DSP/BIOS Timing Benchmarks for Code Composer Studio v2.2

Arnie Reynoso, Sridhar Chakravarthy,
Ashutosh Agrawal, Prashanth L A

Software Development Systems

ABSTRACT

This document provides timing benchmarks for DSP/BIOS functions in the Code Composer Studio 2.22.20.18 update release. Benchmark values are provided for TMS320C5000, TMS320C6000, and TMS320C28x DSPs. 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

1 **DSP/BIOS Timing Benchmarks** 3
1.1 Interrupt Latency 3
1.2 HWI—Hardware Interrupt Benchmarks 3
1.3 SWI—Software Interrupt Benchmarks 4
1.4 TSK—Task Benchmarks 5
1.5 SEM—Semaphore Benchmarks 7
1.6 MBX—Mailbox Benchmarks 8
1.7 LCK—Resource Lock Benchmarks 10
1.8 CLK—System Clock Benchmarks 11
1.9 LOG—Log Benchmarks 11
1.10 STS—Statistics Benchmarks 12
1.11 MEM—Memory Benchmarks 12
1.12 PIP—Pipe Benchmarks 12
1.13 QUE—Queue Benchmarks 13
2 **DSP/BIOS Benchmarking Methodology** 13
2.1 DSP/BIOS Benchmarking Environment 13
2.2 Calculating System Performance 14
3 **DSP/BIOS Timing Benchmarks** 15
3.1 DSP/BIOS Benchmark Results for the TMSC28x Architecture 15
3.2 DSP/BIOS Benchmark Results for the TMSC5000 Architecture 18
3.3 DSP/BIOS Benchmark Results for the TMSC6000 Architecture 21
4 **References** 24

List of Figures

Figure 1 Hardware Interrupt to Blocked Task 4
Figure 2 Hardware Interrupt to Software Interrupt 4
Figure 3 Post of Software Interrupt Again 4
Figure 4 Post Software Interrupt without Context Switch 5
Figure 5 Post Software Interrupt with Context Switch 5
Figure 6 Create a New Task without Context Switch 5
Figure 7 Create a New Task with Context Switch 6
Figure 8 Set a Task’s Execution Priority without Context Switch 6
Figure 9 Lower the Current Task’s Execution Priority, Context Switch 6
Figure 10 Raise a Ready Task’s Execution Priority, Context Switch 7
Figure 11 Task Yield 7
Figure 12 Post Semaphore, Increment Semaphore Count 7
Figure 13 Post Semaphore without Task Switch 7
Figure 14 Post Semaphore with Task Switch 8
Figure 15 Pend on Semaphore, No Context Switch 8
Figure 16 Pend on Semaphore with Task Switch 8
Figure 17 Post Mailbox, No Task Pending on Mailbox 9
Figure 18 Post a Mailbox without Context Switch 9
Figure 19 Post a Mailbox with Context Switch 9
Figure 20 Pend on Mailbox, No Context Switch 9
Figure 21 Pend on Mailbox with Context Switch 10
Figure 22 Post a Resource LCK without Ownership Relinquishment 10
Figure 23 Post a Resource LCK without Context Switch 10
Figure 24 Post a Resource LCK with Context Switch 10
Figure 25 Pend on a Self-Owned Resource LCK 11
Figure 26 Pend on a Resource LCK without Context Switch 11
Figure 27 Pend on a Resource LCK with Context Switch 11

List of Tables

Table 1 Instructions Per Timer Tick on Various Architectures 13
Table 2 Benchmark Programs Environment Setup 14
Table 3 Benchmark Results for C28x 15
Table 4 Benchmark Results for C5000 18
Table 5 Benchmark Results for C6000 21

1 DSP/BIOS Timing Benchmarks

The following sections identify DSP/BIOS modules and describe the APIs for which benchmarks are provided in Section 3.

1.1 Interrupt Latency

Interrupt latency. This is the maximum number of instructions during which the DSP/BIOS kernel disables maskable interrupts. DSP/BIOS minimizes this time as much as possible to allow the fastest possible interrupt response time. The interrupt latency of the kernel is in a known region inside the HWI scheduler. 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_enter. This is the execution time of a HWI_enter function call, which is the hardware interrupt service routine prolog. This document provides benchmarks for the following cases of HWI_enter:

- *Interrupt prolog (minimum)*. This is a measurement of execution time of HWI_enter macro call. The execution time of the HWI_enter macro depends upon the list of registers to be preserved for the ISR, as defined in masks specified by the user. This benchmark shows the minimum execution time for the prolog when no registers are preserved.
- *Interrupt prolog for calling C function*. This is execution time of HWI_enter macro call with preservation register list as C caller preserved register .

HWI_exit. This is the execution time of HWI_exit function call, which is the hardware interrupt service routine epilog. This document provides benchmarks for the following cases of HWI_exit:

- *Interrupt epilog (minimum)*. This is a measurement of the execution time of an HWI_exit macro call. The execution time of HWI_exit depends upon the list of registers the user specifies to be restored. This benchmark shows the minimum execution time for the epilog when no registers are restored and without activation of DSP/BIOS scheduler.
- *Interrupt epilog following C function call*. This is execution time of HWI_enter macro call with preservation register list as C caller preserved register. In this case DSP/BIOS scheduler is not invoked from HWI_exit.

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.

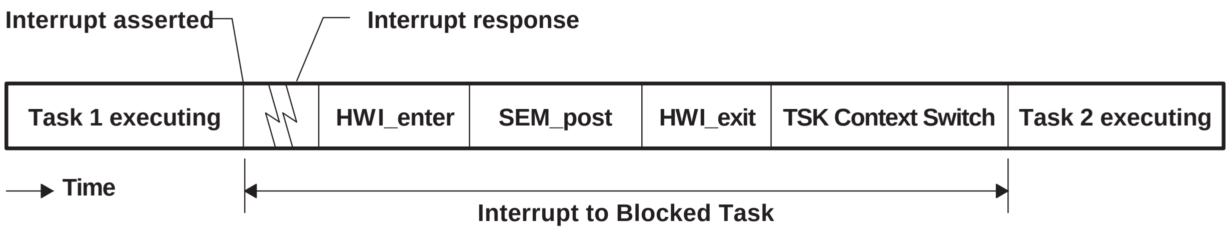

Figure 1. Hardware Interrupt to Blocked Task

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.

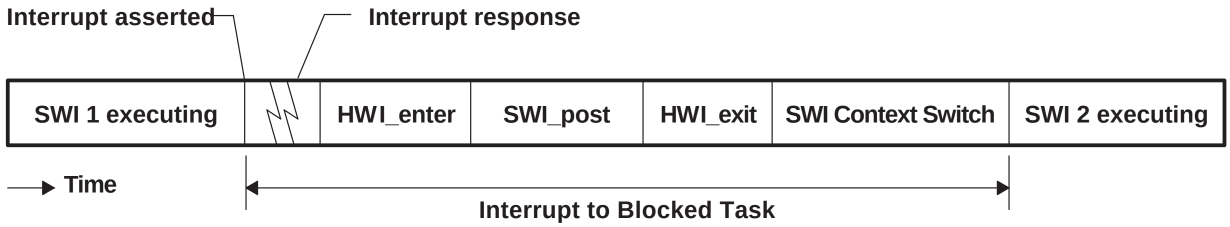

Figure 2. Hardware Interrupt to Software Interrupt

In Figure 2, SWI 2 (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 HWI_exit, and this time is included within the measurement. In this case, the registers saved/restored by HWI_enter/HWI_exit correspond to that of “C” caller saved register.

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.

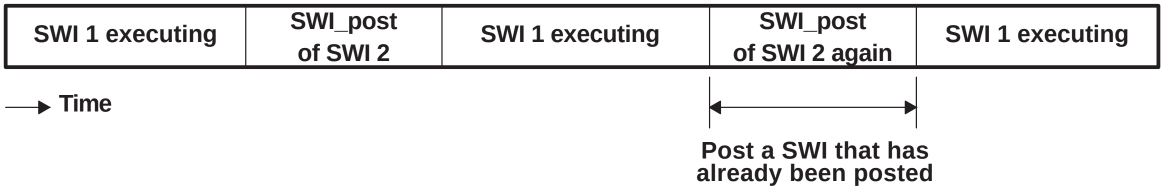

Figure 3. Post of Software Interrupt Again

- *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.

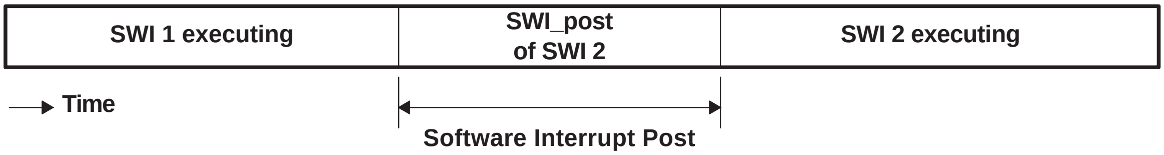

Figure 4. Post Software Interrupt without Context Switch

- *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.

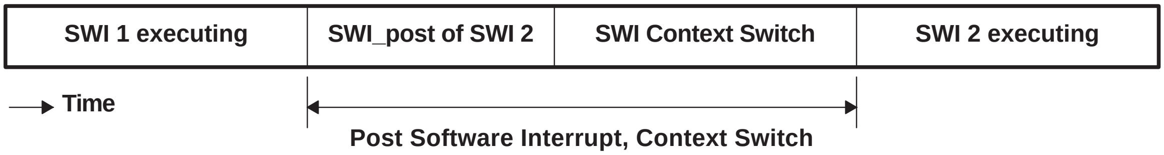

Figure 5. Post Software Interrupt with Context Switch

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 another task of lower or equal priority, which results in no context switch. See Figure 6.

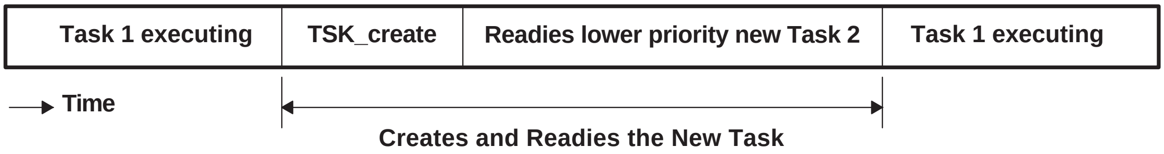
Task 1 executing
TSK_create
Readies lower priority new Task 2
Task 1 executing
Time
Creates and Readies the New Task

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.

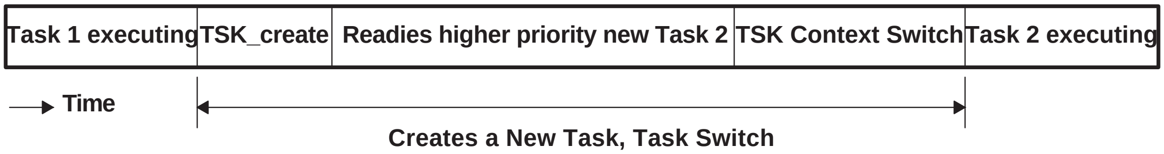
Task 1 executing
TSK_create
Readies higher priority new Task 2
TSK Context Switch
Task 2 executing
Time
Creates a New Task, Task Switch

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's 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.

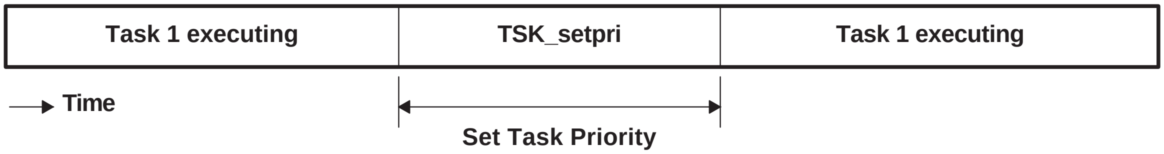

Figure 8. Set a Task's Execution Priority without Context Switch

- *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.

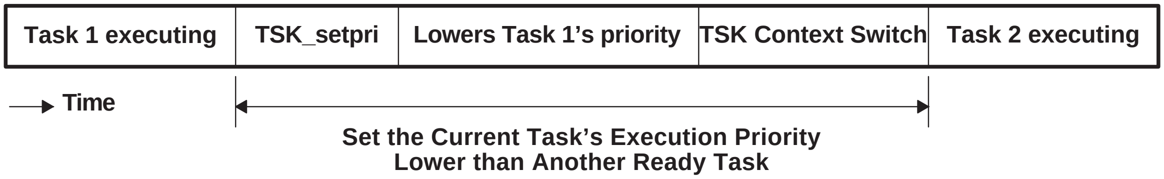

Figure 9. Lower the Current Task's Execution Priority, Context Switch

- *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.

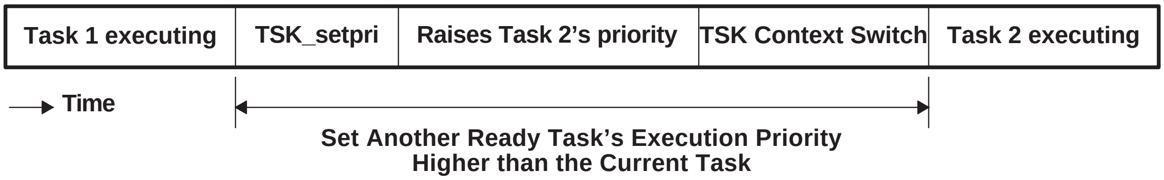

Figure 10. Raise a Ready Task's Execution Priority, Context Switch

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.

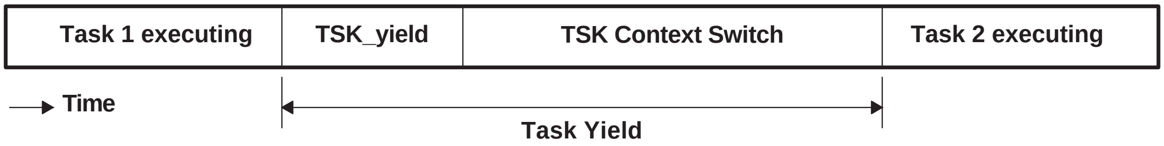

Figure 11. Task Yield

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.

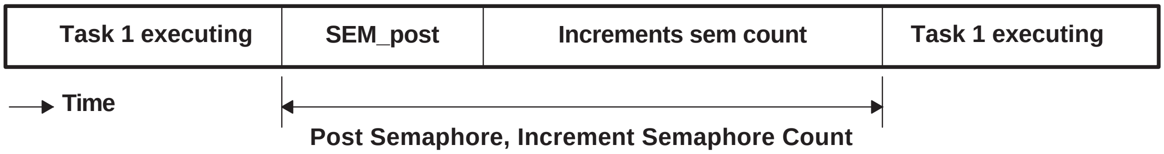

Figure 12. Post Semaphore, Increment Semaphore Count

- *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.

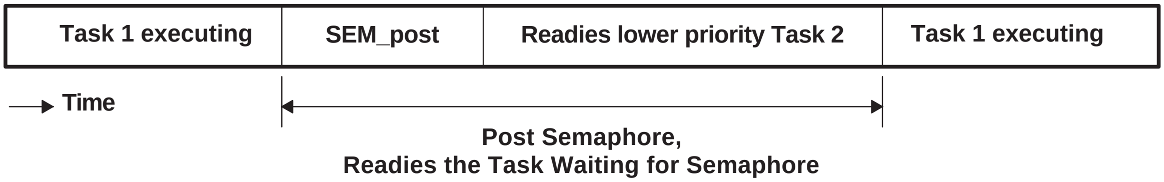
Task 1 executing
SEM_post
Readies lower priority Task 2
Task 1 executing
Time
Post Semaphore,
Readies the Task Waiting for Semaphore

Figure 13. Post Semaphore without Task 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.

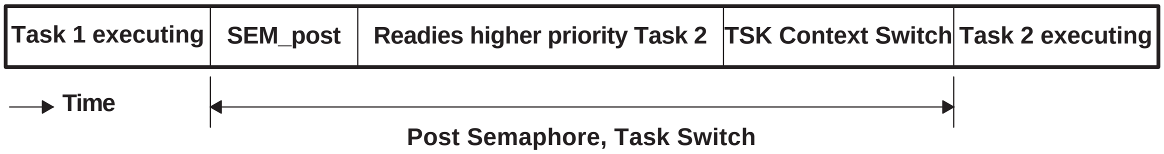

Figure 14. Post Semaphore with Task Switch

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 Figure 15.

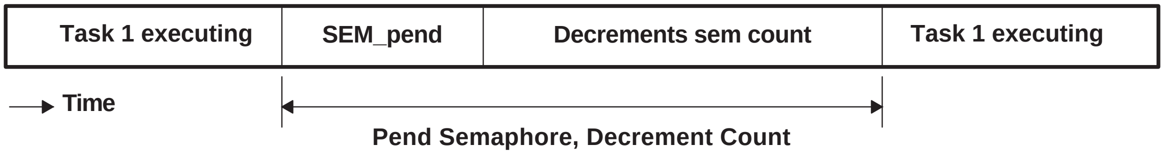

Figure 15. Pend on Semaphore, No Context Switch

- *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 Figure 16.

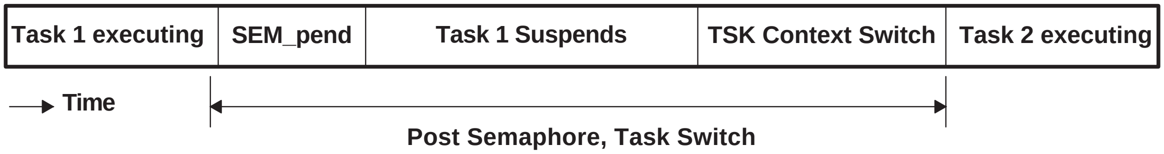

Figure 16. Pend on Semaphore with Task Switch

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.

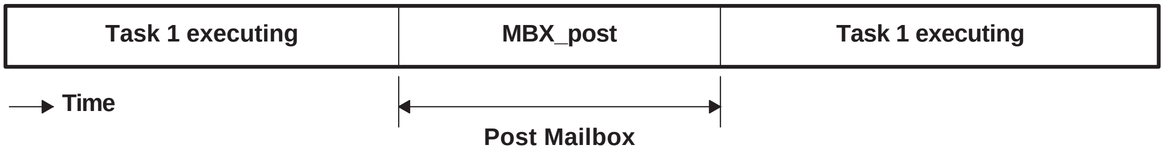

Figure 17. Post Mailbox, No Task Pending on Mailbox

- *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.

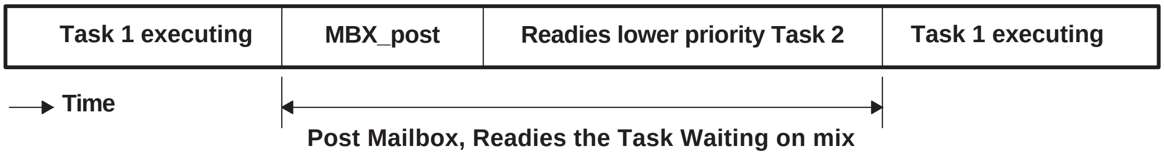
Task 1 executing
MBX_post
Readies lower priority Task 2
Task 1 executing
Time
Post Mailbox, Readies the Task Waiting on mix

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.

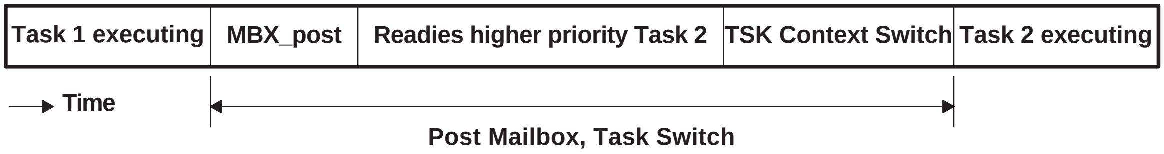

Figure 19. Post a Mailbox with Context Switch

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.

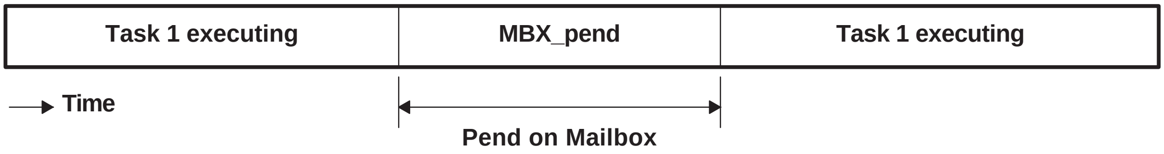

Figure 20. Pend on Mailbox, No Context Switch

- *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.

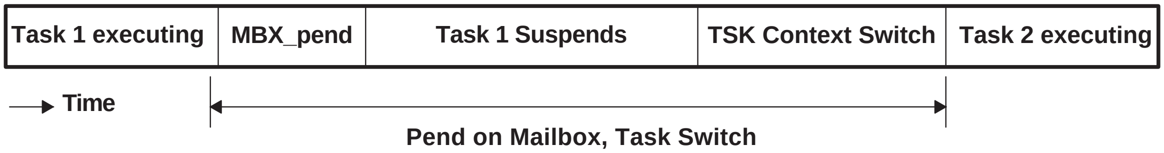

Figure 21. Pend on Mailbox with Context Switch

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.

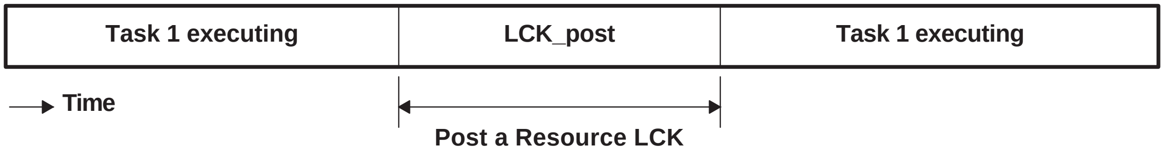

Figure 22. Post a Resource LCK without Ownership Relinquishment

- *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.

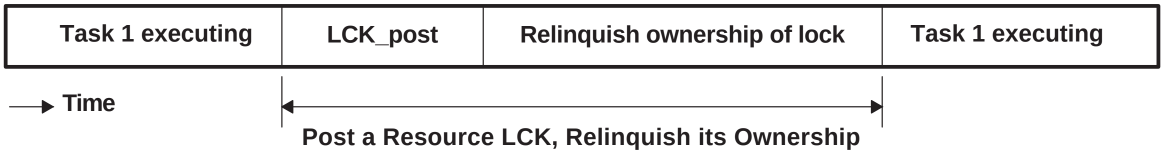
Task 1 executing
LCK_post
Relinquish ownership of lock
Task 1 executing
Time
Post a Resource LCK, Relinquish its Ownership

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.

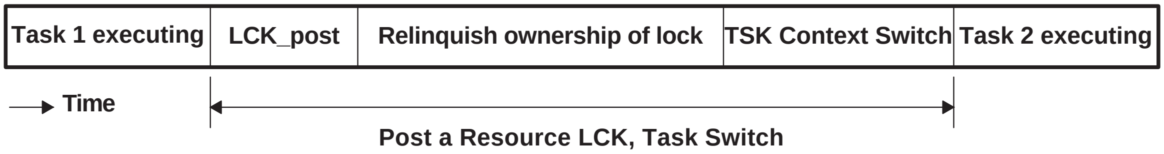

Figure 24. Post a Resource LCK with Context Switch

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.

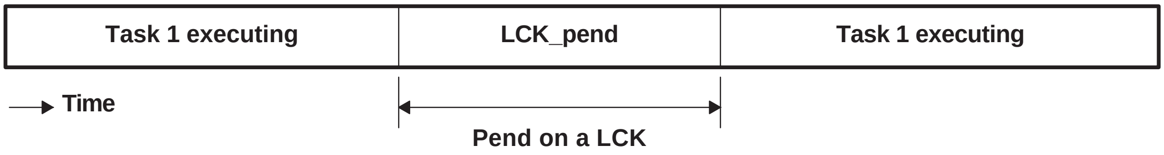

Figure 25. Pend on a Self-Owned Resource LCK

- *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.

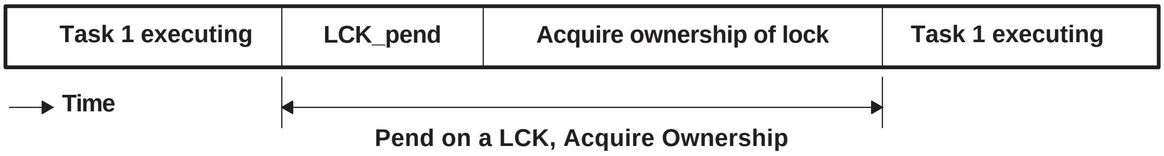

Figure 26. Pend on a Resource LCK without Context Switch

- *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.

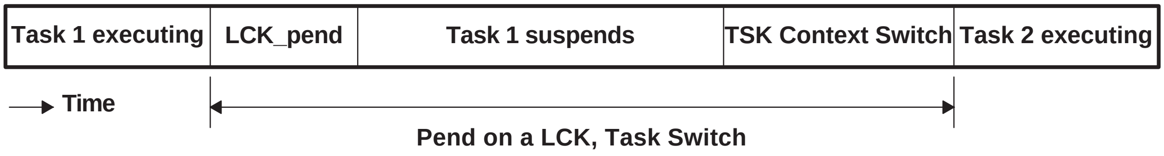

Figure 27. Pend on a Resource LCK with Context Switch

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).

2 DSP/BIOS Benchmarking Methodology

2.1 DSP/BIOS Benchmarking Environment

To obtain the benchmarks, the non-instrumented version of the DSP/BIOS API was used. That is, DSP/BIOS real-time analysis was disabled.

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 of code in conjunction with a specific processor configuration. Table 2 details the memory placement and application configuration.

Table 2. Benchmark Programs Environment Setup

	Memory Placement				
DSP Architecture	**Code**	**Data**	**Heap**	**Application Configuration**	**BoardConfiguration**
TMS320F28x (Large)	H0SARAM	L0SARAM	L0SARAM	Data Model = Large	TMS320F2812 DSK
TMS320C54x (Near)	IPROG	IDATA	IDATA	Code Model = Near	TMS320C5416 DSK
TMS320C54x (Far)	IPROG	IDATA	IDATA	Code Model = Far	TMS320C5416 DSK
TMS320C55x (Small)	SARAM	DARAM	DARAM	Stack Mode = Fast Return Data Model = Small	TMS320C5510 DSK
TMS320C55x (Large)	SARAM	DARAM	DARAM	Stack Mode = Fast Return Data Model = Large	TMS320C5510 DSK
TMS320C62x/C67x† (Best Case)	IDRAM	IDRAM	IDRAM	Flat memory system (Single Cycle Memory access)	TMS320C6201 EVM
TMS320C62x/C67x‡ (Best-Worst Case)	IDRAM	IDRAM	IDRAM	L2 Cache disabled. L1 Data and Program is invalidated before every DSP/BIOS API call	TMS320C6711 DSK
TMS320C64x† (Best Case)	SRAM	SRAM	SRAM	Flat memory system (Single Cycle Memory access)	TMS320C6416 Functional Simulator
TMS320C64x‡ (Best-Worst Case)	SRAM	SRAM	SRAM	L2 Cache disabled. L1 Data and Program cache is invalidated before every DSP/BIOS API call	TMS320C6416 TEB Rev 1.1

† For these “Best Case” benchmarks, a flat memory system is used. A single cycle memory access is used to simulate no cache misses.

‡ For these “Best-Worst Case” benchmarks, L2 is configured as SRAM by disabling L2. 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. The worst case would be to place code and data in external memory. “Best-Worst case” is an intermediate case because although code and data are place in internal memory, the cache is flushed before each benchmark.

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.

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 DSP/BIOS Timing Benchmarks

3.1 DSP/BIOS Benchmark Results for the TMSC28x Architecture

Table 3 provides timing information for DSP/BIOS APIs on the C28x. See Table 2 for a description of the environment used to obtain these benchmarks.

Table 3. Benchmark Results for C28x

	C28x (Large)
Interrupt latency	
Interrupt latency	103
Hardware Interrupts (HWIs)	
HWI_enable	11
HWI_disable	13
Interrupt prolog (minimum)	82
Interrupt prolog for calling C function	94
Interrupt epilog (minimum)	113
Interrupt epilog following C function call	125
Hardware interrupt to blocked task	967
Hardware interrupt to software interrupt	391
Software Interrupts (SWIs)	
SWI_enable	31
SWI_disable	10
Post software interrupt again	45
Post software interrupt, no context switch	93
Post software interrupt, context switch	201
Tasks	
TSK_enable	106
TSK_disable	66
Create a task, no context switch	1886
Create a task, context switch	2016

Table 3. Benchmark Results for C28x (Continued)

	C28x (Large)
TSK_delete	726
Set a task's priority, no context switch	504
Lower the current task's own priority, context switch	695
Raise a ready task's priority, context switch	687
TSK_yield	321
Semaphores	
Post a semaphore, no waiting task	52
Post a semaphore, no context switch	337
Post a semaphore, context switch	427
Pend on a semaphore, no context switch	36
Pend on a semaphore, context switch	393
Mailboxes	
Post a mailbox, no tasks waiting	245
Post a mailbox, no context switch	530
Post a mailbox, context switch	801
Pend on a mailbox, no context switch	244
Pend on a mailbox, context switch	409
Resource Locks	
Post a lock, no ownership relinquishment	17
Post a lock, no context switch	77
Post a lock, context switch	469
Pend on a self-owned lock	49
Pend on a lock, no context switch	88
Pend on a lock, context switch	419
Clock Operations	
CLK_gethtime	35
CLK_getltime	11
Log Operations	
LOG_event	67
LOG_printf	65

Table 3. Benchmark Results for C28x (Continued)

	C28x (Large)
Statistics Operations	
STS_add	15
STS_delta	20
STS_set	12
Memory Operations	
Memory allocated on first block	342
Memory allocated on second block	403
Memory allocated on third block	466
Memory allocated on fourth block	527
Memory coalesces no block	348
Memory coalesces one block	391
Memory coalesces two blocks	416
Pipe Operations	
PIP_alloc	70
PIP_free	78
PIP_get	70
PIP_put	68
PIP_peek	44
Queue Operations	
QUE_dequeue	13
QUE_empty	12
QUE_enqueue	11
QUE_get	40
QUE_insert	7
QUE_put	34
QUE_remove	12

3.2 DSP/BIOS Benchmark Results for the TMSC5000 Architecture

Table 4 provides timing information for DSP/BIOS APIs on the C5000. See Table 2 for a description of the environments used to obtain these benchmarks.

Table 4. Benchmark Results for C5000

	C54x (Near)	C54x (Far)	C55x (Small)	C55x (Large)
Interrupt latency				
Interrupt latency	77	77	142	144
Hardware Interrupts (HWIs)				
HWI_enable	11	12	12	11
HWI_disable	14	15	20	21
Interrupt prolog (minimum)	96	96	94	96
Interrupt prolog for calling C function	110	110	140	137
Interrupt epilog (minimum)	71	72	123	122
Interrupt epilog following C function call	85	86	169	165
Hardware interrupt to blocked task	935	961	1054	1106
Hardware interrupt to software interrupt	386	387	405	416
Software Interrupts (SWIs)				
SWI_enable	48	49	37	38
SWI_disable	21	22	14	13
Post software interrupt again	78	79	46	46
Post software interrupt, no context switch	147	148	90	91
Post software interrupt, context switch	292	293	254	259
Tasks				
TSK_enable	111	118	112	121
TSK_disable	61	69	76	77
Create a task, no context switch	922	986	790	948
Create a task, context switch	1011	1080	973	1185
TSK_delete	673	709	568	649
Set a task’s priority, no context switch	437	461	464	541
Lower the current task’s own priority, context switch	544	569	630	754
Raise a ready task’s priority, context switch	540	565	630	755
TSK_yield	239	258	318	344

Table 4. Benchmark Results for C5000 (Continued)

	C54x (Near)	C54x (Far)	C55x (Small)	C55x (Large)
Semaphores				
Post a semaphore, no waiting task	52	57	53	54
Post a semaphore, no context switch	261	276	249	273
Post a semaphore, context switch	324	340	367	404
Pend on a semaphore, no context switch	32	33	32	31
Pend on a semaphore, context switch	328	335	367	417
Mailboxes				
Post a mailbox, no tasks waiting	219	231	225	228
Post a mailbox, no context switch	428	450	421	447
Post a mailbox, context switch	647	680	703	745
Pend on a mailbox, no context switch	218	230	225	230
Pend on a mailbox, context switch	345	352	381	434
Resource Locks				
Post a lock, no ownership relinquishment	26	28	24	23
Post a lock, no context switch	78	87	76	77
Post a lock, context switch	364	384	399	439
Pend on a self-owned lock	35	36	33	40
Pend on a lock, no context switch	70	72	67	72
Pend on a lock, context switch	346	357	386	439
Clock Operations				
CLK_gethtime	36	37	37	36
CLK_getltime	11	12	13	13
Log Operations				
LOG_event	61	67	54	61
LOG_printf	57	63	61	61
Statistics Operations				
STS_add	43	44	22	22
STS_delta	49	50	24	25
STS_set	20	21	15	16

Table 4. Benchmark Results for C5000 (Continued)

	C54x (Near)	C54x (Far)	C55x (Small)	C55x (Large)
Memory Operations				
Memory allocated on first block	377	394	274	300
Memory allocated on second block	419	436	297	336
Memory allocated on third block	461	478	320	371
Memory allocated on fourth block	503	520	343	300
Memory coalesces no block	390	408	274	318
Memory coalesces one block	413	431	292	374
Memory coalesces two blocks	419	437	292	401
Pipe Operations				
PIP_alloc	104	109	105	105
PIP_free	121	127	105	97
PIP_get	104	109	105	104
PIP_put	123	129	97	99
PIP_peek	38	41	25	28
Queue Operations				
QUE_dequeue	11	11	14	21
QUE_empty	13	13	5	8
QUE_enqueue	13	13	13	12
QUE_get	30	31	38	38
QUE_insert	17	17	15	8
QUE_put	37	38	36	37
QUE_remove	12	12	19	16

3.3 DSP/BIOS Benchmark Results for the TMSC6000 Architecture

Table 5 provides timing information for DSP/BIOS APIs on the C6000. See the footnotes to Table 2 for a description of the “Best Case” and “Best-Worst Case” environments.

Table 5. Benchmark Results for C6000

	C62x/C67x (Best Case)	C62x/C67x (Best-Worst Case)	C64x (Best Case)	C64x (Best-Worst Case)
Interrupt latency				
Interrupt latency	71	110	72	105
Hardware Interrupts (HWIs)				
HWI_enable	12	24	16	32
HWI_disable	12	24	16	40
Interrupt prolog (minimum)	32	80	32	64
Interrupt prolog for calling C function	44	92	56	112
Interrupt epilog (minimum)	40	72	40	88
Interrupt epilog following C function call	52	100	72	144
Hardware interrupt to blocked task	700	1244	752	1296
Hardware interrupt to software interrupt	272	500	288	520
Software Interrupts (SWIs)				
SWI_enable	68	104	72	112
SWI_disable	20	36	24	40
Post software interrupt again	60	100	64	112
Post software interrupt, no context switch	116	188	120	184
Post software interrupt, context switch	232	392	232	384
Tasks				
TSK_enable	104	192	104	192
TSK_disable	64	100	72	128
Create a task, no context switch	1220	1664	744	1152
Create a task, context switch	1316	1796	840	1288
TSK_delete	476	820	480	816
Set a task’s priority, no context switch	348	572	344	576
Lower the current task’s own priority, context switch	440	736	432	696

Table 5. Benchmark Results for C6000 (Continued)

	C62x/C67x (Best Case)	C62x/C67x (Best-Worst Case)	C64x (Best Case)	C64x (Best-Worst Case)
Raise a ready task's priority, context switch	436	732	432	688
TSK_yield	232	424	232	432
Semaphores				
Post a semaphore, no waiting task	24	56	32	64
Post a semaphore, no context switch	200	356	200	360
Post a semaphore, context switch	260	492	264	488
Pend on a semaphore, no context switch	16	32	16	56
Pend on a semaphore, context switch	232	472	240	448
Mailboxes				
Post a mailbox, no tasks waiting	140	260	112	240
Post a mailbox, no context switch	312	560	288	536
Post a mailbox, context switch	476	836	432	768
Pend on a mailbox, no context switch	136	252	112	232
Pend on a mailbox, context switch	244	484	248	472
Resource Locks				
Post a lock, no ownership relinquishment	28	40	32	56
Post a lock, no context switch	44	84	48	96
Post a lock, context switch	292	536	296	520
Pend on a self-owned lock	32	52	32	48
Pend on a lock, no context switch	52	88	48	96
Pend on a lock, context switch	256	496	256	496
Clock Operations				
CLK_gethtime	28	56	32	72
CLK_getltime	16	36	16	40
Log Operations				
LOG_event	32	48	32	64
LOG_printf	36	64	40	88

Table 5. Benchmark Results for C6000 (Continued)

	C62x/C67x (Best Case)	C62x/C67x (Best-Worst Case)	C64x (Best Case)	C64x (Best-Worst Case)
Statistics Operations				
STS_add	16	28	16	32
STS_delta	20	32	24	40
STS_set	12	20	16	32
Memory Operations				
Memory allocated on first block	188	348	184	336
Memory allocated on second block	208	364	208	368
Memory allocated on third block	228	380	224	384
Memory allocated on fourth block	240	404	248	424
Memory coalesces no block	216	388	224	360
Memory coalesces one block	220	380	224	376
Memory coalesces two blocks	220	388	224	376
Pipe Operations				
PIP_alloc	96	140	96	152
PIP_free	92	180	96	168
PIP_get	96	148	96	160
PIP_put	92	164	96	160
PIP_peek	20	32	24	40
Queue Operations				
QUE_dequeue	16	32	24	32
QUE_empty	12	16	16	16
QUE_enqueue	16	28	16	24
QUE_get	16	36	24	48
QUE_insert	12	16	16	16
QUE_put	16	24	16	32
QUE_remove	16	20	16	16

4 References

1. *TMS320 DSP/BIOS User's Guide* (SPRU423B)
2. *TMS320C28x DSP/BIOS API Reference Guide* (SPRU625)
3. *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404E)
4. *TMS320C6000 DSP/BIOS API Reference Guide* (SPRU403E)

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
		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