sequence by using two HLOS threads for each BIOS task and an additional HLOS thread for the hardware interrupt source. The diagram begins with Task A running.

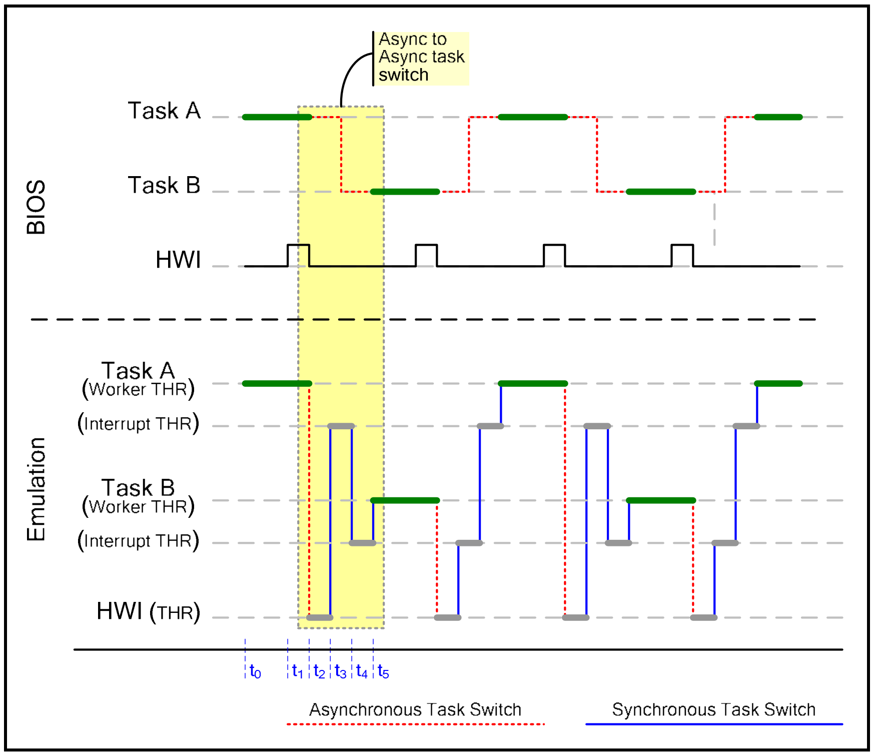

Figure 5: Asynchronous to Asynchronous Task Switch Timing Diagram

2.5.4 Task Startup

- [Maybe delete this section?]
- Timer Proxy startup
- Hwi Startup for EMU startup code

2.5.5 BIOS Timer

2.5.6 BIOS Bench

The BIOS clock is emulated by approximating the number of cycles performed by the CPU between calls to getHTime. This is done by summing the thread execution times of each BIOS emulation thread.

2.5.7 Hardware Interrupts and Peripheral

- Overview of peripheral threads

3 Detailed Design

There are four fundamental cases for task switching in the BIOS scheduler. These are detailed in Section 3.1 with respect to a native DSP implementation and with a BIOS Emulation on a HLOS. SWI scheduling is detailed in Section 3.3, and the Hwi Dispatcher is detailed in Section 3.4. The details of the BIOS clock emulation is described in Section 3.5

On Windows, thread priority will be used to manage which threads run at any given time. Specifically, the worker threads will run at normal priority and their supporting interrupt threads will run at the highest priority. This ensures that when a peripheral thread raises an interrupt, the currently runing worker thread will be preempted by its interrupt thread.

3.1 Emulation Startup

- Okay to use static variable to identify initial EMU call if startup hook not available
- Timer proxy startup
- Hwi proxy startup

3.2 Task Scheduling

BIOS task scheduling is dictated by the BIOS Task Scheduler but implemented by the Emulation Scheduler. The Emulation Scheduler will manage the HLOS threads, which represent the BIOS Task threads, in such a way as to faithfully reproduce the BIOS Scheduler behavior. In addition to these threads, the Emulation Scheduler also manages other HLOS threads which represent the peripheral devices. As a point of clarity, the Emulation Scheduler does not perform the actual thread context switch, this is done by the HLOS kernel. All the Emulation Scheduler can do is manipulate thread priorities and thread synchronizing objects in such a way as to compel the HLOS kernel to switch threads in the order desired by the Emulation Scheduler.

3.2.1 Case 1 – Synchronous task switch to synchronously blocked task

In this instance, the currently running task yields the processor by blocking on a synchronizing object and then calls the task scheduler which switches to a task which had previously yielded the processor in the same manner.

This can be modeled by a writer/reader application as illustrated in Figure 6. In the writer/reader application, there are two tasks: a “writer” task and a “reader” task. Both tasks access a common buffer. The writer task fills the buffer with new data and the reader task drains the buffer. Two semaphores are used to synchronize the tasks. The WriteFlag semaphore is used to indicate when the writer task can start filling the buffer with new data. The writer task will block on this semaphore when it is done filling the buffer and the reader task will signal this semaphore when it is finished draining the buffer. In a similar fashion, the ReadFlag semaphore is used to indicate when the reader task can start draining the buffer. The reader task will block on this semaphore when it is done draining the buffer and the writer task will signal this semaphore when it is done filling the buffer. Both tasks are running at the same priority.

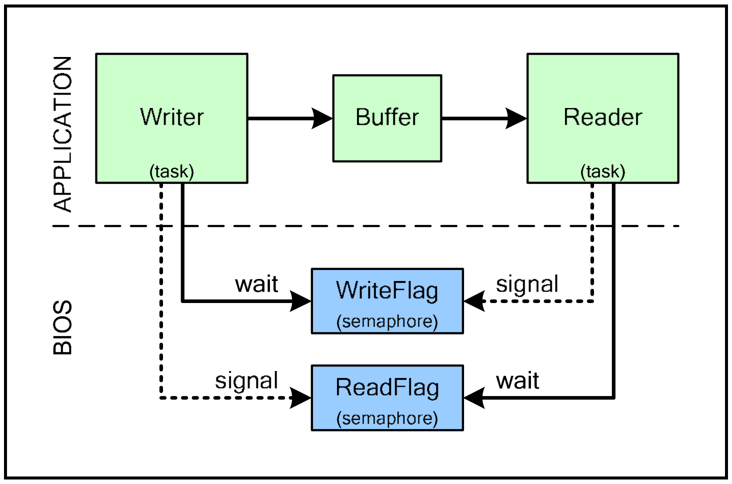

Figure 6: Writer/Reader BIOS Application

Our initial condition has the read task blocked on its semaphore and the write task running and about to start filling the buffer. When the running task signals the other task's semaphore, the task scheduler is not yet invoked because both tasks are running at the same priority and the running task has not yet blocked. But as soon as the running task blocks on its own semaphore, it then calls the task scheduler which switches to the other task.

A sequence diagram of the task switch sequence on a C64x platform is illustrated in Figure 7 and described in Table 4. The same task switch sequence on a HLOS is illustrated in Figure 8 and described in Table 5.

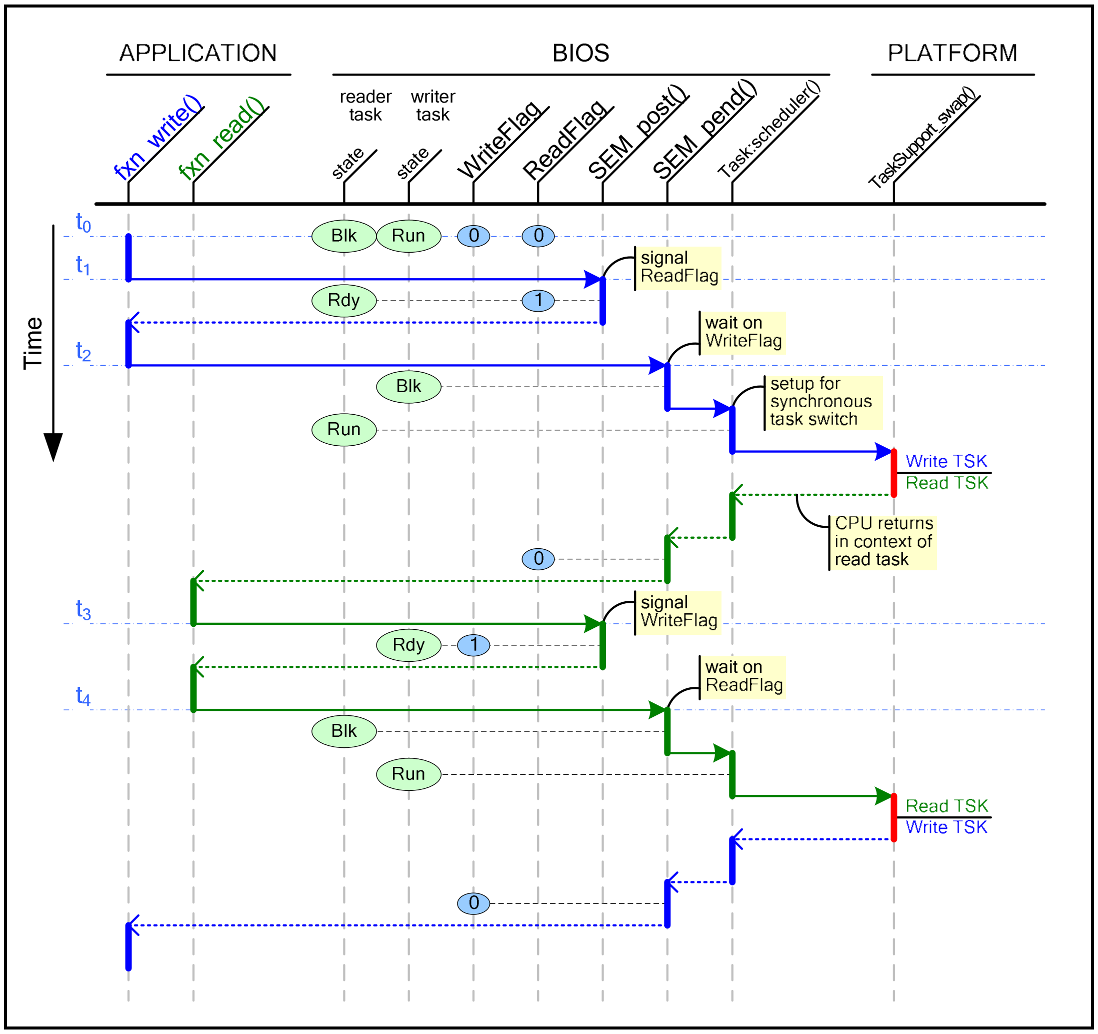

Figure 7: C64x Sync to Sync Task Switch

A detailed description of Figure 7 is given in Table 4.

Time	Description of Events
t_0	Initial condition. The read task is blocked and the write task is running in fxn_write(). Both the WriteFlag and ReadFlag semaphores are cleared.
t_1	The write task has finished filling the data buffer and calls SEM_post() to signal the ReadFlag semaphore. This causes the read task to transition into the “ready” state. The write task returns to fxn_write().
t_2	The write task waits on its WriteFlag semaphore. This causes the write task to transition to the “blocked” state because the WriteFlag semaphore is in the reset state. The write task calls into the task scheduler which transitions the read task into the “ready” state. The scheduler then calls into TaskSupport_swap() to perform the context switch from the write task into the read task. The CPU returns in the context of the read task. As the read task unwinds its stack, the ReadFlag semaphore is cleared and execution returns to the application fxn_read() function.
t_3	The read task has finished draining the data buffer and calls SEM_post() to signal the WriteFlag semaphore. This causes the write task to transition into the “ready” state. The read task returns to fxn_read().
t_4	The read task waits on its ReadFlag semaphore. This causes the read task to transition to the “blocked” state because the ReadFlag semaphore is in the reset state. The read task calls into the task scheduler which transitions the write task into the “ready” state. The scheduler then calls into TaskSupport_swap() to perform the context switch from the read task into the write task. The CPU returns in the context of the write task. As the write task unwinds its stack, the WriteFlag semaphore is cleared and execution returns to the application fxn_write() function.

Table 4: C64x Sync to Sync Task Switch Description

The same task sequence is now illustrated for a HLOS platform (Windows in this example).

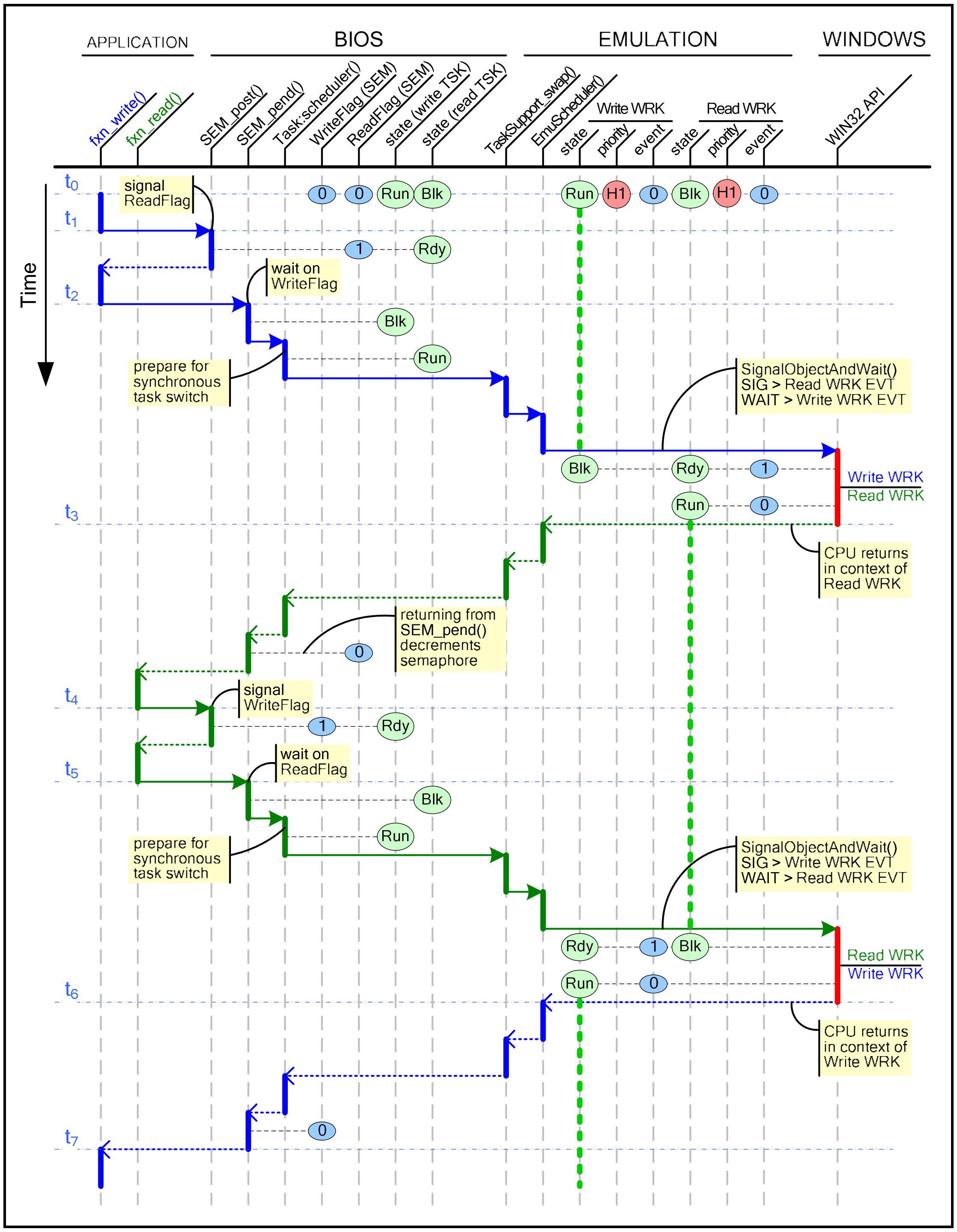

Figure 8: HLOS Sync to Sync Task Switch

A detailed description of Figure 8 is given below in Table 5.

Time	Description of Changes
t_0	Initial condition.
t_1	
t_2	
t_3	
t_4	
t_5	

Table 5: HLOS Sync to Sync Task Switch Description

3.2.2 Case 2 – Asynchronous task preemption to synchronously blocked task

In this instance, the currently running task does not yield the processor, but instead is preempted by the task scheduler (invoked through an interrupt) which switches to a task which had previously yielded the processor by blocking on a synchronizing object. The target task has a higher priority than the running task.

This can be modeled by an audio player application as illustrated in Figure 9. In the audio player application, there are two tasks: a “decode” task and the “idle” task. The idle task is always ready to run and runs only when there are no other tasks ready to run. It is also the lowest priority task. The decode task reads encoded audio data from an input buffer, and writes the decoded data to an output buffer. It then blocks on the DecodeFlag semaphore. A peripheral device, such as a McBSP, reads the decoded audio data from the output buffer and plays it out to a speaker. When the output buffer is empty it raises an interrupt to the CPU to indicate that more data is needed. There are two output buffers which the McBSP reads from; this allows the McBSP to read data from one output buffer while the decode task is filling the other.

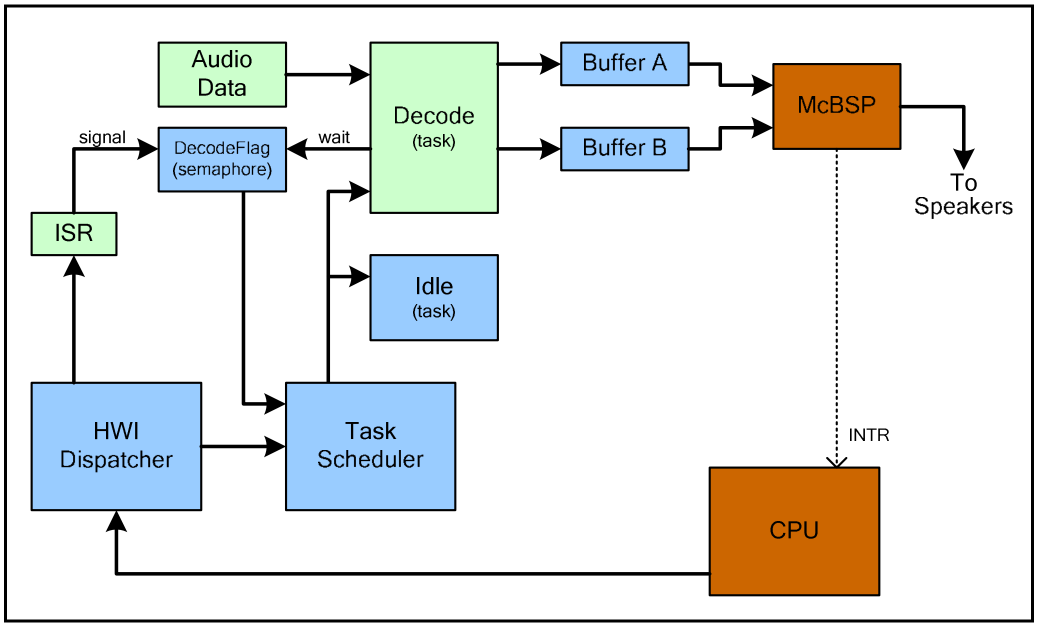

Figure 9: Audio Player BIOS Application

When the McBSP raises an interrupt, the CPU will invoke the Hwi Dispatcher to call the appropriate ISR. In this example, the ISR signals the DecodeFlag semaphore to indicate that an output buffer is empty and ready to be filled. Once the ISR returns to the Hwi Dispatcher, the dispatcher invokes the Task Scheduler which performs an asynchronous preemption of the idle task and switches to the decode task which had previously yielded the processor by blocking on a synchronizing object.

A sequence diagram of the task switch sequence on a C64x platform is illustrated in the first half of Figure 10 and described in Table 6. The same task switch sequence on a HLOS is illustrated in the first half of Figure 11 and described in Table 7.

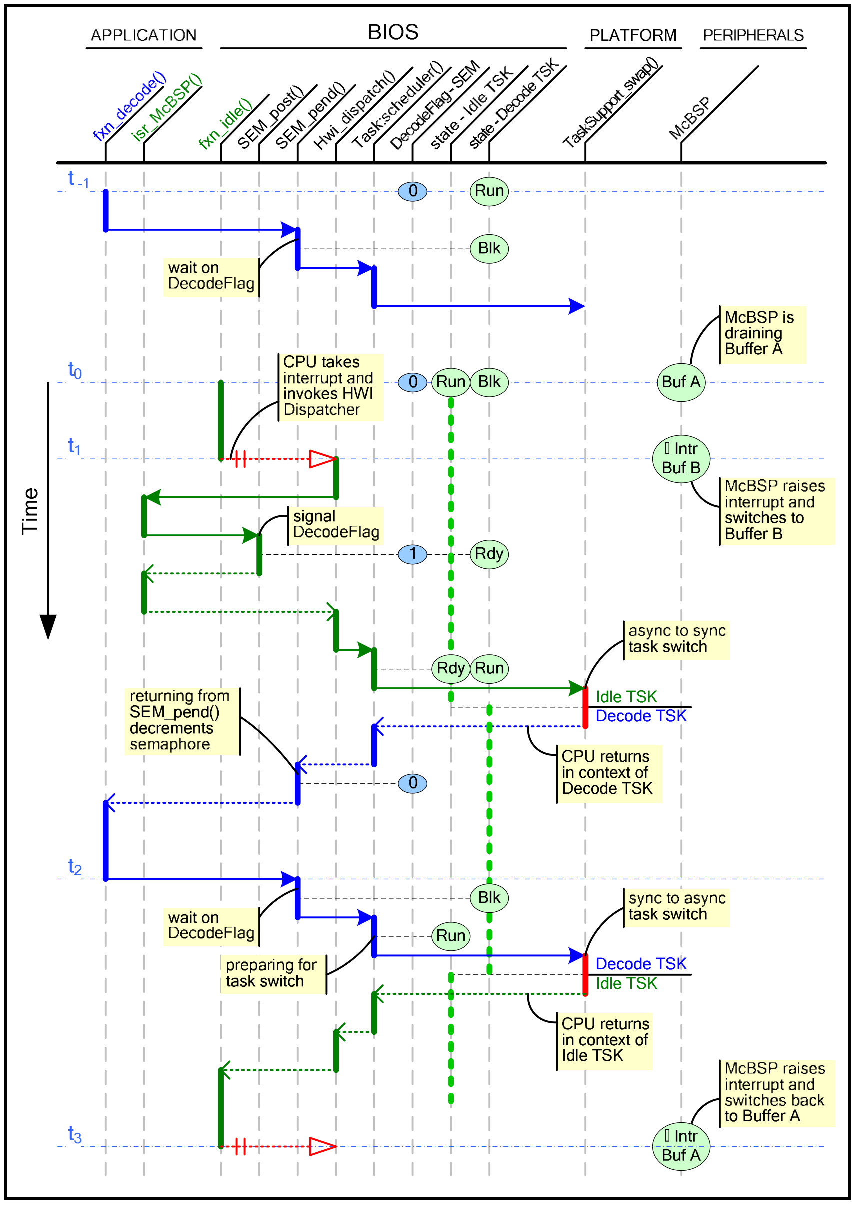
APPLICATION
BIOS
PLATFORM
PERIPHERALS
fxn_decode()
isr_McBSP()
fxn_idle()
SEM_post()
SEM_pend()
Hwi_dispatch()
Task:scheduler()
DecodeFlag - SEM
state - Idle TSK
state - Decode TSK
TaskSupport_swap()
McBSP
Time
t-1
t0
t1
t2
t3
Run
Blk
Rdy
wait on DecodeFlag
McBSP is draining Buffer A
Buf A
CPU takes interrupt and invokes HWI Dispatcher
Intr Buf B
McBSP raises interrupt and switches to Buffer B
signal DecodeFlag
async to sync task switch
Idle TSK
Decode TSK
returning from SEM_pend() decrements semaphore
CPU returns in context of Decode TSK
sync to async task switch
preparing for task switch
CPU returns in context of Idle TSK
McBSP raises interrupt and switches back to Buffer A
Intr Buf A

Figure 10: C64x Async and Sync Task Switch

Time	Description of Changes
t_0	Initial condition.
t_1	
t_2	
t_3	
t_4	
t_5	

Table 6: C64x Async and Sync Task Switch Description

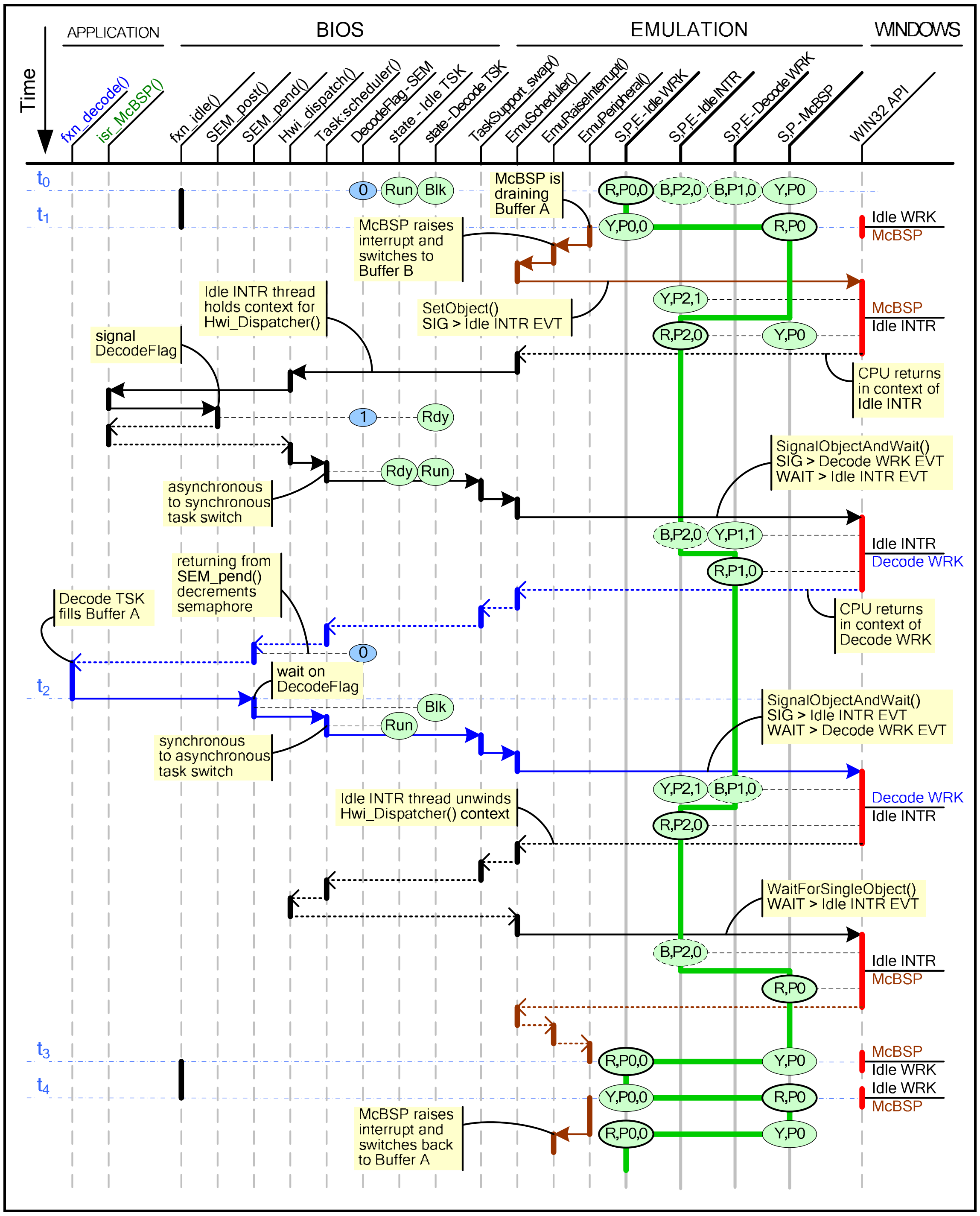
Time
APPLICATION
BIOS
EMULATION
WINDOWS
fxn_decode()
isr_McBSP()
fxn_idle()
SEM_post()
SEM_pend()
Hwi_dispatch()
Task:scheduler()
DecodeFlag-SEM
state - Idle TSK
state-Decode TSK
TaskSupport_swap()
EmuScheduler()
EmuRaiseInterrupt()
EmuPeripheral()
S,P,E- Idle WRK
S,P,E- Idle INTR
S,P,E- Decode WRK
S,P- McBSP
WIN32 API
t0
t1
t2
t3
t4
0
Run
Blk
McBSP is draining Buffer A
R,P0,0
B,P2,0
B,P1,0
Y,P0
Y,P0,0
R,P0
Idle WRK
McBSP
McBSP raises interrupt and switches to Buffer B
Idle INTR thread holds context for Hwi_Dispatcher()
SetObject() SIG > Idle INTR EVT
Y,P2,1
McBSP
Idle INTR
signal DecodeFlag
R,P2,0
Y,P0
CPU returns in context of Idle INTR
1
Rdy
SignalObjectAndWait() SIG > Decode WRK EVT WAIT > Idle INTR EVT
Rdy
Run
asynchronous to synchronous task switch
B,P2,0
Y,P1,1
Idle INTR
Decode WRK
R,P1,0
returning from SEM_pend() decrements semaphore
Decode TSK fills Buffer A
CPU returns in context of Decode WRK
0
wait on DecodeFlag
SignalObjectAndWait() SIG > Idle INTR EVT WAIT > Decode WRK EVT
Blk
Run
synchronous to asynchronous task switch
Idle INTR thread unwinds Hwi_Dispatcher() context
Y,P2,1
B,P1,0
Decode WRK
Idle INTR
R,P2,0
WaitForSingleObject() WAIT > Idle INTR EVT
B,P2,0
Idle INTR
McBSP
R,P0
R,P0,0
Y,P0
McBSP
Idle WRK
Y,P0,0
R,P0
Idle WRK
McBSP
McBSP raises interrupt and switches back to Buffer A
R,P0,0
Y,P0

Figure 11: HLOS Async and Sync Task Switch

Time	Description of Changes
t_0	Initial condition.
t_1	
t_2	
t_3	
t_4	
t_5	

Table 7: HLOS Async and Sync Task Switch Description

3.2.3 Case 3 – Synchronous task switch to asynchronously preempted task

In this instance, the currently running task yields the processor by blocking on a synchronizing object and then calls the task scheduler which switches to a task which had been preempted asynchronously.

This also can be modeled by an audio player application as described above in Section 3.2.2 and illustrated in Figure 9.

This case is the complimentary case to the one described in Section 3.2.2. When the decode task has filled the output buffer, it yields the processor by blocking on the semaphore and then calling the task scheduler which switches to the idle task which had previously been asynchronously preempted.

A sequence diagram of the task switch sequence on a C64x platform is illustrated in the second half of Figure 10 and described in Table 6. The same task switch sequence on a HLOS is illustrated in the second half of Figure 11 and described in Table 7.

3.2.4 Case 4 – Asynchronous task preemption to asynchronously preempted task

In this instance, the currently running task does not yield the processor, but instead is preempted by the task scheduler (invoked through an interrupt) who switches to a task that also had not yielded the processor but had also been preempted.

This can be modeled by an application which uses time-slice based task switching as illustrated in Figure 12. In this application, there are two compute tasks which never block; one task is computing the value of Pi, and the other task is computing the natural logarithm, *e*. A peripheral timer is used to raise an interrupt every 10 milliseconds to signal the CPU that it is time to switch from one compute task to the other.

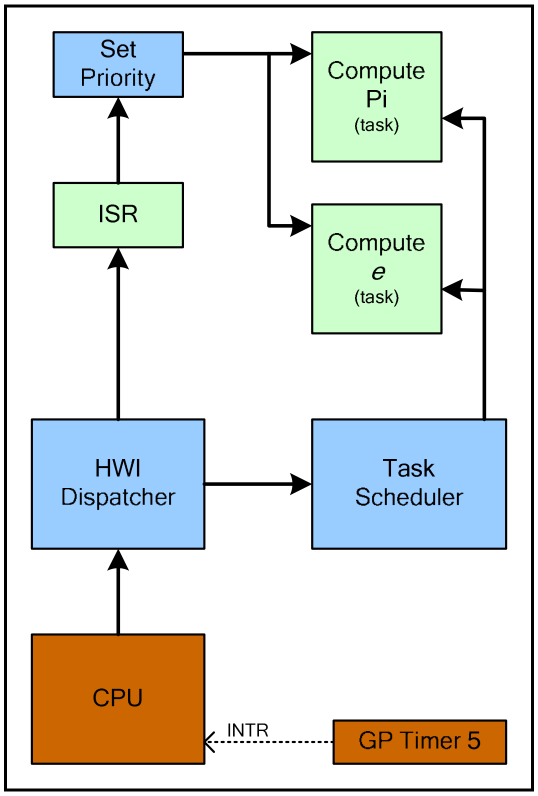

Figure 12: Time-slice BIOS Application

When GP Timer 5 raises an interrupt, the CPU will invoke the Hwi Dispatcher to call the appropriate ISR. In this example, the ISR lowers the priority of the currently running task and raises the priority of the other compute task. Once the ISR returns to the Hwi dispatcher, the dispatcher invokes the Task Scheduler which performs an asynchronous preemption of the currently running task and switches to the target task which had previously been asynchronously preempted.

A sequence diagram of the task switch sequence on a C64x platform is illustrated in Figure 13 and described in Table 8. The same task switch sequence on a HLOS is illustrated in Figure 14 and described in Table 9.

C64x Async to Async Task Switch

Figure 13: C64x Async to Async Task Switch

Time	Description of Changes
t_0	Initial condition.
t_1	
t_2	
t_3	
t_4	
t_5	

Table 8: C64x Async to Async Task Switch Description

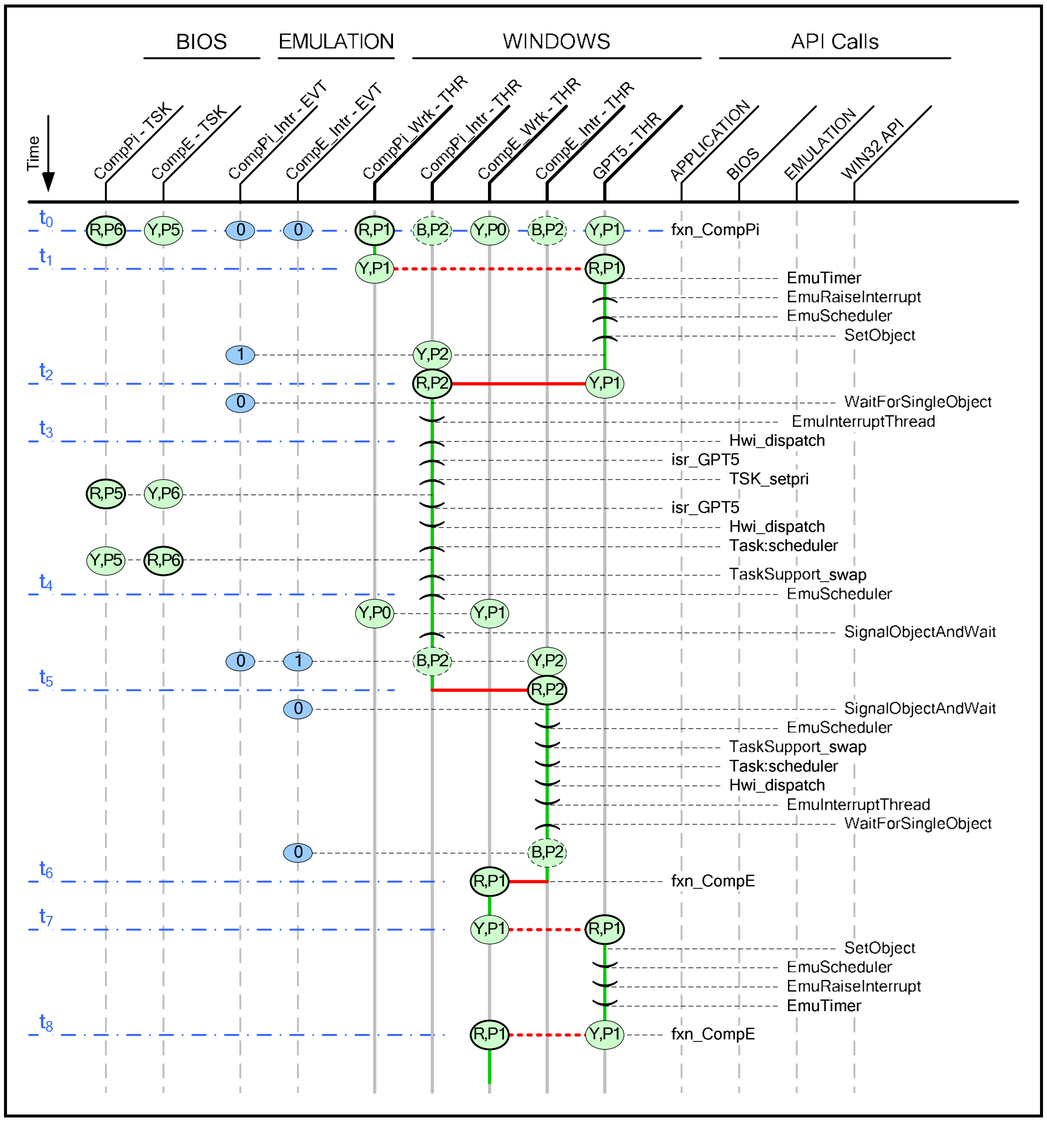

Figure 14: HLOS Async to Async Task Switch

Time	BIOS Events	Emulation Events
t_0	The CompPi (compute Pi) task is running at priority P6 in the application function fxn_CompPi. The CompE (compute E) task is ready and has priority P5.	The CompPI_Wrk (compute Pi worker) thread is running at priority P1. The GPT5 thread is ready and at the same priority. Windows is round-robin scheduling these two threads. All other threads are blocked.
t_1	No events.	Windows has done an asynchronous thread switch to the GPT5 thread which is running in the emulation function EmuTimer. At some point, EmuTimer raises an interrupt by calling EmuRaisInterrupt which calls into EmuScheduler. The emulation scheduler must now asynchronously interrupt the currently running BIOS task. It does this by calling SetObject to signal the CompE_Inter event object which changes the CompE_Inter thread from the blocked to the ready state.
t_2	No events.	Windows switches to the CompE_Inter thread because it is the only thread ready to run at priority P2. This thread resumes in WaitForSingleObject which immediately resets the event object it was blocked on. It then returns to the EmuInterruptThread function and calls Hwi_dispatch to handle the interrupt generated by the GPT5 thread.
t_3		
t_4		
t_5		
t_6		
t_7		
t_8		

Table 9: HLOS Async to Async Task Switch Description

3.3 Swi Scheduling

- This should be handled in BIOS code above the emulation layer.
- Except for the initial interrupt, all subsequent interrupts and all Swis run on the system stack. N.B. the system stack size may need to be increased when running on a HLOS due the extra overhead of the emulation layer.

3.4 Hwi Processing

- Emulation calls Hwi Dispatch
- Interrupt Vector Table

3.5 Clock Emulation

3.6 BIOS Timer

3.7 BIOS Bench

3.8 Peripherals

3.8.1 Timers

3.8.2 I/O Peripherals

3.8.3 Keyboard Peripheral

3.9 Application Configuration

- Use configuration to hook emulation functions
- Use configuration to hook peripheral functions

4 Pseudo Code

4.1 Emulation Layer

4.1.1 EmuInterruptThread

```
emuInterruptThread()
{
    /* initial wait condition */
    wait(self)

    /* main loop */
    while (1) {
        wait(self)
        Hwi_dispatch()
    }
}
```

4.1.2 EmuScheduler

```
emuScheduler()
{


}
```

4.1.3 EmuWorkerThread

```
emuWorkerThread()
{


}
```

5 Supporting Collateral

The material in this section is used to support and implement the design described in this document. This section should be considered as reference material and not part of the design itself.

- Point out that this material is read-only and cannot be changed
- The design must accommodate this material, not the other way around

5.1 BIOS Proxy Modules

The BIOS proxy modules are used to allow different implementations for a given interface definition. The relevant proxy modules needed to support this design are contained in the `ti.bios.family` package and are listed here:

- `ti.bios.family.linux`
- `ti.bios.family.windows`

Each of these packages must implement the following set of interfaces for their proxy modules:

- `ti.bios.interfaces.IHwi`
- `ti.bios.interfaces.ITimer`
- `ti.bios.interfaces.IBench`
- `ti.bios.interfaces.IKernelSupport`
- `ti.bios.interfaces.ITaskSupport`

5.1.1 ti.bios.interfaces.IHwi interface

The `ti.bios.interfaces.IHwi` interface defines the following functions.

API Name	Description
`disable`	Globally disable interrupts.
`enable`	Globally enable interrupts.
`restore`	Globally restore interrupts.
`switchFromBootStack`	Startup function to switch from boot stack to task stack.
`switchToIsrStack`	Used by Swi scheduler and interrupt dispatcher to switch to the ISR stack.
`switchToTaskStack`	Used by Swi scheduler and interrupt dispatcher to switch to the task stack.

Table 10: ti.bios.interfaces.IHwi interface summary

5.1.2 ti.bios.interfaces.ITimer interface

The `ti.bios.interfaces.ITimer` interface defines the following functions.

API Name	Description
`getNumTimers`	Returns number of timer peripherals on the platform.
`getStatus`	Returns timer status.
`startup`	Called during `BIOS_start` which starts statically created timers with startMode = Startup_BIOS.
`getDefaultCountsPmsMeta`	Returns number of timer counts in one millisecond.
`create`	Creates a timer.
`start`	Starts timer created in RunMode_ONESHOT and RunMode_CONTINUOUS.
`trigger`	Timer runs for specified number of instructions.
`stop`	Stops timer.
`reset`	Resets timer to default settings.
`setPeriod`	Sets timer period specified in timer counts.
`getPeriod`	Gets timer period in timer counts.
`getCountsPms`	Returns number of timer counts in one millisecond.

Table 11: ti.bios.interfaces.ITimer interface summary

5.1.3 ti.bios.interfaces.IBench interface

The `ti.bios.interfaces.IBench` interface defines the following functions.

API Name	Description
`getCounts`	Returns high resolution counts since startup.
`getLongCounts`	Returns high resolution counts since startup. Wraps when value reaches size of ULLong.
`getCounterFreq`	Returns frequency of counter used by bench.

Table 12: ti.bios.interfaces.IBench interface summary

5.1.4 ti.bios.interfaces.IKernelSupport interface

The `ti.bios.interfaces.IKernelSupport` interface defines the following functions.

API Name	Description
`maxbit`	Returns bit number of most significant '1' in bits argument.

Table 13: ti.bios.interfaces.IKernelSupport interface summary

5.1.5 ti.bios.interfaces.ITaskSupport

The `ti.bios.interfaces.ITaskSupport` interface defines the following functions.

API Name	Description
`start`	Deals with task startup.
`swap`	Switch from old task to new task.
`createStartupTask`	Create initial task to switch from.
`checkStacks`	Check for stack overflow.
`stackUsed`	Returns the task stack usage.

Table 14: ti.bios.interfaces.ITaskSupport

- Add diagram

5.2 Windows Win32 API functions

The following Win32 API functions will be used to implement the BIOS Emulation layer.

API Name	Description
`GetPriorityClass`	Retrieves the priority class for the specified process.
`GetThreadTimes`	Retrieves timing information for the specified thread.
`ResumeThread`	Decrements a thread's suspend count.
`SetPriorityClass`	Sets the priority class for the specified process.
`SetThreadPriority`	Sets the priority value for the specified thread.
`SuspendThread`	Suspends the specified thread.

Table 15: Windows Win32 API Functions

5.3 Linux User–Level API functions

The following Linux User-Level API functions will be used to implement the BIOS Emulation layer.

[TBD]

5.4 _Subheading_

6 Test Plan

6.1 Unit Tests

6.2 Functional Tests

6.3 System Tests

A Appendix – Updated Design Notes

When moving to the new Visual C++ 8.0 compiler, the current design no longer guaranteed the same scheduling behavior when compared to the C64+ target. This required some changes to the design. This section documents these changes.

The Windows priority boosting scheme caused the emulation layer to run threads in a different sequence than expected. To compensate for this, the design needed to incorporate the thread suspend and resume API's. This allows the emulation to prevent a worker thread from running while processing an interrupt in a new thread.

This prompted another design change. The static interrupt thread is no longer created at task create time. Instead, all interrupt threads are created on-the-fly when a new interrupt is raised and are deleted when the interrupt ISR completes. This also gives a better run-time behavior as the current interrupt can be preempted at any point. With the previous design, the interrupt thread must complete before servicing a new interrupt.

This change allows for the support of nested interrupts. A subsequent interrupt will suspend the current interrupt thread, run to completion and then resume the previous interrupt thread. When the last interrupt thread terminates, it will resume the preempted worker thread.

A future optimization would be to create a pool of interrupt threads at boot time and draw from this pool to service interrupts instead of creating and deleting threads on-the-fly. This would improve run-time performance. It was also observed that Windows fails to create threads after some number of threads have been created and deleted. Hopefully, using a thread pool would avoid this limitation.

«« « § » »»