TEXAS INSTRUMENTS

Application Report

SPRAA16D – May 2006

DSP/BIOS Benchmarks

Software Development Systems

ABSTRACT

This document describes timing benchmarks for DSP/BIOS functions. The actual benchmark values are provided in the Results.htm file, which is installed with DSP/BIOS. These benchmark values are also available on the DSP/BIOS download page (https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html) for each release by following the DSP/BIOS Documentation->Results link. These values may be used to calculate overall system performance or overhead.

Where a particular API call may result in several different situations, benchmarks are provided for each situation. In addition, the methodology used to obtain these benchmarks is described, so that designers may better analyze their system performance.

Contents

Figure

Tables

1 DSP/BIOS Timing Benchmarks

The following sections identify DSP/BIOS modules and describe the APIs for which benchmarks are provided with the DSP/BIOS release. The benchmarks are dependent on the memory system.

See the Results.htm file, which is installed with DSP/BIOS, for the results of these benchmarks for the current release. These benchmark values are also available on the DSP/BIOS download page (https://www-a.ti.com/downloads/sds_support/targetcontent/bios/index.html) for each release by following the DSP/BIOS Documentation→Results link.

1.1 Interrupt Latency

Interrupt latency. This is the maximum number of instructions during which the DSP/BIOS kernel disables maskable interrupts. Interrupts are disabled in order to modify data shared across multiple threads. DSP/BIOS minimizes this time as much as possible to allow the fastest possible interrupt response time. The interrupt latency of the kernel is measured in a specific region within DSP/BIOS. The measurement provided here is the cycle count measurement for executing that region of code.

1.2 HWI—Hardware Interrupt Benchmarks

HWI_enable. This is the execution time of a HWI_enable function call, which is used to globally enable hardware interrupts.

HWI_disable. This is the execution time of a HWI_disable function call, which is used to globally disable hardware interrupts.

HWI dispatcher. These are the execution times of specified portions of the HWI dispatcher code. This dispatcher handles running C code in response to an interrupt. The benchmarks provide times for the following cases:

- *Interrupt prolog for calling C function*. This is the execution time from when an interrupt occurs until the user's C function is called.
- *Interrupt epilog following C function call*. This is the execution time from when the user's C function completes execution until the HWI dispatcher has completed its work and exited.

Hardware interrupt to blocked task. This is a measurement of the elapsed time from the start of an ISR that posts a semaphore, to the execution of first instruction in the higher priority blocked task, as shown in Figure 1.

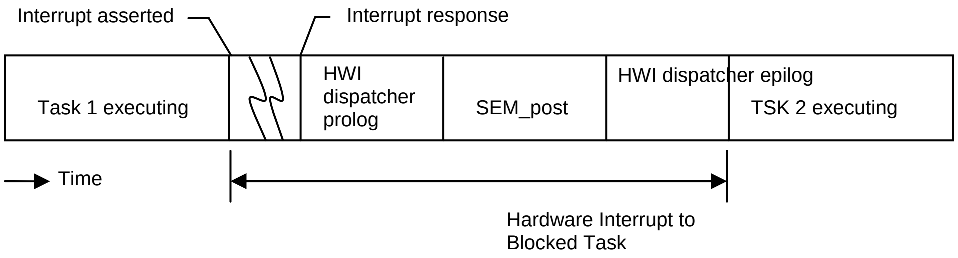

Figure 1. Hardware Interrupt to Blocked Task

Hardware interrupt to software interrupt. This is a measurement of the elapsed time from the start of an ISR that posts a software interrupt, to the execution of the first instruction in the higher-priority posted software interrupt.

This duration is shown in Figure 2. SWI 2, which is posted from the ISR, has a higher priority than SWI 1, so SWI 1 is preempted. The context switch for SWI 2 is performed within the SWI executive invoked by the HWI dispatcher, and this time is included within the measurement. In this case, the registers saved/restored by the HWI dispatcher correspond to that of "C" caller saved registers.

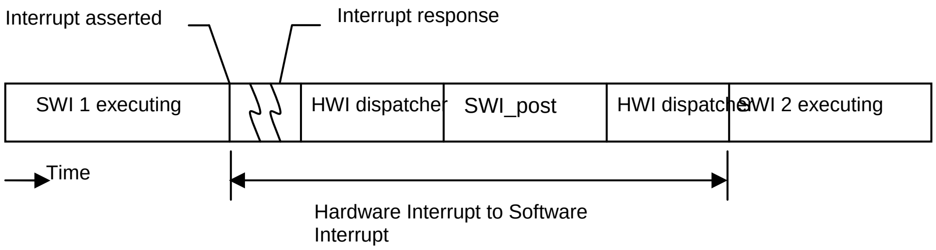

Figure 2. Hardware Interrupt to Software Interrupt

1.3 SWI—Software Interrupt Benchmarks

SWI_enable. This is the execution time of a SWI_enable function call, which is used to enable software interrupts.

SWI_disable. This is the execution time of a SWI_disable function call, which is used to disable software interrupts.

SWI_post. This is the execution time of a SWI_post function call, which is used to post a software interrupt. This document provides benchmarks for the following cases of SWI_post:

- *Post software interrupt again*. This case corresponds to a call to SWI_post of SWI that has already been posted but hasn't started running as it was posted by a higher priority SWI. Figure 3 shows this case. Higher priority SWI1 posts lower priority SWI2 twice. The cycle count being measured corresponds to that of second post of SWI2.

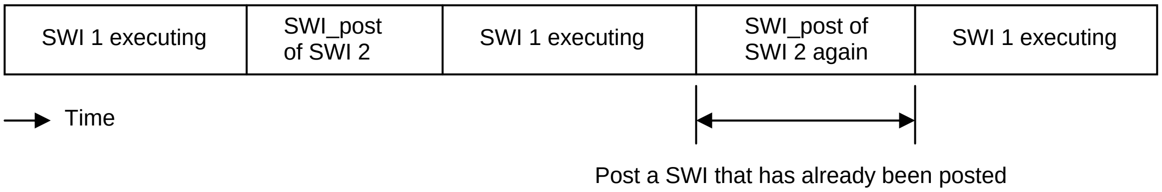

Figure 3. Post of Software Interrupt Again

- *Post software interrupt, no context switch*. This is a measurement of a SWI_post function call, when the posted software interrupt is of lower priority then currently running SWI. Figure 4 shows this case.

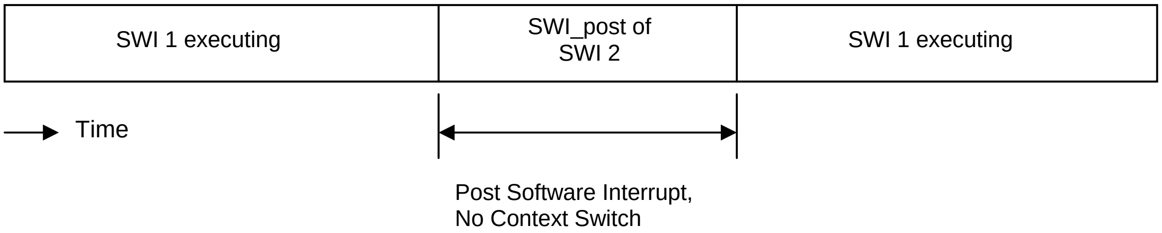

Figure 4. Post Software Interrupt without Context Switch

- *Post software interrupt, context switch.* This is a measurement of the elapsed time between a call to SWI_post (which causes preemption of the current SWI), and the execution of the first instruction in the higher–priority software interrupt, as shown in Figure 5. The context switch to SWI2 is performed within the SWI executive, and this time is included within the measurement.

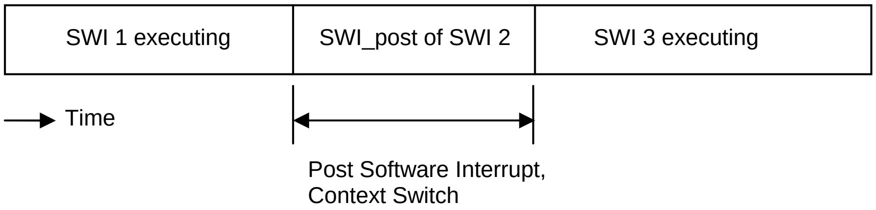

Figure 5. Post Software Interrupt with Context Switch

1.4 TSK—Task Benchmarks

TSK_enable. This is the execution time of a TSK_enable function call, which is used to enable DSP/BIOS task scheduler.

TSK_disable. This is the execution time of a TSK_disable function call, which is used to disable DSP/BIOS task scheduler.

TSK_create. This is the execution time of a TSK_create function call, which is used to create a task ready for execution. This document provides benchmarks for the following cases of TSK_create:

- *Create a task, no context switch*. The executing task creates and readies another task of lower or equal priority, which results in no context switch. See Figure 6.

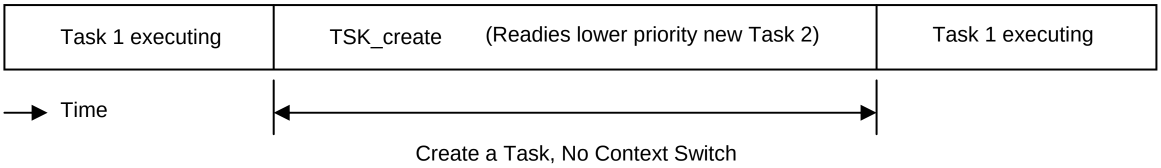
Task 1 executing
TSK_create
(Readies lower priority new Task 2)
Task 1 executing
Time
Create a Task, No Context Switch

Figure 6. Create a New Task without Context Switch

- *Create a task, context switch.* The executing task creates another task of higher priority, resulting in a context switch. See Figure 7.

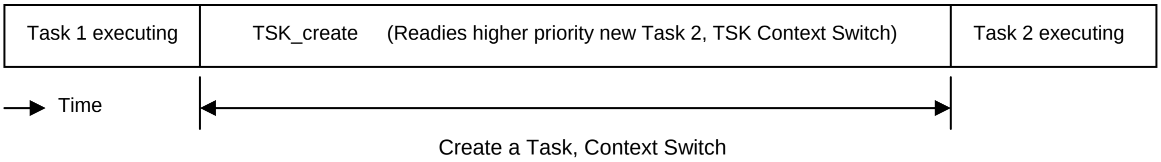
Task 1 executing
TSK_create
(Readies higher priority new Task 2, TSK Context Switch)
Task 2 executing
Time
Create a Task, Context Switch

Figure 7. Create a New Task with Context Switch

NOTE: The benchmarks for TSK_create assume that memory allocated for TSK object is available in the first free list and that no other task holds the lock to that memory. Additionally the stack has been pre-allocated and is being passed as a parameter.

TSK_delete. This is the execution time of a TSK_delete function call, which is used to delete a task. The Task handle created by TSK_create is passed to the TSK_delete API.

TSK_setpri. This is the execution time of a TSK_setpri function call, which is used to set a task's execution priority. This document provides benchmarks for the following cases of TSK_setpri:

- *Set a task priority, no context switch.* This case measures the execution time of the TSK_setpri API called from a task Task1 as in Figure 8 if the following conditions are all true:
  – TSK_setpri sets the priority of a lower priority task that is in ready state.
  – The argument to TSK_setpri is less then the priority of current running task.

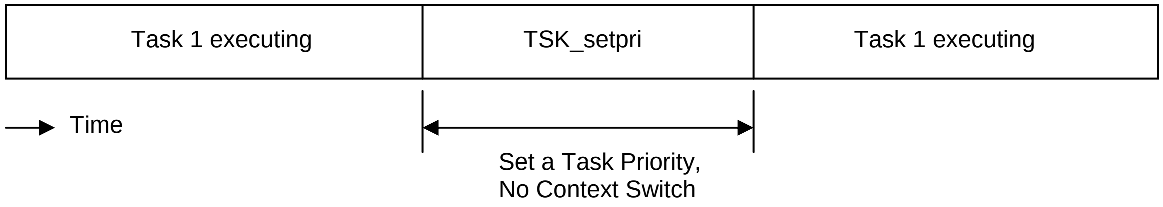

Figure 8. Set a Task's Priority without a Context Switch

- *Lower the current task's own priority, context switch*. This case measures execution time of TSK_setpri API when it is called to lower the priority of currently running task. The call to TSK_setpri would result in context switch to next higher priority ready task. Figure 9 shows this case.

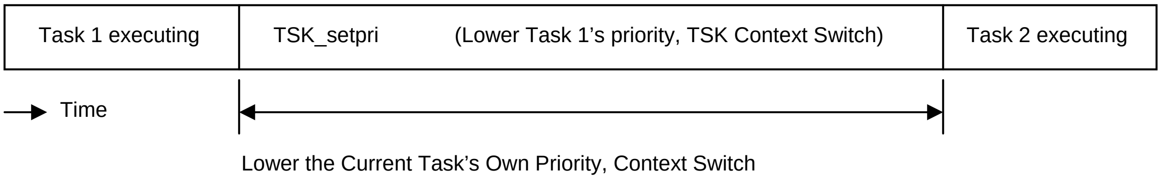
Task 1 executing
TSK_setpri
(Lower Task 1's priority, TSK Context Switch)
Task 2 executing
Time
Lower the Current Task's Own Priority, Context Switch

Figure 9. Lower the Current Task's Priority, Context Switch

- *Raise a ready task's priority, context switch.* This case measures execution time of TSK_setpri API called from a task Task1 if the following conditions are all true:
  – TSK_setpri sets the priority of a lower priority task that is in ready state.
  – The argument to TSK_setpri is greater then the priority of current running task.

  The execution time measurement includes the context switch time as shown in Figure 10.

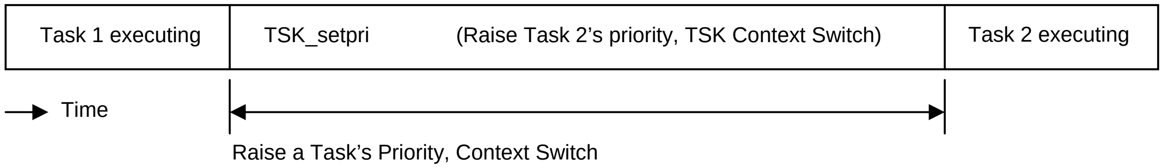
Task 1 executing
TSK_setpri
(Raise Task 2's priority, TSK Context Switch)
Task 2 executing
Time
Raise a Task's Priority, Context Switch

Figure 10. Raise a Ready Task's Priority, Context Switch

TSK_yield. This is a measurement of the elapsed time between a function call to TSK_yield (which causes preemption of the current task), and the execution of the first instruction in the next ready task of equal priority, as shown in Figure 11.

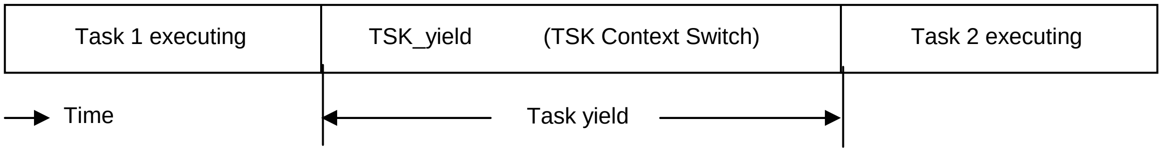

Figure 11. Task Yield

1.5 SEM—Semaphore Benchmarks

Semaphore benchmarks measure the time interval between issuing a SEM_post or SEM_pend function call and the resumption of task execution, both with and without a context switch.

SEM_post. This is the execution time of a SEM_post function call. This document provides benchmarks for the following cases of SEM_post:

- *Post a semaphore, no waiting task*. In this case, the SEM_post function call does not cause a context switch as no other task is waiting for the semaphore. This is shown in Figure 12.

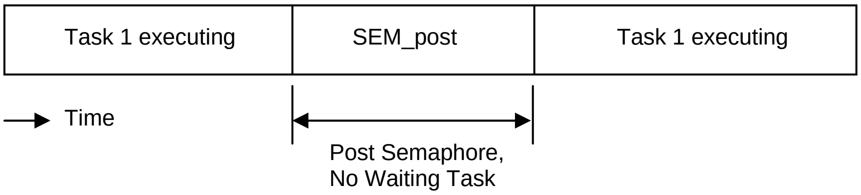

Figure 12. Post Semaphore, No Waiting Task

- *Post a semaphore, no context switch*. This is a measurement of a SEM_post function call, when a lower priority task is pending on the semaphore. In this case, SEM_post readies the lower priority task waiting for the semaphore and resumes execution of the original task, as shown in Figure 13.

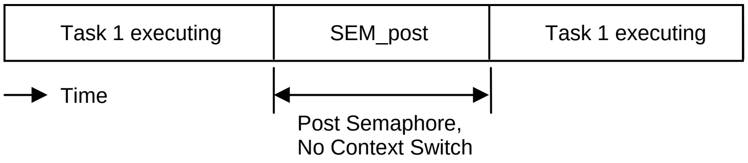

Figure 13. Post Semaphore, No Context Switch

- *Post a semaphore, context switch*. This is a measurement of the elapsed time between a function call to SEM_post (which readies a higher priority task pending on the semaphore causing a context switch to higher priority task), and the execution of the first instruction in the higher–priority task, as shown in Figure 14.

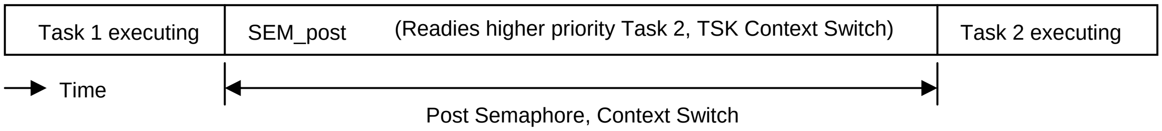
Task 1 executing
SEM_post
(Readies higher priority Task 2, TSK Context Switch)
Task 2 executing
Time
Post Semaphore, Context Switch

Figure 14. Post Semaphore with Task Switch

SEM_pend. This is the execution time of a SEM_pend function call, which is used to acquire a semaphore. This document provides benchmarks for the following cases of SEM_pend:

- *Pend on a semaphore, no context switch*. This is a measurement of a SEM_pend function call without a context switch (as the semaphore is available.) See 0.

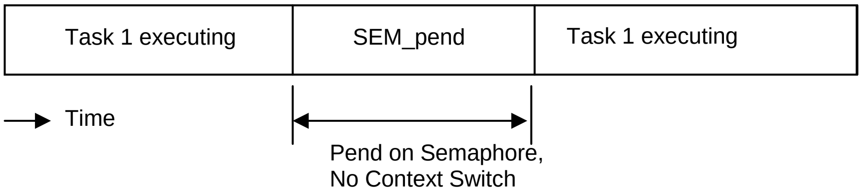

Figure 15. Pend on Semaphore, No Context Switch

- *Pend on a semaphore, context switch*. This is a measurement of the elapsed time between a function call to SEM_pend (which causes preemption of the current task), and the execution of first instruction in next higher–priority ready task. See 0.

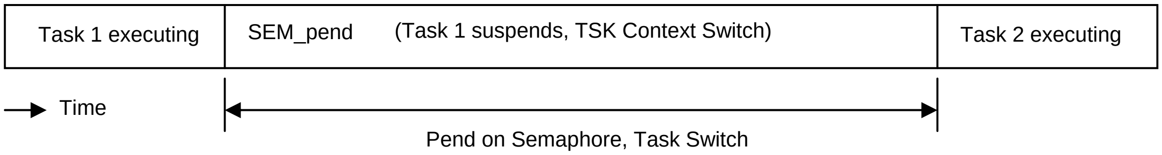
Task 1 executing
SEM_pend
(Task 1 suspends, TSK Context Switch)
Task 2 executing
Time
Pend on Semaphore, Task Switch

Figure 16. Pend on Semaphore with Task Switch

1.6 MBX—Mailbox Benchmarks

Messages are copied in and out of the MBX. Therefore, the message length of the MBX is significant when benchmarking it. A message length of 1 MADU was used in the measurement of the MBX APIs.

MBX_post. This is the execution time of an MBX_post function call, which is used to post a message to mailbox. This document provides benchmarks for the following cases of MBX_post:

- *Post a mailbox, no tasks waiting*. This is a measurement of an MBX_post function if the following conditions are all true:
  - Mailbox has an empty slot.
  - No task is pending on the mailbox.

  This MBX_post function call does not cause a context switch. Figure 17 shows this case.

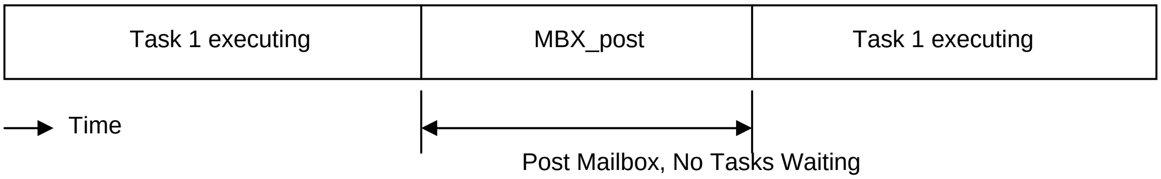

Figure 17. Post Mailbox, No Task Pending on Mailbox

- *Post a mailbox, no context switch.* This is a measurement of an MBX_post API made from a higher priority task that readies a lower priority task pending on the same mailbox. Figure 18 shows this case. Task1 is the higher priority task that posts a mailbox to ready lower priority Task2 task.

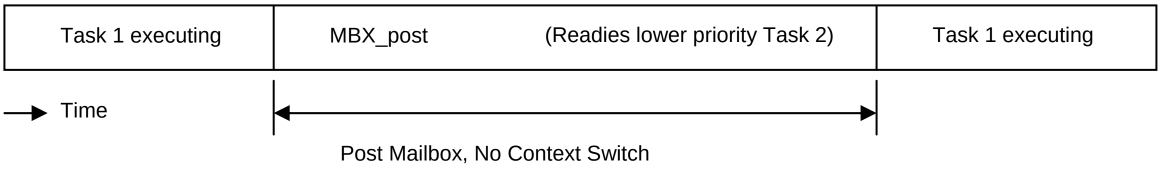
Task 1 executing
MBX_post
(Readies lower priority Task 2)
Task 1 executing
Time
Post Mailbox, No Context Switch

Figure 18. Post a Mailbox without Context Switch

- *Post a mailbox, context switch*. This is a measurement of the elapsed time between a function call to MBX_post (which readies a higher priority task pending on the mailbox causing a context switch to higher priority) and the execution of first instruction in the higher priority task. Figure 19 shows this case.

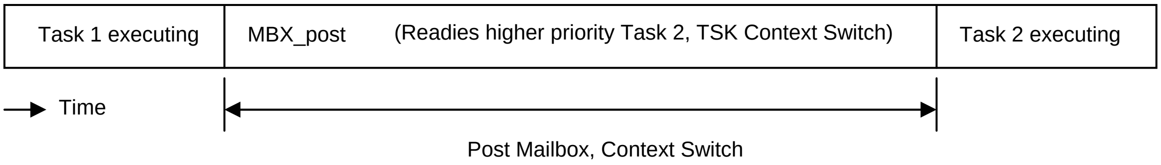
Task 1 executing
MBX_post
(Readies higher priority Task 2, TSK Context Switch)
Task 2 executing
Time
Post Mailbox, Context Switch

Figure 19. Post a Mailbox with Context Switch

MBX_pend. This is the execution time of an MBX_pend function call, which obtains message from mailbox. This document provides benchmarks for the following cases of MBX_pend:

- *Pend on a mailbox, no context switch*. This is a measurement of an MBX_pend function call that obtains a message without blocking. See Figure 20.

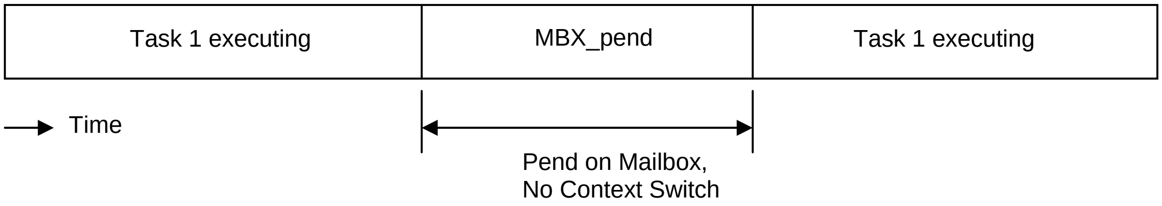

Figure 20. Pend on Mailbox, No Context Switch

- *Pend on a mailbox, context switch*. This is a measurement of the elapsed time between a function call to MBX_pend (which causes preemption of the current task) and a switch to a higher–priority task is blocked on MBX_post function call. See Figure 21.

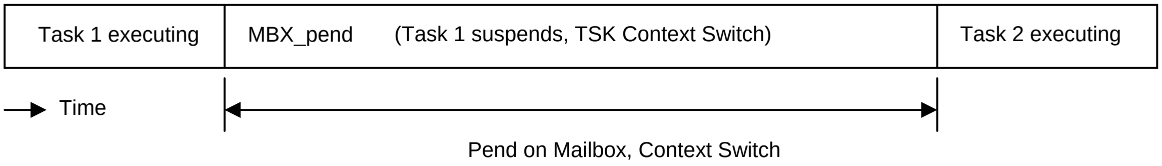
Task 1 executing
MBX_pend
(Task 1 suspends, TSK Context Switch)
Task 2 executing
Time
Pend on Mailbox, Context Switch

Figure 21. Pend on Mailbox with Context Switch

1.7 LCK—Resource Lock Benchmarks

LCK_post. This is the execution time of a LCK_post function call, which is used to relinquish ownership of a resource lock. This document provides benchmarks for the following cases of LCK_post:

- *Post a lock, no ownership relinquishment*. In this case the current running task that owns the lock (due to multiple prior calls to LCK_pend) calls LCK_post. This call to LCK_post is benchmarked as shown in Figure 22.

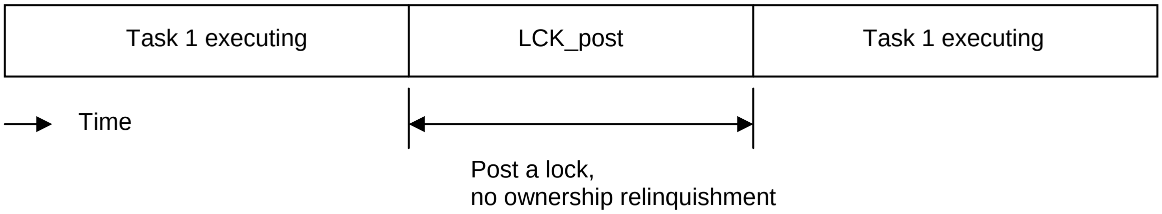

Figure 22. Post a Resource LCK without Ownership Relinquishment

- *Post a lock, no context switch*. In this case LCK_post relinquishes ownership of a resource lock, and continues execution of the current task. LCK_post does not result in a context switch because no task is pending on the lock. See Figure 23.

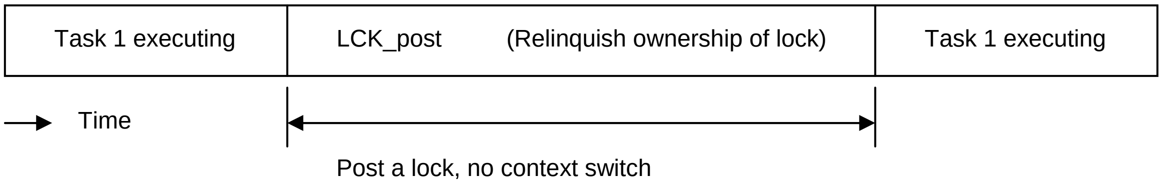
Task 1 executing
LCK_post
(Relinquish ownership of lock)
Task 1 executing
Time
Post a lock, no context switch

Figure 23. Post a Resource LCK without Context Switch

- *Post a lock, context switch*. In this case, LCK_post relinquishes ownership of a resource lock, and results in a context switch because a higher priority task is currently pending on the lock. See Figure 24.

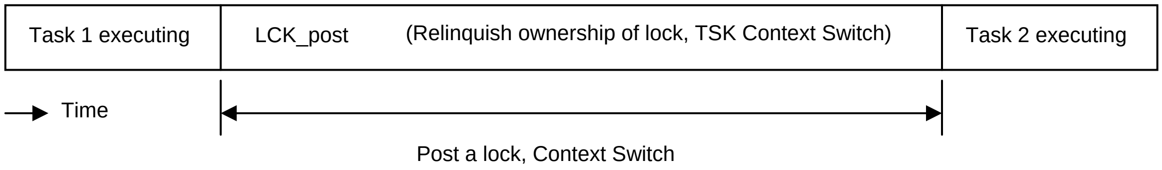
Task 1 executing
LCK_post
(Relinquish ownership of lock, TSK Context Switch)
Task 2 executing
Time
Post a lock, Context Switch

Figure 24. Post a Resource LCK with Context Switch

LCK_pend. The execution time of a LCK_pend function call, which is used to acquire ownership of a resource lock. This document provides benchmarks for the following cases of LCK_pend:

- *Pend on a self-owned lock*. This is the execution time of a LCK_pend when a task already owns the resource lock. See Figure 25.

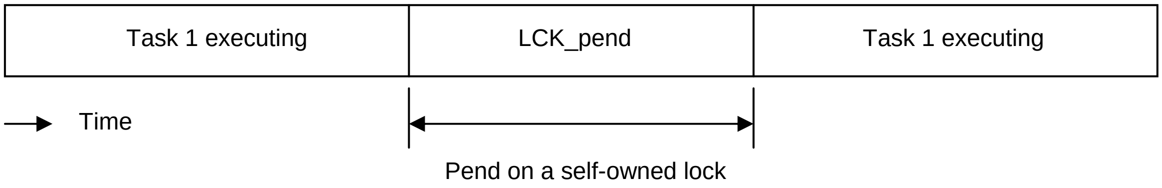

Figure 25. Pend on a Self-Owned LCK

- *Pend on a lock, no context switch*. The lock is not owned by any task, and the current task calls LCK_pend. The current task succeeds in acquiring ownership of lock, which grants the current task exclusive access to the corresponding resource. See Figure 26.

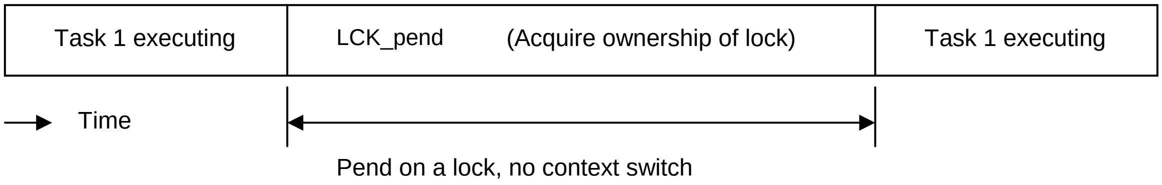
Task 1 executing
LCK_pend
(Acquire ownership of lock)
Task 1 executing
Time
Pend on a lock, no context switch

Figure 26. Pend on a Resource LCK without Context Switch

- *Pend on a lock, context switch*. The resource lock is owned by another task, LCK_pend suspends execution of the current task until the resource becomes available and results in a context switch. See Figure 27.

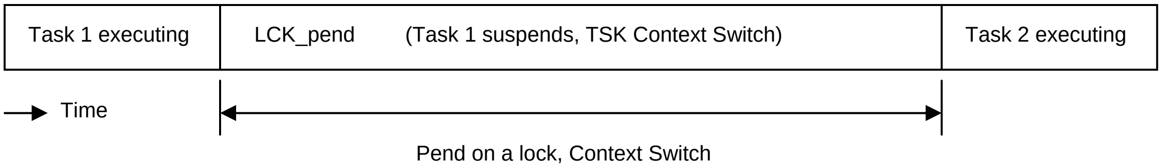
Task 1 executing
LCK_pend
(Task 1 suspends, TSK Context Switch)
Task 2 executing
Time
Pend on a lock, Context Switch

Figure 27. Pend on a Resource LCK with Context Switch

1.8 CLK—System Clock Benchmarks

CLK_gethtime. This is the execution time of a CLK_gethtime function call.

CLK_getltime. This is the execution time of a CLK_getltime function call.

1.9 LOG—Log Benchmarks

LOG_event. This is the execution time of a LOG_event function call, which is used to append an unformatted message to an event log.

LOG_printf. This is the execution time of a LOG_printf function call, which is used to append a formatted message to an event log. The execution time of the function is not dependent on the number of arguments specified in the function call.

1.10 STS—Statistics Benchmarks

STS_add. This is the execution time of an STS_add function call, which is used to update the total, count, and max fields of a statistics object.

STS_delta. This is the execution time of an STS_delta function call, which is used to update a statistics object, using the difference between a provided value and a previous set point value.

STS_set. This is the execution time of an STS_set function call, which is used to set the previous value for a statistics object.

1.11 MEM—Memory Benchmarks

MEM_alloc. This is the execution time of a MEM_alloc function call, which is used to allocate a contiguous block of storage from a specified memory section. This document provides benchmarks for the following cases of MEM_alloc:

- *Memory allocated on first block*. Memory block to be allocated fits on the first block of the MEM_free list.
- *Memory allocated on second block*. Memory block to be allocated does not fit on the first block, but fits on the second block of the MEM_free list.
- *Memory allocated on third block*. Memory block to be allocated does not fit on the first, nor the second block, but fits on the third block of the MEM_free list.
- *Memory allocated on fourth block*. Memory block to be allocated does not fit on the first, second and third block of MEM_free list but fits on the fourth block of the MEM_free list.

MEM_free. This is the execution time of a MEM_free function call, which places the memory block specified back into the free pool of the section specified. This document provides benchmarks for the following cases of MEM_free:

- *Memory coalesces no block*. Memory block to be freed cannot coalesce with either of its neighboring memory segments.
- *Memory coalesces one block*. Memory block to be freed coalesces with one neighboring memory segment either above it or below it.
- *Memory coalesces two blocks*. Memory block to be freed coalesces with both neighboring memory segments above and below it.

1.12 PIP—Pipe Benchmarks

NOTE: Each of the following pipe benchmarks includes the execution time of a minimal notifyWriter (or notifyReader) C function call—that is, a function that simply returns.

PIP_alloc. This is the execution time of a PIP_alloc function call, which is used to allocate an empty frame from a pipe.

PIP_free. This is the execution time of a PIP_free function call, which is used to recycle a frame back into a pipe.

PIP_get. This is the execution time of a PIP_get function call, which is used to get a full frame from a pipe.

PIP_put. This is the execution time of a PIP_put function call, which is used to put a full frame into a pipe.

PIP_peek. This is the execution time of a PIP_peek function call, which is used to get the pipe frame size and address without actually claiming the pipe frame.

1.13 QUE—Queue Benchmarks

QUE_dequeue. This is the execution time of a QUE_dequeue function call, which is used to remove the element from the front of a queue (non-atomically).

QUE_empty. This is the execution time of a QUE_ empty function call, which is used to test for an empty queue.

QUE_enqueue. This is the execution time of a QUE_enqueue function call, which is used to insert an element at the end of a queue (non-atomically).

QUE_get. This is the execution time of a QUE_get function call, which is used to remove the element from the front of a queue (atomically).

QUE_insert. This is the execution time of a QUE_insert function call, which is used to insert an element in the middle of a queue (non-atomically).

QUE_put. This is the execution time of a QUE_put function call, which is used to put an element at the end of a queue (atomically).

QUE_remove. This is the execution time of a QUE_remove function call, which is used to remove an element from the middle of a queue (non-atomically).

1.14 MSGQ—Message Queue Benchmarks

All the MSGQ benchmarks were run with the following configuration:

- Default notification attributes for all message queues were used. That is, the pend() function was SYS_zero() and the post function was FXN_F_nop().
- The STATICPOOL allocator was used.
- All tests were run on a uni-processor system. That is, all message queues resided on the same processor.

MSGQ_alloc. This is the execution time of a MSGQ_alloc function call. As noted above, the STATICPOOL allocator was used for this benchmark.

MSGQ_put. This is the execution time of a MSGQ_put function call. The number of existing messages in the message queue does not affect this benchmark.

MSGQ_get with messages. This is the execution time of a MSGQ_get function call when there is at least one message already present. Therefore the message queue's pend() is not called.

MSGQ_get with no messages. This is the execution time of a MSGQ_get function call when there no message present. Therefore the message queue's pend() is called.

MSGQ_free. This is the execution time of a MSGQ_free function call. As noted above, the STATICPOOL allocator was used for this benchmark.

1.15 PWRM—Power Manager Benchmarks

PWRM_getCapabilities. This is the execution time of a PWRM_getCapabilities function call to get information on PWRM's capabilities on the current platform.

PWRM_getCurrentSetpoint. This is the execution time of a PWRM_getCurrentSetpoint function call to get the current setpoint in effect.

PWRM_getNumSetpoints. This is the execution time of a PWRM_getNumSetpoints function call to get the number of setpoints supported for the current platform.

PWRM_getSetpointInfo. This is the execution time of a PWRM_getSetpointInfo function call to get the corresponding frequency and CPU core voltage for a setpoint.

PWRM_getTransitionLatency. This is the execution time of a PWRM_getTransitionLatency function call to get the latency to scale from one setpoint to another setpoint.

PWRM_configure. This is the execution time of a PWRM_configure function call to set new configuration parameters for PWRM.

PWRM_registerNotify. This is the execution time of a PWRM_registerNotify function call to register a function to be called on a specific power event.

PWRM_unregisterNotify. This is the execution time of a PWRM_unregisterNotify function call to unregister for an event notification from PWRM.

PWRM_sleepDSP. This is the execution time of a PWRM_sleepDSP function call to transition the DSP to a new sleep state and wake up from that deep sleep state.

PWRM_idleClocks. This is the execution time of a PWRM_idleClocks function call to immediately idle clock domains.

2 DSP/BIOS Benchmarking Methodology

2.1 DSP/BIOS Benchmarking Environment

DSP/BIOS real-time analysis was disabled when benchmarks were obtained.

The benchmark numbers were obtained using the DSP's hardware timer. The API benchmark numbers were obtained by clearing and starting the timer, calling the API, and reading the timer value again. The overhead of the timer has been factored out of the timer reading and multiplied by the number of instructions per timer tick. The number of instructions performed during a single timer tick varies on the different DSP architectures as shown in Table 1.

Table 1. Instructions Per Timer Tick on Various Architectures

C28x	C54x	C55x	C62x/C67x	C64x
1 instruction / tick	1 instruction / tick	1 instruction / tick	4 instructions / tick	8 instructions / tick

DSP/BIOS benchmarks presented in this paper corresponds to particular placement of application code in conjunction with a specific processor configuration. Table 2 details the memory placement and application configuration.

Table 2. Benchmark Programs Environment Setup

DSP Architecture	Memory Placement			Application Configuration
	Code	Data	Heap	
TMS320F28x (Large)	H0SARAM	L0SARAM	L0SARAM	Data Model = Large Instrumented Kernel
TMS320C54x (Near)	IPROG	IDATA	IDATA	Code Model = Near Instrumented Kernel
TMS320C54x (Far)	IPROG	IDATA	IDATA	Code Model = Far Instrumented Kernel
TMS320C55x (Large)	SARAM	DARAM	DARAM	Stack Mode = Fast Return Data Model = Large Non-Instrumented Kernel
TMS320C55x (Huge)	SARAM	DARAM	DARAM	Stack Mode = Fast Return Data Model = Large Non-Instrumented Kernel
TMS320C621x/C671x[1] (functional simulator)	IRAM	IRAM	IRAM	Flat memory system (Single Cycle Memory access) Non-Instrumented Kernel
TMS320C621x/C671x[2] (on-chip)	IRAM	IRAM	IRAM	L2 configured as SRAM. L1 Data and Program cache is invalidated before every DSP/BIOS API call Non-Instrumented Kernel
TMS320C64x[1] (functional simulator)	IRAM	IRAM	IRAM	Flat memory system (Single Cycle Memory access) Non-Instrumented Kernel
TMS320C64x[2] (on-chip)	IRAM	IRAM	IRAM	L2 configured as SRAM. L1 Data and Program cache is invalidated before every DSP/BIOS API call Non-Instrumented Kernel

2.2 Calculating System Performance

We can estimate the amount of DSP/BIOS overhead in terms of CPU load in any application. This is possible since all DSP/BIOS operations are visible to the developer. That is, the developer specifies which DSP/BIOS components and function calls to include into the application, either in the Configuration Tool, or explicitly in the code. The developer needs only to compute the sum of the components and frequency of occurrence to determine the overhead analytically. By using the RTA tools in CCS, developers may also directly measure the overhead on their specific hardware platform.

To calculate the amount of memory consumed by the DSP/BIOS kernel, the developer again needs to identify the DSP/BIOS components and API calls in the program. By summing the components, the developer can estimate the memory usage, both data and program. By using the memory map from the application, the exact amount can be determined.

[1] For these benchmarks, the functional simulator provides a flat memory system, in which all memory accesses take one cycle. The L1 and L2 caches are not involved.

[2] For the on-chip benchmarks, L2 is configured as SRAM. All code and data is placed in L2 SRAM. The L1P and L1D are invalidated prior to every API benchmark. This forces L1P and L1D cache misses to occur. The processor loads L1P and L1D from L2 SRAM.

In a similar fashion, developers can analytically determine the overhead attributed to the DSP/BIOS kernel. However, since it is the nature of software to change over time, analytical calculation can be tedious. The real-time analysis tool provided by the DSP/BIOS kernel allows developers to measure the overhead directly. Finally, since developers can choose the amount of the DSP/BIOS kernel to use and include in their applications, they have full control over the overhead.

3 References

1. *TMS320 DSP/BIOS User's Guide* (SPRU423)
2. *TMS320C28x DSP/BIOS API Reference Guide* (SPRU625)
3. *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404)
4. *TMS320C6000 DSP/BIOS API Reference Guide* (SPRU403)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265