TEXAS INSTRUMENTS

Application Report

SPRA829 - July 2002

DSP/BIOS Timers and Benchmarking Tips

Harish Thampi S *Software Development Systems*

ABSTRACT

Digital signal processors (DSPs) typically have one or more on-chip timers that generate hardware interrupts at periodic intervals. DSP/BIOS™ normally uses one of the available on-chip timers as a source for its system clock. The on-chip timer can also be used to generate periodic hardware interrupts from the user application. DSP/BIOS also has a PRD module that allows the user to trigger periodic functions based on events that are triggered by certain sources. This application report describes the DSP/BIOS timers, clock (CLK), and periodic function (PRD) modules of DSP/BIOS. This document also explains how to configure a periodic function, and how to configure an on-chip timer to generate periodic hardware interrupts. The report also gives some tips on benchmarking with regard to the timers and the clock module. Examples that run on TMS320C6711 DSP starter kit (DSK) and TMS320C5402 DSK are included. The examples demonstrate how to configure timers, and periodic and clock functions using the configuration tool.

Contents

List of Figures

List of Tables

1 Introduction

Digital signal processors typically have one or more on-chip timers that generate hardware interrupt at periodic intervals. DSP/BIOS normally uses one of the available on-chip timers as a source for its system clock. The on-chip timer can also be used to generate periodic hardware interrupts from the user application. DSP/BIOS also has a PRD module that allows the user to trigger periodic functions based on events that are triggered by certain sources. The purpose of this application report is to help DSP/BIOS users to understand more about the system clock, understand the difference between a clock function and a PRD function, and also learn how to use the on-chip timer to generate periodic interrupts for a user application. The document also gives some tips on benchmarking with regard to the timers and the clock module. Examples that run on TMS320C6711 DSK and TMS320C5402 DSK are included. The examples demonstrate how to configure timers, and periodic and clock functions using the configuration tool. The essence of what is covered in this application report is described in Figure 1. After going through this application report, you will be able to understand the relation between the DSP/BIOS system clock, PRDs, and timers.

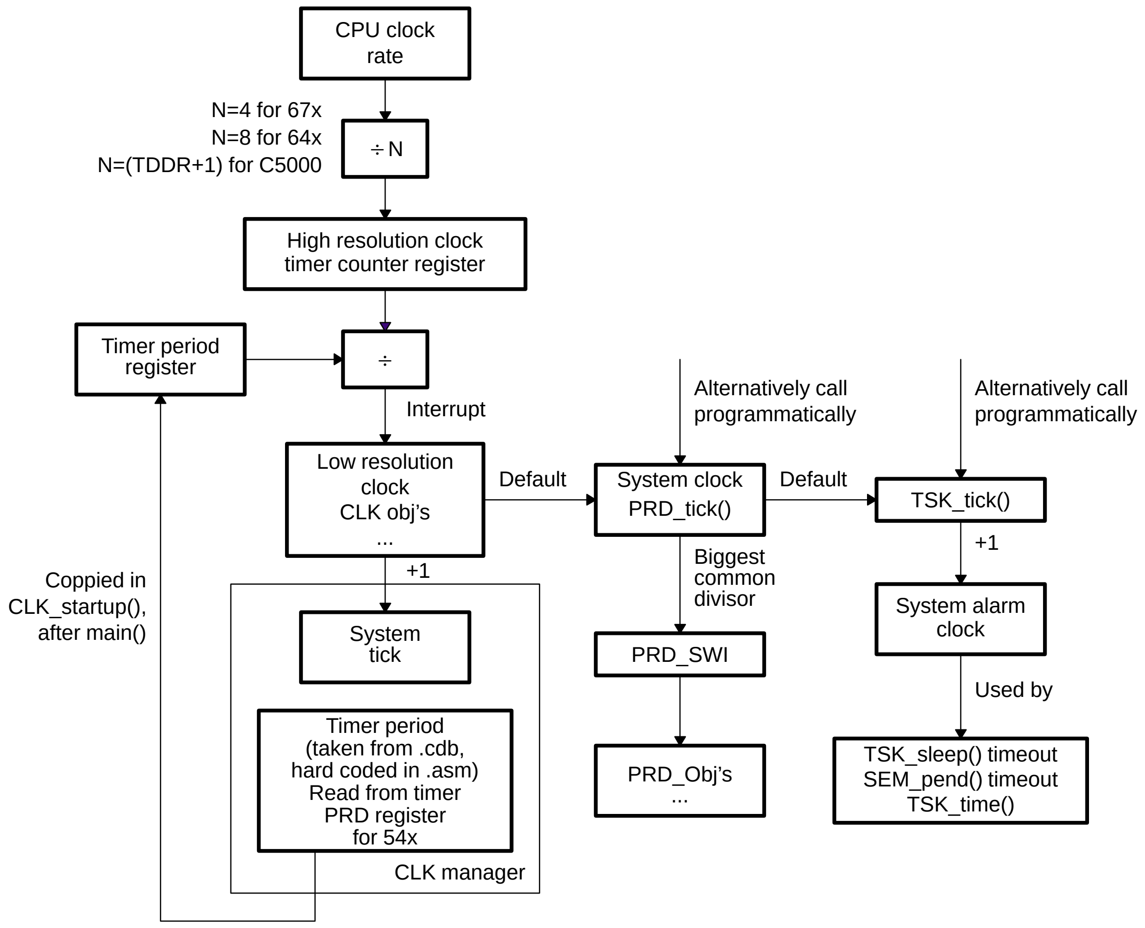

CPU clock rate
N=4 for 67x
N=8 for 64x
N=(TDDR+1) for C5000
÷ N
High resolution clock timer counter register
Timer period register
÷
Interrupt
Alternatively call programmatically
Alternatively call programmatically
Low resolution clock CLK obj's ...
Default
System clock PRD_tick()
Default
TSK_tick()
+1
+1
Coppied in CLK_startup(), after main()
System tick
Biggest common divisor
System alarm clock
PRD_SWI
Used by
Timer period (taken from .cdb, hard coded in .asm) Read from timer PRD register for 54x
PRD_Obj's ...
TSK_sleep() timeout SEM_pend() timeout TSK_time()
CLK manager

Figure 1. Relation Between Timers, PRDs, and CLKs

By default, an on-chip timer drives the DSP/BIOS system clock. Alternatively, It may also be driven by an external source. The DSP/BIOS application programming interfaces (APIs) get the timer period values from the configuration files, except for TMS320C54x™, where the value is actually read from the PRD register. The on-chip timers are driven by the divided-down central processing unit (CPU) clock. The divide-down ratio for the various TI chips is shown in Figure 1.

2 DSP/BIOS Timers

2.1 DSP On-Chip Timers

Digital signal processors typically have one or more on-chip timers that generate the hardware interrupt at periodic intervals. They can be used to time or count events, generate pulses or interrupt the CPU. The timers have two signaling modes, and can be clocked by an external clock or the CPU clock. By default they are clocked internally. The timer output can be configured as a timer output or a general-purpose output. When an internal clock drives the timer, the frequency on the timer input clock varies across the processor generations. Table 1 gives the timer input frequency for different TI chips as a ratio of the CPU clocking rate.

TMS320C54x is a trademark of Texas Instruments.

Table 1. Timer Clock for DSPs

Feature	TMS320C620x/ C670x	TMS320C621x/ C671x	TMS320C64x	TMS320C54x/C55x
Timer input clock frequency	CPU rate / 4	CPU rate / 4	CPU rate / 8	CPU rate / (TDDR +1)

2.2 Timer Operation

For TMS320C6000™ devices, the on-chip timer has a period count register (PRD), timer count register (CNT), and a timer control register (CTL). The timer's CTL register contains the control bits for configuring the timer. It also contains bits for setting the mode of operation, input source, and function. Apart from setting the appropriate values in the control register (CTL), the timer period value is set in the timer PRD register. When the timer starts from reset, the timer CNT register is incremented once every tick of the timer input clock. When the value in the timer CNT register equals the timer PRD register, the timer is reset to 0 in the next CPU clock. Thus, the counter counts from 0 to the value set in the timer PRD register. Every time the timer CNT register value equals the value of the timer PRD register, a timer interrupt occurs. In the default configuration, this timer interrupt advances the DSP/BIOS system clock by one tick.

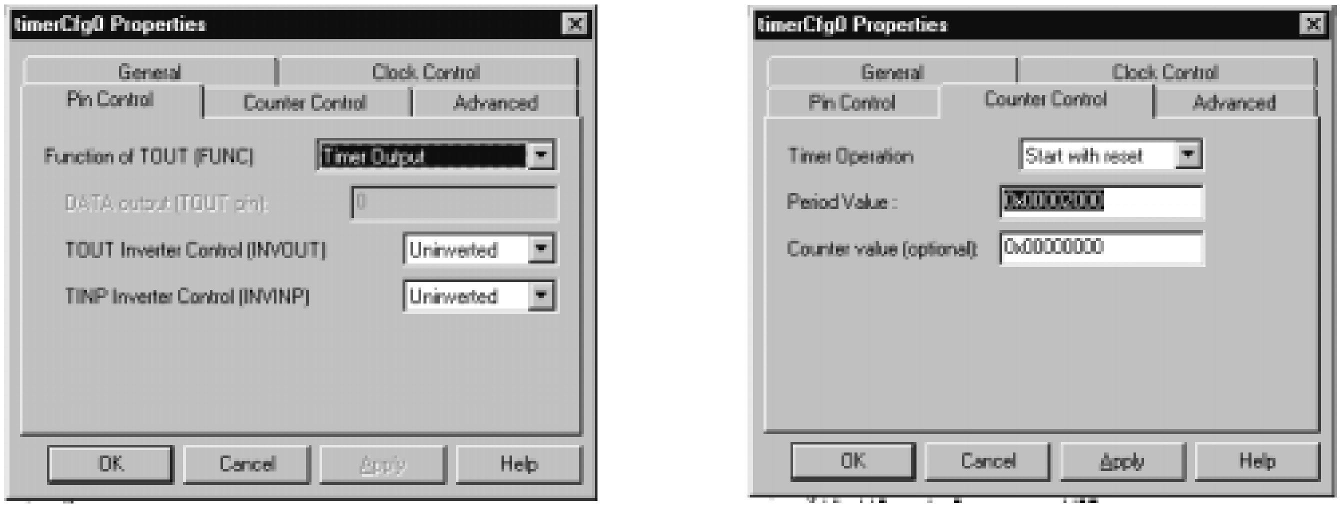
For TMS320C5000™ devices the on-chip timer has a period count register (PRD), timer register (TIM), and a timer control register (TCR). Bits 0 to 3 of the TCR is for the timer divide-down ratio (TDDR) field, and bits 6 to 9 are for the prescalar (PSC) field. Unlike the C6000™ devices, where the CPU clock is divided by a fixed value and fed to the timer input, C5000™ devices make use of the TDDR value to divide down the CPU clock. The value in the TDDR field specifies the divide-down ratio of the CPU clock. The value in TDDR is copied to the prescaler counter (PSC) field and decremented on each CPU clock tick. Whenever the PSC reaches zero, the value in TDDR is reloaded to PSC, and the TIM register is decremented by one. When the TIM register reaches zero, a timer interrupt is triggered and the value in PRD is reloaded to the TIM register. Effectively, the CPU clock is divided by (TDDR +1) and fed to the timer.

2.3 Configuring the On-Chip Timer to Generate Periodic Hardware Interrupts

This section shows how to configure the on-chip timer 1 to generate periodic hardware interrupts on a TMS320C6711DSK. Copy the hello2 example, shipped with Code Composer Studio™ and available in the folder <install directory>\tutorials\dsk6711\hello2, to the myprojects directory, and open the hello2 project. Open the configuration file of the hello2 project, and in the Chip Support Library (CSL) section of the configuration file, expand **TIMER – Timer Device**. There are two subsections: a **TIMER Configuration Manager** and a **TIMER Resource Manager**. Right-click on the TIMER Configuration Manager, and insert a **timercfg** object called "timercfg0". Right-click, and select the properties of the new timercfg object. In the properties window, the **General** tab can be ignored; it has only a comment field. Choose the **Pin Control** tab, and set the FUNC field to "timer output". This sets the TOUT pin to be used as a timer output. The value in the INVOUT does not matter because the internal CPU clock is used here. Now select the **Counter Control** tab, and set the value of 0x2000 in the Period Value field. This means that the timer interrupt will occur every time the timer CNT register reaches 0x2000. The counter value is optional. The Timer Operation field determines the state of the timer. It can be set to start when reset, or restart or start with the reset option. Select the "start with reset" option.

TMS320C6000, TMS320C5000, C6000, C5000, and Code Composer Studio are trademarks of Texas Instruments.

Now, move on to the **Clock Control** tab. In the CLKSRC field, you can select the source of the timer input clock. Select (CPU clock)/4 option because for the TMS3206711 DSK, the timer input clock is one-fourth the frequency on the CPU clock. The timer can be set to run in clock mode or pulse mode by the appropriate selection of the CP field. Select clock mode. If pulse mode is selected, the pulse width will have to be specified. Move on to the **Advanced** tab. The PRD and CNT fields will reflect the values in the **Counter control** tab. In this section, you need to configure the timer control register appropriately. Set the timer control register to 0x3C1. This field will be automatically set according the settings selected in the properties window. The value of 0x3C1 in the timer control register will set the clock source to internal clock, operating mode to clock mode, TOUT pin to be a timer output, HLD to 1, and the GO bit to one. Since the HLD and GO bits are set to 1, the timer is reset, and it starts counting from 0. Apply the settings. See Figure 2 and Figure 3.



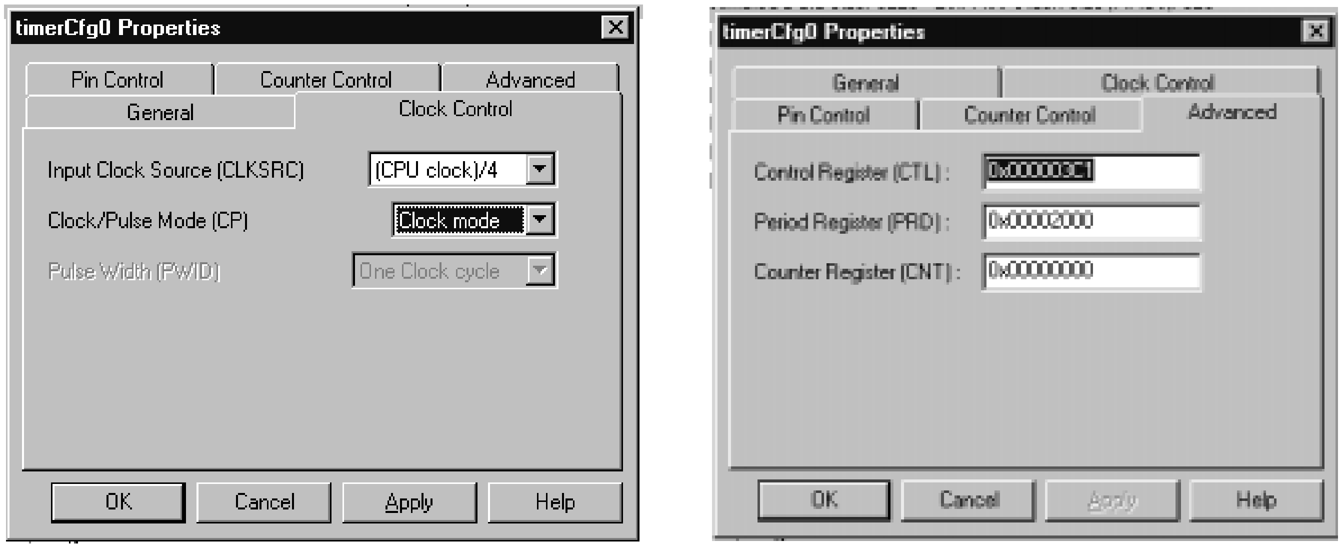
timerCfg0 Properties – Pin Control

Function of TOUT (FUNC): Timer Output
DATA output (TOUT pin): 0
TOUT Inverter Control (INVOUT): Uninverted
TINP Inverter Control (INVINP): Uninverted

timerCfg0 Properties – Counter Control

Timer Operation: Start with reset
Period Value: 0x00002000
Counter value (optional): 0x00000000

Figure 2. Timer Object Properties (a)



timerCfg0 Properties – Clock Control

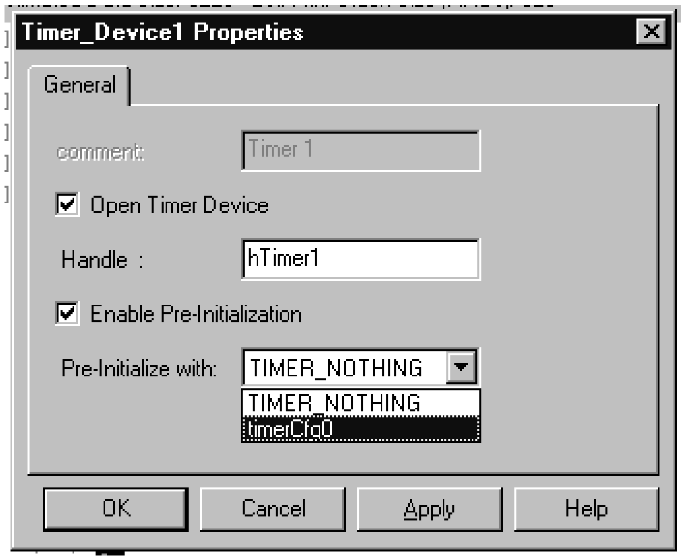
Input Clock Source (CLKSRC): (CPU clock)/4
Clock/Pulse Mode (CP): Clock mode
Pulse Width (PWID): One Clock cycle

timerCfg0 Properties – Advanced

Control Register (CTL): 0x000003C1
Period Register (PRD): 0x00002000
Counter Register (CNT): 0x00000000

Figure 3. Timer Object Properties (b)

You have now configured the timer object; however, this does not mean that the timer itself is configured. To configure the timer, expand the **TIMER – Resource Manger** list. You are configuring timer 1, so right-click on **Timer Device1**, and select “properties”. In the properties window, click to select the “Open Timer Device” option; this will activate the edit box to input name of timer handle. Use the default handle name “hTimer1”. Now select the “Enable Pre-initialization” option. This activates the “Pre-initialization with” drop-down options. Select the timercfg0 from the list of available options. Apply the settings. This will make sure that timer 1 is configured according to the settings in the configuration object, timerCfg0. See Figure 4.


Timer_Device1 Properties

comment: Timer 1

[✓] Open Timer Device

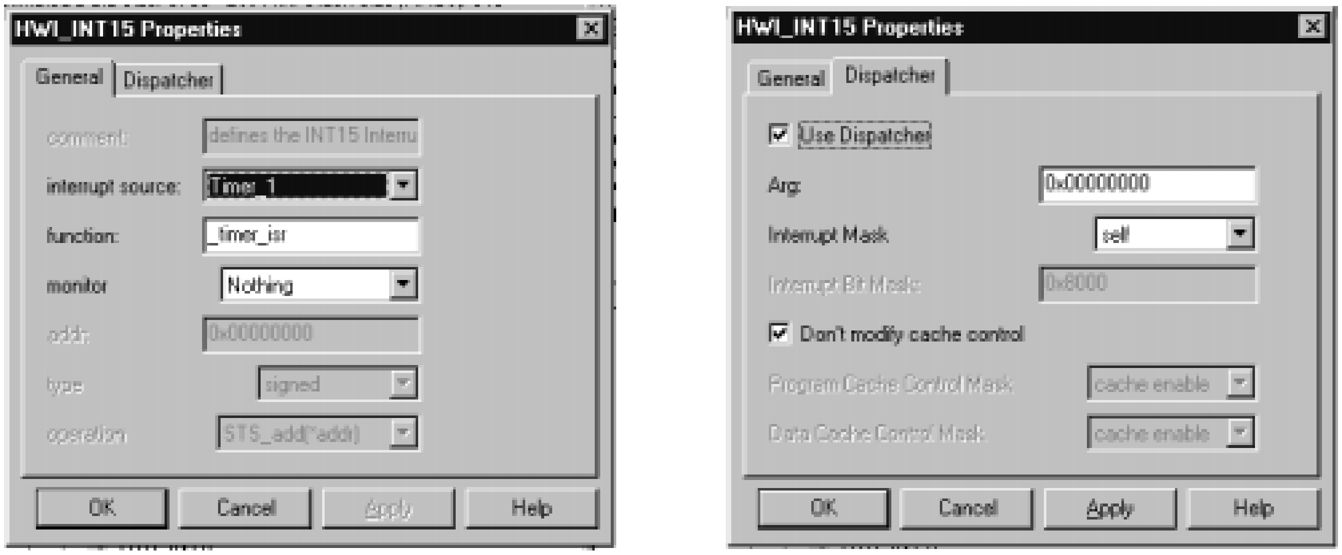
Handle : hTimer1

[✓] Enable Pre-Initialization

Pre-Initialize with: TIMER_NOTHING

Figure 4. Timer Device Properties

At this point, you have finished configuring timer 1 on the chip. To generate periodic hardware interrupts, you need to tie this interrupt to one of the hardware interrupt manager (HWI) objects. Open the properties window of HWI_INT15. Set the “interrupt source” to Timer 1 and the timer ISR “function” to *timer_isr ()*. Note the use of a leading underscore, as this interrupt service routine (ISR) is written as a C function. The leading underscore is required because the C compiler places a leading underscore to all C symbols during compilation. In the Dispatcher tab of the HWI_INT15 properties, select the “Use Dispatcher” option so that DSP/BIOS will take care of the context save operation. See Figure 5.



HWI_INT15 Properties – General

comment: defines the INT15 Interru
interrupt source: Timer_1
function: _timer_isr
monitor: Nothing
addr: 0x00000000
type: signed
operation: STS_add(*addr)

HWI_INT15 Properties – Dispatcher

[✓] Use Dispatcher
Arg: 0x00000000
Interrupt Mask: self
Interrupt Bit Mask: 0x8000
[✓] Don't modify cache control
Program Cache Control Mask: cache enable
Data Cache Control Mask: cache enable

Figure 5. HWI_INT15 Properties

In the global settings of the configuration file, select CSL as “6711”, and define the symbol CHIP_6711 in the project build options. In the file hello.c, define the ISR function as shown below. This ISR will be invoked every 0x2000 ticks of the timer clock.

```
Void timer_isr (Void)
{
      LOG_printf (&trace, “In a timer 1 ISR”);
}
```

In the same file, include the following CSL header files related to the CSL and timer.

```
#include <csl.h>
#include <csl_timer.h>
```

Declare the global variable as shown below. This variable is used to store the timer event ID.

```
static Uint32 TimerEventId1;
```

Now, inside the function *main ()*, add the following lines of code. The code fragment below will get the timer event ID of timer 1, enable the particular timer event, and then start the timer:

```
TimerEventId1 = TIMER_getEventId (hTimer1);
IRQ_enable (TimerEventId1);
TIMER_start (hTimer1);
```

Save and build the hello2 project. Load and run the executable. You can see that *timer_isr ()* is invoked periodically. The interval at which the timer 1 interrupt occurs is described below:

TMS320C6711 CPU clock period = 1/150 Mhz = 6.67 ns

Timer 1 input clock = 1/150 Mhz) * 4 = 26.67 ns

Timer 1 period = 0x2000 = 8192

Time period for timer 1 interrupt = 8192 * (1/150 Mhz) * 4 = 218.45 us

Now the hello2 project has a timer1 ISR that is invoked every 218.45us. So, the log output would be like the one shown below. The complete source code listing of hello.c is given in Appendix A.

```
0    In a timer 1 ISR
```

Configuring the on-chip timer to generate periodic hardware interrupts on a TMS320C5402 DSK is very similar to that on a TMS320C6711 DSK. The only difference is that you can specify the timer divide-down ratio (TDDR). In the example provided with this application report, the TDDR value is set to zero. So, the timer will be driven by the DSP clock at 100Mhz. The PRD register of the timer is set to 0x55f0, which is equal to 22000 DSP clock ticks. So, the timer interrupt will occur every 22000 ticks of the DSP clock, which is equal to 220us. In the C5000 example, you tie the timer interrupt to HWI_SINT7. You need to use the dispatcher for HWI_SINT7.

3 DSP/BIOS System Clock

3.1 System Clock

DSP/BIOS need a heartbeat because many of its APIs have a time-out parameter. This heartbeat is called the system clock. The system clock tick is different from the DSP clock tick. Apart from the system clock, DSP/BIOS provides a high- and low-resolution time. These clocks are used to measure passage of time, to generate time-stamp messages, and serve as the default heartbeats for driving the execution of periodic function.

DSPs typically have more than one on-chip timer that generates hardware interrupts at periodic intervals. DSP/BIOS normally uses one of these available on-chip timers as a source for its on-chip system clock. In the default configuration, the system clock has the same value as the low-resolution time. There is no requirement that the system clock needs to be driven by an on-chip timer. An external clock, or an ISR triggered by an on-chip peripheral, can drive the system instead of the timer. The functioning of the system clock is explained in detail in section 3.3. The pre-configured CLK object, PRD_clock, can be removed from the DSP/BIOS configuration. To do this, un-check the “Use CLK Manager to drive PRD” option in the PRD objects “General Properties” window.

If an external clock is used, it should call PRD_tick () to advance the system clock. The system clock can also be triggered by a periodic interrupt from an on-chip peripheral. In this case the interrupt’s hardware ISR needs to call PRD_tick ().

3.2 High- and Low-Resolution Times

The on-chip timer has two registers called timer period register (PRD) and timer counter register (CNT). The high- and low-resolution times are dependent on these two registers. The timer CNT register increments by one every four CPU clock ticks in the case of TMS320C6711. The frequency at which the timer CNT increment depends on the DSP generation. See Table 2.

Table 2. Timer Clock for DSPs

Feature	TMS320C620x/ C670x	TMS320C621x/ C671x	TMS320C64x	TMS320C54x/C55x
Timer input clock frequency	CPU rate / 4	CPU rate / 4	CPU rate / 8	CPU rate / (TDDR + 1)

When the value in the timer CNT register equals the value in the timer PRD register, the timer CNT register is reset to zero, and a timer interrupt is generated. DSP/BIOS increments the kernel variable **CLK_R_time**, whenever a timer interrupt occurs. The low-resolution time can be obtained by calling the API CLK_getltime (). It returns the number of timer interrupts that had occurred until the point of time when the API was called. The high-resolution time gives a more precise value by multiplying the number of timer interrupts held in CLK_R_time with the timer period, and adding the value of the timer CNT register to it. This gives a time with resolution very close to one instruction cycle. Potential timer counter rollover during the process of register reads will appropriately compensated before returning the high-resolution timer value to the user. The low-resolution time is used to add time stamps to event logs when the events happen over a long period of time. The high-resolution time is used in conjunction with STS_set () and STS_delta () APIs to benchmark code. It can also be used for adding time stamps to event logs. When the timer period is set to 0xFFFF on the C5000 and 0xFFFFFFFF on the C6000 devices, an optimized version of CLK_gethtime and CLK_getltime is used by DSP/BIOS:

CLK_R_time = Number of timer interrupts

Low-resolution time = CLK_R_time

High-resolution time = (CLK_R_time * Timer PRD) + Timer CNT

3.3 Clock Functions and Operation

The CLK module provides four APIs: CLK_gethtime (), CLK_getltime (), CLK_countspms (), and CLK_getprd (). The first two have already been discussed in the previous section. The API CLK_countspms () returns the programmed number of hardware timer register ticks per millisecond, while the CLK_getprd () returns the configured timer PRD register value. The hardware interrupt 14 is tied to the timer 0 interrupt by default. Also by default, the DSP/BIOS system clock is tied to timer 0.

When a timer interrupt occurs, the ISR corresponding to HWI_INT14 is run. CLK_F_isr () is the ISR that is invoked when the timer interrupt happens. CLK_F_isr () does some basic interrupt service operations, increments CLK_R_time (low-resolution clock), transfers control to clock hook functions that eventually return to the context where the interrupt occurred. The hook functions are wrapped with prescribed (HWI_enter, HWI_exit) hardware interrupt prolog and epilog. The usually used hook function is the CLK_F_run () function, which basically calls a function FXN_F_run (). The function FXN_F_run () calls all the configured clock functions in sequence. So, when a timer interrupt occurs, all the clock functions are executed in the context of the hardware ISR. Therefore, the amount of processing done by any CLK function should be minimal, and these functions may invoke only the DSP/BIOS APIs that are allowable from an ISR. The default PRD_clock () is explained in section 4.2. Figure 6 explains the sequence of operations that takes place in the event of a timer interrupt. The box “CLK functions” denotes all the clock functions configured. See Figure 6 for the CLK function execution sequence.

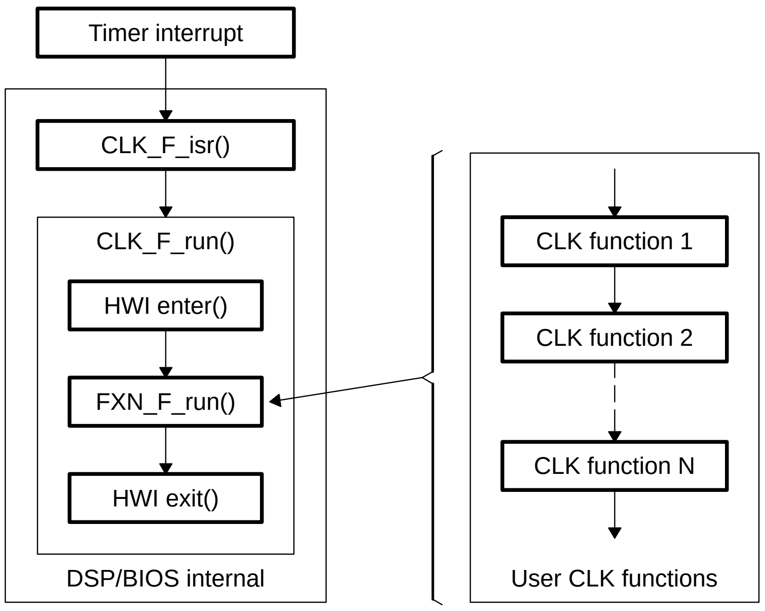


Figure 6. Clock Function Sequence

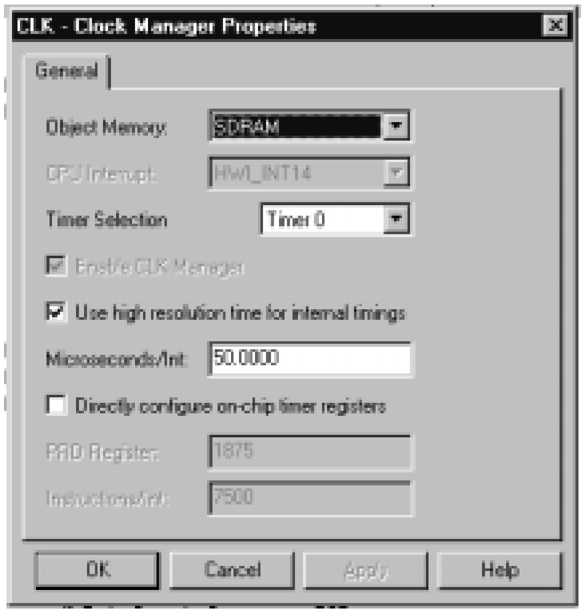
3.4 Configuring and Understanding a CLK Function

In this section, you will configure a clock function that will be executed every 50us. Open the configuration file hello.cdb. Right-click on **CLK – Clock Manager**, and select properties. In the properties window, the CPU interrupt is set to HWI_INT14 by default. Also, the clock is set to be triggered by on-chip timer 0. In the microsecond/Int field, enter the value 50. This value specifies the time period at which the timer interrupt should occur. Setting the microseconds/Int field sets the PRD register with an appropriate value. Alternatively, you can set the timer period register directly by selecting the Directly Configure on-chip timer register option. Apply the setting, and now the timer 0 is configured to interrupt every 50us. This means that the DSP/BIOS system clock will tick every 50us and, therefore, all the configured CLK functions will be executed every 50us. When you set the microseconds/Int field to 50us, the PRD register field and Instruction per interrupt field are set automatically. The DSP speed in Mhz serves as the basis reference for this computation. These values are calculated as follows:

Microseconds/Int	$= 50\text{us} = 50 * 10^{-6}$
Time for 1 DSP clock tick	$= 1 / (150 * 10^{6}) = 10^{-6} / 150$
Time for 1 PRD register increment	$= 4 / (150 * 10^{6}) = (4 * 10^{-6}) / 150$

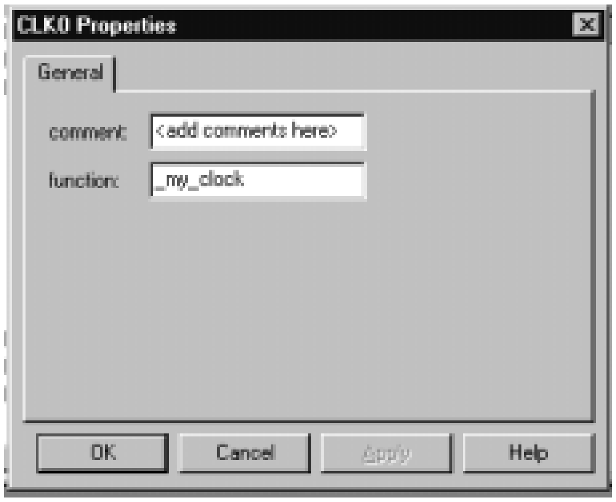
Number of PRD increments in 50us $= (50 * 10^{-6} * 150) / (4 * 10^{-6}) = 1875 = 7500$ DSP clocks

Again, right-click on the **CLK – Clock Manager** and select Insert CLK. After inserting a CLK object, you may rename the object if you wish to. Now right click the newly created CLK object, CLK0, and tie up a clock function, *my_clock ()*, to this CLK object. A leading underscore should be added to the function name if it is written in C. Apply the settings and, at this point, you have finished configuring a clock function. See Figure 7.



CLK - Clock Manager Properties

Object Memory: SDRAM
CPU Interrupt: HWI_INT14
Timer Selection: Timer 0
[✓] Enable CLK Manager
[✓] Use high resolution time for internal timings
Microseconds/Int: 50.0000
[ ] Directly configure on-chip timer registers
PRD Register: 1875
Instructions/Int: 7500



CLK0 Properties

comment: <add comments here>
function: _my_clock

Figure 7. CLK Manager Properties

Now, in the file hello.c, define the function *my_clock ()* as follows:

```
Void my_clock (Void)

{

LOG_printf (&trace, ”In clock function my_clock ()”);

}
```

Save and build the hello2 project. Load the executable. Run the program, and you can see that the function, *my_clock ()*, is called repeatedly from the LOG window transcript.

4 DSP/BIOS PRDs and Periodic Functions

4.1 The PRD Module

The main purpose of the PRD module is to manage PRD objects that represent individual program functions that execute periodically, based on events that are triggered by certain sources. These periodic functions can be either time-based or space-based. Time-based functions are triggered by timer interrupts. This means that the PRD module uses the DSP/BIOS clock. In this scenario, the option “Use CLK Manager to drive PRD” in the **PRD – Periodic Function Manager** is enabled. Enabling this option introduces a new CLK object PRD_clock. Space-based functions are the ones that are triggered by events such as input/output (I/O) availability or other programmatic events. Activation of the PRD module is driven by regular calls to the kernel function PRD_F_tick ().

The PRD module may be driven by the system clock or by calls to PRD_F_tick () from specific events. The same period counter, PRD_D_tick, drives all the PRD objects configured. Each object can execute its functions at different intervals, based on the period counter. The difference between PRD and CLK functions is that CLK functions get executed at the frequency of DSP/BIOS system clock ticks, while PRD allows the functions to be executed repeatedly at any multiple of system clock ticks. Another difference is that the CLK functions are executed in the context of a hardware interrupt, while PRD functions are executed in the context of a software interrupt. DSP/BIOS allows the user to drive the task manager (TSK) module with PRD by enabling the check box in the Task Manager properties.

4.2 PRD Functions and Operation

The period of a function is the time between successive invocations. It is expressed in terms of ticks, where a single tick is defined as a single invocation of the PRD_F_tick () function. So, if the period for a function ‘f’ is set to be ‘n’, then PRD module invokes the function ‘f’ once every ‘n’ calls to the PRD_F_tick () function. The DSP/BIOS API PRD_tick () internally calls PRD_F_tick () to increment the variable PRD_D_ tick. The PRD module executes in the context of a special software interrupt object automatically created by the DSP/BIOS configuration tool. This software interrupt is implicitly posted through calls to PRD_F_tick (). The configuration tool automatically inserts a PRD_swi object when a PRD object is created. It is the PRD_swi that invokes the configured periodic functions. When there are no periodic functions configured, the PRD_swi is automatically removed.

The two main functions in the PRD module are PRD_F_swi () and PRD_F_tick (). PRD_F_swi () is the function that is bound to the software interrupt posted by the PRD_F_tick () function. The configuration tool uses PRD_swi as the name for the software interrupt that binds and calls PRD_F_swi (). When the target program is loaded, PRD_swi is statically stored in the PRD_D_swihandle variable. The PRD module uses a software interrupt (swi) object (called PRD_swi by default), which itself is triggered on a periodic basis to manage execution of PRD objects. The PRD_swi is posted only if the PRD_D_tick is equal to the greatest common divisor that is a power of two of the periods of all the configured PRD functions. You can get a better performance of PRDs if the tick value of the PRD functions is a multiple of 2. For example, if there are two PRD functions that have period values of 1024 and 1025, the PRD_swi will be posted every cycle. If the tick values are 1024 and 512, the PRD_swi will be posted only once every 512 ticks. Thus, in the latter case, the overhead of running the PRD_swi for every tick is reduced.

PRD_Obj has a “count” field in it. PRD_F_swi () decrements the value in the count field of all “enabled” PRD objects. If the count field value of any PRD object reaches 0 or a value less than 0, the function bound to the PRD object is executed, and the PRD object’s count field is reloaded with the value in its “period” field, if it is a continuous PRD. PRD_F_tick () is a function that increments the PRD timer count, PRD_D_timer. See Figure 8 and Figure 9.

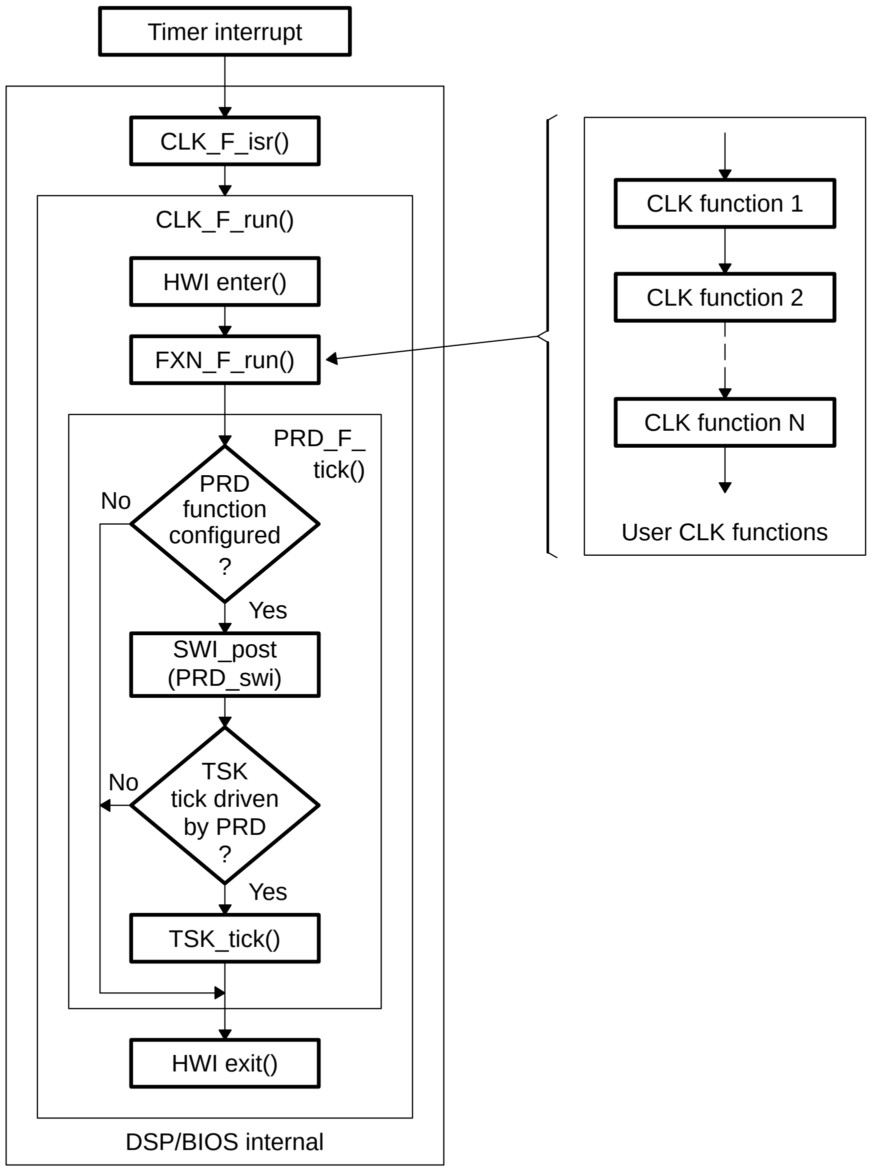


Figure 8. PRD Function Sequence (a)

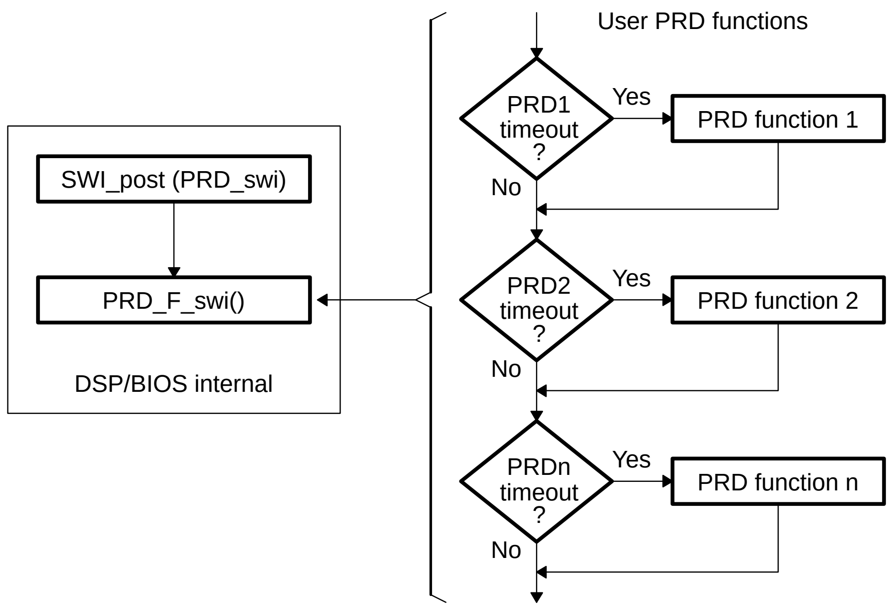


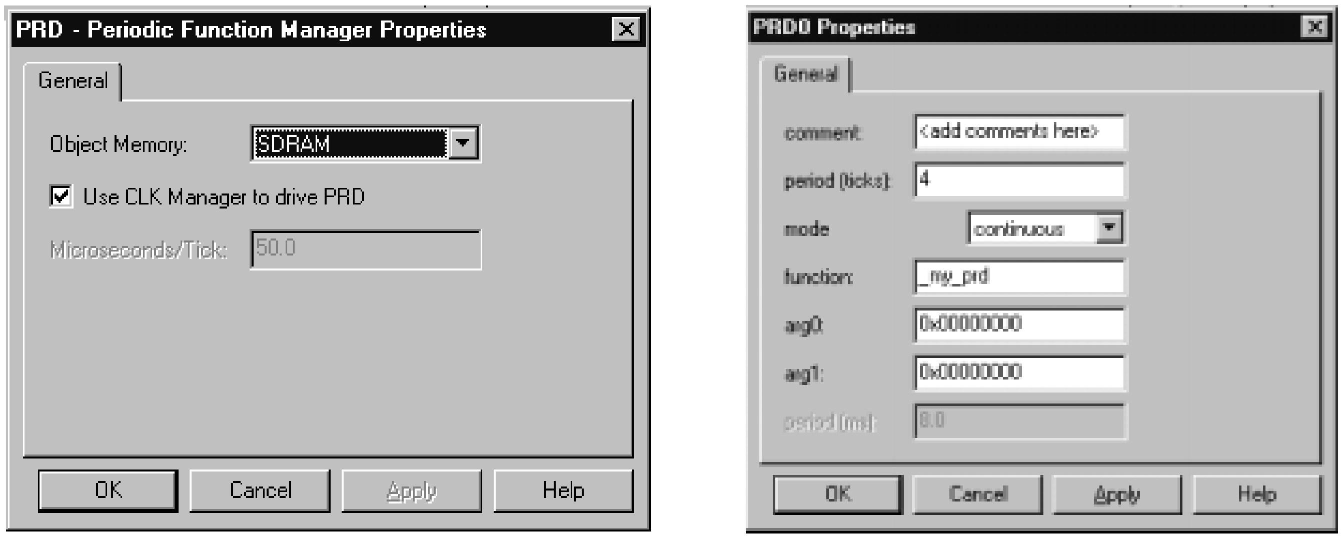
Figure 9. PRD Function Sequence (b)

When the PRD module is driven by the DSP/BIOS system clock, the configuration tool inserts a CLK object called PRD_clock. The function PRD_F_tick () is tied to this CLK object. This is done to provide an interface between the DSP/BIOS system clock and the PRD module. The operation of the PRD module is very clear from Figure 8 and Figure 9. If the Task Manager is configured to be driven by the PRD module, PRD_F_tick () will call a function TSK_tick (). TSK_tick () increments the system alarm clock which serves the time out needs of the TSK module.

4.3 Configuring and Understanding a Periodic Function

In this section, you will configure a periodic function that will be executed every 4 system clock ticks. Open the configuration file for the hello2 project. Right-click on the PRD – Periodic Function Manager, and select properties. In the properties windows, the “Use CLK Manager to drive PRD” option is enabled, and the “Microseconds/tick” is set to 50, which is equal to the time between successive timer interrupts.

Now, right-click on PRD – Periodic Function Manager, and insert a PRD object. You may rename the newly created object, if you wish to. Right-click on the newly created object, PRD0, and open the properties window of PRD0. In the PRD0 properties window, set the period ticks to 4, select continuous mode, and set the periodic function as *my_prd ()*. Remember to put a leading underscore before the function, if the function is written in C. Apply the settings. The configuration tool allows you to enter two arguments to the PRD function. See Figure 10.


PRD - Periodic Function Manager Properties

General

Object Memory: SDRAM

[✓] Use CLK Manager to drive PRD

Microseconds/Tick: 50.0

PRD0 Properties

General

comment: <add comments here>

period (ticks): 4

mode: continuous

function: _my_prd

arg0: 0x00000000

arg1: 0x00000000

period (ms): 8.0

Figure 10. PRD Manager Properties

In the file hello.c, define the function *my_prd ()* as follows:

```
Void my_prd (Void)
{
      LOG_printf (&trace, “In periodic function my_prd ()”);
}
```

The periodic function, *my_prd ()*, will be executed every 4 ticks of the BIOS system clock. In section 3.4, you have configured the BIOS clock to be driven by timer 0, which generates timer interrupts at intervals of 50us. The calculation shown below is self-descriptive.

Time for one timer interrupt = Time for one BIOS clock tick = 50us

Time for one PRD_F_tick () = 50us

Period for the PRD function my_prd () = 4 PRD ticks

Time interval at which my_prd () is called = 4 * 50us = 200us

The function my_prd () is invoked every 200us.

From the calculations above, you can conclude that the periodic function, *my_prd (),* is invoked every 200us, which is equal to the time for four timer 0 interrupts to occur. In the log output, you can see that the output from clock function, *my_clock ()*, is printed 4 times, and then the output of the function *my_prd ()* is printed. This keeps repeating.

5 Benchmarking Tips

5.1 Why Tips on Benchmarking

DSP/BIOS kernel performance benchmarks, including a comparison of the instrumented versus non-instrumented kernel's performances are provided in the application report *DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP* (SPRA662). DSP/BIOS offers a set of APIs to benchmark and profile the user code. The APIs provided by the statistics object manager (STS) and CLK modules can be used to profile code fragments. There are certain circumstances under which these APIs may return inconsistent or incorrect values. This section explains the various circumstances under which the instrumentation APIs may give incorrect values. You must take care of these conditions while benchmarking or profiling your code.

5.2 Timer-Free Runs

In an application where the timer is being used as a periodic interrupt or to drive the DSP/BIOS system clock, the benchmark and profile results may differ from time to time and from DSP to DSP. The reason is that the timer may be running while the CPU is halted. On the C620x/C670x the timer is halted when the CPU is halted, if the timer clock source is set to internal CPU clock. In such a case, the timer counter is enabled to count only when the CPU is not stalled by emulation driver halt. In a scenario like this, the profile values may be reasonable. In the case of the C621x/C671x and TMS320C64x™, the timer continues counting as programmed, even when the CPU is stalled due to emulation driver halt. In this case the profile/benchmark values obtained will not be any use if the execution is halted in between. An example of timer-free runs giving incorrect profile is as follows. Suppose the user puts a break point at the beginning and end of the code fragment being profiled. The programs runs and hits the first breakpoint. At this point when the program is halted, a timer interrupt may occur, and when you run the program again, the timer ISR code is inserted between the code fragments being profiled. Due to this, the profile values obtained will be the sum of the time taken by the code fragment, and the ISR for execution.

5.3 Benefits of Instrumentation APIs Over Profiler

Real-time analysis is the analysis of data acquired during real-time operation of a system. The purpose of real-time analysis is to determine if a system is operating within the design constraints. Cyclic debugging is not effective for a real-time system, due to non-deterministic execution and stringent time constraints. DSP/BIOS provides a set of instrumentation APIs to complement cyclic debugging. Using these APIs are better than the profiler because they have fixed and short execution times. Since the overhead time is fixed, the time taken by these APIs are known, and overhead time can be factored out of the measurements. *DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP* (SPRA662) gives the benchmarks for these instrumentation APIs. The CPU load increases only marginally when all the implicit instrumentation is enabled.

TMS320C64x is a trademark of Texas Instruments.

The DSP/BIOS instrumentation APIS have advantages over the Code Composer Studio profiler. The profiler collects program analysis data during the execution of the program and may not be desirable for measuring the performance of real-time systems. This is where the real power of instrumentation APIs comes into picture. Instrumentation APIs communicate between the target and the host in a background idle (IDL) thread that has the lowest priority. The IDL thread executes only when the application has nothing else to do. This makes sure that the real-time behavior of the application is preserved. The instrumentation data is always formatted on the host. This relieves the overhead of instrumentation data processing on the target. DSP/BIOS provides a variety of instrumentation APIs that give more reliable performance data over the profiler for a real-time system.

5.4 Limitations of CLK_gethtime/CLK_getltime

The APIs, CLK_gethtime () and CLK_getltime (), are used extensively for benchmarking and profiling. These APIs may return incorrect values if the interrupts are disabled for a long period of time. CLK_gethtime () provides the high-resolution time in the system. The timer peripheral has a period register and a counter register. Whenever the timer count register hits 0, a timer interrupt occurs. The low-resolution time is equal to the number of timer interrupts that occurred until the point of time the API was called. The high-resolution time is calculated by multiplying the number of timer interrupts to the timer period register, and adding the timer count register.

If the code being profiled takes more time than the time period between consecutive timer interrupts, some of the timer interrupts may be missed, if the interrupts are masked inside the code being profiled. Missing timer interrupts are equivalent to corrupting the low-resolution time. When the low-resolution time is corrupted, it automatically corrupts the high-resolution time because the number of timer interrupts recorded is wrong. To overcome this, if the code being profiled is expected to take a huge amount of cycles, then make sure that the timer interrupt rate in the configuration file is decreased, to avoid missed interrupts. This may not be a good solution if the application uses timer interrupt to trigger other events in the system. This essentially means that STS objects cannot be used to profile a code, if it disables interrupts for a duration longer than the timer interrupt rate.

A different scenario where the CLK_gethtime () and CLK_getltime () may return incorrect values is the wraparound of the values. The low- and high-resolution time are represented by a 32-bit value, and may reach their maximum value of $2^{32} – 1$. These time values wrap around to zero after the maximum value of $2^{32} – 1$. So, if the code being profiled takes more time than the time required for the low or high–resolution time to reach $2^{32} – 1$, wraparound occurs, and time values returned are no longer correct. You can get a better time resolution by setting the timer period to 0xFFFF on the C5000, and 0xFFFFFFFF on the C6000 devices. Setting these values in the timer-period register will link an optimized version of CLK_gethtime() and CLK_getltime() to be used by DSP/BIOS.

5.5 Timer ISR Overhead

The timer ISR has an overhead on the system. When a timer ISR is serviced, it calls the CLK_F_isr () function, which executes within the context of a hardware interrupt. After doing some basic operations, it executes all the clock functions. Once the clock functions have been executed, the control is transferred via a SWI to the PRD module. In the PRD module, the system checks if any of the PRD function counters have hit zero, and executes the functions for the ones that have hit zero. So, in effect, the amount of processing done in the event of a timer interrupt is dependent on the number of clock functions configured. The time required for servicing the timer interrupt increases, as the number of clock function increases. So, it is always advisable that you may optimize your system so that only the required CLK functions are there in the system. Unwanted clock functions will do nothing other than put an extra overhead on the timer ISR.

6 References

1. *DSP/BIOS II Timing Benchmarks on the TMS320C6000 DSP* (SPRA662).
2. *TMS320C6000 DSP/BIOS User's Guide* (SPRU303).
3. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide* (SPRU403).
4. *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals* (SPRU131).
5. *TMS320C55x DSP Peripherals Reference Guide* (SPRU317).

Appendix A Hello.c listing

```
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 */
/* ”@(#) DSP/BIOS 4.60.22 12–07–01 (barracuda–j15)” */
/***************************************************************************/
/*                                                                         */
/*     H E L L O . C                                                       */
/*                                                                         */
/*     Basic LOG event operation from main.                                */
/*                                                                         */
/***************************************************************************/

#include <std.h>
#include <log.h>
#include ”hellocfg.h”

/* Included as part of timer 1 configuration */
#include <csl.h>
#include <csl_timer.h>
#include <csl_irq.h>

/* Global Declarations – part of timer 1 configuration */
static Uint32 TimerEventId1;

/* Declarations of CLK, PRD functions and timer 1 isr */
Void my_clock(Void);
Void my_prd(Void);
Void timer_isr(Void);

/*
 *  ======== main ========
 */
Void main()
{
  LOG_printf(&trace, ”hello world!”);
  /* Obtain the event IDs for the timer devices */
               TimerEventId1 = TIMER_getEventId(hTimer1);

               /* Enable the timer events */
               IRQ_enable(TimerEventId1);

               /* Start the timers */
               TIMER_start(hTimer1);

  /* fall into DSP/BIOS idle loop */
  return;
}

/* Definition of clock function for CLK0 object */
Void my_clock(Void)
{
               LOG_printf(&trace, ”In clock function my_clock()”);
}

/* Definition of periodic function for PRD0 object */
Void my_prd(Void)
{
               LOG_printf(&trace, ”In periodic function my_prd()”);
}

/* Definition of timer 1 ISR */
Void timer_isr(Void)
{
               LOG_printf(&trace, ”In a timer 1 ISR”);
}
```