Lab 4 – Introduction to Using SYS/BIOS Real-Time Operating System on TI TMS320F28027 Piccolo Microcontroller

NOTE: THIS DOC IS WRITTEN ASSUMING A 28027 LAUNCHPAD IS BEING USED

NOTE: THIS DOC IS WRITTEN ASSUMING CODE COMPOSER STUDIO VERSION 5 IS BEING USED

Code Composer Studio versions 6 and 7 do not easily support the "Raw Log" functions that appear in the code in this lab, therefore you need to install CCSv5 to do this lab

In this lab you will:

- experiment with code that uses SYS/BIOS RTOS
- explore the graphical configuration environment for setting up a project that uses SYS/BIOS
- explore C code structure for the various thread-types
- use existing code templates – there is no need to use your files from previous labs
    1. experiment with "Idle Example" code
    2. experiment with "Swi Example" code
    3. experiment with "Task Example" code
    4. modify the Task Example code by adding a new task and additional semaphore
- observe print messages in Raw Logs window

What to submit to D2L (use .zip file):

- answers to any questions posed in this handout or on the whiteboard
- any snapshot(s) you feel would help make your lab report better
- code you write or modify for Item 4 above, i.e., the .c and .cfg files
- snapshot(s) of information on the new task you add for Item 4
- snapshot of the Raw Logs display for Item 4

Some Notes on Tsk Threads in This Lab

- In this lab, Tsk threads are created statically, i.e., they are created by the RTOS once only at the time that the system starts up.

- In this lab, Tsk threads are always in one of three states: READY, BLOCKED, or RUNNING. They are never TERMINATED.

- In SYS/BIOS, the state of a semaphore is defined as follows:

  = 0 → not available (blocking)
  = 1 → available

Some Other Notes

- In order to use SYS/BIOS in a project, you must pick a SYS/BIOS template to start with to get the appropriate Include files for SYS/BIOS.

- The xds100 Emulator on the Piccolo is inexpensive, but a bit slow. It can take some time for the Build and Load processes to be performed. Be patient.

- When you want to re-Build your code, I recommend that you explicitly **Terminate** the Debug session first, otherwise I noticed the Raw Logs might not update properly thereafter.

0. Install Code Composer Studio v5 (including SYS/BIOS)

Copy the .exe install file from Share Out\ELEX\7820 to your desktop and run the installer.

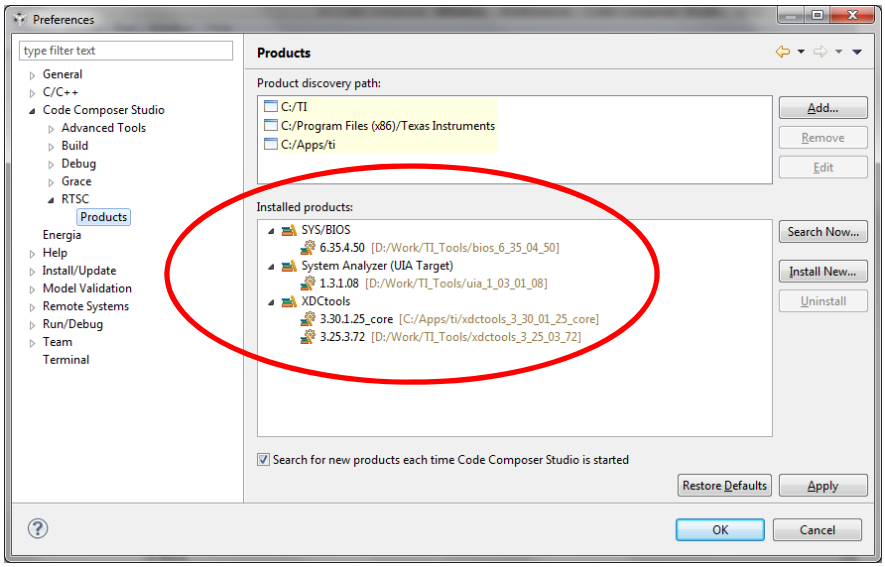
Select Custom install. You only need to select the c2000 tools and the XDS emulator packages.

The install takes about 15 min on the lab PC (and you may need to say yes to a firewall message part way through).

Note that, when you set up your CCS project later in this lab, it will need to use a pre 3.30 version of XDC tools.

You can check the packages that get installed in CCS by doing:

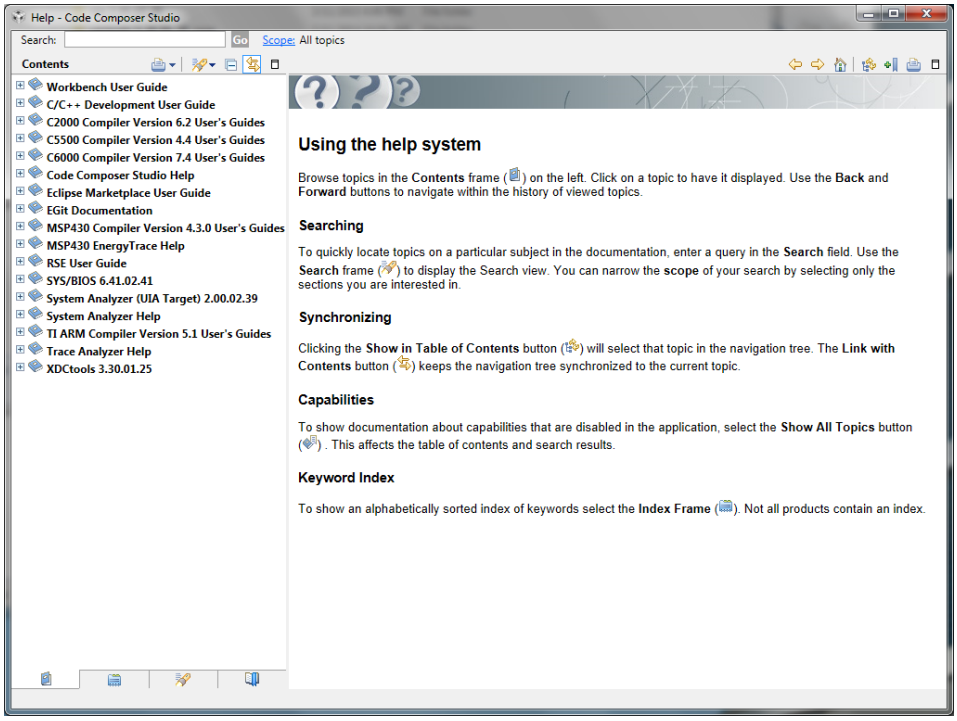
Window – Preferences – Code Composer Studio – RTSC – Products

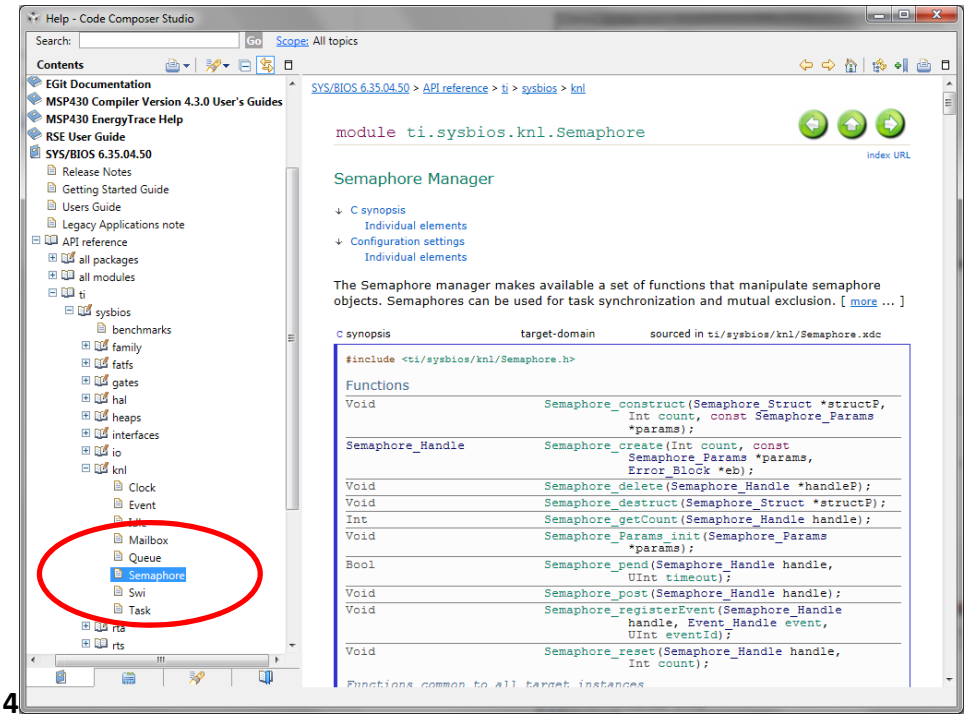


Note: pre 3.30 version of XDCtools required for this lab

- To see Help Info on SYS/BIOS, e.g. API functions:

In Code Composer: **Help – Help Contents**





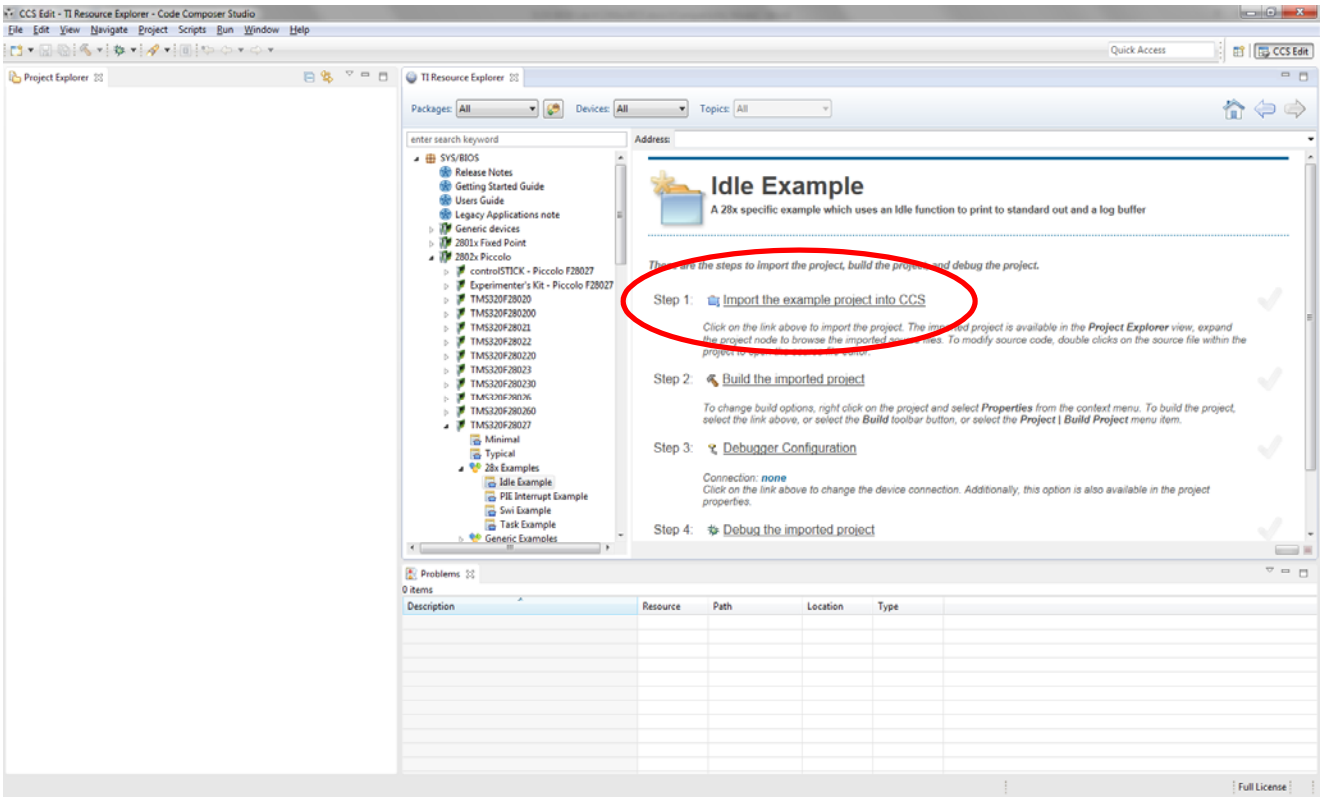
4

1. Create New Project Based on SYS/BIOS Template "Idle Example"

Create a new project called "Lab4Idle" according to the following steps:

View – Resource Explorer (Examples) or similar

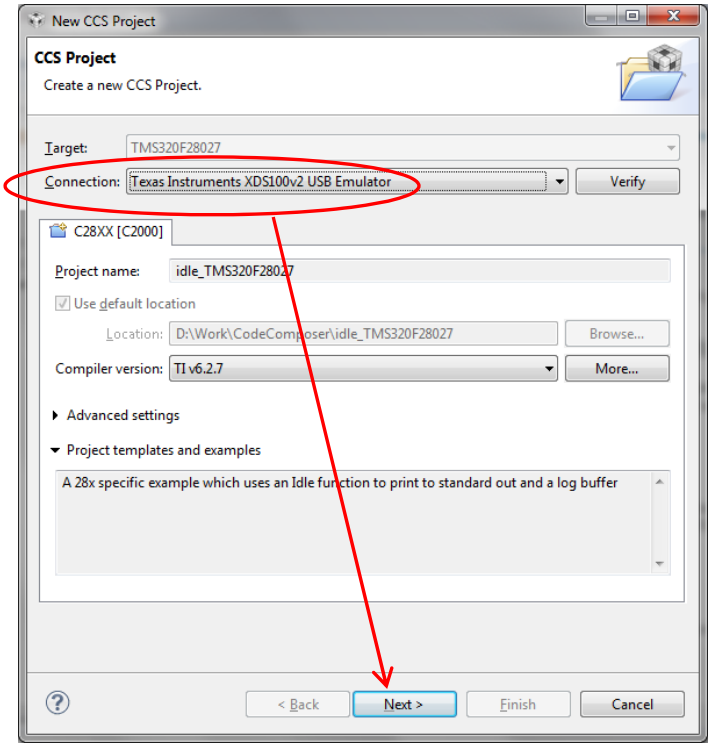
And navigate to the 28x Idle Example

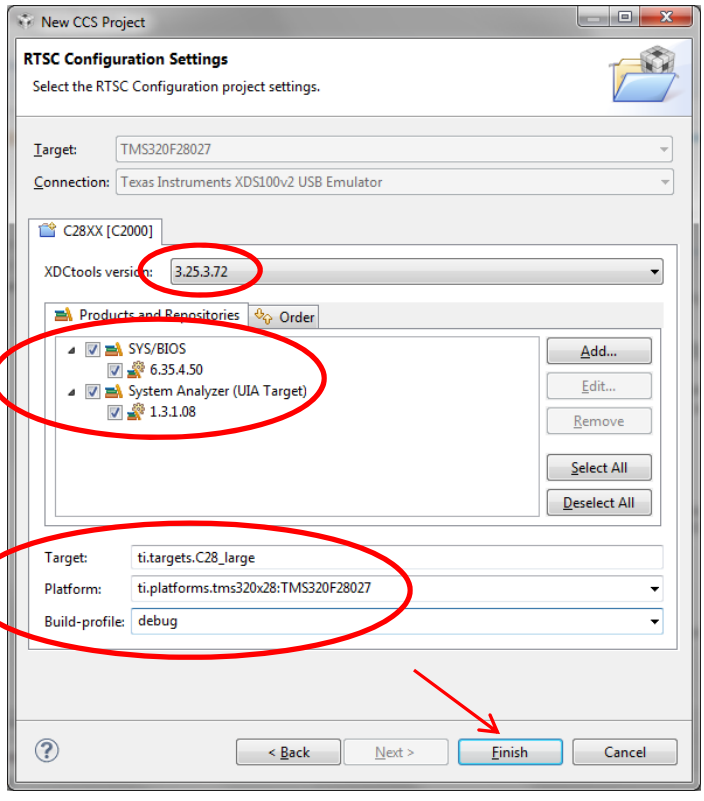


And **Import the example project into CCS.**

And **Build** it.

Set up the **Debugger Configuration** as necessary.





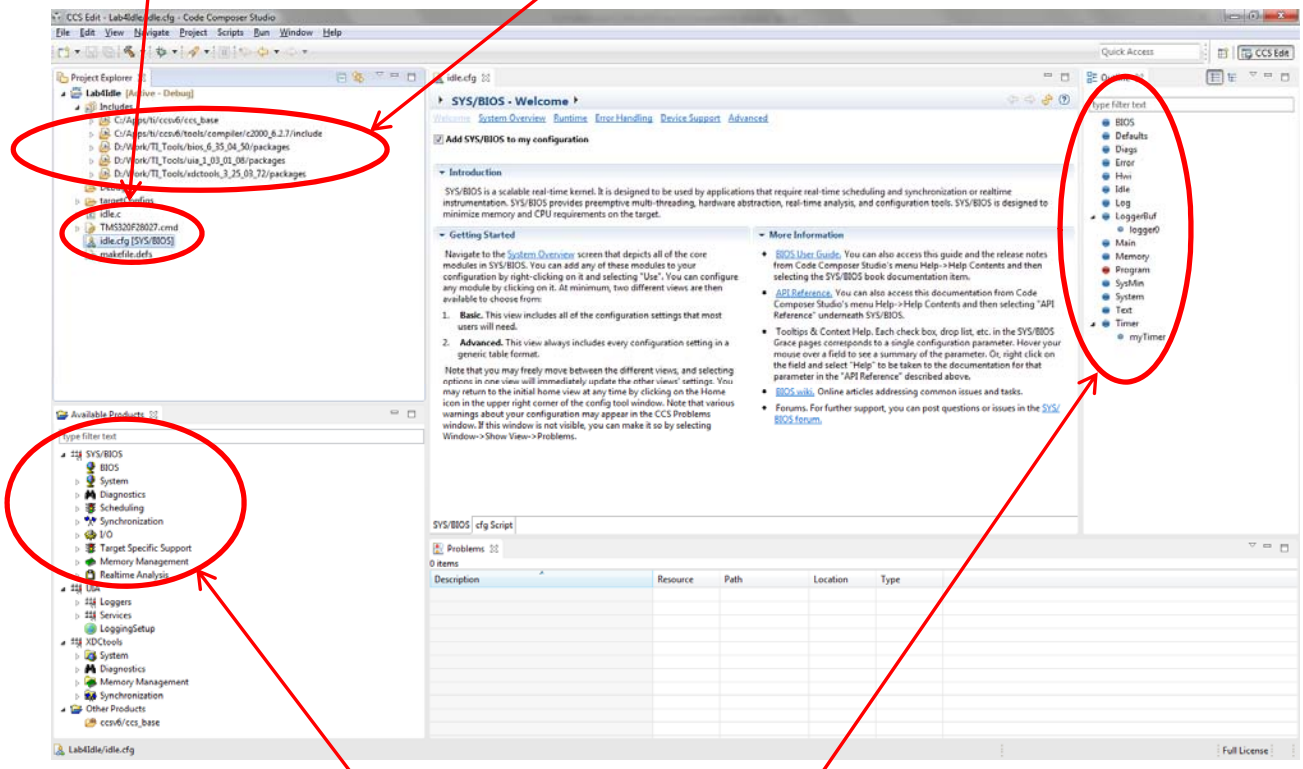
Rename the project from **idle_TMS320F28027** to **Lab4Idle**

.cfg File

.cfg file can be viewed via a graphical interface – this makes it easier to develop code that uses SYS/BIOS

Observe:

- packages to support SYS/BIOS and UIA have been installed
- your two files: idle.c and idle.cfg



Use this tree to select ("**use**") items (e.g. Hwi, Swi, Tsk) to add to your code

OR USE THE "SYSTEM OVERVIEW" GRAPHICAL ENVIRONMENT TO USE/INSTANTIATE NEW ITEMS

Use this tree to use ("**instantiate**") new items (e.g. Hwi, Swi, Tsk) into your code

Question 1A: How can you see the non-graphical, XDC-script-based version of the .cfg file?

After creating your project and observing the .cfg file…

1. Peruse the idle.c file

Question 1B: What headers are included at the top of the code and what are they used for?

`Log_info()` is a function that is much more efficient than **`printf()`** and can be used to print information to the Raw Log buffers.

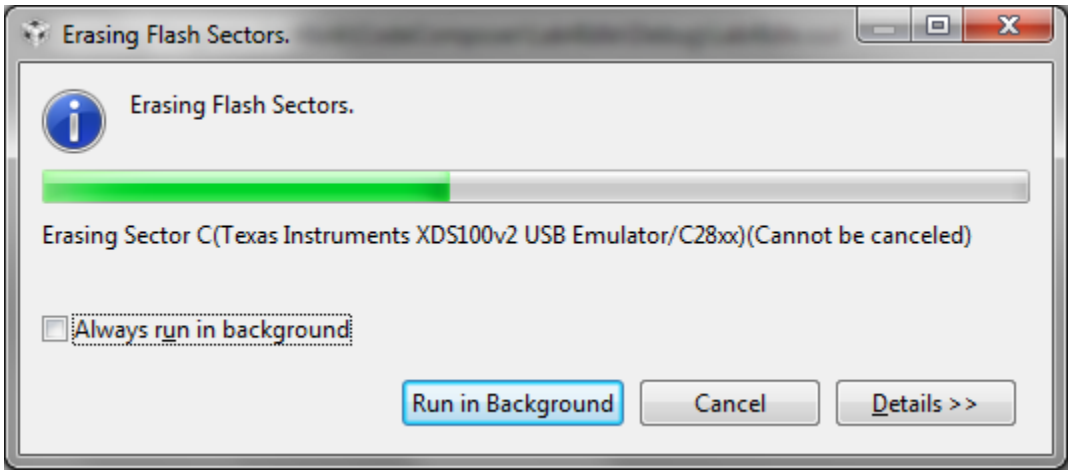
Observe that there is a fundamental difference between the **`main`** code from previous labs and the **`main`** code when SYS/BIOS is invoked:

Without SYS/BIOS, the main code stayed in an infinite loop.

With SYS/BIOS, the last thing the main code does is to branch to the RTOS, never to return unless the system is re-booted.

2. Launch the Debugger.

Observe that SYS/BIOS gets programmed into the Flash - this might take some time…



3. Open the Raw Logs window:

Tools – RTOS Analyzer – RTA (Legacy) – Raw Logs

or Tools – RTA – Raw Logs

then Resume (Run) the code for a few seconds and then Suspend (Halt) it, and observe the information in the **formattedMsg** column of Raw Logs.

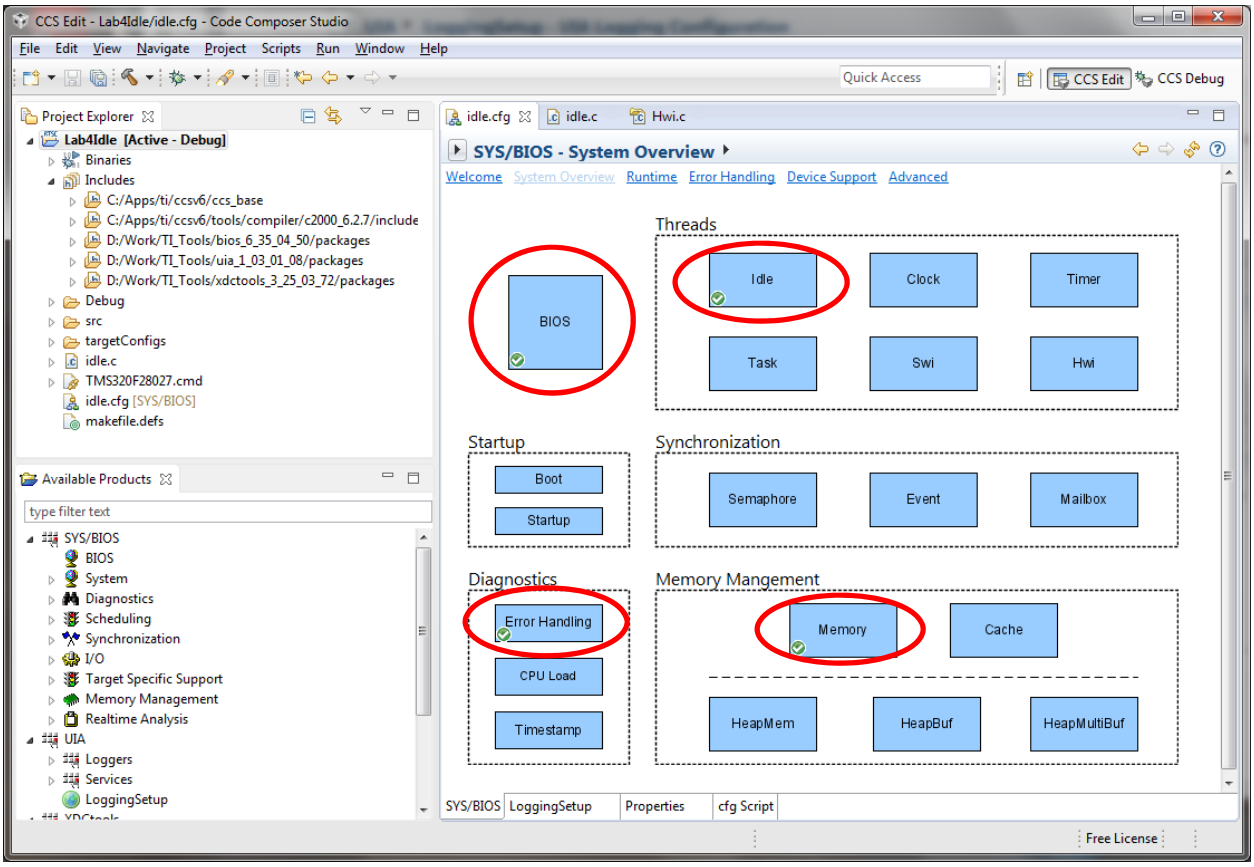
Question 1C: What does the* time *column in Raw Logs represent?

When you Suspend the code execution in the Debug perspective, notice that a window will open where the code has temporarily stopped. It might be in idle.c or Idle.c or other:

idle.c → your user code

Idle.c → SYS/BIOS kernel code

Observe the graphical representation of your system showing what parts of the RTOS are used in this application:



Question 1F: Navigate in the graphical environment to find where you can change the rate at which the timer interrupt occurs. Change it, re-build the project, and verify that your change worked.

2. Create New Project Based on SYS/BIOS Template "Swi Example"

Close your Idle project and create a new project called "Lab4Swi" based on the 'Swi Example" template.

Peruse the swi.c file. Observe that the Swi function starts and ends (i.e., it does not have an infinite loop cf. Tsk function).

(Remember to rename the project.)

Build the code.

Set up the Debugger Configuration as necessary.

Launch the Debugger.

Run the code for a while and then observe the Raw Logs buffer.

Question 2A: What* two *functions are in the Idle thread?

Question 2B: Draw a processor timeline showing
the* main*,* idle*,* myTickFxn*, and* mySwi *threads.
(for simplicity, you do not need to show the divide-by-10)

3. Create New Project Based on SYS/BIOS Template "Task Example"

Close your Swi project and create a new project called "Lab4Task" based on the 'Task Example" template.

Peruse the task.c file. Observe that the Task function has an infinite **`for`** or **`while`** loop.

(Remember to rename the project.)

Build the code.

Set up the Debugger Configuration as necessary.

Launch the Debugger.

Run the code for a while and then observe the Raw Logs buffer.

Recall from a previous lecture that each task in a multi-tasking system has a Task Control Block (TCB) associated with it. The TCB contains information like the task priority, task state, and other info. (In SYS/BIOS, it is not called TCB.)

Question 3A: In the SYS/BIOS kernel code, find the structure definition equivalent to the Task Control Block and describe what each member represents.

4. Modify the Task Example Project

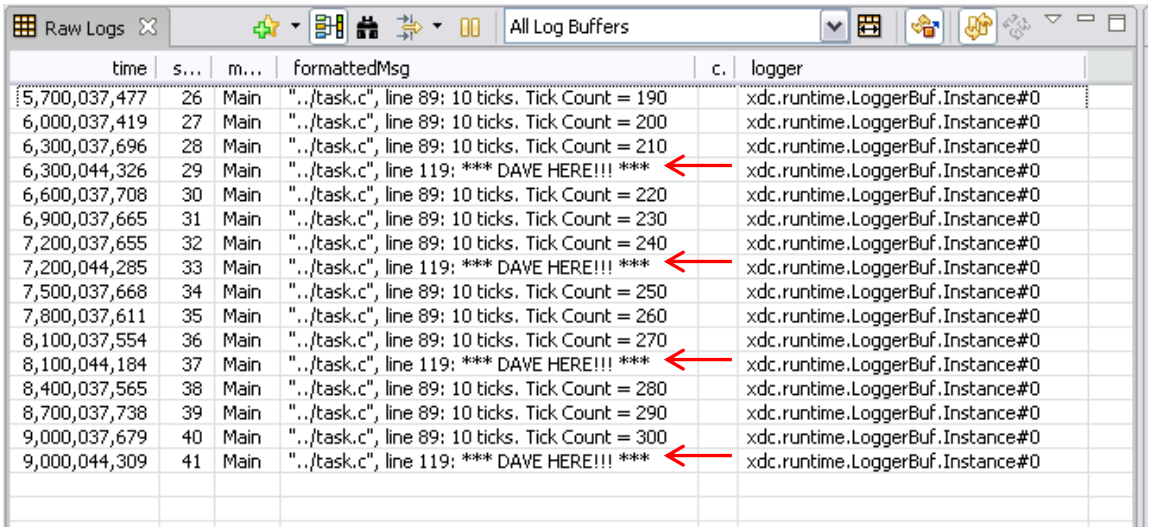
Close your Task project and create a new project called “Lab4Taskmod” based on the ‘Task Example” template.

In the following, replace “Dave” with your first name.

1. Graphically modify the task.cfg file by adding an additional task instance called “newTask” that will correspond to the function “DaveTaskFxn” that you will add to your code. Set the Priority of the new task to 2.

2. Graphically modify the task.cfg file by adding an additional binary semaphore called “DaveSem”.

3. Modify the task.c file by adding code for the function “DaveTaskFxn” and modifying myTaskFxn. The task is to print to the Raw Logs buffer the message shown below. DaveTaskFxn is to pend the semaphore DaveSem. The semaphore DaveSem is to be posted from the function “myTaskFxn” once every three times that myTaskFxn logs its info.



time	s...	m...	formattedMsg	c.	logger
5,700,037,477	26	Main	"../task.c", line 89: 10 ticks. Tick Count = 190		xdc.runtime.LoggerBuf.Instance#0
6,000,037,419	27	Main	"../task.c", line 89: 10 ticks. Tick Count = 200		xdc.runtime.LoggerBuf.Instance#0
6,300,037,696	28	Main	"../task.c", line 89: 10 ticks. Tick Count = 210		xdc.runtime.LoggerBuf.Instance#0
6,300,044,326	29	Main	"../task.c", line 119: *** DAVE HERE!!! ***		xdc.runtime.LoggerBuf.Instance#0
6,600,037,708	30	Main	"../task.c", line 89: 10 ticks. Tick Count = 220		xdc.runtime.LoggerBuf.Instance#0
6,900,037,665	31	Main	"../task.c", line 89: 10 ticks. Tick Count = 230		xdc.runtime.LoggerBuf.Instance#0
7,200,037,655	32	Main	"../task.c", line 89: 10 ticks. Tick Count = 240		xdc.runtime.LoggerBuf.Instance#0
7,200,044,285	33	Main	"../task.c", line 119: *** DAVE HERE!!! ***		xdc.runtime.LoggerBuf.Instance#0
7,500,037,668	34	Main	"../task.c", line 89: 10 ticks. Tick Count = 250		xdc.runtime.LoggerBuf.Instance#0
7,800,037,611	35	Main	"../task.c", line 89: 10 ticks. Tick Count = 260		xdc.runtime.LoggerBuf.Instance#0
8,100,037,554	36	Main	"../task.c", line 89: 10 ticks. Tick Count = 270		xdc.runtime.LoggerBuf.Instance#0
8,100,044,184	37	Main	"../task.c", line 119: *** DAVE HERE!!! ***		xdc.runtime.LoggerBuf.Instance#0
8,400,037,565	38	Main	"../task.c", line 89: 10 ticks. Tick Count = 280		xdc.runtime.LoggerBuf.Instance#0
8,700,037,738	39	Main	"../task.c", line 89: 10 ticks. Tick Count = 290		xdc.runtime.LoggerBuf.Instance#0
9,000,037,679	40	Main	"../task.c", line 89: 10 ticks. Tick Count = 300		xdc.runtime.LoggerBuf.Instance#0
9,000,044,309	41	Main	"../task.c", line 119: *** DAVE HERE!!! ***		xdc.runtime.LoggerBuf.Instance#0