i.MX31 SOM-LV Windows Embedded CE 6.0
Lab 5: Performance Analysis
BSP Documentation

Logic // Products
Published: June 2009

Revision History

REV	EDITOR	DESCRIPTION	APPROVAL	DATE
A	MH	Initial release	JCA	06/09/09

Table of Contents

1 Introduction

1.1 Learning Objectives

- Learn to use the CeLog/Kernel Tracker and the Monte Carlo profiler
- Understand how threads and processes are scheduled in Windows CE 6.0
- Find Performance bottlenecks

1.2 Prerequisites

Knowledge of the vocabulary used in OS design, Visual Studio, and the Platform Builder Plug-in for Visual Studio

Knowledge of C and C++ languages

1.3 Lab Setup

To complete this lab, you must have:

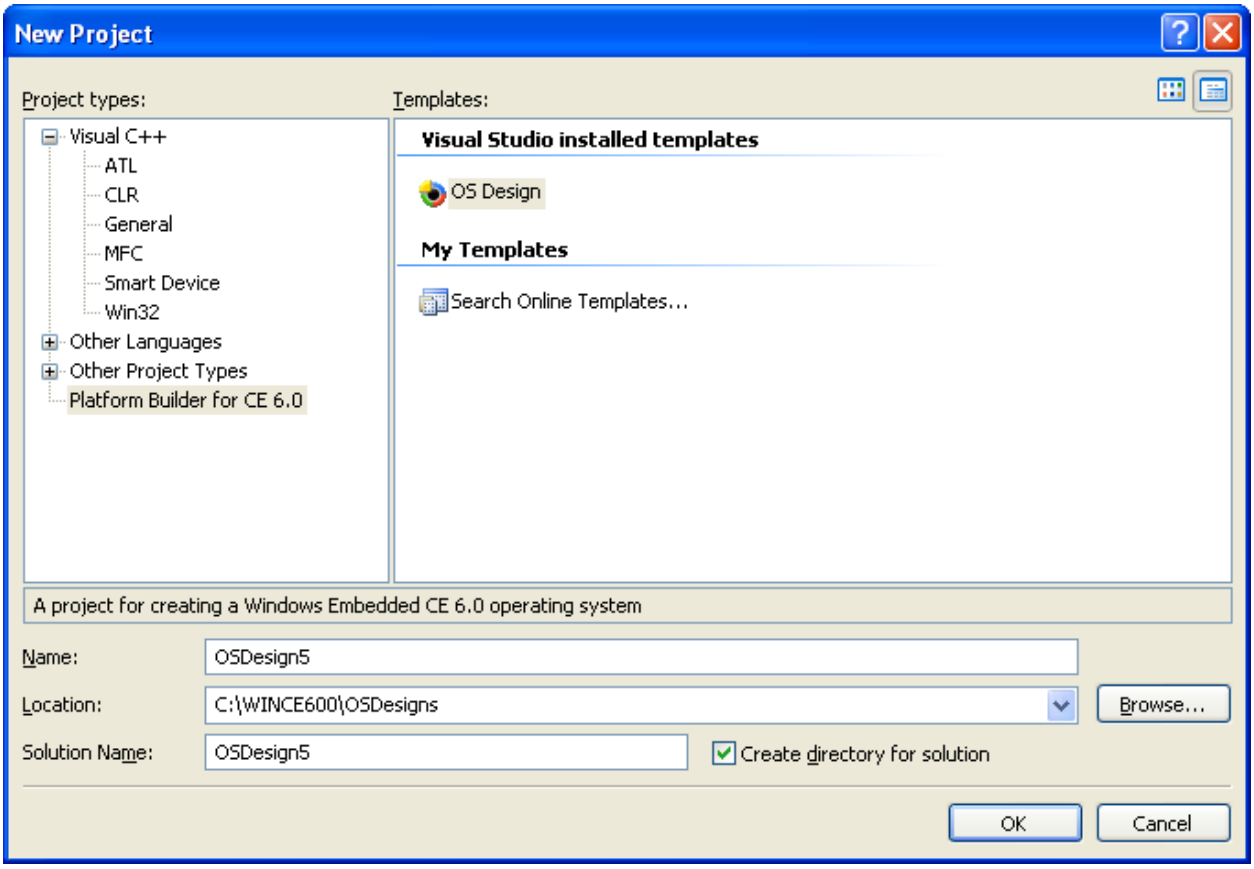
- A development workstation running Windows XP or Vista
- Visual Studio 2005 (Version 8)
- Service Pack 1 for Visual Studio 2005
- Service Pack 1 for Visual Studio 2005 on Vista (if you are running Vista)
- Windows CE 6.0 R2 Platform Builder plug-in with Service Pack 1
- A Zoom i.MX31 LITEKIT
- The i.MX31 SOM-LV Windows CE 6.0 BSP

2 Creating the Platform Image

2.1 Create a New OS Design

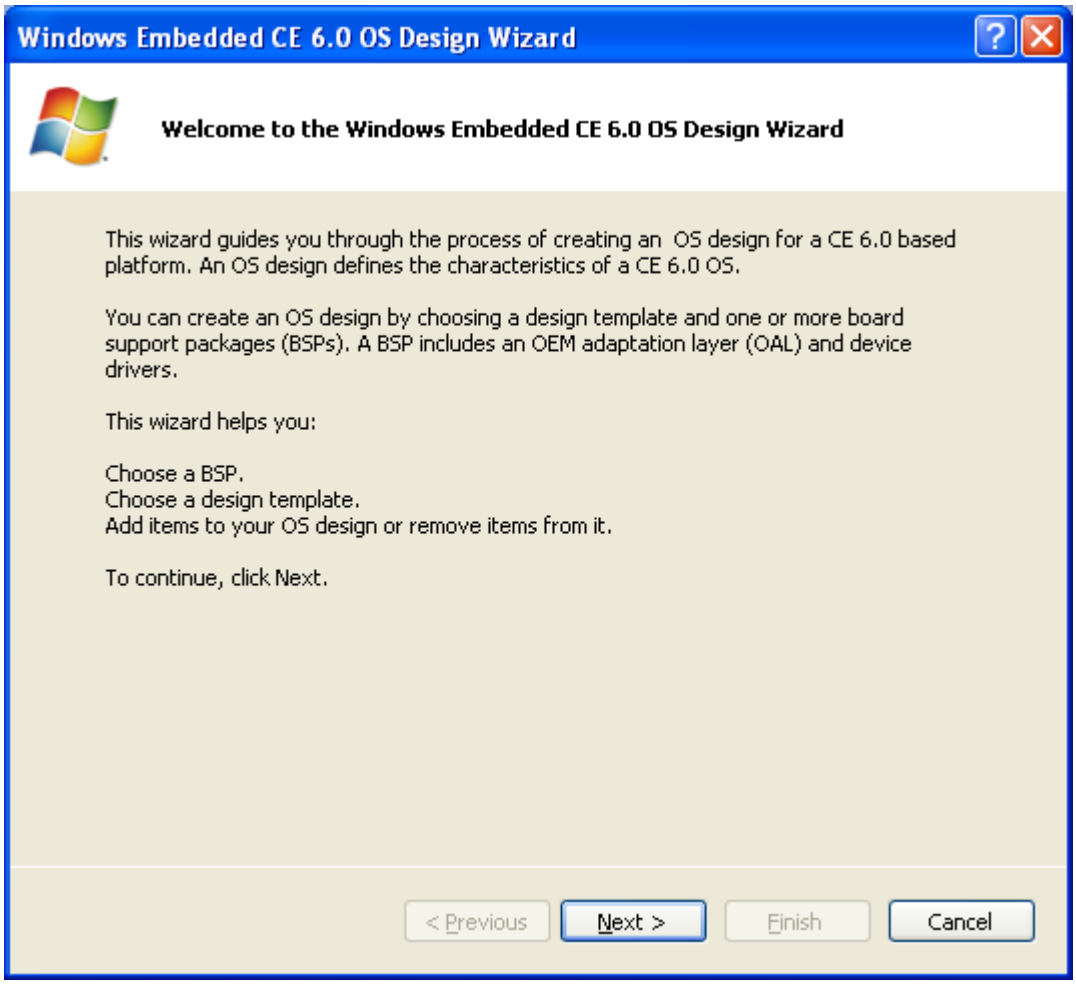
Launch the Visual Studio 2005 New Project dialog.

- Select **File** | **New…** | **Project…**



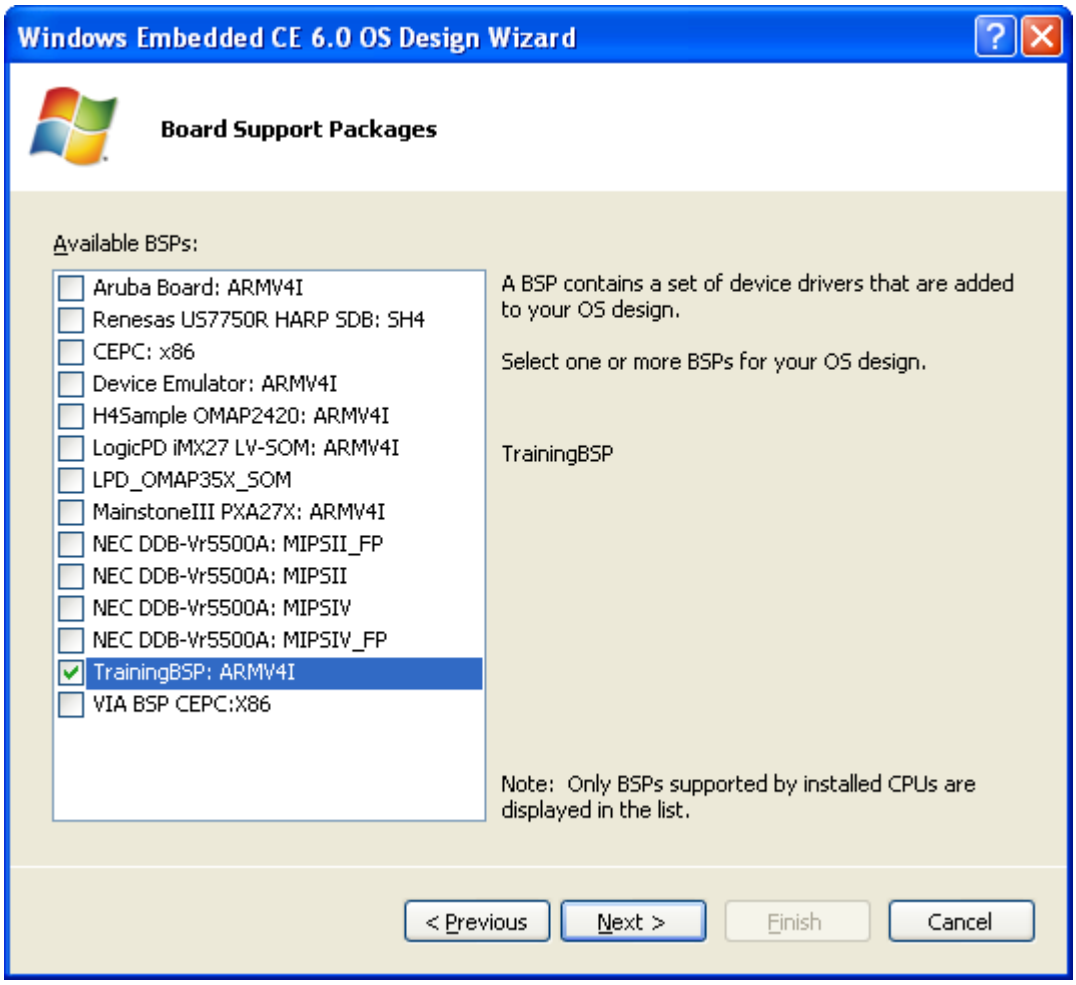
- Select the **Platform Builder for CE 6.0** project type.
- Choose the **OS Design** template
- Enter a name for your project, or go with the default name
- Click **OK**

The initial Dialog that is displayed below outlines the process of creating an OS Design. There are a number of steps involved and you will step through the wizard and make selections as appropriate.



We will build an operating system for the i.MX31 SOM-LV device.

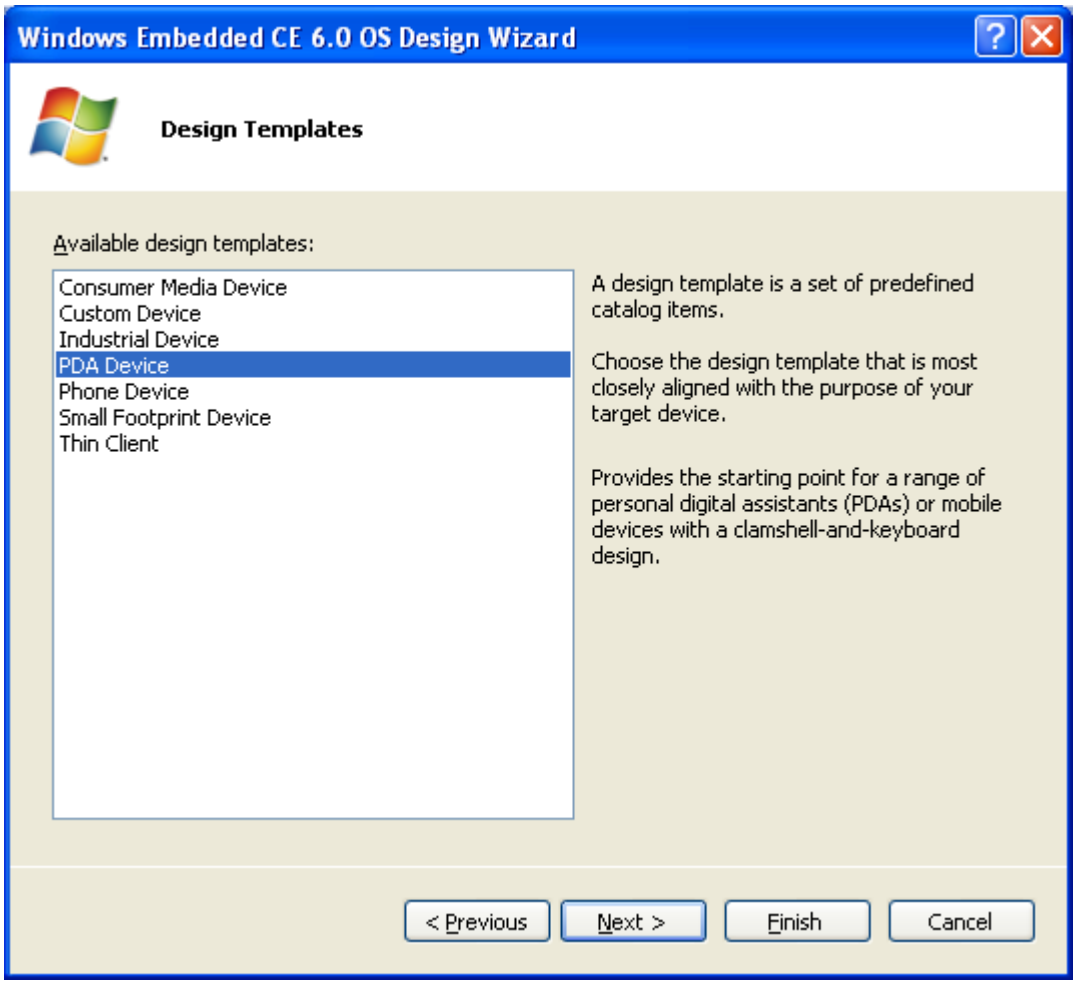
You could select more than one BSP (Board Support Package). For example, it may be useful to select the Device Emulator and a hardware reference platform; the emulation environment can be used by your internal/external application developers to build applications for your device while hardware is still being developed.



- Select **TrainingBSP: ARMV4I**
- Click **Next**

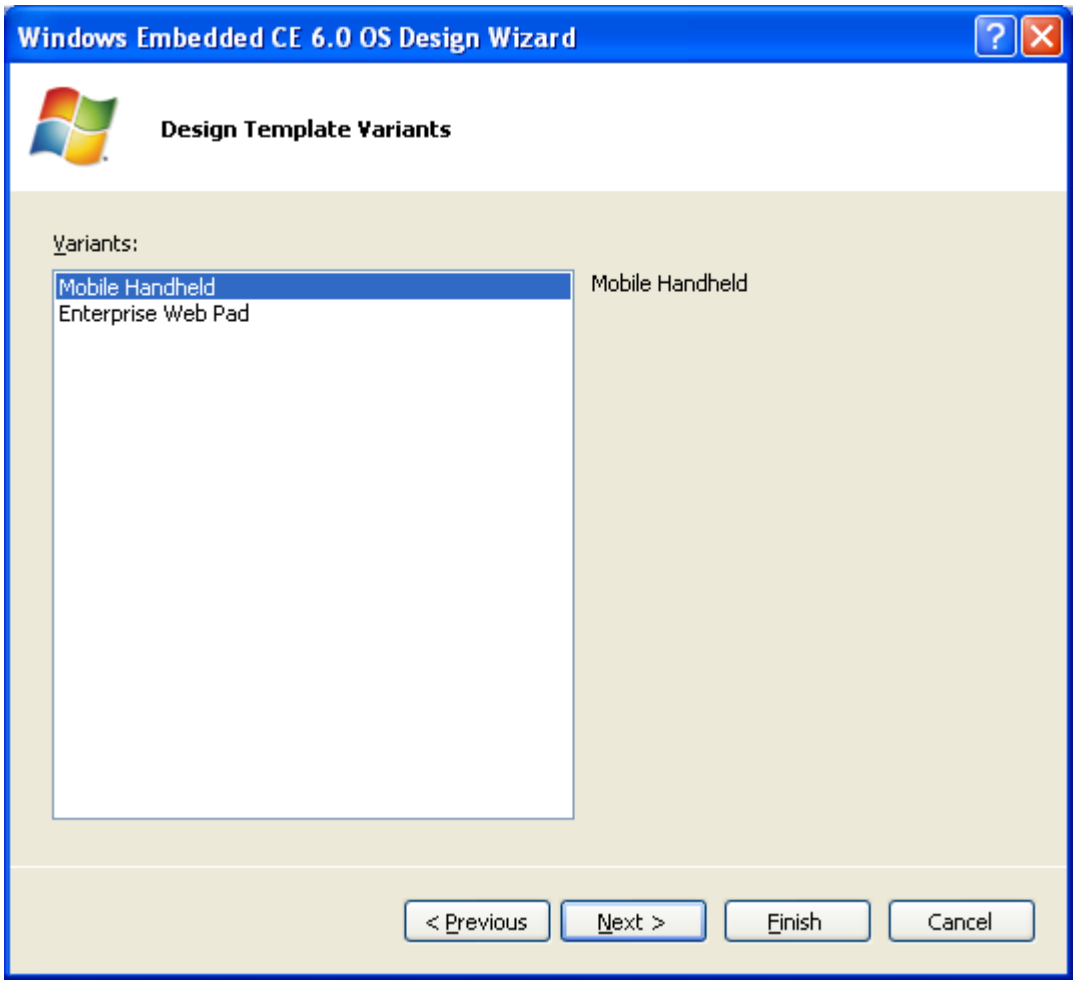
You are now given a number of Design Templates to choose from as the starting point for this project. If none of the options match your needs you can simply select **'Custom Device'** and build the image based on the components you select from the catalog.

For the purposes of this tutorial you will be using the **PDA Device** template.



- Select **PDA Device** from the list of Design Templates
- Click **Next**

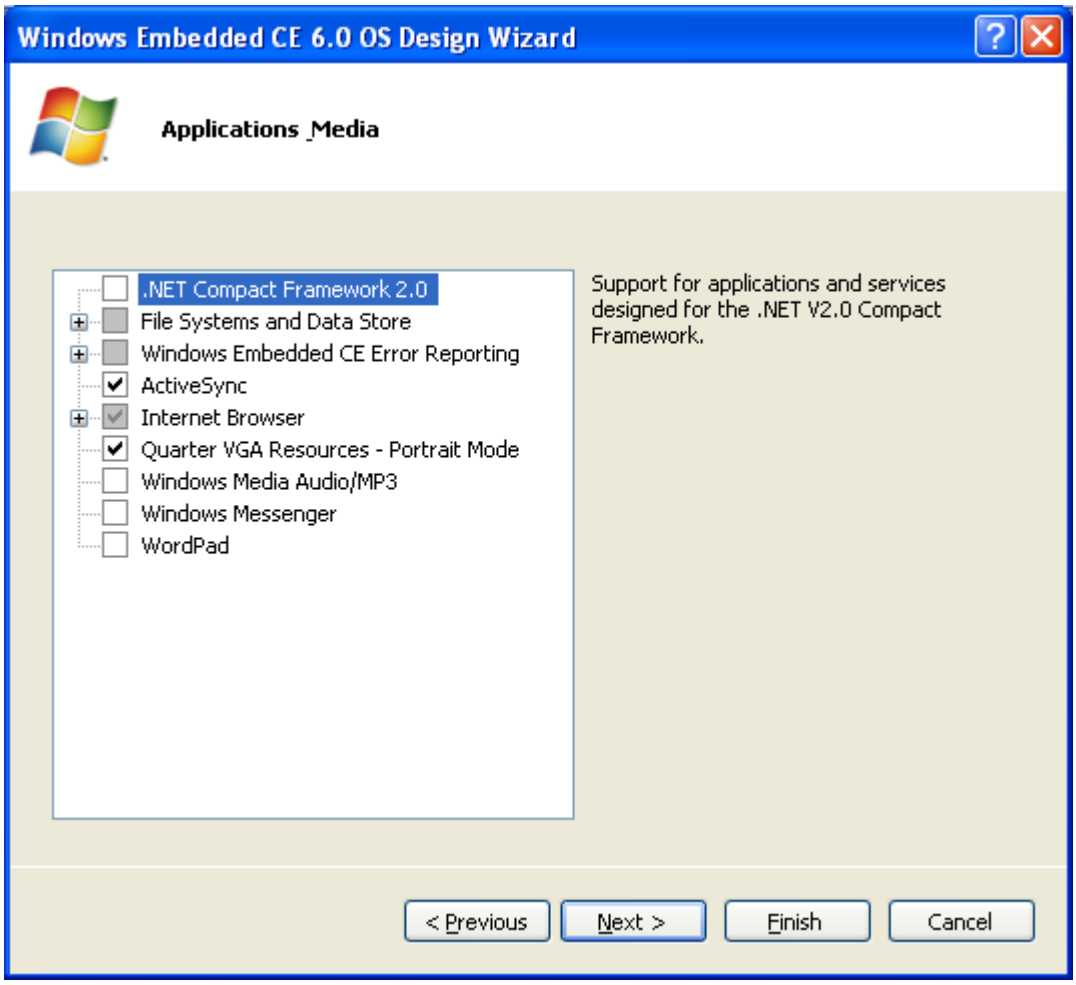
The Wizard now gives you the option of choosing a variant of the Design Template. For this tutorial you will be using the Mobile Handheld variant.



- Select **Mobile Handheld** from the list of Design Template Variants
- Click **Next**

There are a number of options which can be selected for each of the sample platforms, the OS Design Wizard will only show options that are relevant to the platform you are building. For example, it would not make sense to include Internet Explorer or WordPad in a headless device such as a Gateway. The Mobile Handheld device can include applications such as Internet Explorer, WordPad, and Windows Messenger.

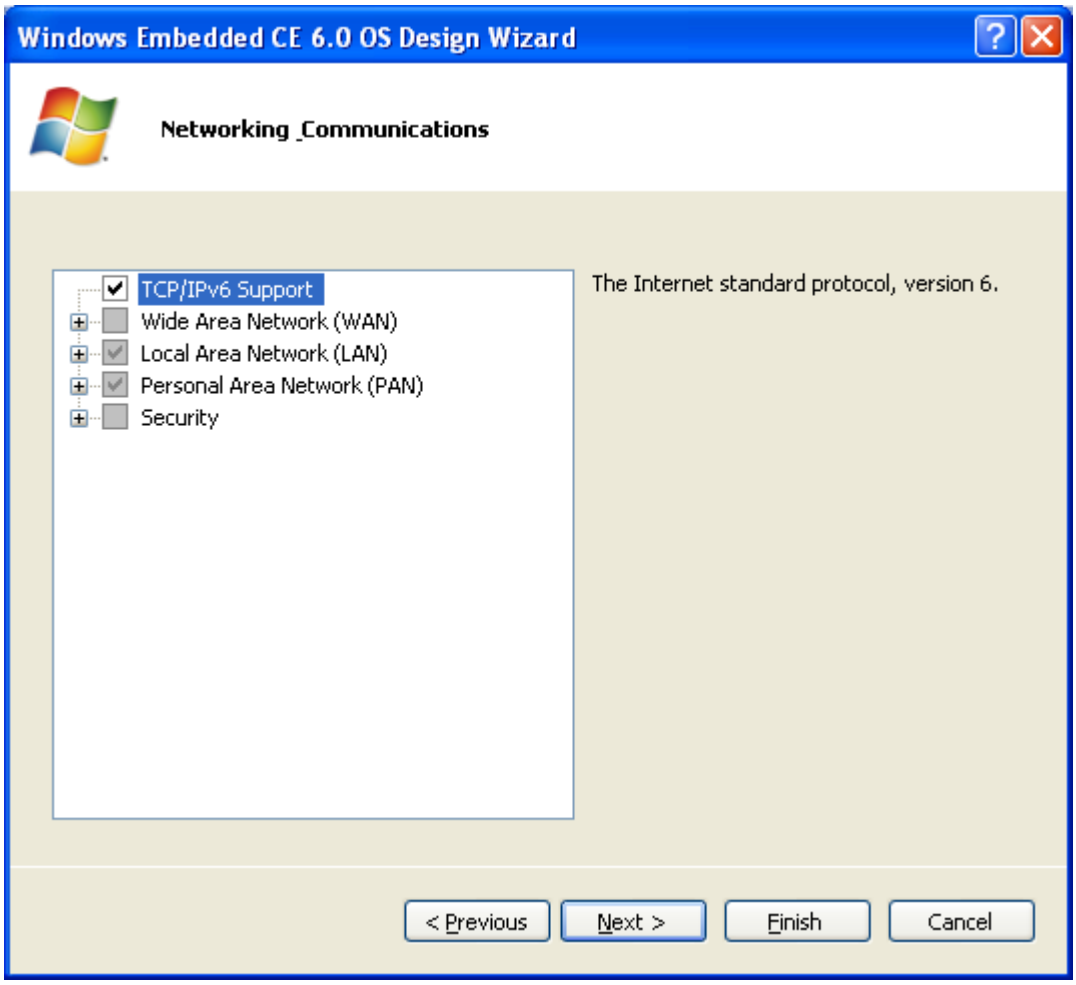
In this case we don't need the .NET Compact Framework. De-select this item.



- De-select the **.NET Compact Framework**
- Click **Next**

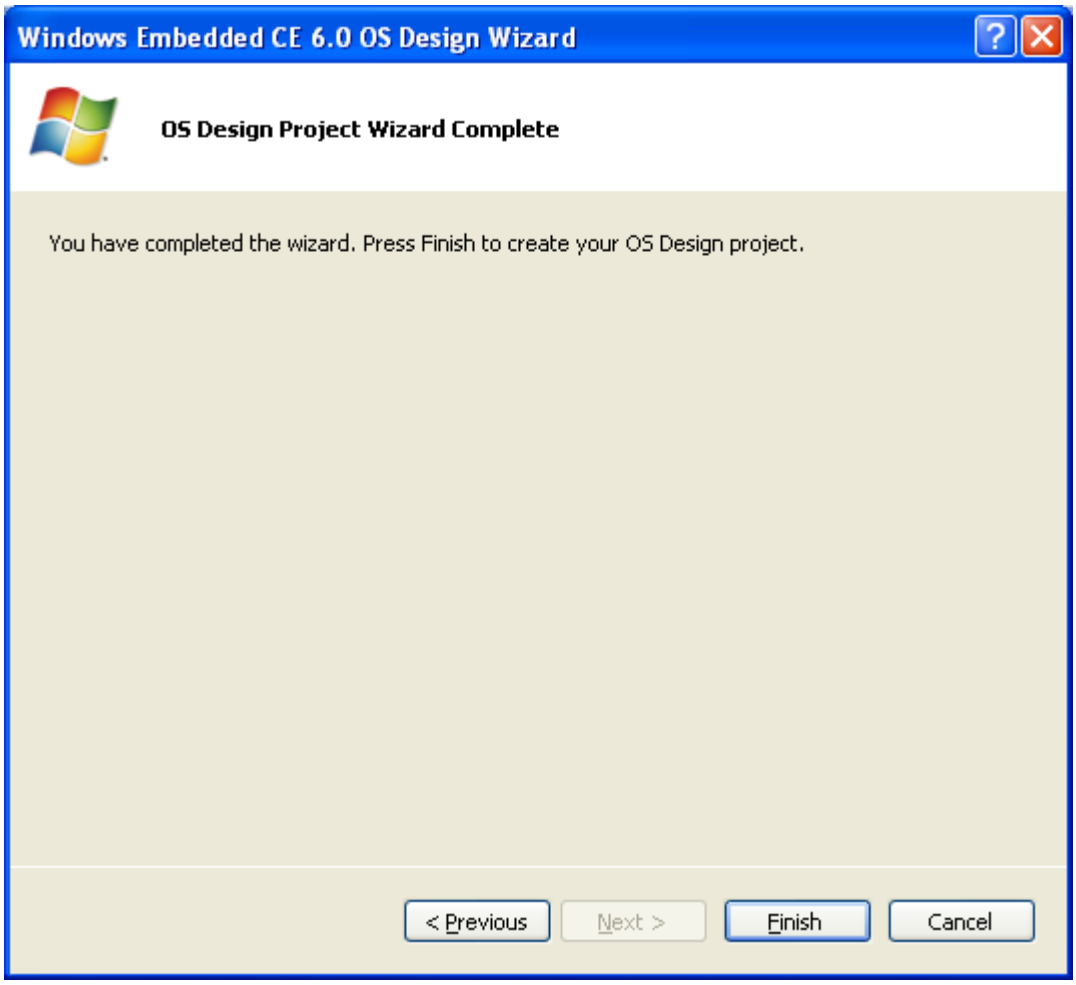
Next comes networking and communications, you can see that Windows CE provides support for Personal, local, and wide area networking through technologies such as Bluetooth, IrDA, Wired and Wireless networking, and VPN.

In our case the default options are fine.



- Click **Next**

We're done! – The wizard is complete. You've configured the OS Design and can now further customize it by adding and/or removing components from the Catalog.



- Click **Finish**

The Wizard will now prompt you with any security warnings or other notifications related to the components you selected. Acknowledge these notifications.

- Click **Acknowledge**

At this point you have an OS Design containing all of the OS components which have been selected from the Wizard. In the next section we will further customize the OS Design using the Catalog.

3 Customize and Build the OS Design

3.1 Using the Catalog

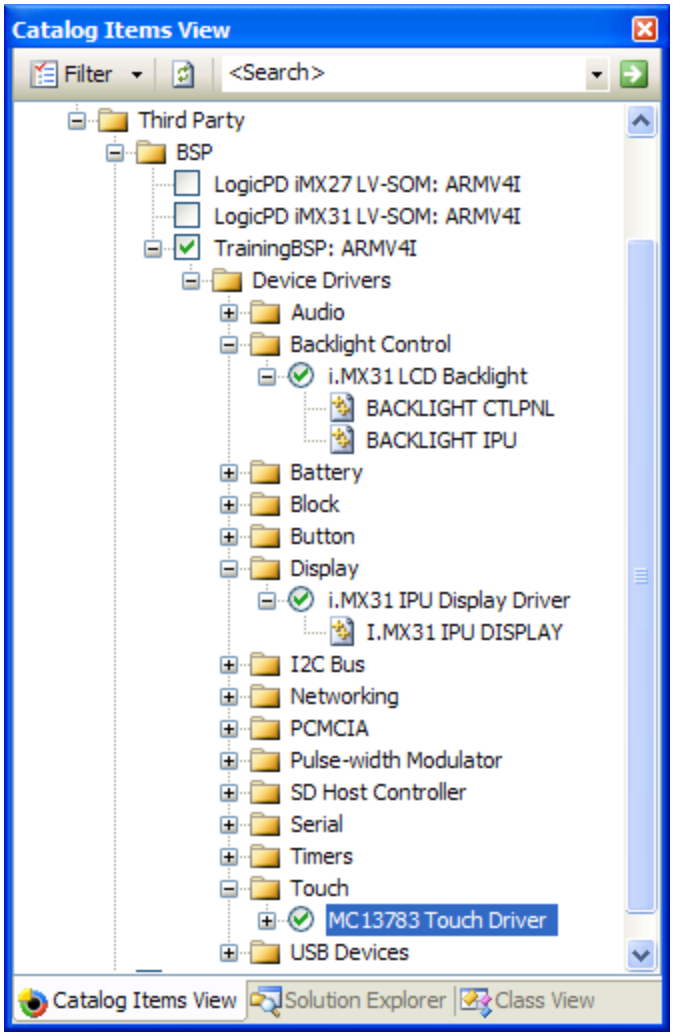
You will now make changes to the items included in your OS Design using the Catalog View.

First, ensure that the Catalog View is visible.

- Select the **View** menu
- Choose **Other Windows** | **Catalog Items View**

We need to select a few components for our i.MX31 SOM-LV device.

- In the search box, type **i.MX31 LCD Backlight**
- Select the **i.MX31 LCD Backlight** component
- Note that there are other device drivers that are a part of this BSP.
- Under **Display**, select **i.MX31 IPU Display Driver**
- Under **Touch**, select **MC13783 Touch Driver**
- Your selections should match the following image:

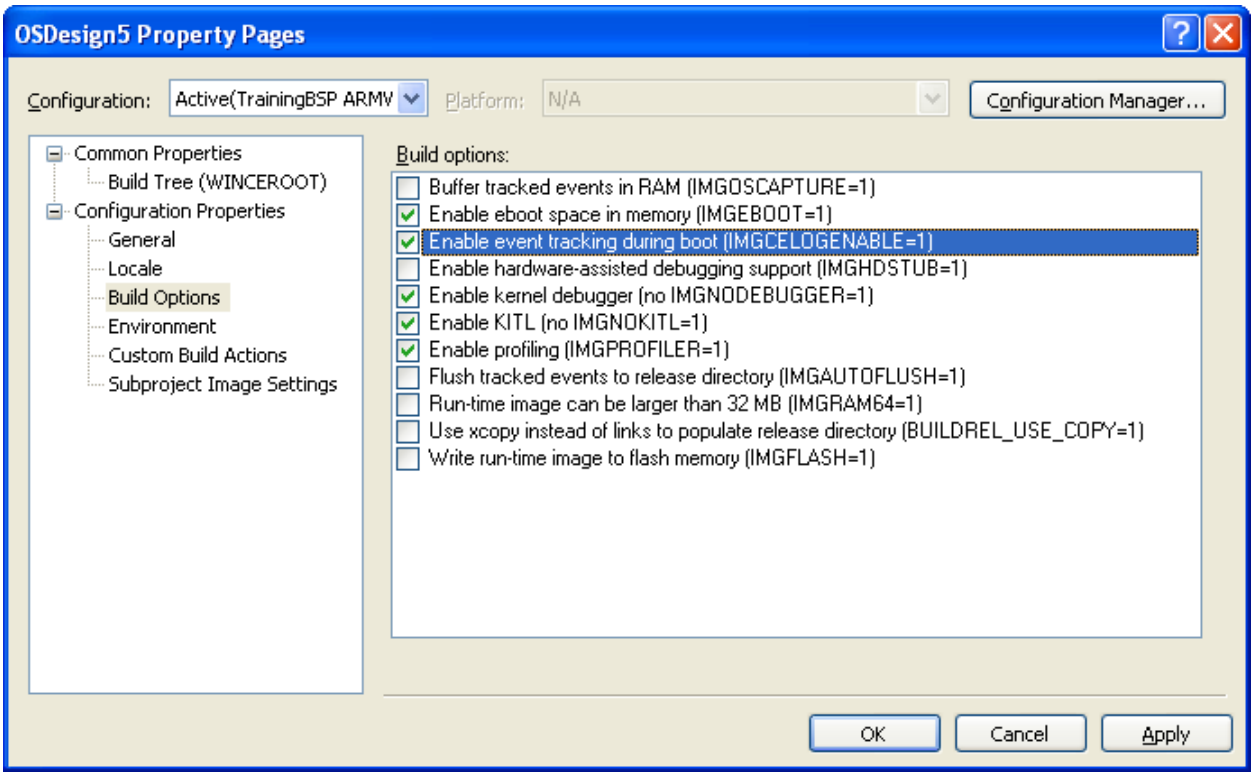


3.2 Enabling Profiling Kernel and Event Tracking

In the next section we will be using the Kernel Tracker. The Kernel Tracker tool requires that we have enabled the Profiling Kernel.

You enable the Profiling Kernel by selecting the following options:

- Select the **Solution Explorer** view
- Right Click on **OSDesign5** (or the name of your project)
- Select **Properties**
- Expand **Configuration Properties**
- Select the **Build Options** item



You will notice that the tools are currently set to build a Debug image of the platform. You have "Kernel Debugging" enabled, but in order to use the Kernel Tracker tool you also need to enable the Profiling Kernel and Event Tracking.

- Select **Enable Event Tracking During Boot**
- Select **Enable Profiling**
- Click **OK**

The profiler operates by taking an interrupt at a regular interval of time, at which point it records the program counter of the currently running thread. It then maps the program counter's instruction back to its module and function. Building up a collection of these profiling samples provides a statistical view of the code running in the system. We know the percentage of samples in each executable and in each function. The samples aren't guaranteed to accurately reflect true running time – it's possible, for example, that there's a chunk of code that runs at exactly the same frequency as the profiler interrupt, in between each interrupt, so that we never catch a sample in that code. But it's very unlikely. It's also possible that someone will turn off interrupts for a long time, preventing the profiler from capturing samples. However that's a more advanced problem and is beyond the scope of this lab.

Note: For more information, see the Platform Builder help document "Information Not Reported by the Kernel Profiler," http://msdn2.microsoft.com/en-us/library/Aa462494.aspx

When you enable profiling in your image (set IMGPROFILER=1), you include the Monte Carlo profiler into the kernel AND include all the symbols for all the executables in the "MODULES" section of ROM in a big table inside the image. The profiler uses that big symbol table to convert addresses like "0x12345678" to symbols like "foo.dll!MySlowFunction". The profiler reports "unknown" for any hits that are in modules that are not in the MODULES section of ROM because those symbols are not in the big table.

The kernel profiler can operate in three different modes: Monte Carlo mode, system call mode and kernel call mode. We will only be using Monte Carlo mode during this lab, but we'll touch on the other two modes briefly.

Let's jump in and start looking at data. The first thing you need to learn about the profiler and CeLog output is what idle time looks like.

3.3 Add the Performance Monitor Application

We need to add the PerfSample application now so that we can use it later in this lab.

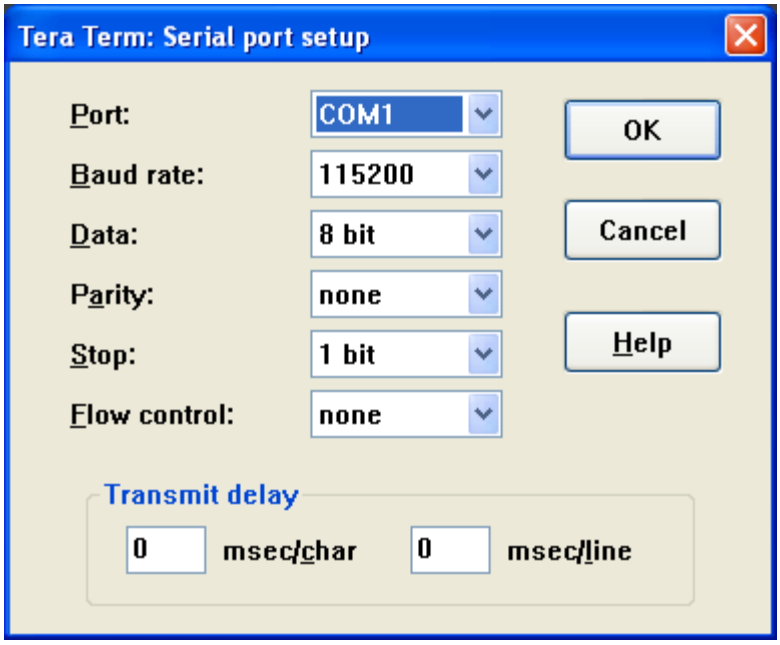
- Copy the **PerfSample** folder from C:\Labs\Lab5\LabFiles\ into your OSDesign folder at **C:\WINCE600\OSDesigns\OSDesign5\OSDesign5**
- In the Solution Explorer window right click on the **Subprojects** folder and select **Add Existing Subproject…**
- Navigate to the **PerfSample** folder that you just copied.
- Select **PerfSample.pbpxml** and click **Open**. Visual Studio will add the project to your current solution.
- Select **Build** | **Build Solution**

4 Download the Platform Image

4.1 Configure Download/Debug Transport

You are now ready to download and debug the Windows CE operating system image – before you download let's check to make sure the debug and kernel transports are configured and that we can view debug information over the serial terminal.

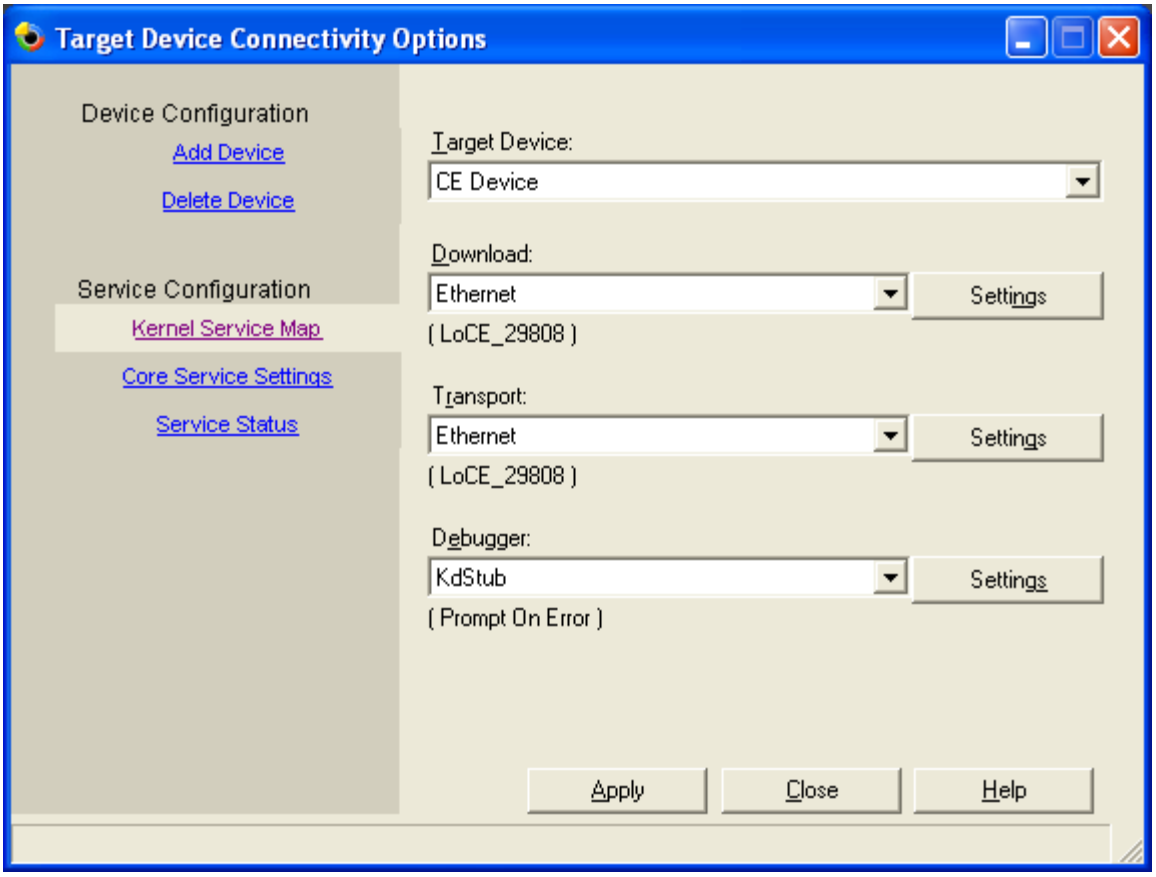
- Open **TeraTerm** by double clicking the icon on the desktop.
- Verify the serial port settings by clicking the **Setup** | **Serial Port** menu item (Your physical Port may differ):



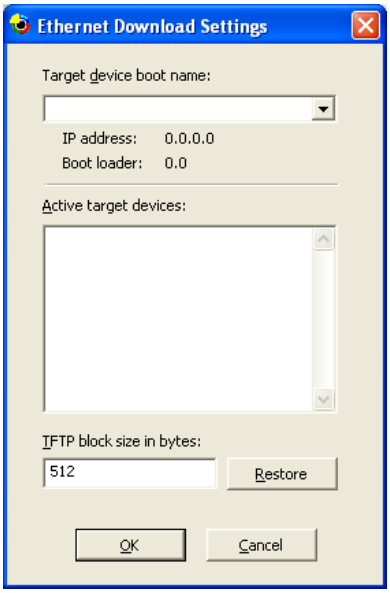
Ok now that you have configured your serial terminal we can continue to setup the device and Platform Builder. Switch back to your Visual Studio 2005 instance.

- Select **Target** | **Connectivity Options**

The following dialog will be displayed:



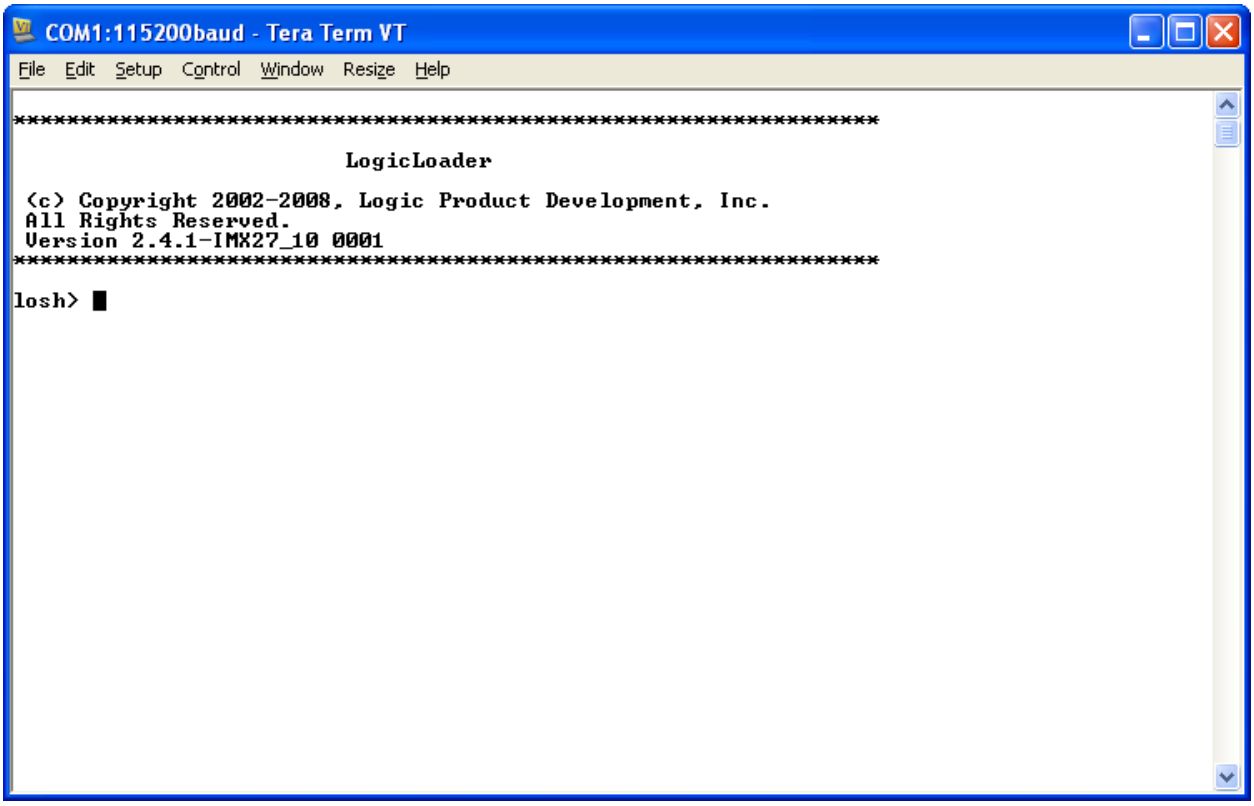
- In the **Download** box select **Ethernet**
- In the **Transport** box select **Ethernet**
- In the **Debugger** box select **KdStub**
- Click on the **Settings** box next to the **Download**: dropdown box. A box like this will appear:



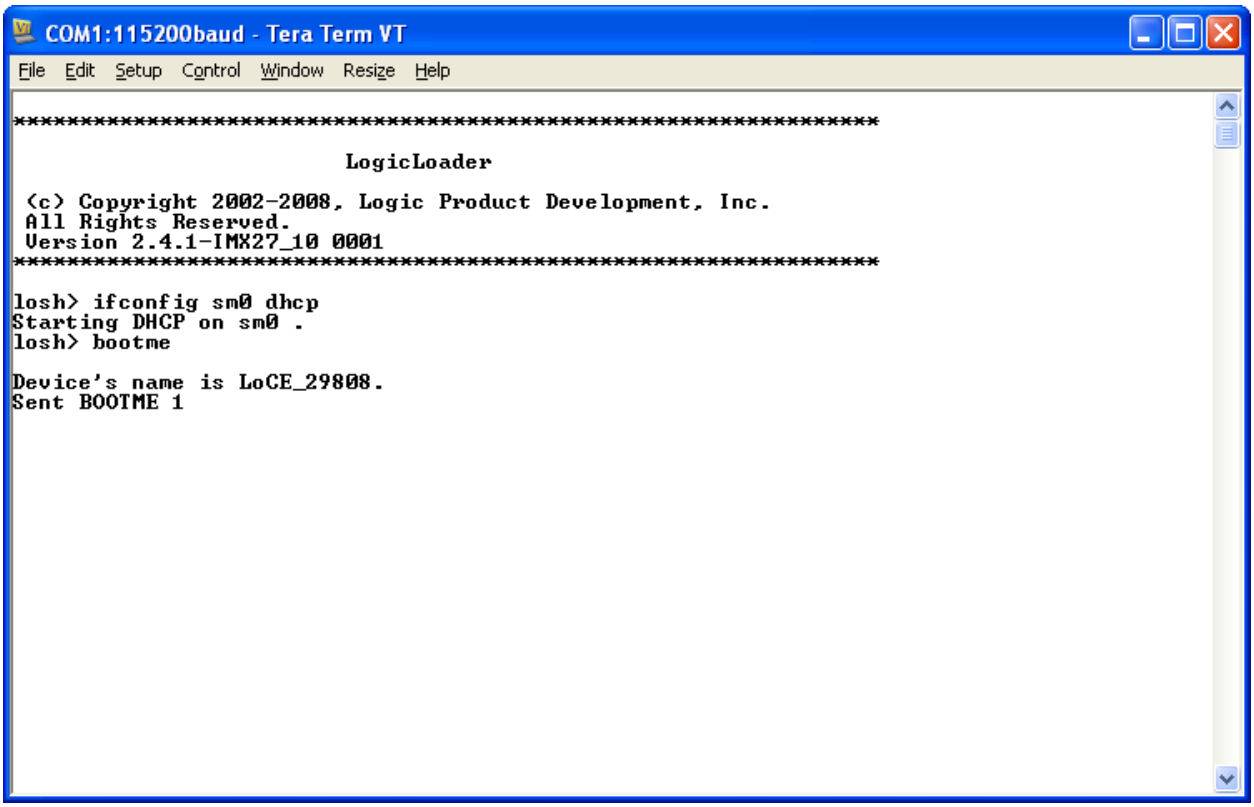
- Leave this box open and move on to the next step

Now we will switch to the **Tera Term** window to configure our device.

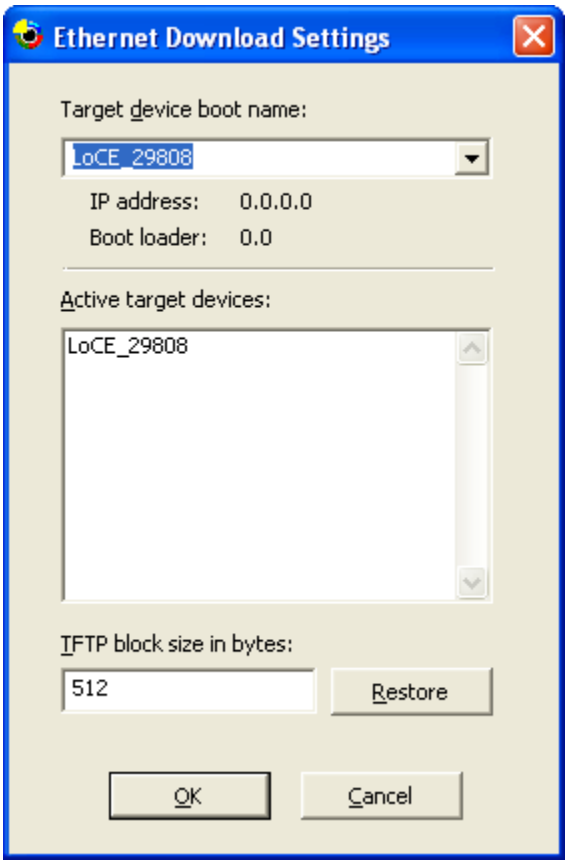
- Power on the device.
- You should see a **losh** prompt display.



- Configure the network interface using either DHCP or a statically assigned address:
  - For DHCP, type: **ifconfig sm0 dhcp**
  - For a static IP address, use the following command: **ifconfig <interface> <ip> <netmask> <gw>**
    - Example: **ifconfig sm0 192.168.1.1 255.255.255.0 192.168.1.0**
- Type the command **bootme** and press the Enter key. This will allow visual studio to connect to this device. Note the name of the device that is displayed:



- Switch back to Visual Studio and the box you left open should now be populated with a device. If there are multiple devices in the list, click on them and view the IP address. Make sure it matches up with the one that you wrote down.
- **Note**: if the device does not appear in Platform Builder's "Ethernet Download Settings" window, please check your firewall settings and anti-virus software. The "bootme" protocol uses raw UDP packets for communication between the device and the tools.



- Click **OK** in the small box, and then click **Apply**. Once you have done this, click the **Close** button.

4.2 Download the Operating System

- Select **Target** | **Attach Device**
- The device should still be executing the **bootme** command, but if it is not, type **bootme** at the **losh** prompt
- When the image is finished transferring (a **losh** prompt will display), type the following command to boot the image:
  exec rtc:rtc_imx31_pmic:dbg_leds:1:disp_num:3:kitl:true:share_eth:1:
- The debugger may pause with a **DEBUGCHK** message. In the **Debug** menu, click **Start** or press **F5** on the keyboard to continue execution

The image will load quite slowly because we are in a DEBUG build.

5 Running the Profiler

5.1 Run the Profiler

- Select **Target** | **Target Control**
- Type **prof on** to start the profiler
- You will see output like the following in the Windows CE debug log:

```
186532 PID:400002 TID:ed0002 Kernel Profiler: Gathering MonteCarlo data
in buffered mode
```



```
186532 PID:400002 TID:ed0002 +OEMProfileTimerDisable()

186532 PID:400002 TID:ed0002 -OEMProfileTimerDisable

186959 PID:400002 TID:ed0002 ProfileStart() : Allocated 21348 KB for
Profiler Buffer (0xD0D50000)

186959 PID:400002 TID:ed0002 +OEMProfileTimerEnable(200)

186959 PID:400002 TID:ed0002 -OEMProfileTimerEnable
```

- Wait some time for the profiler to collect data – at least 10 seconds
- Type **prof off** to stop profiling

When you stop profiling, you'll get a lot of information printed to the debug output window. You'll notice that the output is separated into three segments. The first one is the summary:

```
================== MONTE CARLO HIT REPORT ==================

Total samples recorded =    69013

Total samples dropped  =        0

Ticks elapsed          =    14098

IDLE TIME:

* Total non-idle samples recorded                  =      363

* Total idle samples recorded                      =    68650

* Percentage of recorded samples in idle           =     99.4

 INTERRUPTS:

* Total interrupts during sample period            =    69868

* Non-profiler interrupts                          =      855

* Non-profiler interrupts per tick                 =      0.0
```

The first detail to check in the header is the number of samples that was recorded. If the profiler is run for too short a time, the sample size is too small and the data is too unreliable. If you only have 100 samples, don't waste your time looking. If you have 1000, be cautious about drawing conclusions. More data is better. For this report, the profiler ran for 14 seconds (14098 1ms timer ticks) to generate 69013 samples.

The next detail is the number of samples dropped. If you use buffered profiling (which is the default), it is possible that you'll run the profiler so long you overfill the buffer. All of the samples after that would be dropped. If you ever see a non-zero drop count then you might need to increase the size of the buffer or switch to un-buffered profiling. Those techniques are outside the scope of this lab, but for more information see the Platform Builder help documentation under "Buffered and Unbuffered Profiling." http://msdn2.microsoft.com/en-us/library/ms893176.aspx

The rest of the header information summarizes idle time and interrupt activity that occurred while the profiler was running.

The second segment of the profiler output is a sorted list of all the modules that had profiler hits.

```
MODULES:  (Does not include dropped samples!)
```



```
Module            Hits        Percent
----------------  ----------  -------
IDLE                   68650     99.4
kernel.dll               203      0.2
kitl.dll                  37      0.0
k.coredll.dll             29      0.0
cxport.dll                19      0.0
tcpstk.dll                19      0.0
ndis.dll                  16      0.0
tcpip6.dll                11      0.0
afd.dll                    8      0.0
relfsd.dll                 5      0.0
celog.dll                  4      0.0
coredll.dll                2      0.0
fsdmgr.dll                 2      0.0
VMini.dll                  2      0.0
netbios.dll                2      0.0
UNKNOWN                    4      0.0
```

“Idle” is not a module, but a counter of samples when the CPU was idle (no code was running). “Unknown” is code that is not part of the MODULES section of ROM, not in the “big table” of symbols.

The last segment of the profiler output is a sorted list of all the functions that had profiler hits. Note that the following data was truncated solely for the purposes of display here.

```
SYMBOLS:  (Does not include dropped samples!)


Hits       Percent Address  Module           Routine
---------- ------- -------- ----------------:---------------------
     68650    99.4 -------- ----------------:IDLE
        30     0.0 80256568 kernel.dll      :INTERRUPTS_ENABLE
        12     0.0 8026c770 kernel.dll      :KCNextThread
        11     0.0 802a0e38 kernel.dll      :CheckTakeCritSec
         9     0.0 802737bc kernel.dll      :WaitOneMore
         8     0.0 802a12d8 kernel.dll      :DoWaitForObjects
```



```
         8     0.0 c0881a74 ndis.dll
:ethFilterDprIndicateReceivePacket

         7     0.0 80268cec kernel.dll      :MakeRun

         7     0.0 802696f0 kernel.dll      :RunqDequeue

         7     0.0 c010f468 k.coredll.dll   :EnterCriticalSection

         6     0.0 80254e24 kernel.dll      :KCALLPROFON

         6     0.0 802afe74 kernel.dll      :NKHandleCall

         6     0.0 c010f754 k.coredll.dll   :LeaveCriticalSection

         5     0.0 80256128 kernel.dll      :SaveAndReschedule

         5     0.0 8029a0e4 kernel.dll      :EnterCriticalSection

         5     0.0 8029cc2c kernel.dll
:CELOG_WaitForMultipleObjects

<truncated for display>

         4     0.0                          :<UNACCOUNTED FOR>
```

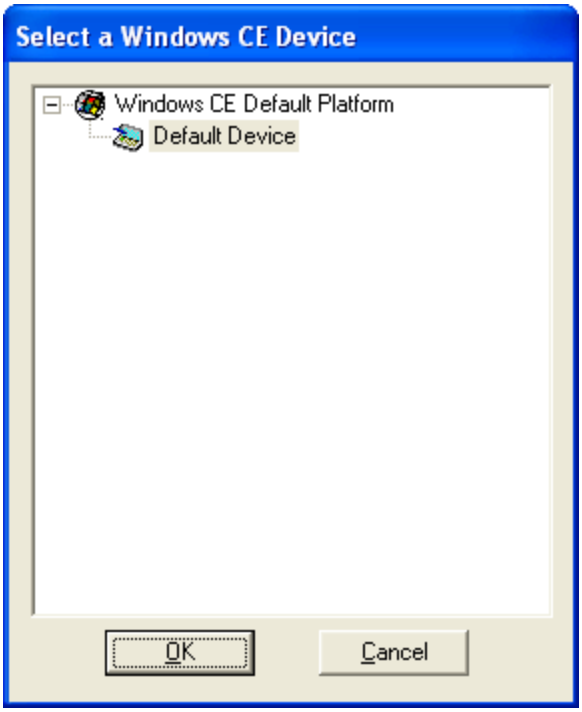
Similar to the way idle time is represented in the module list, there's a line representing idle in the function list. The rest of the functions in the list tell you how the samples from the module list above break down to samples inside functions from those modules. You could add up all the samples from one module and see that they equal the number of samples for that module in the module list above.

When you look at the module and function lists reported by the profiler, be very careful not to get too carried away with small numbers.

Okay, so now you've seen what an idle system looks like under the kernel profiler. Let's take a completely different look at the same system.

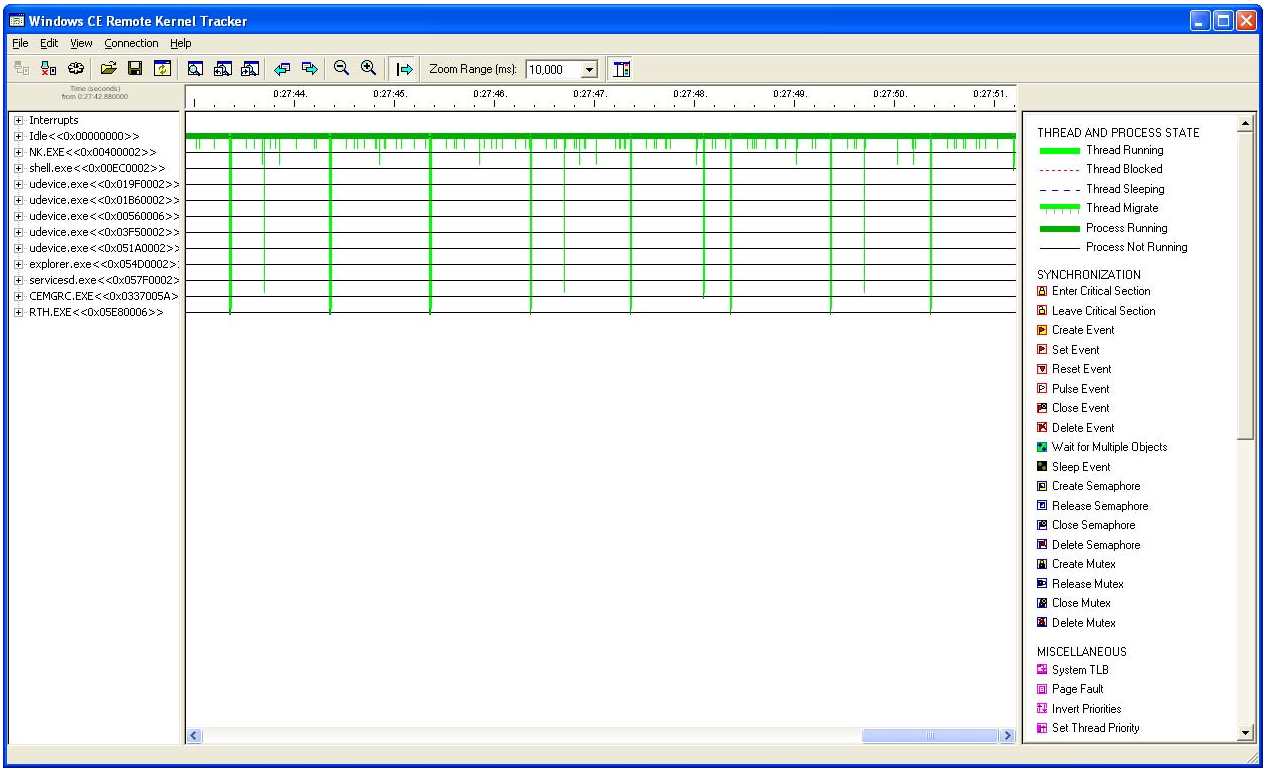
5.2 Collect Data with Remote Kernel Tracker

- Select **Target** | **Remote Tools** | **Kernel Tracker**



- Choose **Default Device** and click **OK** to connect

You'll see the Platform Manager connect and download some tools onto the device, and then data should start appearing in the Kernel Tracker window.



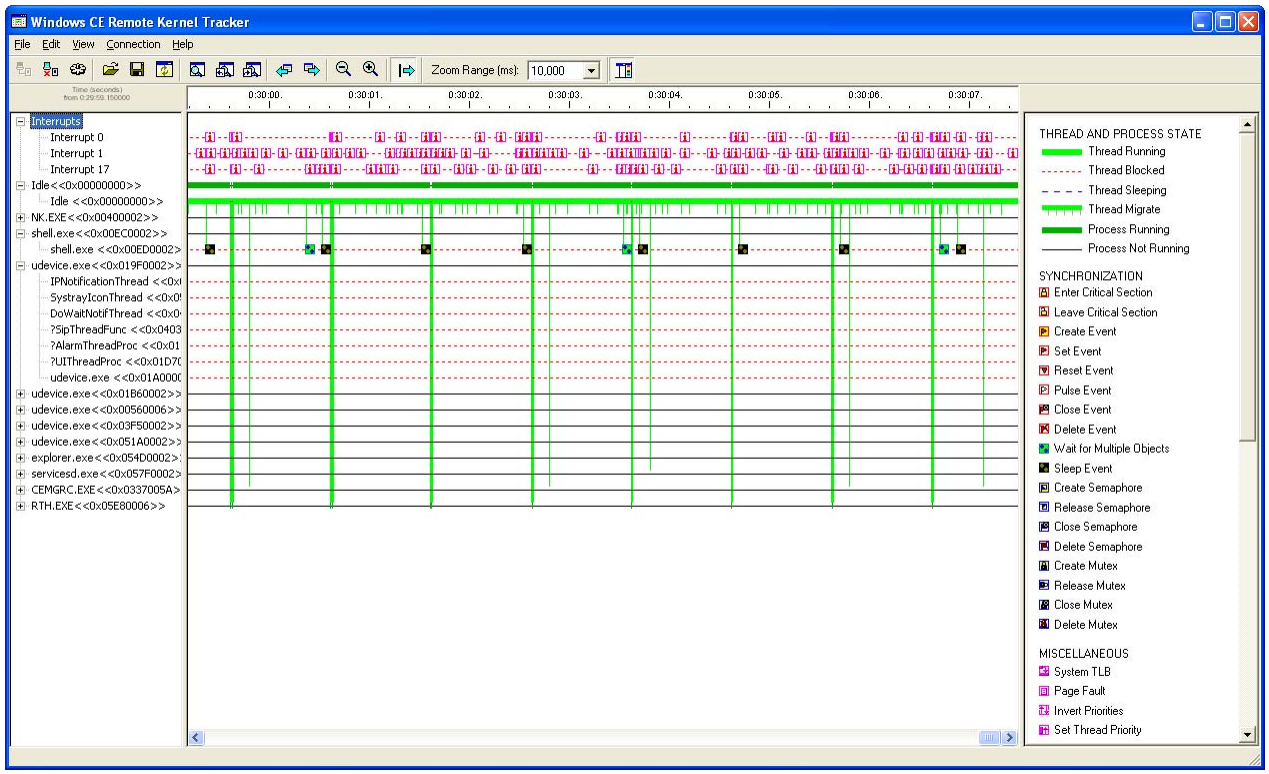
Each process in the system has a horizontal line. When one of the threads owned by that process is running, the process line is wide and dark green. Otherwise the process line is thin and black. (In case it's not clear, only one thread ever runs at a time, so only one process ever

runs at a time too.) So in the example above, threads owned by kernel (nk.exe) are running almost all the time. The vertical lines you see are due to threads in other processes running for very short periods of time. Remote Kernel Tracker draws vertical lines to represent switches between threads, to make it easier to see which threads ran before and after the thread switch.

- If you'd like to see more data at once, hide the key by clicking the icon in the top right next to the zoom range.

Underneath each process are all the threads owned by that process. Processes can be expanded and collapsed to show and hide the threads, for easy viewing.

- Click on the "+" sign next to the interrupts and processes to expand the process lines.



5.3 Disconnect and Save the Log File

- In Remote Kernel Tracker, choose **Connection** | **Disconnect from Device**
- Select **File** | **Exit**
- It will ask whether you want to save the collected data. Click **Yes**
- Save the log file as **C:\idle.clg**

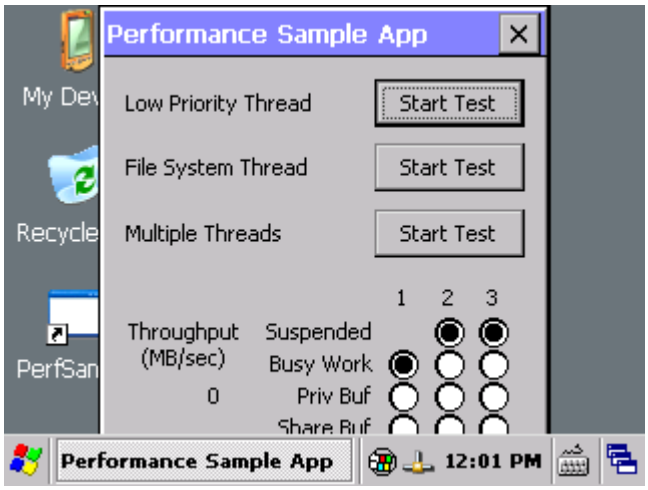
6 Viewing Thread Performance

Now that you know what our performance tool output looks like for an idle system, let's compare that against what the tools show when there's a low-priority thread running continuously. A low-priority thread can keep the CPU from going idle, but it should not prevent any other system activity from taking place. All the interrupts and thread activity we just observed on an idle system should stay exactly the same on a system where there's a low-priority thread running. The low-priority thread only affects the execution of other threads with equal or lesser priority – and the fact that the CPU won't enter idle. So you might expect to see exactly the same results

that you saw during idle, except with idle time replaced by time on the continuously-running thread. Let's check that assumption by analyzing that scenario.

6.1 Start the Low-Priority Thread

- On your running hardware, double click on the **PerfSample** icon on the desktop to start the sample program.



- Next to **Low Priority Thread**, click **Start Test** to run the test.

You won't see a visible difference except for the fact that the button switches to say "Stop Test," and the debug output contains a message confirming that the thread is running. The low-priority thread is now running continuously in the background. You could click on the desktop or move the UI around, with no visible difference compared to the response you got on an idle system. That's because all of that activity happens at a priority that is higher than that of the running thread. UI activity simply preempts the thread until the UI events are handled.

Once you've satisfied yourself that nothing outwardly seems different, run the profiler to see what's going on in the system.

6.2 Run the Profiler

- Type **prof on** in the Target Control window to start the profiler.
- Wait some time for the profiler to collect data – at least 10 seconds.
- Type **prof off** to stop profiling.

Below are the results of 11.5 seconds of data collection. Note that the data has again been truncated for length in this document:

```
Module             Hits         Percent

---------------   ----------   -------

kernel.dll              53043      94.8

PerfSample.exe           1042       1.8

coredll.dll               919       1.6

celog.dll                  35       0.0

kitl.dll                   33       0.0
```


Did you expect to see the 99.4% idle be replaced by 99.4% of time in PerfSample.exe, with all the rest of the results staying the same? It's a pretty natural expectation. Instead, only 1.8% of the run time is spent in PerfSample.exe, while the majority of time is spent inside the kernel.

Take a look in the SYMBOLS section of the output (only the relevant PerfSample lines are shown):

```
604     1.0 00011a48 PerfSample.exe  :?LowPriorityThread
422     0.7 00012988 PerfSample.exe  :WaitForSingleObject
```

Why doesn't the time show up as time inside PerfSample.exe? Because the profiler is telling you what code is running, not what thread is running. The thread is calling WaitForSingleObject, which is a kernel API. So a lot of time is spent executing kernel code:

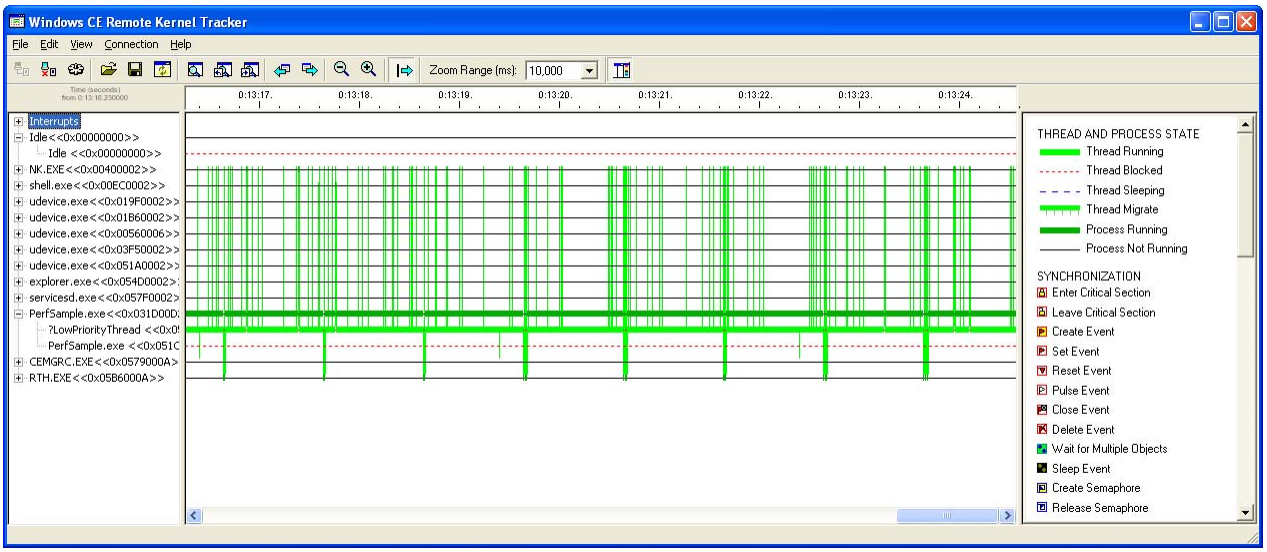
```
static DWORD WINAPI LowPriorityThread (LPVOID lpvParam)
{
    // Nobody will ever set this event
    HANDLE hEvent = CreateEvent (NULL, TRUE, FALSE, NULL);
    if (hEvent) {
        // Repeatedly wait for the event, timeout=zero
        do {
            WaitForSingleObject(hEvent, 0);
        } while (!g_LowPriThreadExit);
        CloseHandle (hEvent);
    }
    return 0;
}
```

Now let's go back to Remote Kernel Tracker view of the system.

6.3 Collect data with Remote Kernel Tracker

- In Visual Studio, select **Target** | **Remote Tools** | **Kernel Tracker** and then click **OK** to connect

After a few seconds you should see data start appearing. The results match our expectations. The time in kernel idle "thread" is replaced with time in the PerfSample program, and otherwise things look pretty much the same as before.



The way the PerfSample run-time resembles the idle time in our previous log verifies the earlier statement that the PerfSample thread will keep running even when it calls a system API call.

Note that when you look at the system with Remote Kernel Tracker, you can tell which thread is running, but not what that thread is doing. Just because you see a lot of events in the Kernel Tracker does not mean those events are taking a significant part of the thread run-time!

Kernel Tracker tells you what thread is running. Monte Carlo profiling tells you what that thread is doing.

6.4 Disconnect and Save the Log file

- In **Remote Kernel Tracker**, choose **File** | **Exit**
- It will ask whether you want to save the collected data. Click **Yes**
- Save the log file to **C:\lowpri.clg**

Remote Kernel Tracker will give you a visualization of how long each thread ran, but in cases it is useful to have actual totals of thread and process run-times. Let's take a quick look at quantitative totals. We'll use the "readlog" command-line tool to parse the log and generate summary information for the two logs we've gathered. Since it is a command-line tool, we use a release directory build window to run it.

6.5 Read the Log Files

- Select **Build** | **Open Release Directory in Build Window**
- In the command window, type **cd C:** to go to the location where the log files are saved.
- Type **readlog -s idle.clg idle.txt**
- Type **readlog -s lowpri.clg lowpri.txt**

Here is an example from idle.txt. You will find the following lines near the end of the file:

```
Processor utilization, from idle thread running time:      98. 0%

Processor utilization, from running time of other threads:  1. 0%

Total time  0:06:46.477.026
```

This is reporting the idle time. You can compare the total time to that of the Idle thread below to see that indeed the Idle process took most of the clock time.

Next you will see thread information. Not all threads will have a known name. For example, the RTH.EXE and CEMGRC.EXE executables were copied over to the system by the Remote Kernel Tracker.

```
Thread information:

Thread:     StartProc:  CurProc:    Run Time:         Status:  Name:

0x00000000  0x00000000  0x00000000  0:06:38.693.683   Active   Idle

0x04FF000A  0x05E80006  0x05E80006  0:00:00.255.603   Active   UNKNOWN
THREAD
```

However, we can match these threads to a process by looking in the process information table:

```
Process information:

Process:    Thread Run Time:  Code Run Time:   Status:  Name:

0x05E80006  0:00:04.953.866   0:00:04.951.565  Active   RTH.EXE
```

Note that the process number matches that of the CurProc number of the thread information. Visit http://msdn.microsoft.com/en-us/library/aa935935.aspx for information on how to get readlog to do this match work for you.

Now selections from lowpri.txt:

```
Processor utilization, from idle thread running time:       0. 0%

Processor utilization, from running time of other threads: 99. 0%

Total time  0:03:10.332.036

Thread information:

Thread:     StartProc:  CurProc:    Run Time:         Status:  Name:

0x0568000A  0x031D00D2  0x031D00D2  0:03:04.161.381  Active
?LowPriorityThread

Process information:

Process:    Thread Run Time:  Code Run Time:   Status:  Name:

0x031D00D2  0:03:04.161.381   0:03:04.161.381  Active   PerfSample.exe
```

Here we can see that PerfSample.exe took most of the processor time, and spent that time in LowPriorityThread.

7 Measuring Thread Performance

Let's take a look at another example where a thread keeps looping. The thread is enumerating through the contents of the file system release directory. It's looping and calling the FindNextFile API over and over. The release directory file system is implemented by calling desktop file system APIs over KITL. So, we'll expect the system to be spending at least some amount of time performing KITL(Kernel Independent Transport Layer) communication.

```
static DWORD WINAPI FSEnumThread (LPVOID lpvParam)

{

    do {
```



```
        WIN32_FIND_DATA fdFile;
        HANDLE hFind = FindFirstFile (TEXT("\\release\\*"), &fdFile);
        if (hFind) {
            do {
                // Not actually doing anything with the data
            } while (!g_FSEnumThreadExit && FindNextFile (hFind,
&fdFile));
            FindClose (hFind);
        }
    } while (!g_FSEnumThreadExit);

    return 0;
}
```

7.1 View the File System Thread with the Kernel Profiler

- In the PerfSample UI by **File System Thread**, click **Start Test** (stop the low priority thread if it is running). You won't see much of a difference in the system behavior.
- Type **prof on** in the Target Control window to start the profiler.
- Wait some time for the profiler to collect data – at least 10 seconds.
- Type **prof off** to stop profiling.

Again, the results might surprise you: Instead of being 100% busy, the system is spends a lot of time idle. And very little time is spent inside file system routines!

Here is a sample of the output:

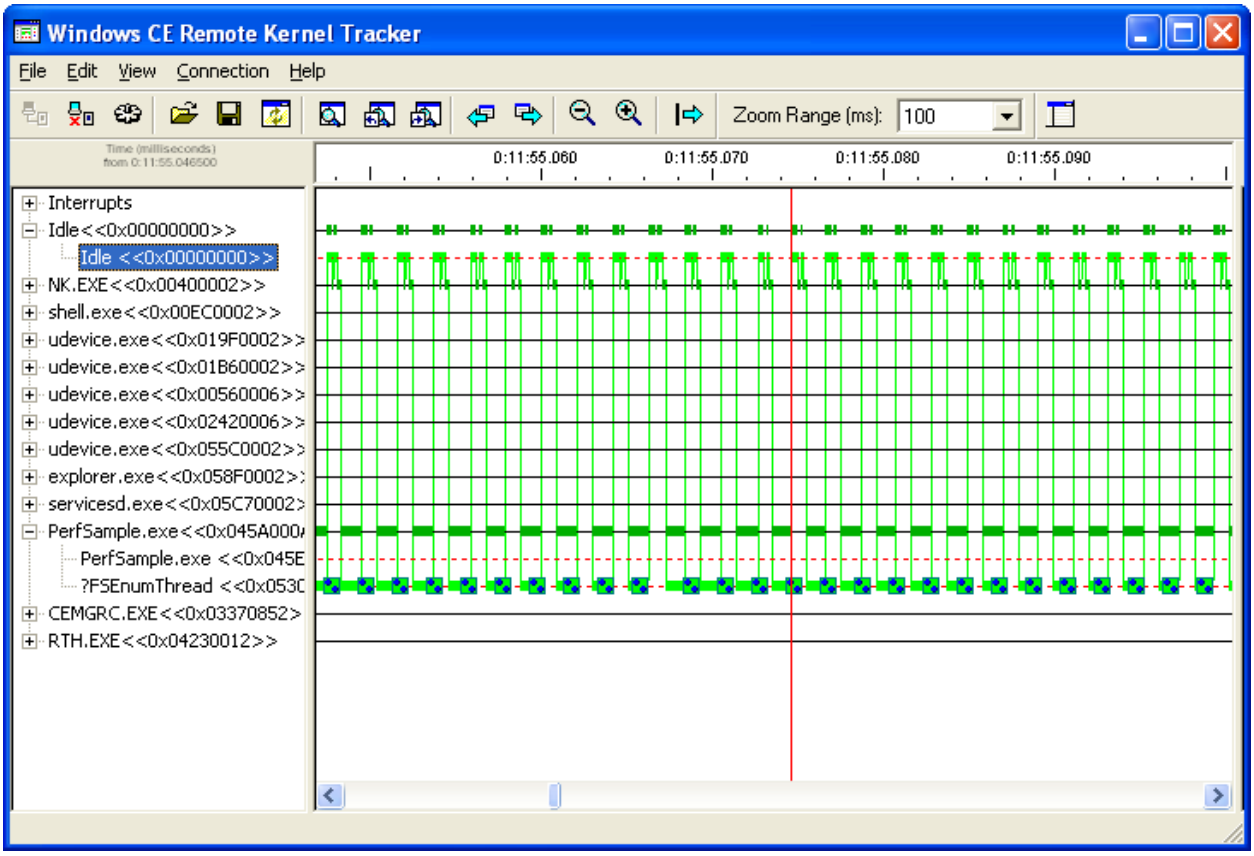
```
MODULES:  (Does not include dropped samples!)


  89245 PID:400002 TID:ed0002 Module            Hits        Percent
  89246 PID:400002 TID:ed0002 ----------------  ----------  -------
  89246 PID:400002 TID:ed0002 kernel.dll             14789     31.5
  89246 PID:400002 TID:ed0002 IDLE                   14275     30.4
  89256 PID:400002 TID:ed0002 relfsd.dll              7240     15.4
  89256 PID:400002 TID:ed0002 kitl.dll                4296      9.1
  89256 PID:400002 TID:ed0002 k.coredll.dll           3284      7.0
  89257 PID:400002 TID:ed0002 fsdmgr.dll              1459      3.1
  89259 PID:400002 TID:ed0002 UNKNOWN                  727      1.5
```


Relfsd.dll is the release directory file system driver, and fsdmgr.dll is the file system driver manager. These two comprise all the file system activity that is taking place on the device, and they only add up to 18.5% of the test run time.

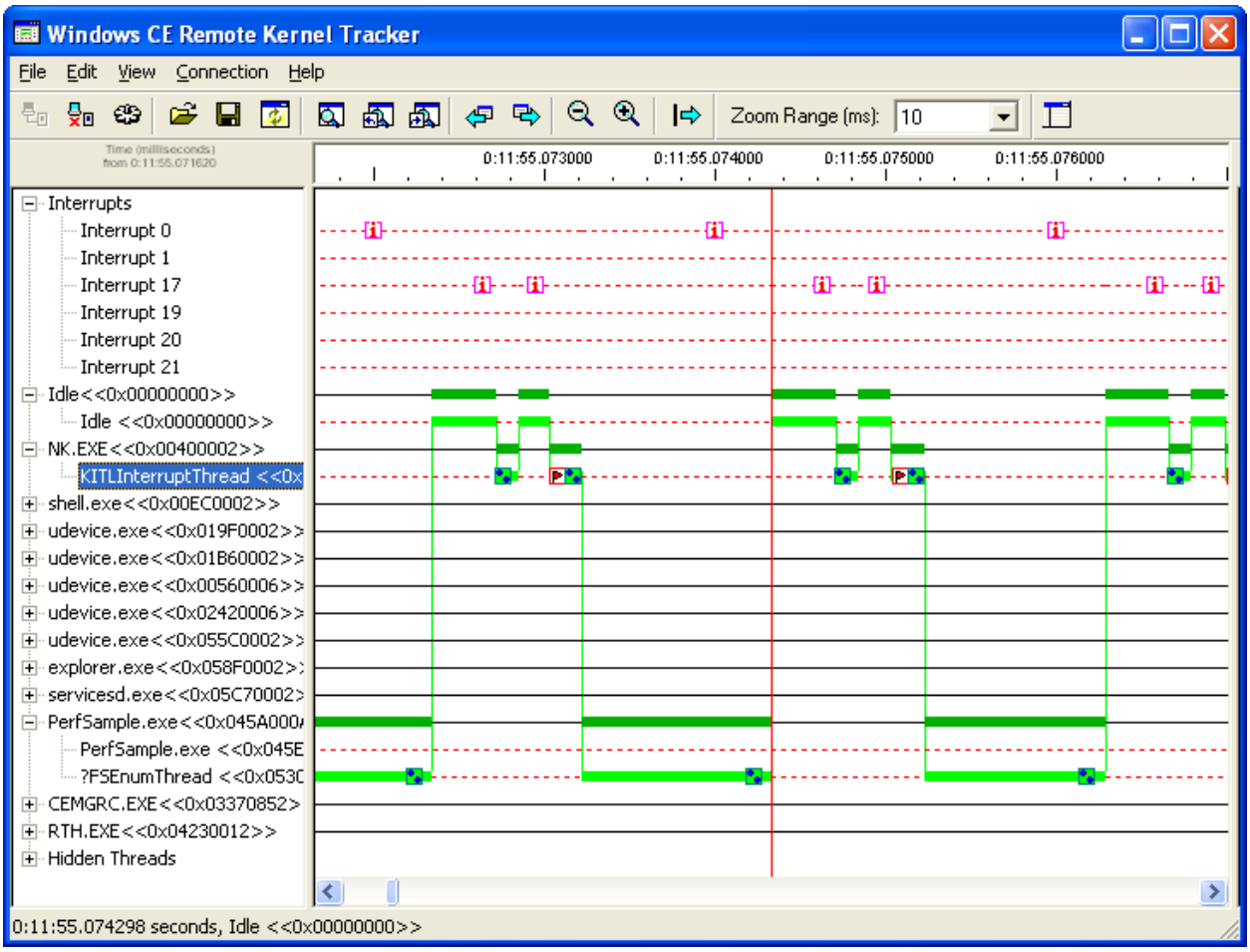
7.2 View the File System Thread with Remote Kernel Tracker

- In the PerfSample UI by **File System Thread**, click **Stop Test**. The Kernel Tracker can have difficulty connecting during heavy activity on the remote device.
- Open the Kernel Tracker by selecting **Tools** | **Remote Kernel Tracker** and then click **OK** to connect.
- In the PerfSample UI by **File System Thread**, click **Start Test**.
- Wait for the activity shown in Kernel Tracker to look heavier. If it takes too long, click **Stop Test** again to shut down the device side activity and let Kernel Tracker catch up.
- Click on the region of heavy activity, to set a vertical red cursor line.
- Zoom in to see the thread behavior. In the **Zoom Range (ms)** drop-down, choose **100**.
- Click the **+** next to **PerfSample.exe** to show the threads in this process.
- Click the **+** next to Idle to show the threads from the idle process, and click on **Idle <<0x00000000>>**
- Right-click the idle thread and select **Find Next time this Thread Runs**.

This will take you to an example of CPU idle time. You should be able to find activity similar to the following screen shot. (You may need to search a few times.)



Here is a screenshot showing a close up of what the system is busy doing:



The **FSEnumThread** runs for a while, then it waits on an event. The CPU goes idle, which represented as a thread switch to idle. After a while interrupt 17 occurs, which is the KITL interrupt in the device. You can see the KITL interrupt occur and the resulting context switch from idle to the interrupt service thread (IST). As you can see in Kernel Tracker, that thread is appropriately named **KITLInterruptThread**. The KITL IST runs briefly, goes idle and runs again, then it sets an event and waits on a different event. Since it set the event that the **FSEnumThread** was waiting for, that thread executes as soon as the **KITLInterruptThread** blocks.

How do I know which event is being set or waited on? You can hover or double-click the event icons to see their information. If you look, you'll see that the **WaitForMultipleObjects** in **FSEnumThread** uses the same handle value as the **SetEvent** in **KITLInterruptThread**. So the **PerfSample** thread waits until the KITL thread wakes it up. In turn, the KITL thread waits until the KITL interrupt fires. So, the idle time we observed is spent waiting for the KITL interrupt. After the **PerfSample** thread wakes it soon asks to enumerate another file, and it blocks in **WaitForMultipleObjects** again waiting for the next KITL interrupt.

So, what's going on during this test scenario is that the **PerfSample** thread is waiting for the release directory file system information across the KITL connection to Platform Builder, and is waiting for Platform Builder to get the information and send the result back to the device.

The lesson of this example is that threads can block when you don't expect. Call a file system API and your thread could end up waiting for your KITL connection. Call a heap API and your thread could block waiting for a different thread to release heap locks. Call someone else's code and they could put your thread to sleep. It is never a bad idea to validate your assumptions by observing actual behavior using the system tools.

- In Remote Kernel Tracker, Select **File** | **Exit**
- Click **No** unless you want to save the data

Lessons learned during this exercise:

Threads can block at unexpected times. It is never a bad idea to validate your assumptions by observing actual behavior.

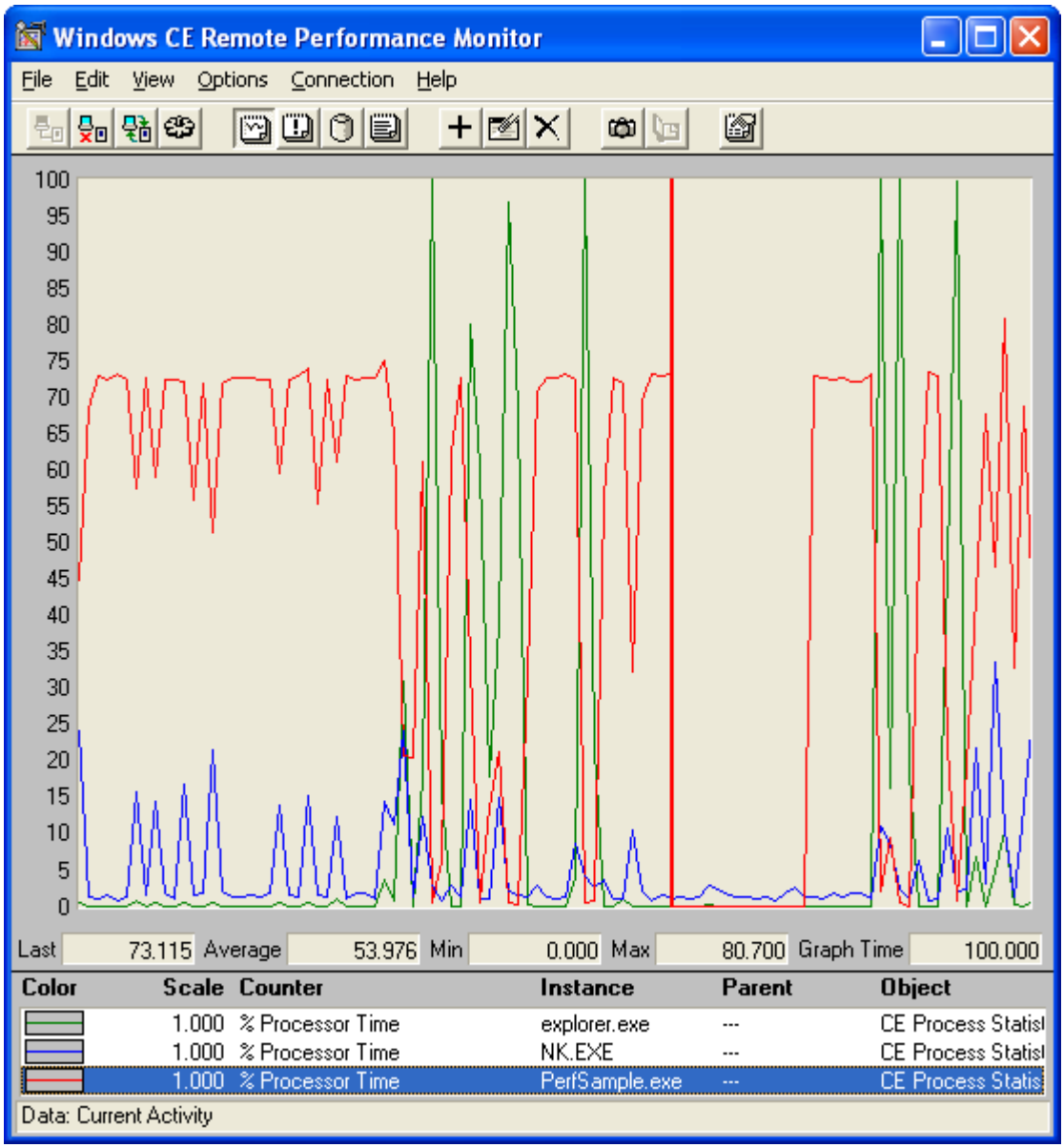
Using Remote Kernel Tracker you can identify why threads block and what causes them to resume running.

8 Using the Performance Monitor

Finally, we'll provide a brief introduction to using the Performance Monitor remote tool for a graphical measurement of system performance. The Performance Monitor enables you to chart different aspects of system performance relative to one another and to time.

8.1 Start the Performance Monitor

- Start the performance monitor remote tool by selecting **Target** | **Remote Tools** | **Performance Monitor**
- Connect to the device by clicking **OK**
- In the Performance Monitor on the development PC, select **Edit** | **Add to Chart**. This will bring up a window from which you can add a number of different statistics to track
- Select **CE Process Statistics** next to Object, then for Instance select PerfSample.exe, then for Counter select **% Processor Time**, and click Add at the top right corner to add it to the display
- Add the other programs (explorer.exe, NK.exe, etc) with the same way
- Use the different programs to increase the processor load and observe the results



9 Summary

You have now completed all the steps in this lab. Here's what we have covered:

- Measuring activity on an idle system
- Measuring activity while a low-priority thread is running
- Measuring activity while a thread is enumerating the file system
- Spending time inside OS API calls
- Un-contended critical section usage
- Contended critical section usage
- Unexpected thread blocking
- Unexpected thread interaction