DSP/BIOS Timing Benchmarks on the TMS320C6000/ TMS320C54x DSPs for Code Composer Studio 2.0

Shawn Dirksen *Software Development Systems*

ABSTRACT

This document describes each of the DSP/BIOS™ performance benchmarks and the accompanying results, followed by a technique used for calculating overall system performance or overhead. To help designers better analyze their system performance, the methodology used has been detailed for obtaining each of the benchmarks, along with the number of CPU cycles to execute each of the DSP/BIOS functions. The designers can then compute the sum of these components and the frequency of occurrence to determine the total system performance for their application.

Contents

List of Figures

List of Tables

1 DSP/BIOS Timing Benchmarks

DSP/BIOS functions have been described along with the approach taken to measure each performance benchmark.

1.1 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.

1.2 STS – Statistics Benchmarks

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

1.3 TSK – Task Yield Benchmarks

TSK_yield. This is a measurement of the elapsed time between a function call to TSK_yield (which causes preemption of the current thread yielding control of the processor), and the execution of the next instruction in a currently blocked task of equal priority, as shown in Figure 1.

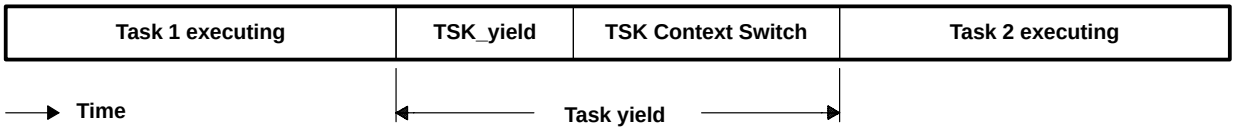

Figure 1. Task Yield Benchmarks

1.4 SEM – Semaphore Benchmarks

The semaphore benchmarks measure the time interval between the issuance of a post (SEM_post) or pend (SEM_pend) function call and the resumption of task execution, both with and without a context switch. The results are independent of task priority, an inherent characteristic of the DSP/BIOS kernel that makes it ideal for signal-processing applications that require predictable, consistent real-time response.

Post of a Semaphore, no context switch. This is a measurement of a SEM_post function call, when the posted task is not higher-priority than the currently running TSK, and no preemption occurs, as shown in Figure 2.

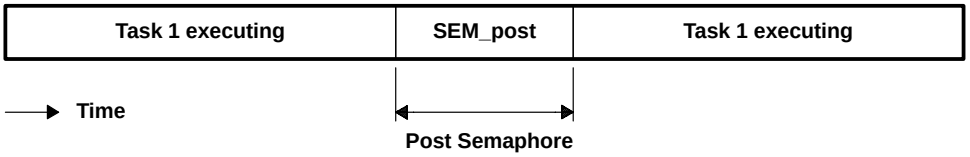

Figure 2. Semaphore Benchmarks

Post of a Semaphore, with context switch. This is a measurement of the elapsed time between a function call to SEM_post (which causes preemption of the current task), and the execution of the first instruction in the higher-priority task, as shown in Figure 3.

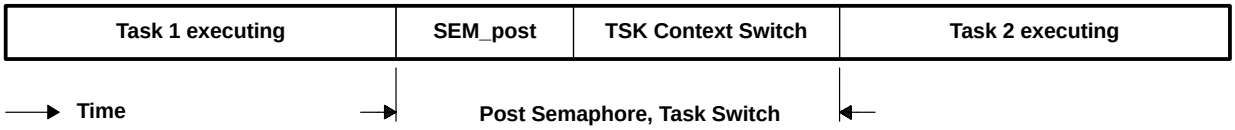

Figure 3. Post of Semaphore Task Switch

Pend of a Semaphore, no context switch. This is a measurement of a SEM_pend function call without a context switch.

Pend of a Semaphore, with 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 the higher-priority task.

1.5 SWI – Software Interrupt Benchmarks

Post of Software Interrupt, no context switch. This is a measurement of a SWI_post function call, when the posted software interrupt is not higher priority than the currently running SWI, and no preemption occurs. See Figure 4.

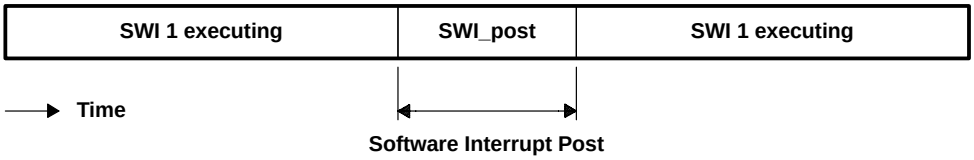

Figure 4. Software Interrupt Benchmarks

Post of Software Interrupt, with context switch. This is a measurement of the elapsed time between a function call to SWI_post (which causes preemption of the current thread), and the execution of the first instruction in the higher-priority software interrupt, as shown in Figure 5. The context switch for SWI 2 is performed within the SWI executive, and this time is included within the measurement.

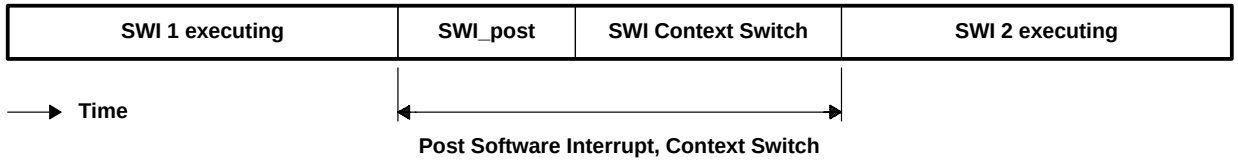

Figure 5. Post of Software Context Switch

1.6 HWI – Hardware Interrupt Benchmarks

These benchmarks exhibit the interrupt latency typical of most interrupt processing applications independent of a kernel being used. The interrupt latency provides a useful measure of worst-case interrupt response, but does not reflect the scheduling capability of the DSP/BIOS kernel (launching threads to perform background processing for the ISR). This is further demonstrated in the Hardware Interrupt to Software Interrupt and the Hardware Interrupt to Blocked Task benchmarks.

Interrupt prolog. This is a measurement of the execution time of an HWI_enter macro call. HWI_enter must be called in an ISR prior to any DSP/BIOS API calls that can post or affect a software interrupt (SWI). The execution time of the HWI_enter macro depends upon the list of registers to be saved for the ISR, as defined in masks specified by the user. This benchmark shows the minimum execution time for the prolog, with no registers saved.

Interrupt prolog for calling C function. This measurement is similar to the previous (interrupt prolog), but in this case the time shown in the data sheet corresponds to all C caller-preserved registers being saved, so that a C function can be called from the assembly stub.

Interrupt epilog. This is a measurement of the execution time of an HWI_exit macro call. HWI_exit must be the last statement of any ISR that calls HWI_enter. 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, with no registers restored, and no higher-priority SWIs posted in the ISR (i.e., following the ISR, execution resumes with the thread that was preempted by the hardware interrupt).

Interrupt epilog following C function call. This measurement is similar to the previous (Interrupt epilog), but in this case the time shown in the data sheet corresponds to all C caller-preserved registers being restored, with no higher-priority SWIs posted in the ISR.

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 the blocked task, as shown in Figure 6.

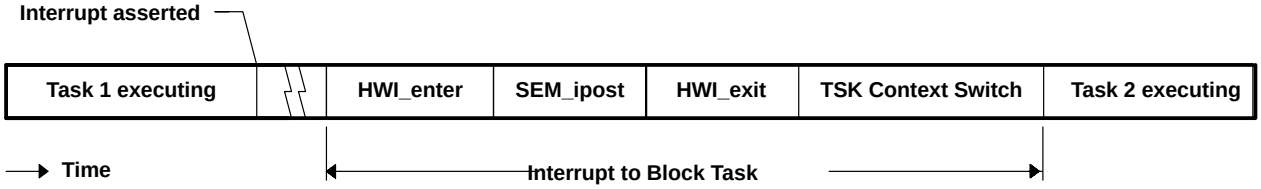

Figure 6. Hardware Interrupt to Blocked Task

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 posted software interrupt, as shown in Figure 7.

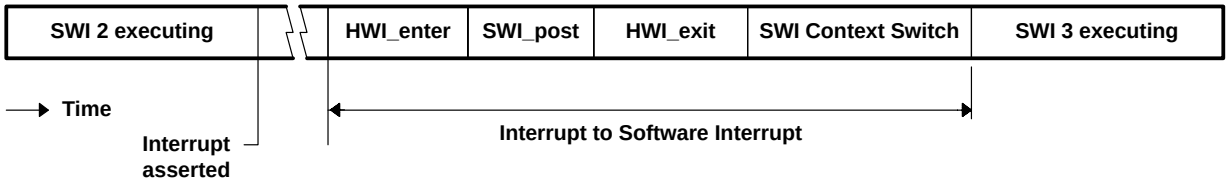

Figure 7. Hardware Interrupt to Software Interrupt

In Figure 7, SWI 3 has a higher priority than SWI 2, so SWI 2 is preempted. The context switch for SWI 3 is performed within the SWI executive, and this time is included within the measurement. In this case, the registers saved/restored by HWI_enter/HWI_exit are only those modified by the SWI_post assembly macro.

Interrupt Latency. This is the maximum latency time during which the DSP/BIOS kernel disables maskable interrupts.

1.7 MBX – Mailbox Benchmarks

Post of a mailbox, no context switch. This is a measurement of a MBX_post function call when the posted mailbox copies a message into the unfilled mailbox, and no higher-priority task is pending on the mailbox. See Figure 8.

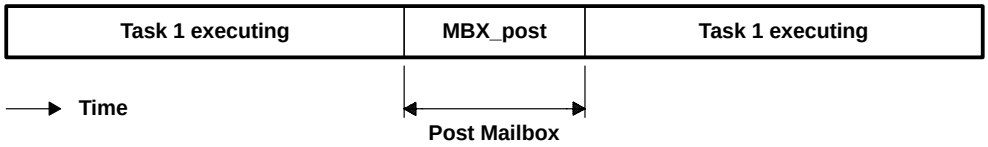

Figure 8. Mailbox Benchmarks

Post of a mailbox, with context switch. This is a measurement of the elapsed time between a function call to MBX_post (which causes preemption of the current task), and a context switch to a higher-priority task pending on the mailbox. See Figure 9.

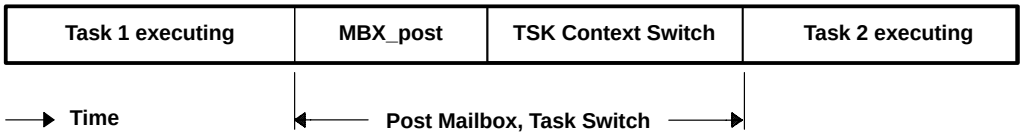

Figure 9. Post of a Mailbox With Context Switch

Pend of a mailbox, no context switch. This is a measurement of a MBX_pend function call when the unfulfilled pending mailbox copies a message, and no higher-priority task is pending on the mailbox.

Pend of a mailbox, with context switch. This is a measurement of the elapsed time between a function call to MBX_pend (which causes preemption of the current task) if the mailbox is empty or a higher-priority task is blocked on a MBX_post.

1.8 PIP – Pipe Benchmarks

NOTE: Each of the following pipe benchmarks includes the execution time of a minimal notifyWriter (or notifyReader) C function call, i.e., a function that just does a return, but is considered to have modified all C caller-preserved registers.

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.

2 DSP/BIOS Timings

2.1 TMS320C6000 DSP Benchmark Results

This data contains timing information of the DSP/BIOS kernel for TMS320C6000™ digital signal processors. These timings apply to the floating-point processor as well.

Environment Testing Platform: TMS320C6201 EVM, using internal memory for both code and data.

Software: DSP/BIOS version 4.51, built with TI Code Generation Tools, version 4.10.

Table 1. Benchmark Results for C6000 DSPs

	Non-Instrumented CPU Cycles	Non-Instrumented† Time (usec) at 100 MHz‡	Instrumented CPU Cycles	Instrumented Time (usec) at 100 MHz‡
LOG Operations				
LOG_event	30	0.150	30	0.150
LOG_printf	36	0.18	36	0.180
Statistics Operations				
STS_set	12	0.060	12	0.060
STS_add	15	0.075	15	0.075
STS_delta	21	0.105	21	0.105

† These timings were performed using the non-instrumental and instrumental kernel. Refer to *DSP/BIOS Sizing Guidelines in Code Composer Studio 2.0* (SPRA772) for details regarding scaling and code size of DSP/BIOS.
‡ For a 200 MHz C6201 processor, the CPU cycle period is 5 nanoseconds.
§ These measurements relate to the DSP/BIOS assembly language API, not the C language API.

TMS320C6000 is a trademark of Texas Instruments.

Table 1. Benchmark Results for C6000 DSPs (Continued)

	Non-Instrumented CPU Cycles	Non-Instrumented† Time (usec) at 100 MHz‡	Instrumented CPU Cycles	Instrumented Time (usec) at 100 MHz‡
Task Yield				
TSK_yield	232	1.160	336	1.680
Semaphores				
Post semaphore, no task switch	173	0.865	259	1.295
Post semaphore, task switch	279	1.395	429	2.145
Pend semaphore, no task switch	149	0.745	200	1.000
Pend semaphore, task switch	267	1.335	382	1.910
Software Interrupts (SWIs)				
Post of software interrupt, no context switch	120	0.600	120	0.600
Post of software interrupt, with context switch	243	1.215	249	1.245
Hardware Interrupts (HWIs)§				
Interrupt prolog (minimum)	33	0.165	33	0.165
Interrupt prolog for calling C function	41	0.205	41	0.205
Interrupt epilog (minimum)	37	0.185	37	0.185
Interrupt epilog following C function call	49	0.245	49	0.245
Hardware interrupt to blocked task	731	3.655	871	4.355
Hardware interrupt to software interrupt	288	1.44	294	1.470
Interrupt latency	88	0.44	92	0.460
Mailboxes				
Post mailbox, no task switch	421	2.105	558	2.790
Post mailbox, task switch	780	3.900	1066	5.330
Pend mailbox, no task switch	424	2.120	560	2.800
Pend mailbox, task switch	276	1.380	391	1.955
Pipe Operations				
PIP alloc	96	0.480	96	0.480
PIP free	93	0.465	93	0.465
PIP get	96	0.48	96	0.480
PIP put	95	0.475	95	0.475

† These timings were performed using the non-instrumental and instrumental kernel. Refer to *DSP/BIOS Sizing Guidelines in Code Composer Studio 2.0* (SPRA772) for details regarding scaling and code size of DSP/BIOS.
‡ For a 200 MHz C6201 processor, the CPU cycle period is 5 nanoseconds.
§ These measurements relate to the DSP/BIOS assembly language API, not the C language API.

2.2 TMS320C54x DSP Benchmark Results

This data contains timing information of the DSP/BIOS kernel for TMS320C54x™ digital signal processors.

Environment Testing Platform: TMS320C5410 EVM, using internal memory for both code and data.

Software: DSP/BIOS version 4.51, built with TI Code Generation Tools, version 3.70.

Table 2. Benchmark Results for C54x DSPs

	Non-Instrumented CPU Cycles	Non-Instrumented† Time (usec) at 100 MHz‡	Instrumented CPU Cycles	Instrumented Time (usec) at 100 MHz‡
LOG Operations				
LOG_event	61	0.61	61	0.61
LOG_printf	61	0.61	61	0.61
Statistics Operations				
STS_set	20	0.20	20	0.20
STS_add	43	0.43	43	0.43
STS_delta	49	0.49	49	0.49
Task Yield				
TSK_yield	281	2.81	365	3.65
Semaphores				
Post semaphore, no task switch	180	1.80	244	2.44
Post semaphore, task switch	360	3.60	482	4.82
Pend semaphore, no task switch	176	1.76	218	2.18
Pend semaphore, task switch	371	3.71	454	4.54
Software Interrupts (SWIs)				
Post of software interrupt, no context switch	98	0.98	98	0.98
Post of software interrupt, with context switch	246	2.46	246	2.46

† These timings were performed using the non-instrumental and instrumental kernel. Refer to *DSP/BIOS Sizing Guidelines in Code Composer Studio 2.0* (SPRA772) for details regarding scaling and code size of DSP/BIOS.
‡ For a 100 MHz C5402 processor, the CPU cycle period is 10 nanoseconds.
§ These measurements relate to the DSP/BIOS assembly language API, not the C language API.

TMS320C54x is a trademark of Texas Instruments.

Table 2. Benchmark Results for C54x DSPs (Continued)

	Non-Instrumented CPU Cycles	Non-Instrumented† Time (usec) at 100 MHz‡	Instrumented CPU Cycles	Instrumented Time (usec) at 100 MHz‡
Hardware Interrupts (HWIs)§				
Interrupt prolog (minimum)	82	0.82	82	0.82
Interrupt prolog for calling C function	96	0.96	96	0.96
Interrupt epilog (minimum)	88	0.88	88	0.88
Interrupt epilog following C function call	102	1.02	102	1.02
Hardware interrupt to blocked task	928	9.28	1025	10.25
Hardware interrupt to software interrupt	323	3.23	323	3.23
Interrupt latency	100	1.00	100	1.00
Mailboxes				
Post mailbox, no task switch	491	4.91	597	5.97
Post mailbox, task switch	961	9.61	1189	11.89
Pend mailbox, no task switch	492	4.92	598	5.98
Pend mailbox, task switch	386	3.86	469	4.69
Pipe Operations				
PIP alloc	104	1.04	104	1.04
PIP free	121	1.21	121	1.21
PIP get	104	1.04	104	1.04
PIP put	123	1.23	123	1.23

† These timings were performed using the non-instrumental and instrumental kernel. Refer to *DSP/BIOS Sizing Guidelines in Code Composer Studio 2.0* (SPRA772) for details regarding scaling and code size of DSP/BIOS.
‡ For a 100 MHz C5402 processor, the CPU cycle period is 10 nanoseconds.
§ These measurements relate to the DSP/BIOS assembly language API, not the C language API.

3 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 estimate the overhead in DSP/BIOS applications, the developer must first identify all the DSP/BIOS components and API calls within the application. In the sample application audio I/O example, the DSP/BIOS components are:

- One HWI object mapped to the Audio;
- One SWI object to do the processing (copy) operation; and
- Two data pipes; one for input, and one for output.

The component overhead in instruction cycles may be taken from the DSP/BIOS timings as listed in Table 1. To process a single buffer of audio data requires the total overhead of 1106 cycles on a C6000. The processing period is 4 ms, so the frequency of occurrence is 250 times per second. Therefore, the total number of cycles in one second, attributed to DSP/BIOS overhead running the audio thread on a C6000 DSP is 276,500 or 0.276500 MIPS. On a 200 MHz C6000 DSP, this equates to a 0.14% CPU load. Further explanation of this calculation is demonstrated in the *DSP/BIOS Kernel Technical Overview* (SPRA780).

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.

4 References

1. *DSP/BIOS Sizing Guidelines in Code Composer Studio 2.0* (SPRA772).
2. *DSP/BIOS Kernel Technical Overview* (SPRA780).

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265