i.MX31 SOM-LV Windows Embedded CE 6.0
Lab 4: Debugging
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 about the advanced debugging features in Platform Builder
- Monitor the operating system on the device from your workstation
- Familiarize yourself with the Remote Tools

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:

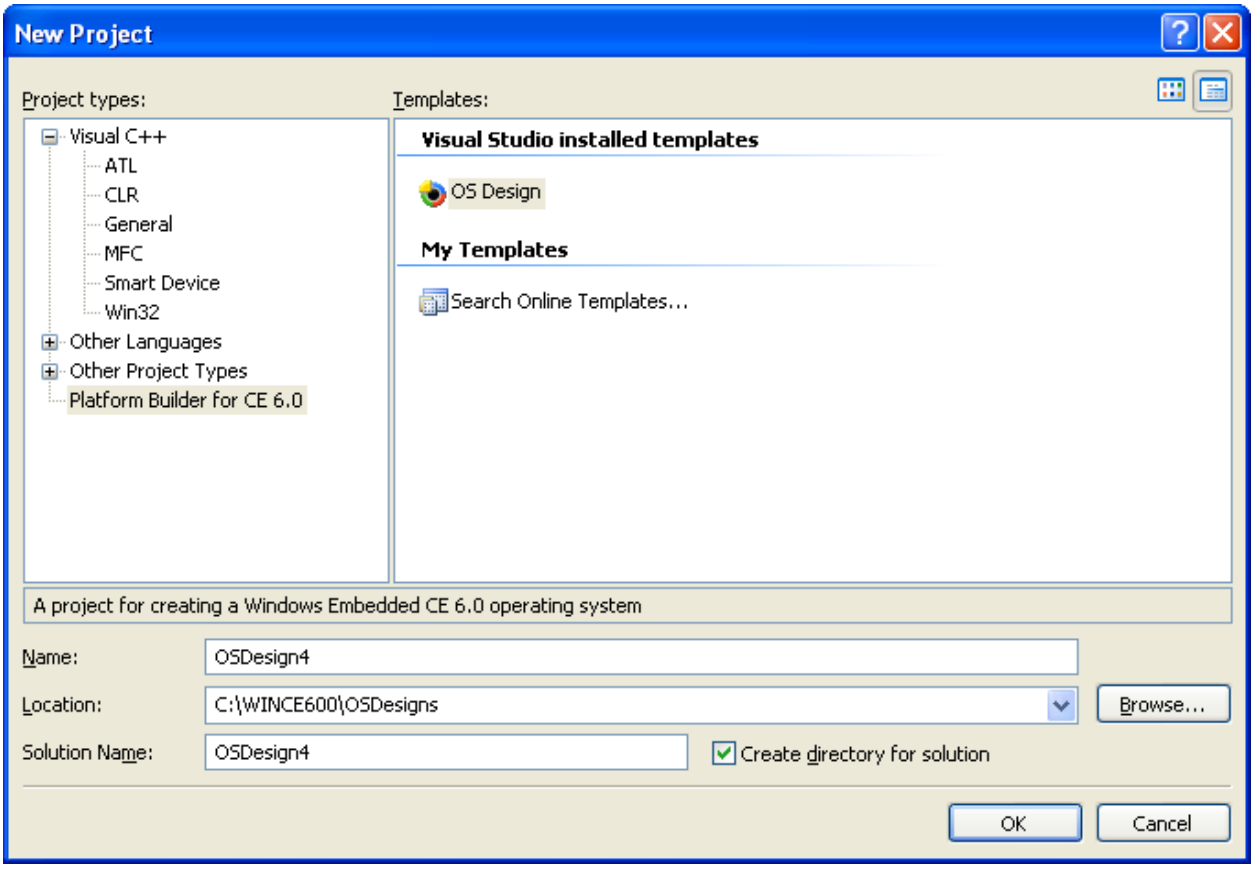
- A development workstation running Windows XP or Vista
- Visual Studio 2005 (Version 8)
- Service Pack 1 for Visual Studio 2005
- Service Pack 1 for Visual Studio 2005 on Vista (if you are running Vista)
- Windows CE 6.0 R2 Platform Builder plug-in with Service Pack 1
- A Zoom i.MX31 LITEKIT
- The i.MX31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

2 Creating the Platform Image

2.1 Create a new OS Design

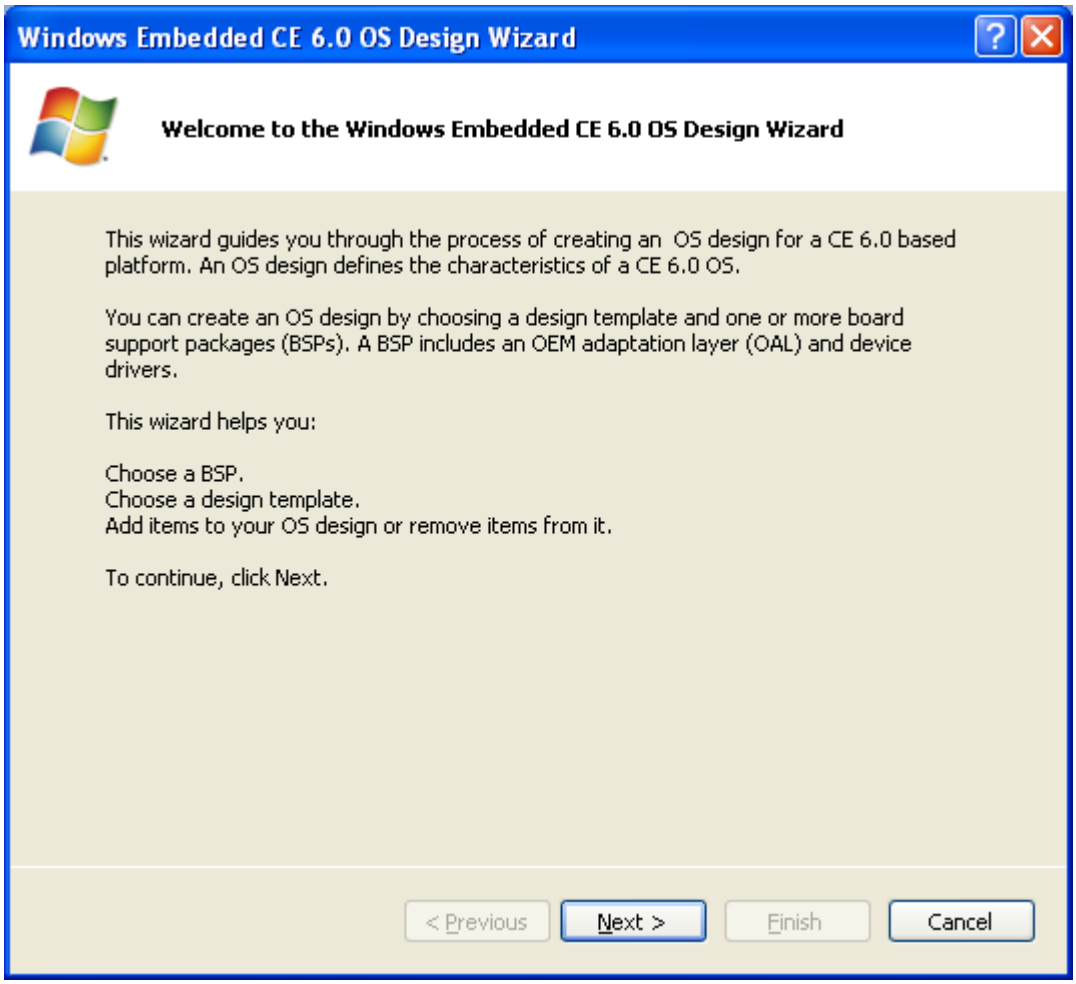
Launch the Visual Studio 2005 New Project dialog.

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



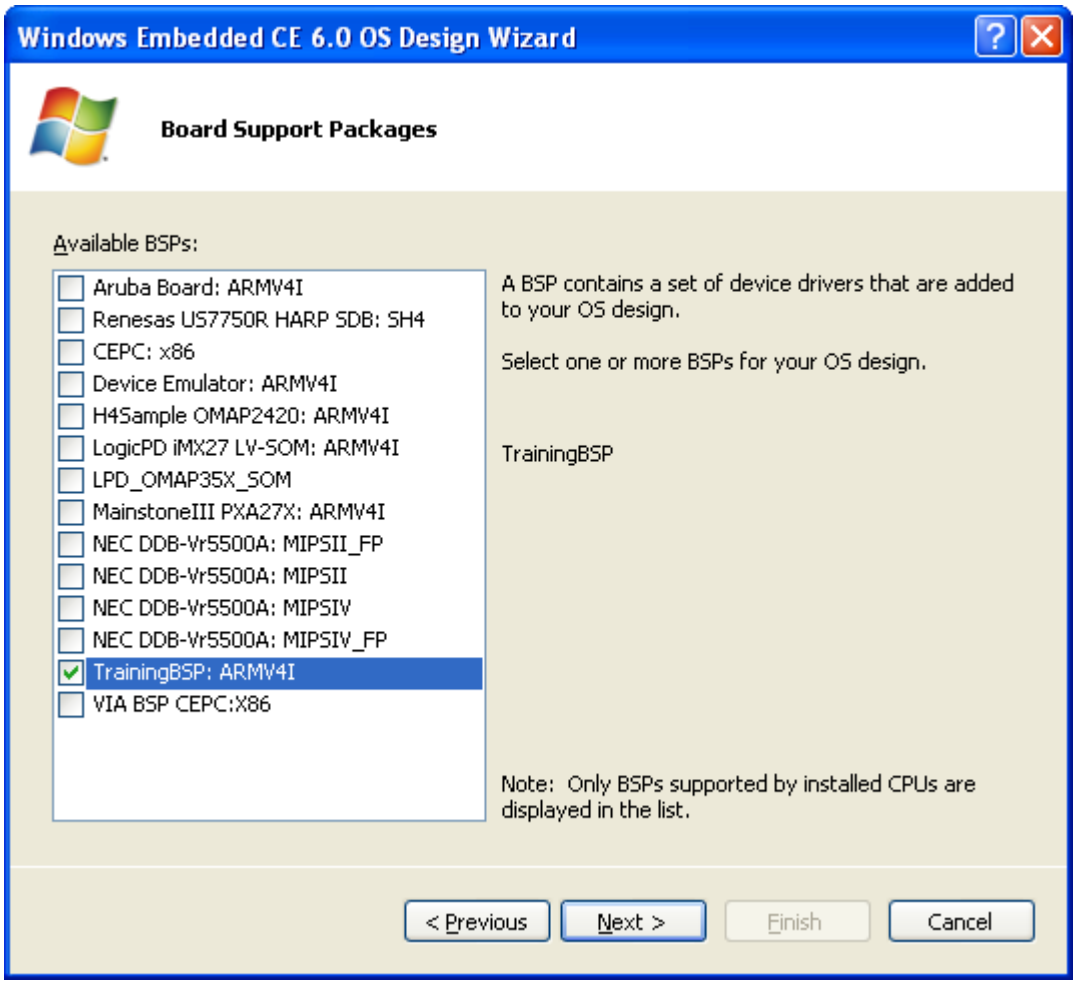
- Select the **Platform Builder for CE 6.0** project type.
- Choose the **OS Design** template
- Enter a name for your project, or go with the default name
- Click **OK**

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.

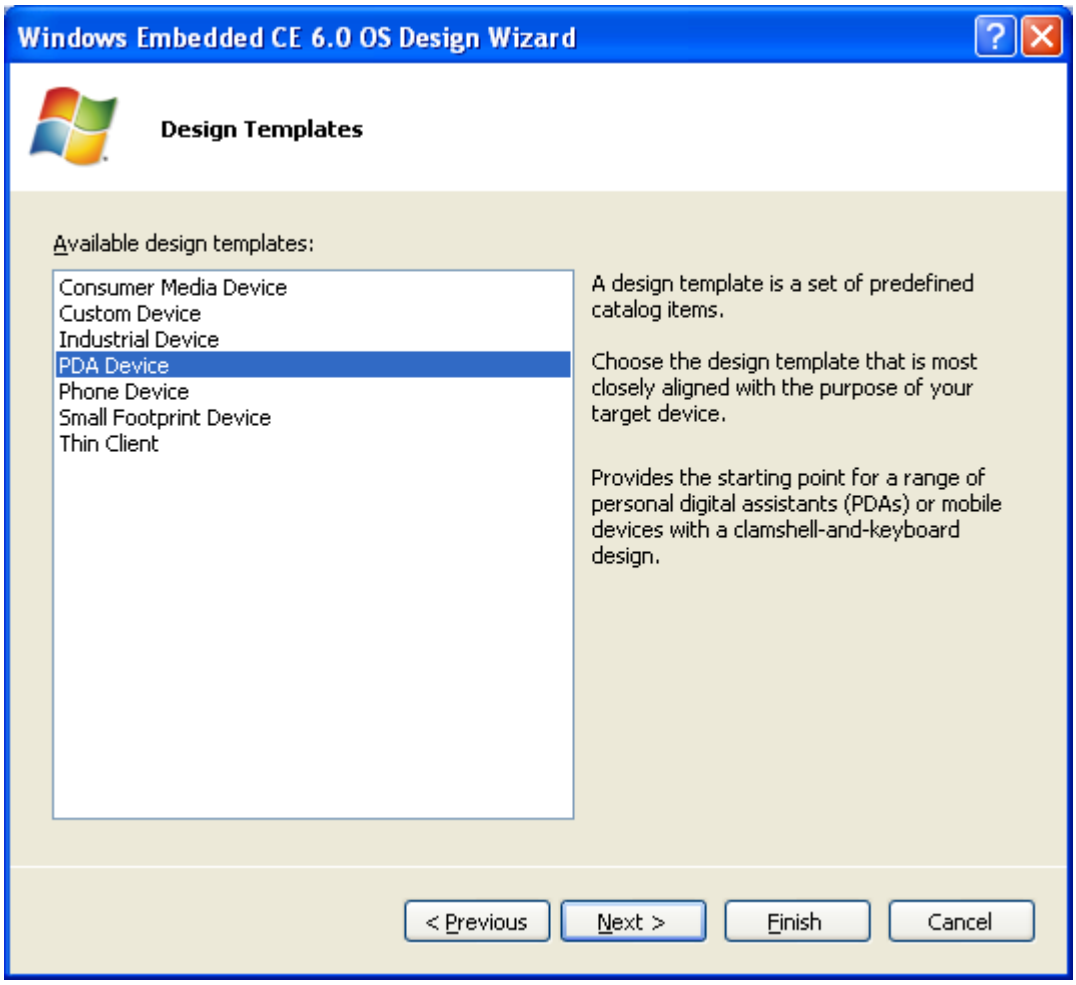
You could select more than one BSP (Board Support Package). For example, it may be useful to select the Device Emulator and a hardware reference platform; the emulation environment can be used by your internal/external application developers to build applications for your device while hardware is still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



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

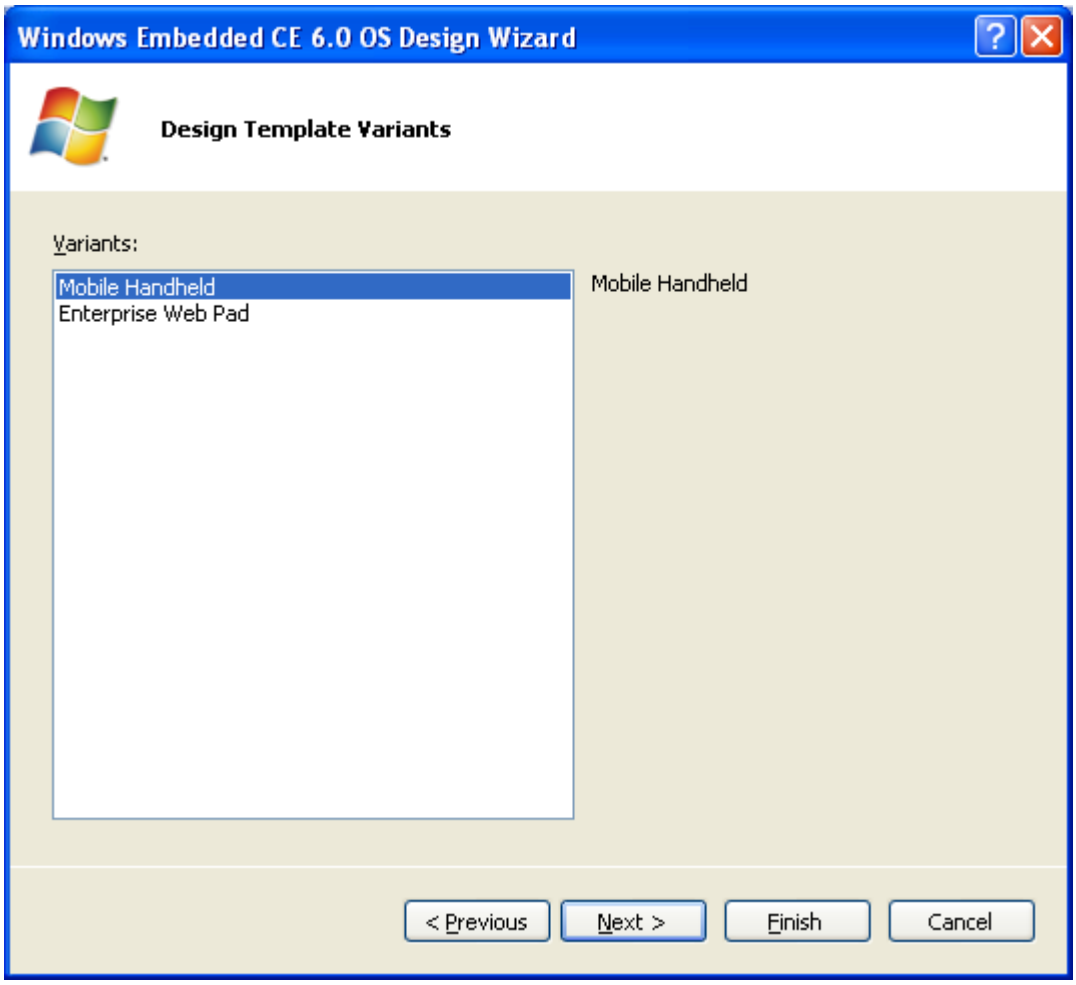
You are now given a number of Design Templates to choose from as the starting point for this project. If none of the options match your needs you can simply select **'Custom Device'** and build the image based on the components you select from the catalog.

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**

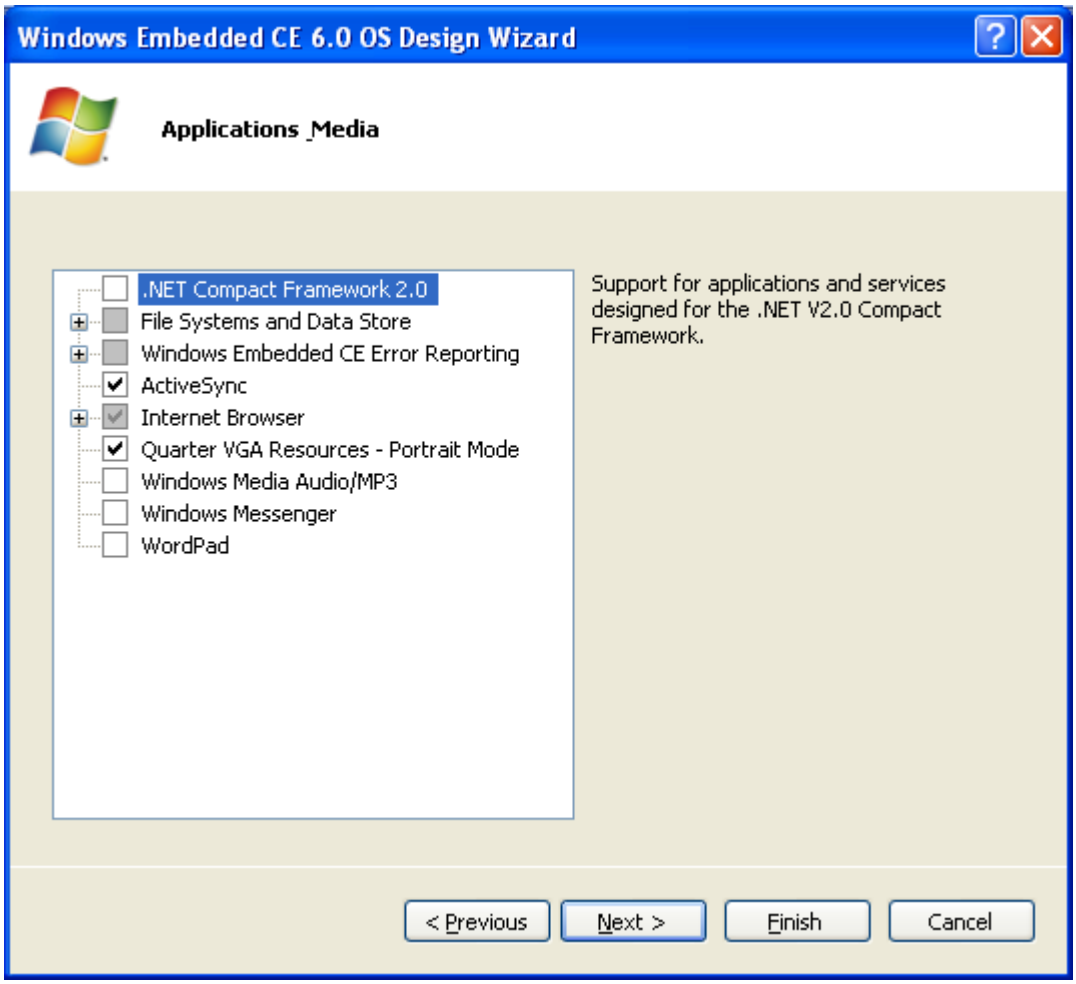
The Wizard now gives you the option of choosing a variant of the Design Template. For this tutorial you will be using the Mobile Handheld variant.



- 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.

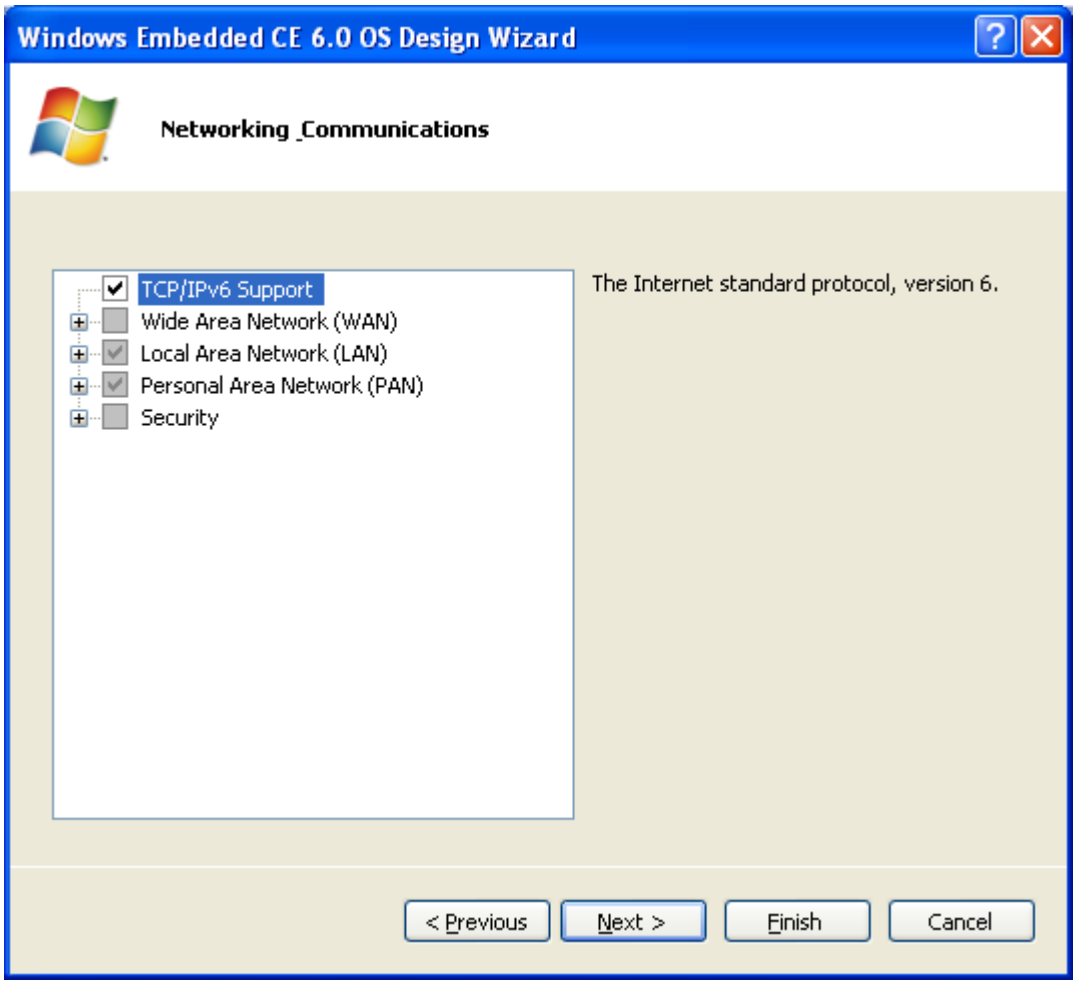
In this case we don't need the .NET Compact Framework. De-select this item.



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

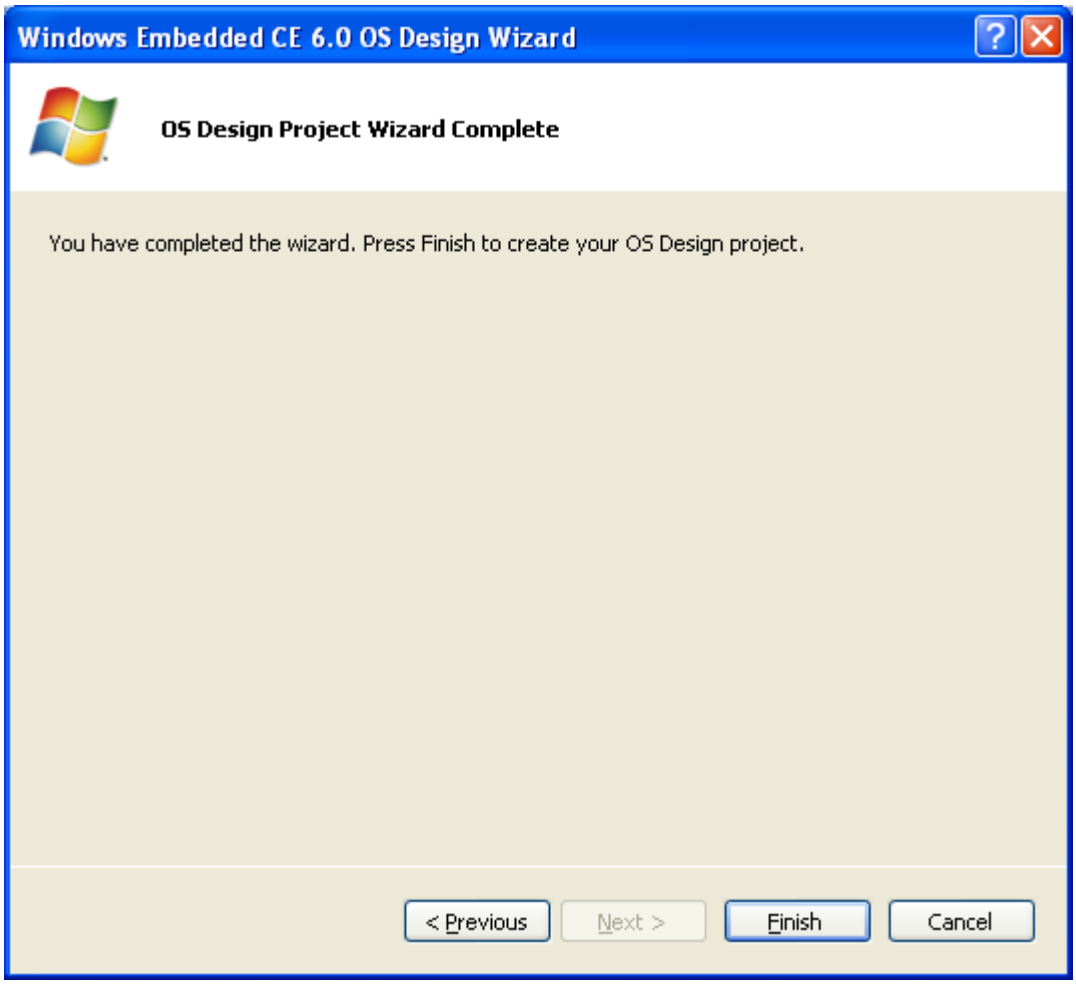
Next comes networking and communications, you can see that Windows CE provides support for Personal, local, and wide area networking through technologies such as Bluetooth, IrDA, Wired and Wireless networking, and VPN.

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

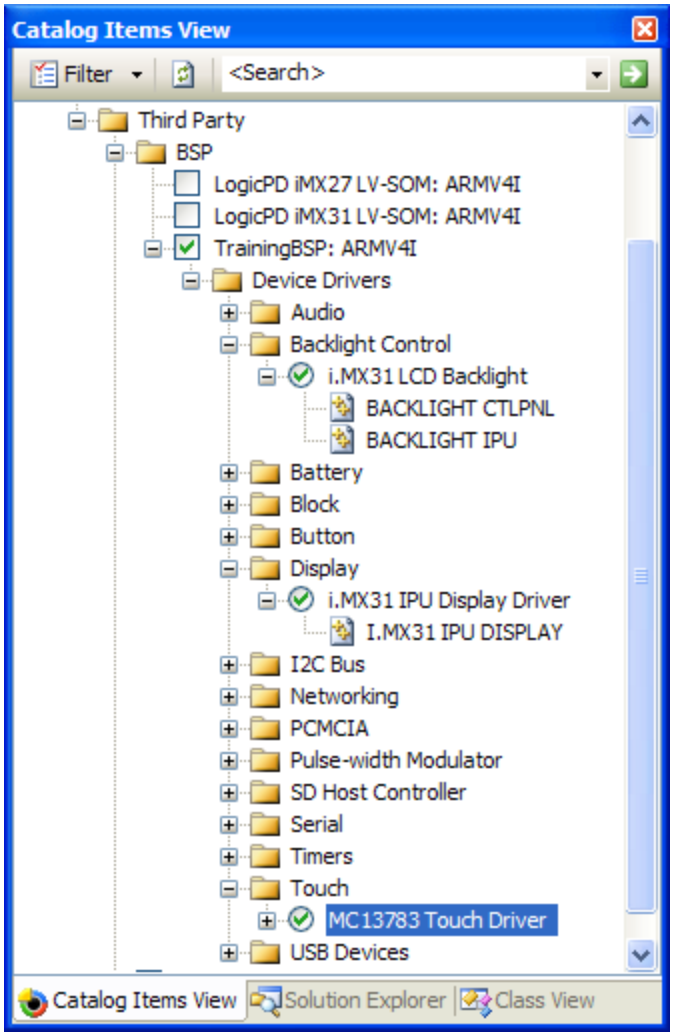
You will now make changes to the items included in your OS Design using the Catalog View.

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:

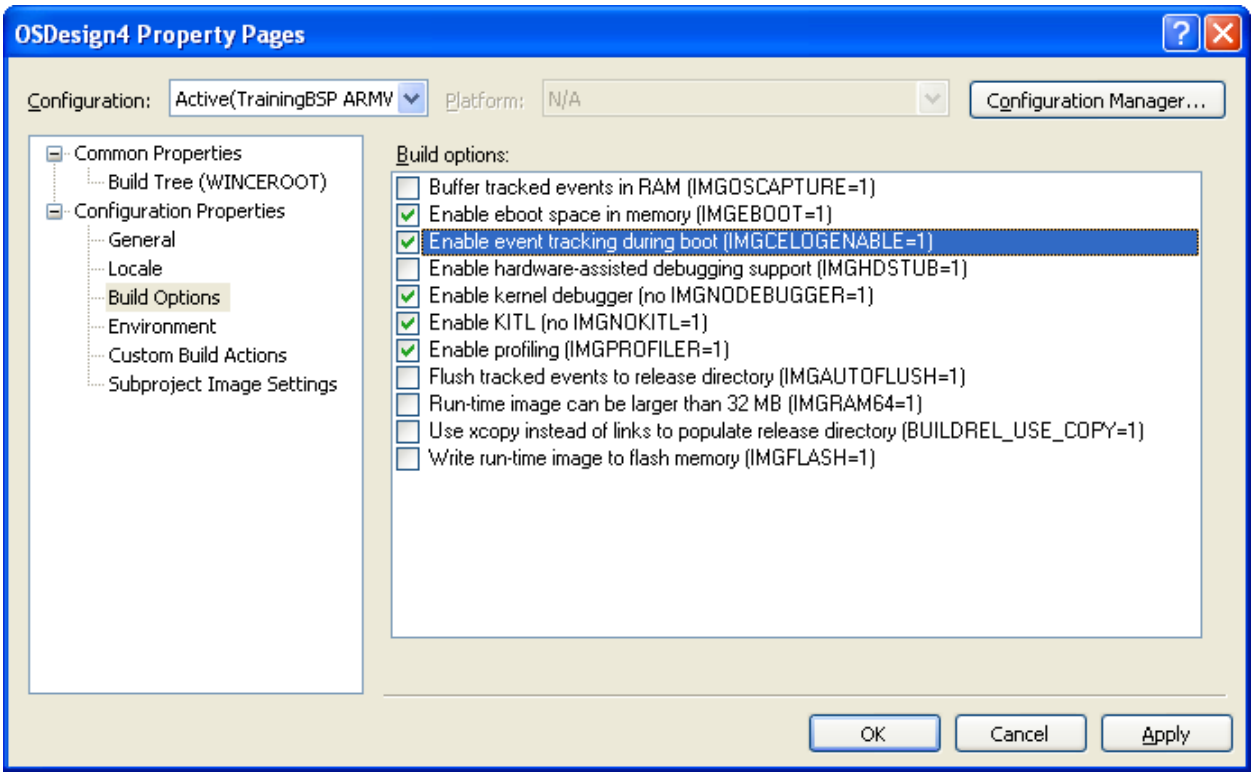


3.2 Enabling Profiling Kernel and Event Tracking

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 **OSDesign4** (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**

4 Adding Applications to the Platform

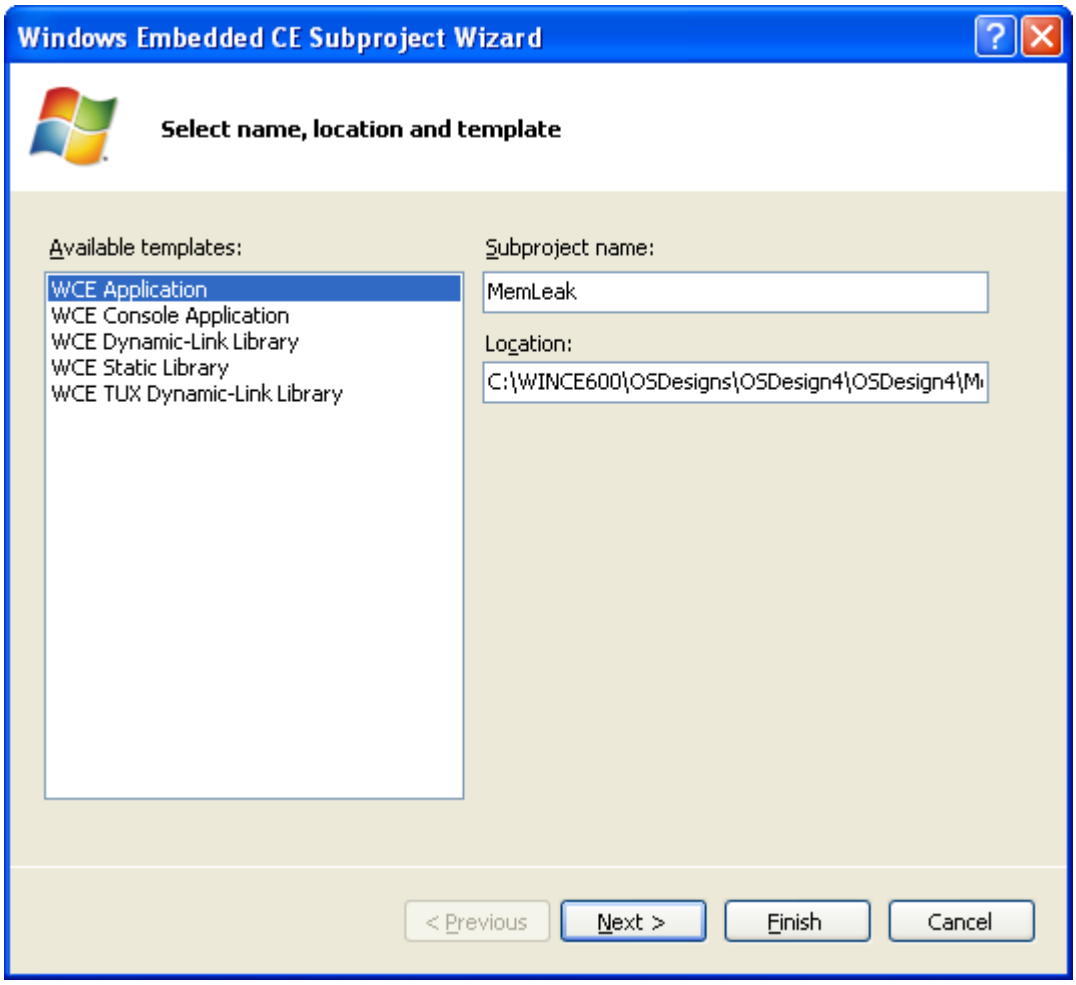
We have two applications to add to our platform, these are MemLeak and Philosophers. Let's now create these applications and add them to the platform workspace.

4.1 MemLeak

We're now ready to add our first application to the platform; this application will be a Windows application with UI. The application will be used to show Debug Zones to display focused debug information from an application or driver, Remote Performance Monitor to monitor memory load within the operating system, and to also show how CELOGDATA can be used to output custom data items into the Kernel Tracker data stream. Memleak is a multi-threaded application which will be created using the Platform Builder 'Application Wizard' – we will examine the code used in this application later in the lab.

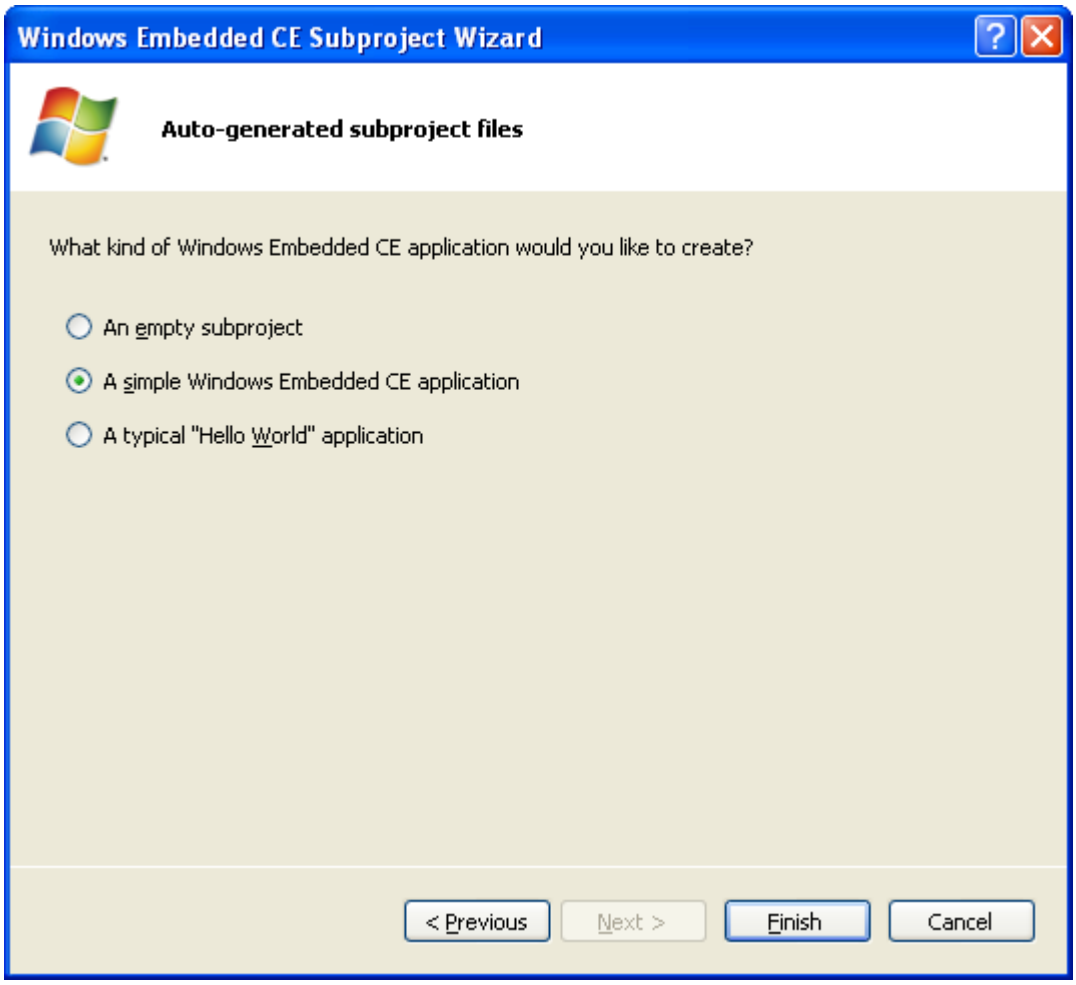
- In the Solution Explorer, right click on the **Subprojects** folder
- Select **Add New Subproject…**

The Subproject Wizard is displayed and shows a number of different subproject types. Each of the available types is described in the Platform Builder online help.



- Choose **WCE Application**
- Type the name of your application – **MemLeak**
- Click **Next**

You will be prompted for the type of application you want to add. There are three types of applications. An empty project assumes that you will add all the source files needed for the project. A “Simple” Windows CE application will create the core source files needed to build the project and will have an entry point of WinMain. Finally, a typical “Hello World” application will output the string “Hello World” to the screen when executed.



- Select **A Simple Windows Embedded CE application**
- Click **Finish**

At this point you have application source written for you that simply exposes the entry point for the application, you are now responsible for writing the rest of the application code.

- Under **Subprojects** Expand **MemLeak**
- Expand **Source Files**
- Double Click on **MemLeak.cpp**

You will now replace the existing ‘skeleton’ code created by the application wizard.

- Select all of the code in the MemLeak application (use CTRL+A or the mouse)
- Delete all of the MemLeak source code.
- Open the file **C:\Labs\Lab4\LabFiles\MemLeak.cpp**
- Copy the contents of this file into the MemLeak.cpp file in visual studio

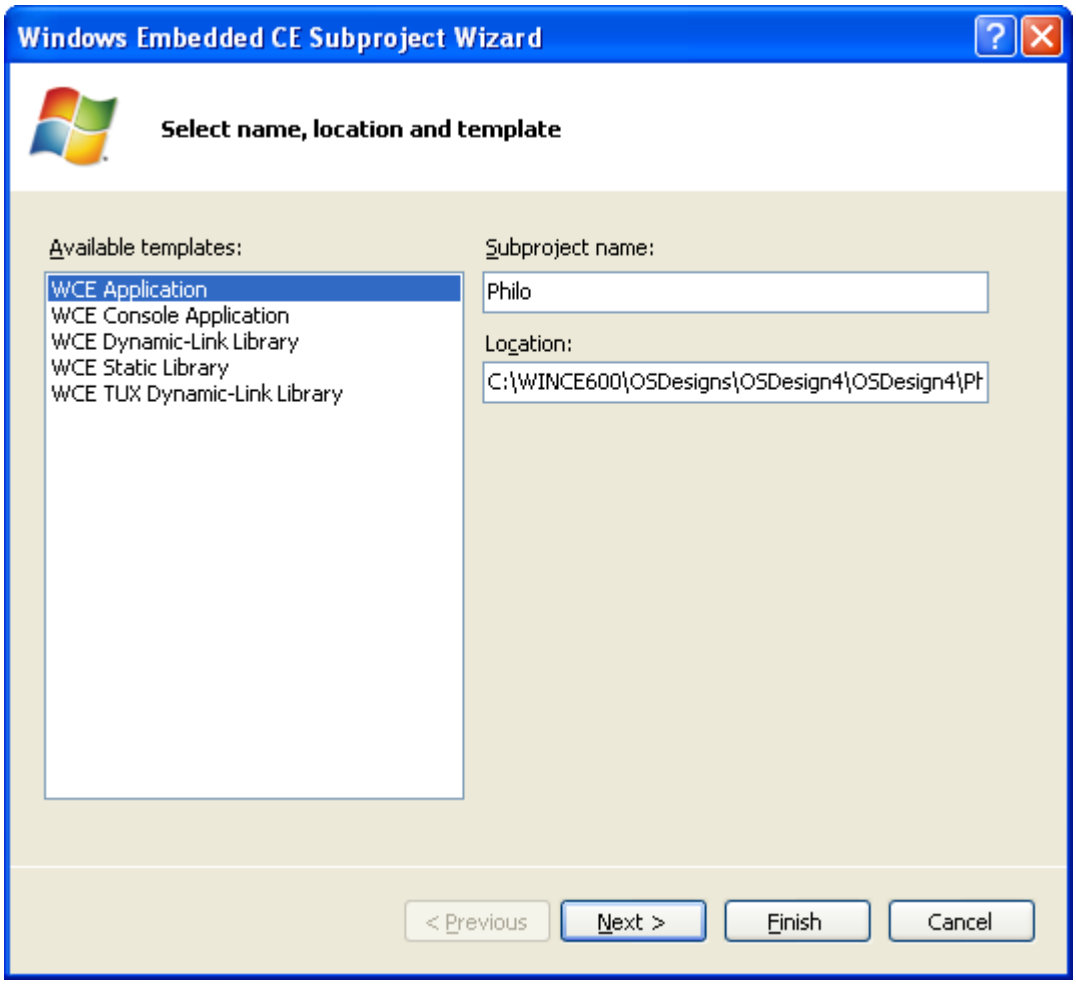
4.2 Philosophers

We’re now ready to add our second application to the platform. In this example we will be using the Dining Philosophers problem which is a classic multi-process synchronization problem. The problem consists of five philosophers sitting at a table who do nothing but think and eat. Between each philosopher, there is a single stick. In order to eat, a philosopher must have both sticks. A

problem can arise if each philosopher grabs the stick on the right, then waits for the stick on the left. In this case a deadlock has occurred, and all philosophers will starve. Also, the philosophers should be fair and not hold onto the chopsticks for too long, therefore each philosopher should be able to eat as much as the rest.

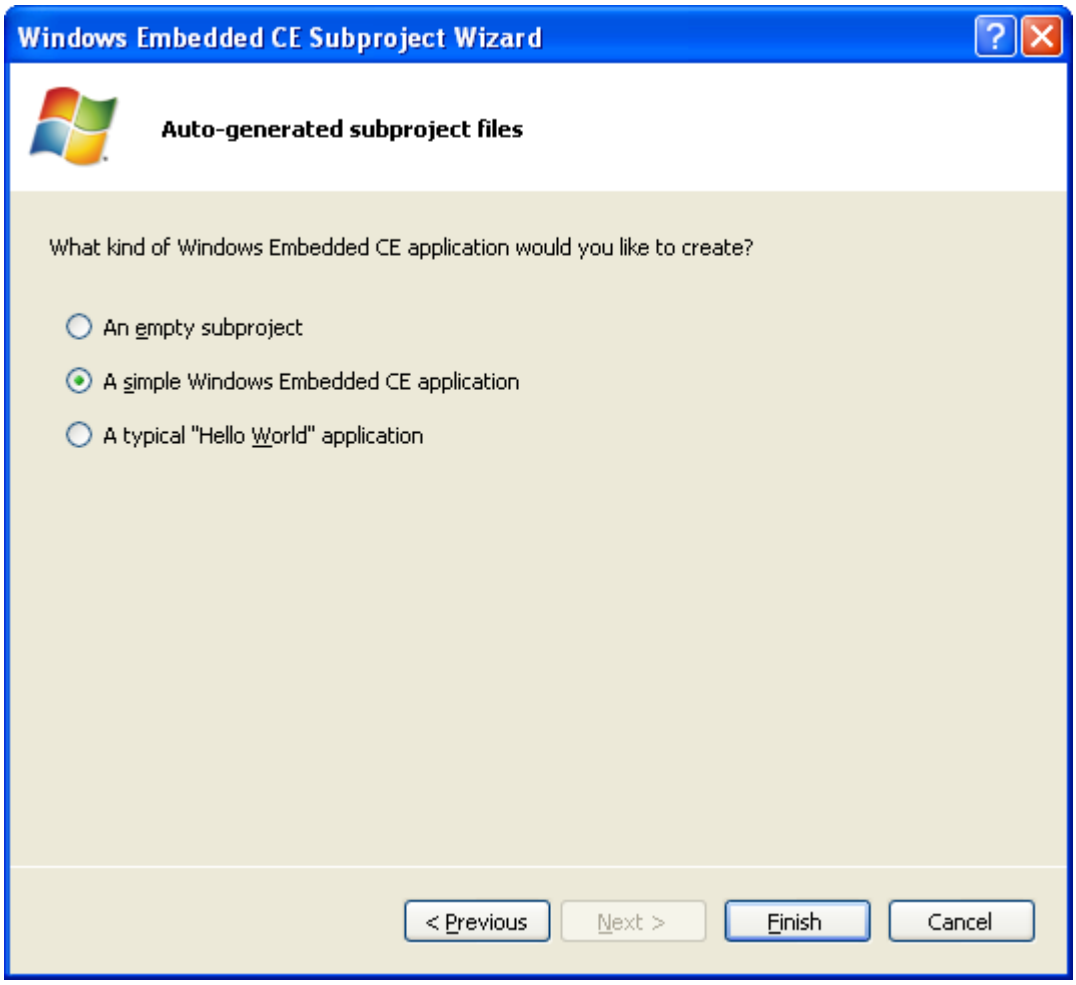
- In the Solution Explorer, right click on the **Subprojects** folder.
- Select **Add New Subproject…**

The Subproject Wizard is displayed and shows a number of different subproject types. Each of the available types is described in the Platform Builder online help.



- Choose **WCE Application**
- Type the name of your application – **Philo**
- Click **Next**

You will be prompted for the type of application you want to add. There are three types of applications. An empty project assumes that you will add all the source files needed for the project. A “Simple” Windows CE application will create the core source files needed to build the project and will have an entry point of WinMain. Finally, a typical “Hello World” application will output the string “Hello World” to the screen when executed.



- Select **A Simple Windows Embedded CE application**
- Click **Finish**

At this point you have application source written for you that simply exposes the entry point for the application, you are now responsible for writing the rest of the application code.

- Under **Subprojects** Expand **Philo**
- Expand **Source Files**
- Double Click on **Philo.cpp**

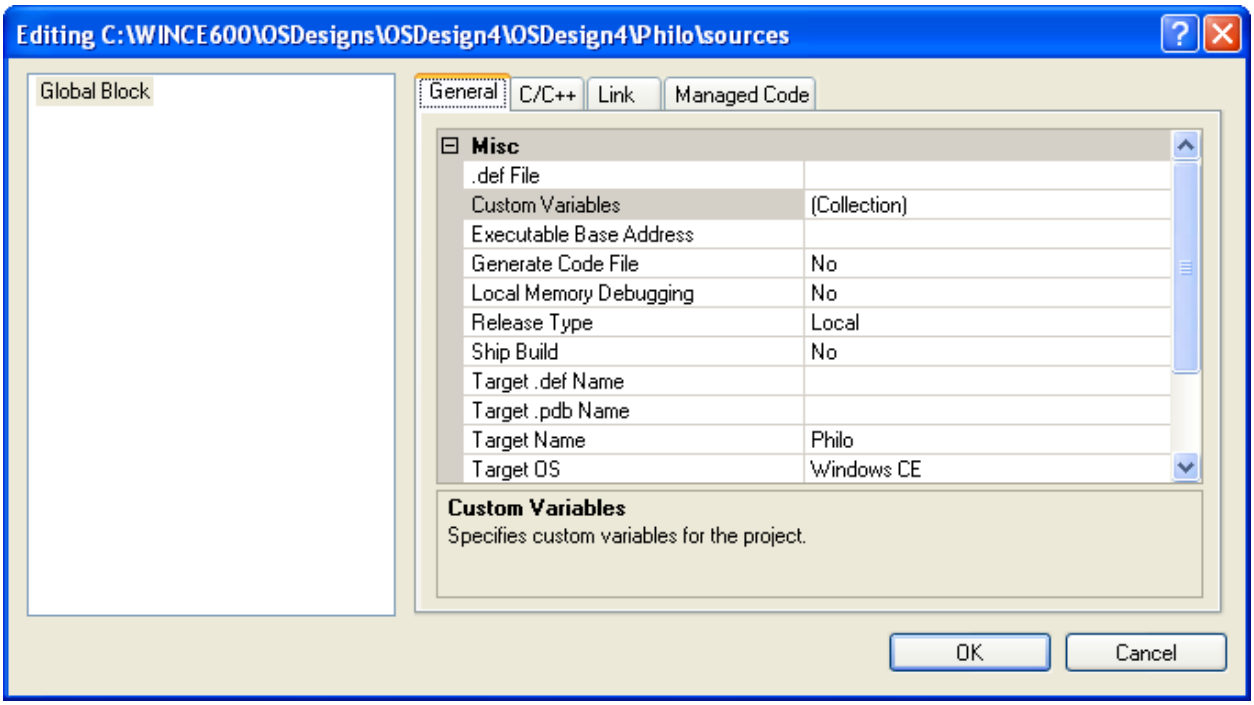
You will now replace the existing ‘skeleton’ code created by the application wizard.

- Select all of the code in the Philo application (use CTRL+A or the mouse)
- Delete all of the Philo source code.
- Open the file **C:\Labs\Lab4\LabFiles\Philo.cpp**
- Copy the contents of this file into the Philo.cpp file in visual studio

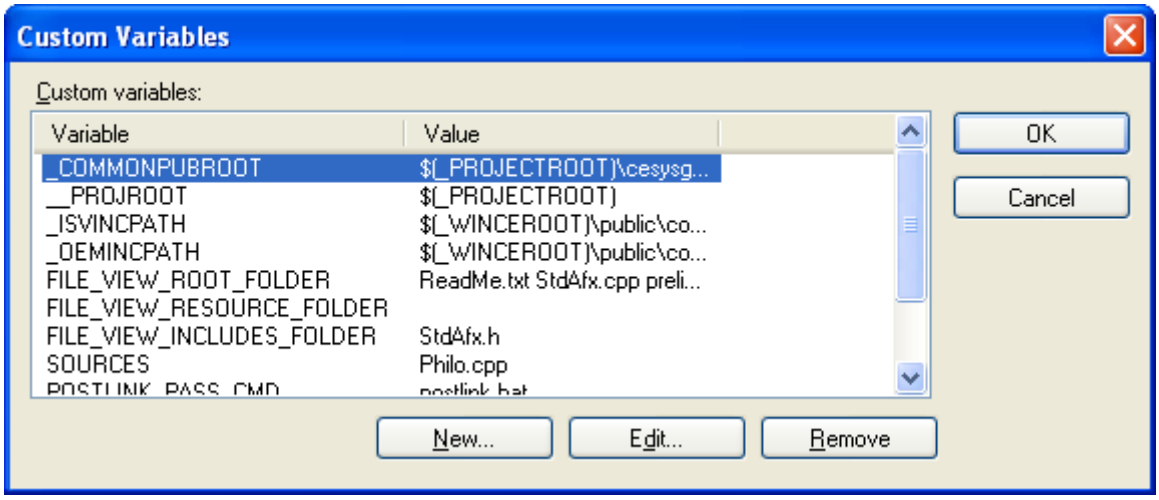
We’re going to run the Remote Call Profiler against the Philosophers application, in order to do this we need to instrument the application.

- In **Solution View**, right click on the **Philo Project** (the top most node of the Philo application)
- Select **Properties**

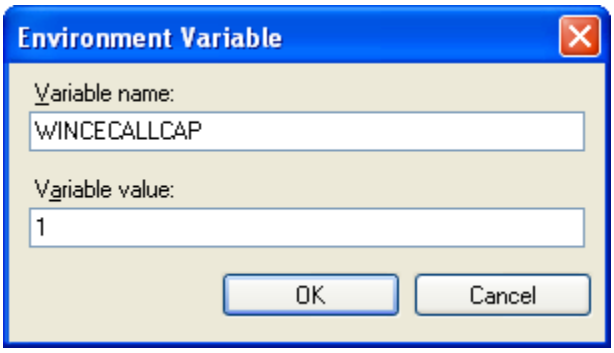
This will display the Project Settings dialog for the Philosophers application.



- Select and open **Custom Variables**



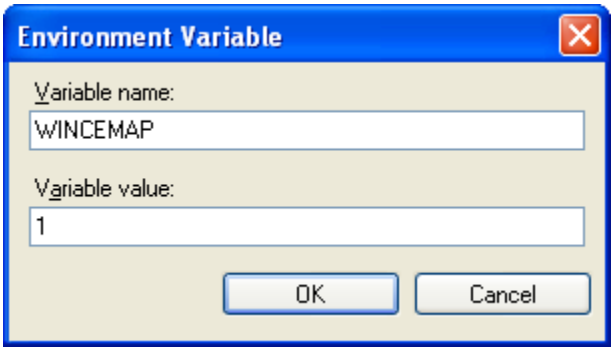
- Select **New**
- Enter **WINCECALLCAP** for the Variable Name
- Enter **1** for the Variable value



- Click **OK**

We also need to enable the generation of a .MAP file for this project.

- On the **Custom Variables Dialog**, click **New**
- Enter **WINCEMAP** for the variable name
- Enter **1** for the Variable value



- Click **OK** on all the open dialogs
- Select **Build** | **Build Solution**

5 Examining Build Errors

In this exercise you will analyze the build results and various build errors to help you understand any errors you may encounter in the real world.

- Observe the text in the build output window from this lab. Identify the **SYSGEN**, **BUILD**, **BUILDREL**, and **makeimg** steps (These keywords will be on the far left of the Output window with a ':' next to them

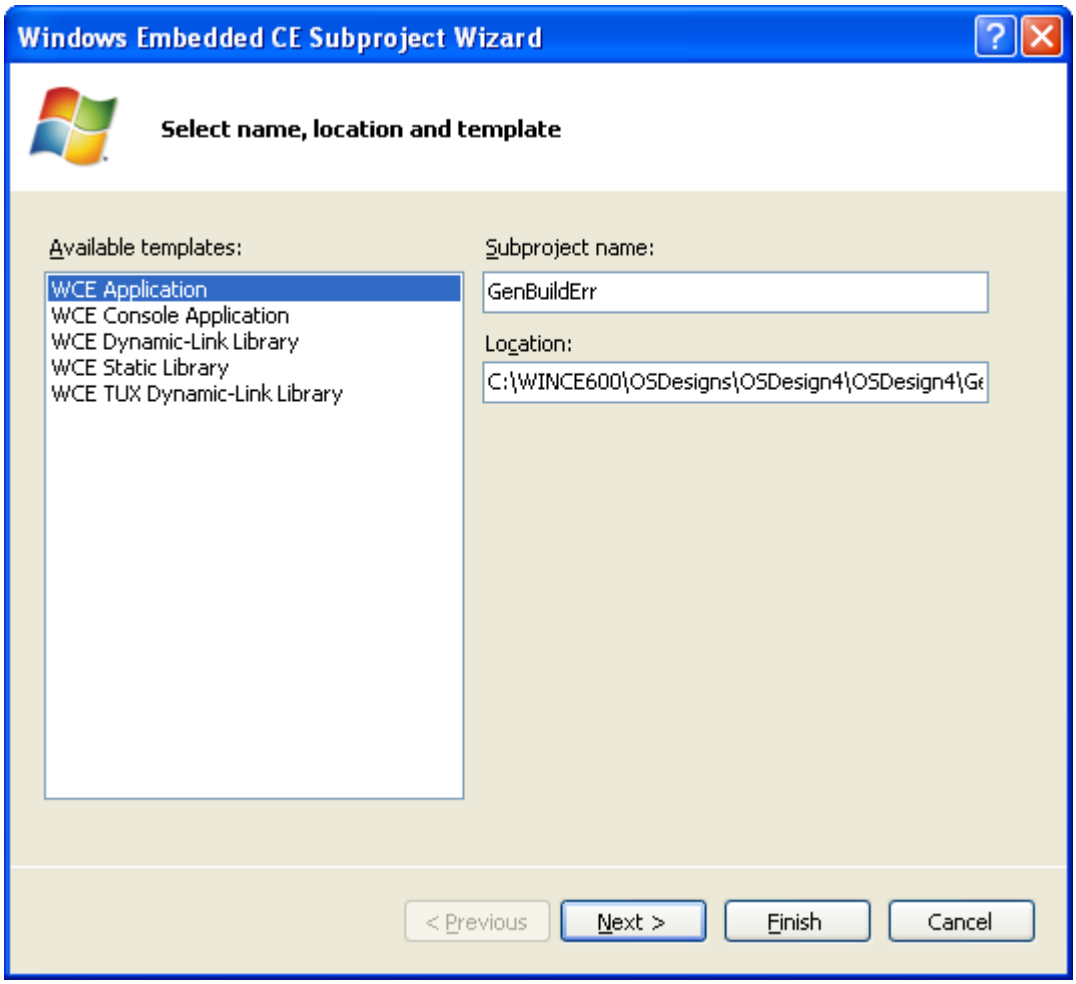
The **SYSGEN** steps start with the line "`Starting sysgen phase for project`". The **BUILD** steps start with a "`Build started with parameters`". The **BUILDREL** steps start with "`C:\WINCE600\build.log`," and the makeimg starts with "`BLDDEMO: Calling Makeimg -- Please Wait`". Any of these phrases may be search for with a **Ctrl + F** to find the beginning of each section.

- Open the **C:\WINCE600** folder using Windows Explorer
- Observe the build.* files. You may open any of them in a simple text editor

The **Build.log** file contains all the information generated during the **BUILD** step. From the end of the **SYSGEN** step to the beginning of the **BUILDREL** step. **Build.wrn** isolates all the warnings that occurred during the build, and a **Build.err** file is generated to show errors if they occur. **Build.dat** contains other information not relevant to this lab.

5.1 Create the GenBuildErr Subproject

- Click on the **Solution Explorer** tab to display the Solution Explorer
- Locate the **Subprojects** node below the OSDesign4 project in the Solution Explorer window
- Right click on the **Subprojects** node and select **Add New Subproject...** The Windows Embedded CE Subproject Wizard will appear
- Select the **WCE Application** template
- Type **GenBuildErr** in the **Subproject name** text box

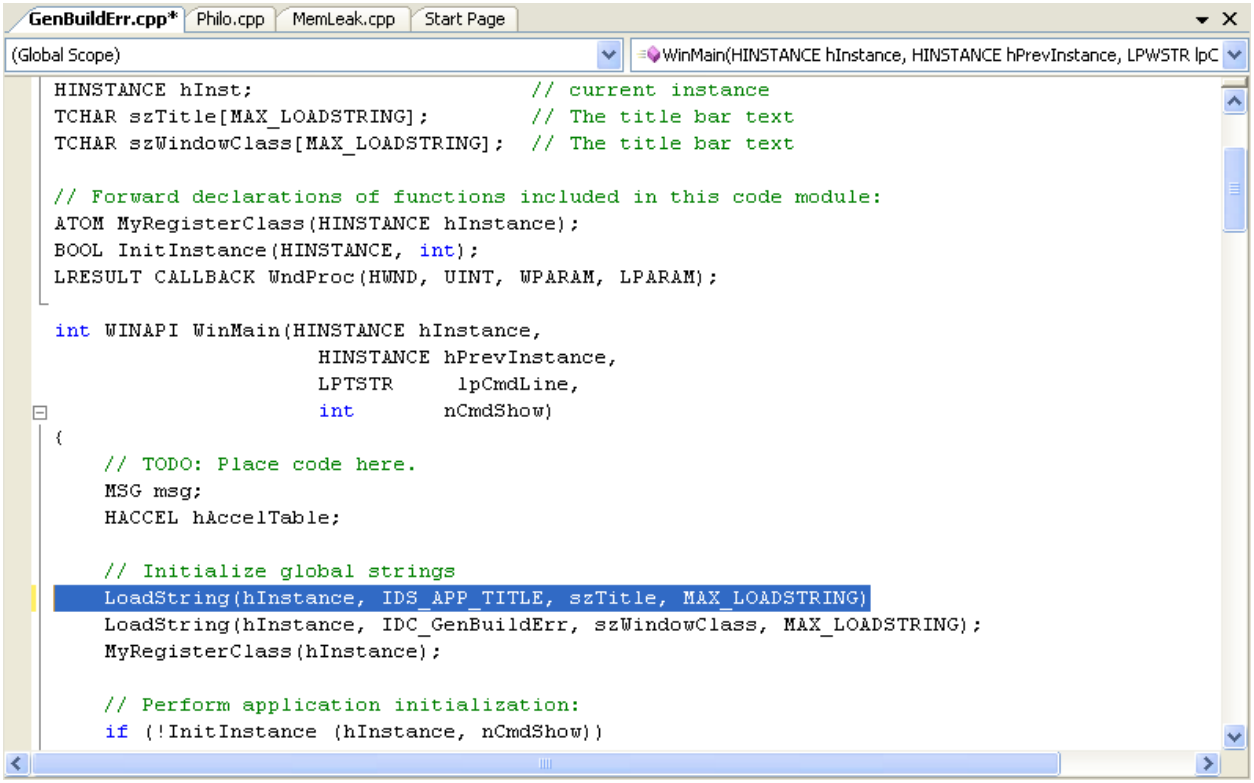


- Click **Next**
- Select **A typical "Hello World" application** and click **Finish**. The wizard will create the files necessary our subproject
- Select **Build** | **Targeted Build Settings** from the Visual Studio menu
- Ensure that **Make Run-Time Image After Building** does NOT have a check mark beside it. If it does, deselect it by clicking on the menu item

This step will prevent the OS run-time image from being rebuilt after we build individual subprojects. This setting will persist for all targeted builds throughout the life of this OS design.

5.2 Generating a Syntax Error

- Right click the **GenBuildErr** subproject in the Solution Explorer and select **Build**. The application will build and should complete with 0 errors and 0 warnings in the build output window
- Navigate to **C:\WINCE600\OSDesigns\OSDesign4\OSDesign4\GenBuildErr** using Windows Explorer
- Observe the build.* files. These files contain the subproject specific build data
- Locate and expand the **GenBuildErr** subproject in the **Subprojects** node of the Solution Explorer
- Expand the **Source files** node of the **GenBuildErr** subproject
- Double click the **GenBuildErr.cpp** file. The file will load in the Visual Studio editor
- Locate the first call to the **LoadString(…)** function near the top of the file
- Delete the semicolon at the end of the line. This will create a syntax error



- Save and close **GenBuildErr.cpp**
- Right click the **GenBuildErr** subproject and select **Build**. The build should fail, with one error
- Notice that a **Build.err** file has been created in you Windows Explorer window

You may need to refresh the Windows Explorer window for the **Build.err** file to appear.

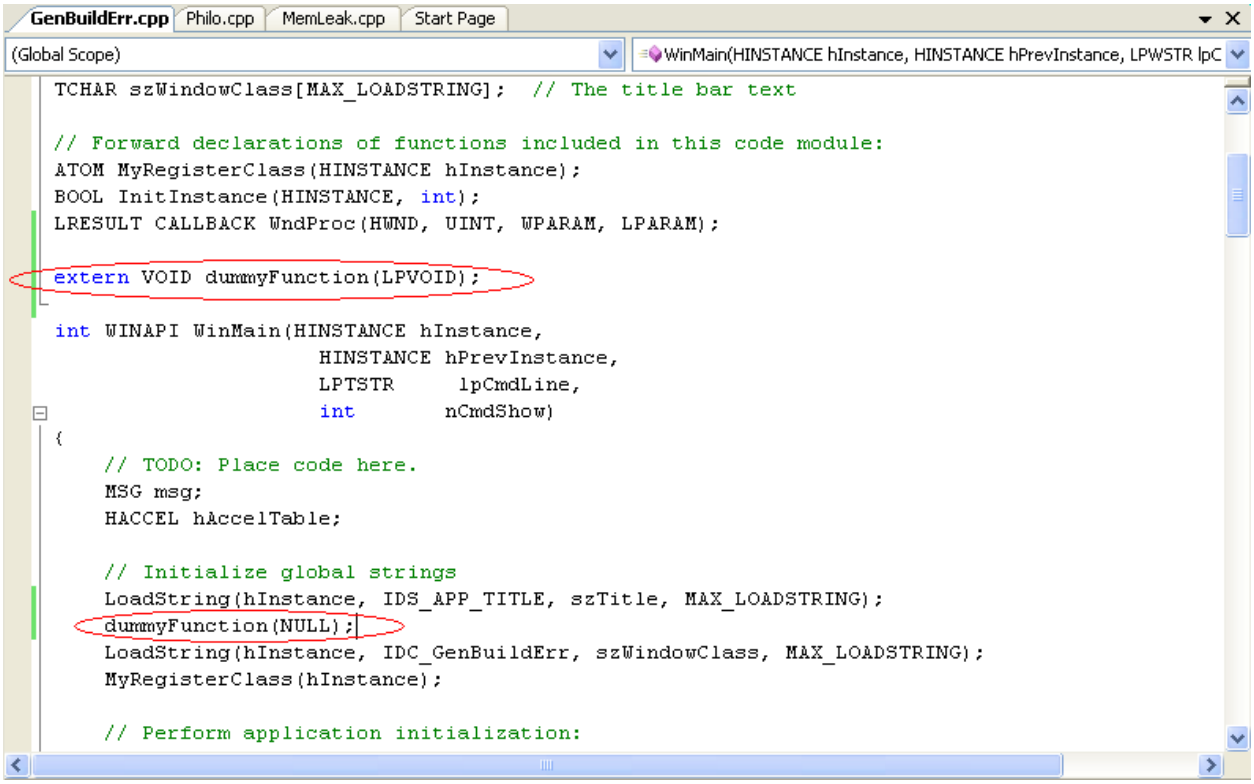
- Open **Build.err** and note the type of error generated
- Switching back to **Visual Studio**, observe the build output window
- Scroll up, until you see the error shown in the **Build.err** file
- Double click on the error. This should reopen **GenBuildErr.cpp** and place the cursor at the line after the missing semicolon

- Replace the semicolon and save.
- Right click the **GenBuildErr** subproject and select **Rebuild**. The build should succeed.

5.3 Generating a Linking Error

- While still in GenBuildErr.cpp, add the following pieces of code where shown:

```
extern VOID dummyFunction(LPVOID);

dummyFunction(NULL);
```



- Save and close **GenBuildErr.cpp**
- Right click the **GenBuildErr** subproject and select **Rebuild**. The build should fail with three errors

By refreshing the Windows Explorer window you should be able to view the new **Build.err** file.

- Locate the errors in the build output window in Visual Studio. Notice that double clicking on the errors will not take you to the issue
- Also note that the first error gives the function name that Visual Studio cannot identify, and a general location

This error is a link error because you have called a function and told Visual Studio with the extern call that it is defined somewhere else, but Visual Studio cannot find the definition.

- Reopen **GenBuildErr.cpp** and delete the lines of code added above
- Save and **rebuild** the subproject

5.4 Generating a Makeimg Error

- Expand the **Parameter files** node of the GenBuildErr subproject.

- Double click the **GenBuildErr.bib** file. The file will load in the Visual Studio editor
- Delete the 'u' in `$(_FLATRELEASEDIR)\GenBuildErr.exe`
- Save and build the **GenBuildErr** subproject
- Note that the build succeeds
- Select **Build** | **Make Run-Time Image** from the Visual Studio menu. Data should be output to the build output window, and the **makeimage** should fail. Note that no Build.err file is generated because the **makeimage** step occurs after the build process
- Observe the errors in the build output window
- Reopen **GenBuildErr.bib** and undelete the 'u' in **GenBuildErr.exe**
- Save and rebuild the subproject
- Select **Build** | **Make Run-Time Image** from the Visual Studio menu. The **makeimage** should succeed
- Select **Build** | **Targeted Build Settings** from the Visual Studio menu
- Enable **Make Run-Time Image After Building** by clicking on this item

You have now successfully debugged various build errors!

6 Setting Breakpoints

You will now set a breakpoint that will be hit once the OS image has been downloaded. The platform has a file system driver, so we can use the Platform Builder IDE to locate the code that's loading the driver and set a breakpoint on this code.

- Within the Solution Explorer, expand the following folder in this order
  - OSDesign4
  - C:\WINCE600
  - PLATFORM
  - COMMON
  - SRC
  - Soc
  - MX31_LDP_V1
  - DRIVERS
  - MXARM11
  - BACKLIGHT
  - DRIVER
  - Source Files
- Open **backlight.cpp**
- Search for the function `UINT32 BKL_Init(UINT32 dwContext)`
- Set a breakpoint on the following line (by clicking on the line and pressing the F9 key)

```
InitializeCriticalSection(&cs);
```



```
    DEBUGMSG(ZONE_FUNCTION, (TEXT("BKL_INIT() +!\r\n")) );

    InitializeCriticalSection(&cs);

    // get our activation information
    dwStatus = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCTSTR)dwContext, 0, 0,
                            &hDriverKey);
```

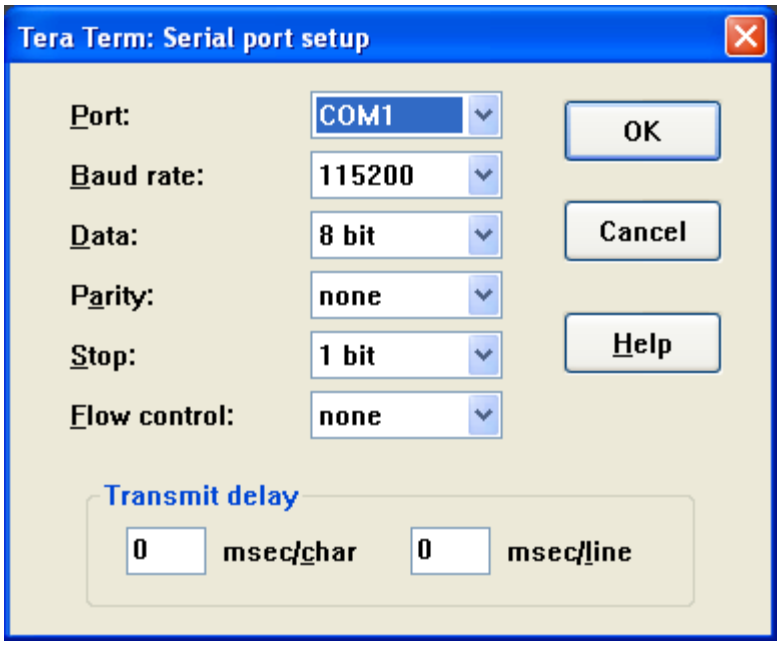
We will now download the image and examine how to interact with the breakpoint.

7 Download the Platform Image

7.1 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.

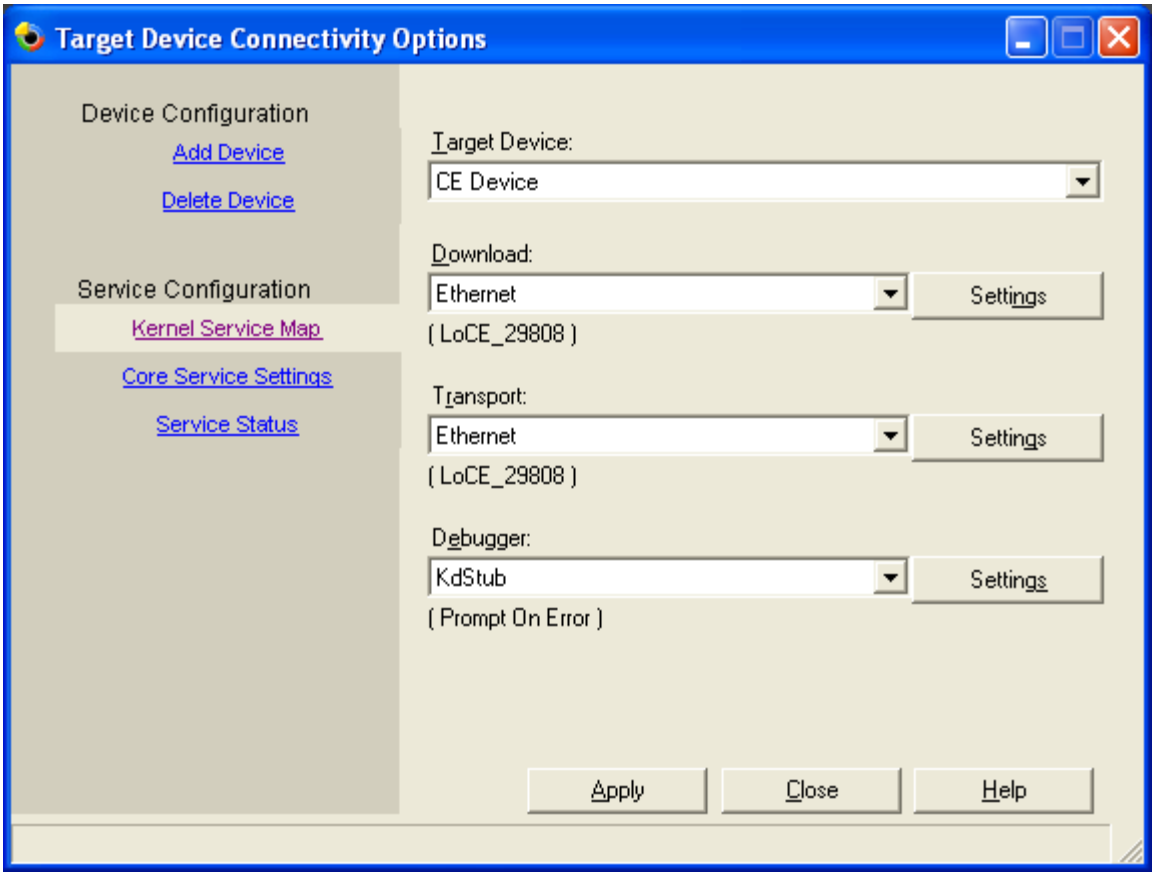
- Open **TeraTerm** by double clicking the icon on the desktop.
- Verify the serial port settings by clicking the **Setup** | **Serial Port** menu item (Your physical Port may differ):



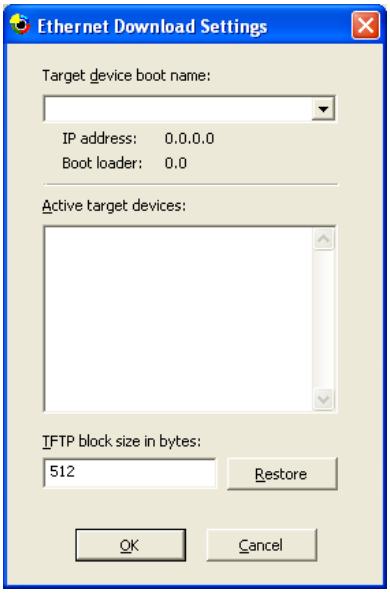
Ok now that you have configured your serial terminal we can continue to setup the device and Platform Builder. Switch back to your Visual Studio 2005 instance.

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

The following dialog will be displayed:



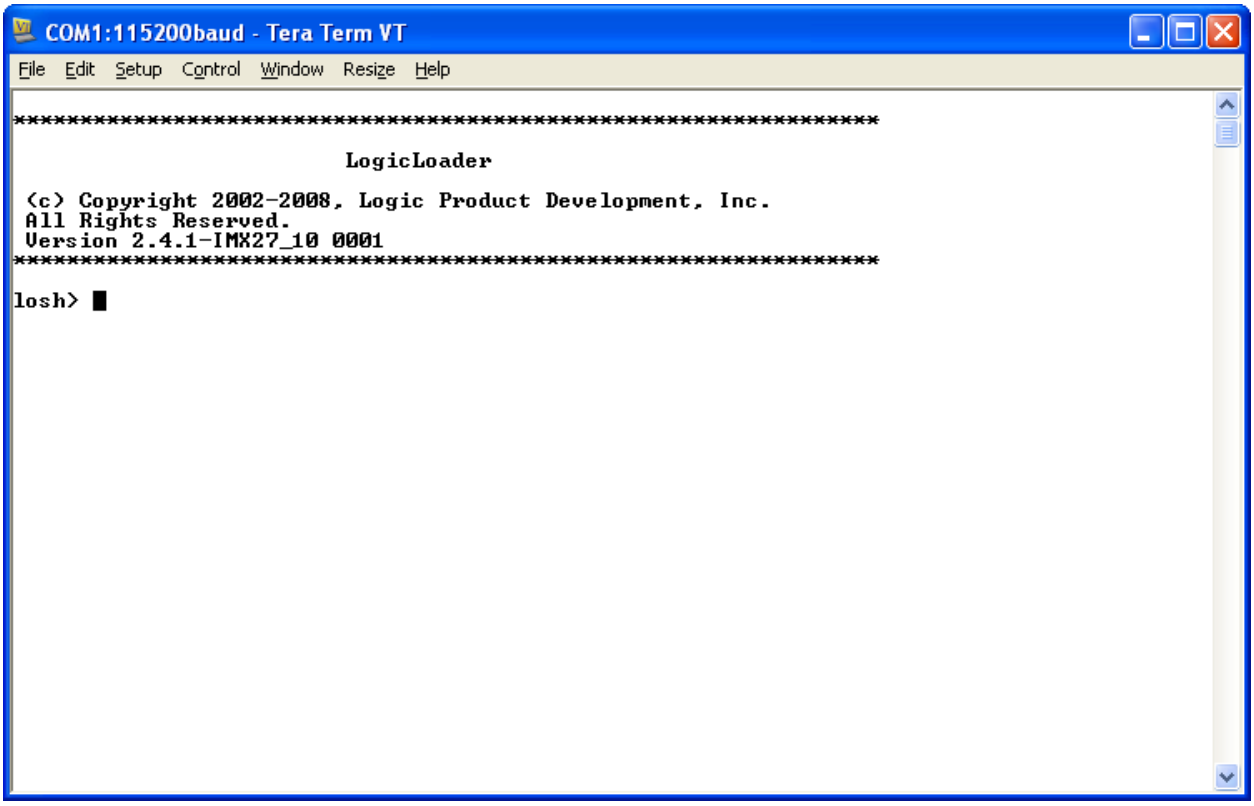
- In the **Download** box select **Ethernet**
- In the **Transport** box select **Ethernet**
- In the **Debugger** box select **KdStub**
- Click on the **Settings** box next to the **Download**: dropdown box. A box like this will appear:



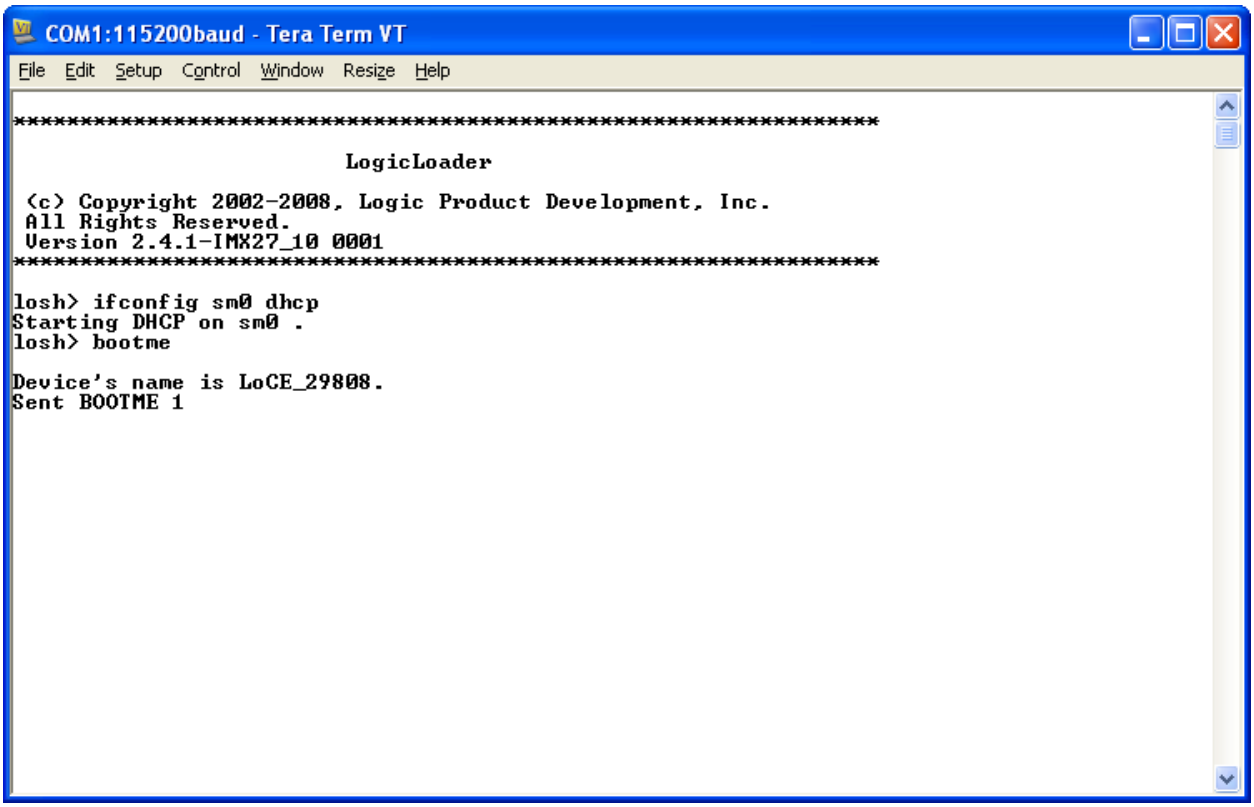
- Leave this box open and move on to the next step

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

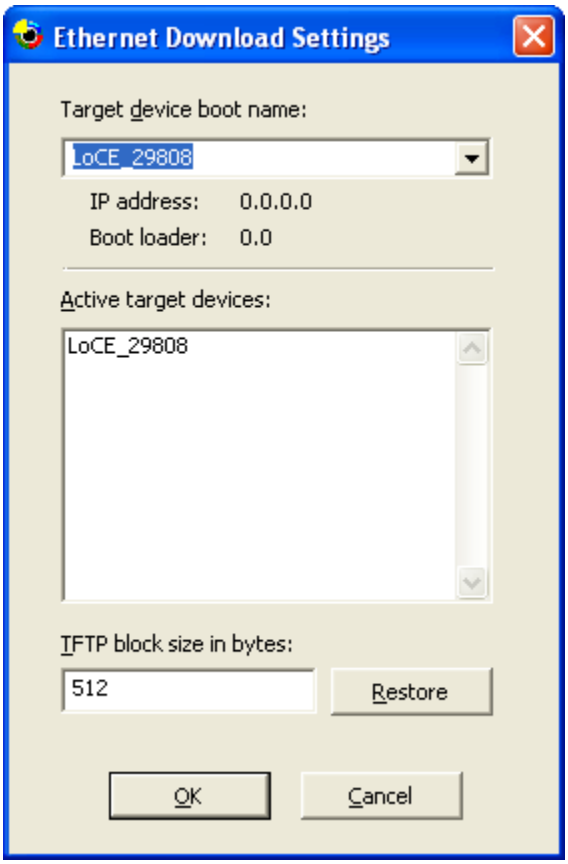
- Power on the device.
- You should see a **losh** prompt display.



- 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.

7.2 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 the **F5** key on the keyboard to continue execution

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

8 Working with Breakpoints

You've set a breakpoint in shared source for the backlight driver. The breakpoint will get hit before the OS has completed loading. Once the breakpoint has been hit you can do a number of things, including View memory, View Registers, Call Stack, Single Step Code, and View local variables.


```
    DEBUGMSG(ZONE_FUNCTION, (TEXT("BKL_INIT() +!\r\n")) );

    InitializeCriticalSection(&cs);

    // get our activation information
    dwStatus = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCTSTR)dwContext, 0, 0,
                            &hDriverKey);
```

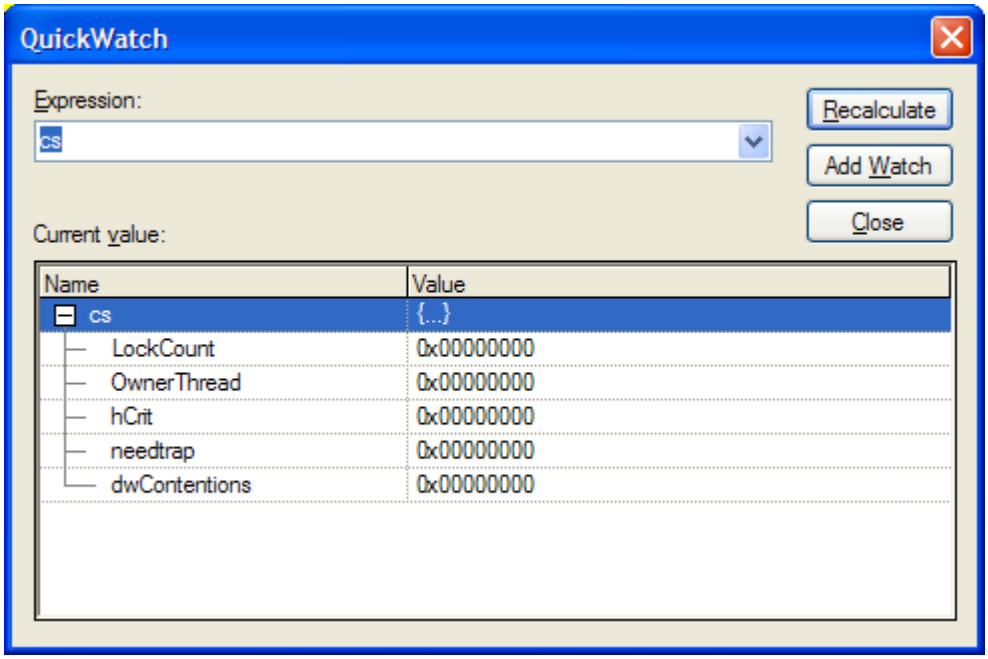
You can see that a breakpoint has been hit because you have a yellow “Current Instruction” pointer displayed over our breakpoint.

- Select **Debug** | **Windows** | **Call Stack**

The call stack window shows the function chain that brought you to the current breakpoint.

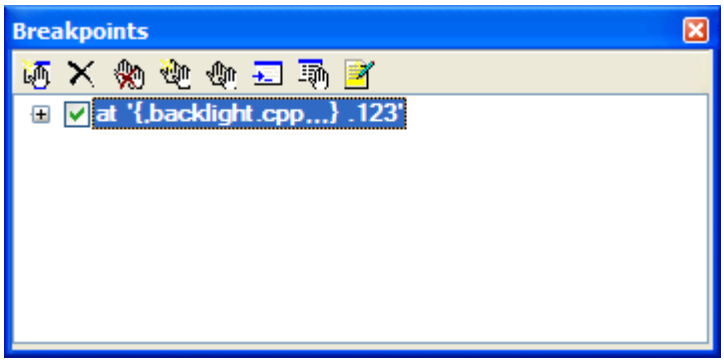
- Back in the source window **backlight.cpp**, right-click on **cs** (it is shown on the current line where the breakpoint was hit) and select **QuickWatch**

The following data will be displayed:



- This shows the contents of this variable

- Select **Close**
- Select **Debug** | **Windows** | **Breakpoints** (or press ALT+F9 on the keyboard).



- Click the **Clear All Breakpoints** ( ) toolbar button
- **Close** the Breakpoint Dialog
- Select **Debug** | **Start** to allow the OS to finish loading

At this point, the operating system will finish loading.

9 Remote Tools & Memory Leaks

9.1 Remote Performance Monitor

The Remote Performance Monitor is a graphical tool for measuring the performance of a remote Windows CE system. You can view the behavior of performance objects, such as processors, memory, threads, and processes.

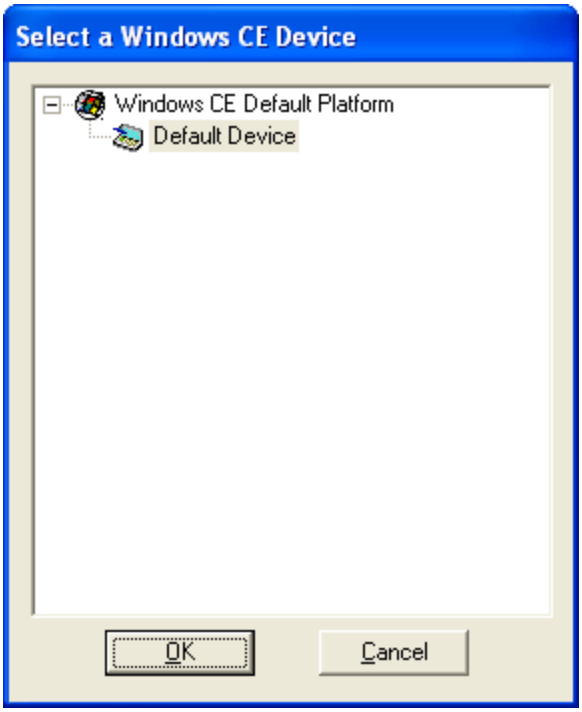
We will be using Remote Performance Monitor to monitor memory load on the reference board, this is useful to track memory load over a period of time, if memory load increases over time this could be normal behavior of the currently running processes and drivers, or this could indicate a memory leak. We will use the MemLeak application to show how Remote Performance Monitor can be used.

To start PerfMon:

- Within Visual Studio, select **Target** | **Remote Tools** | **Performance Monitor**

You will be prompted for the device to connect to.

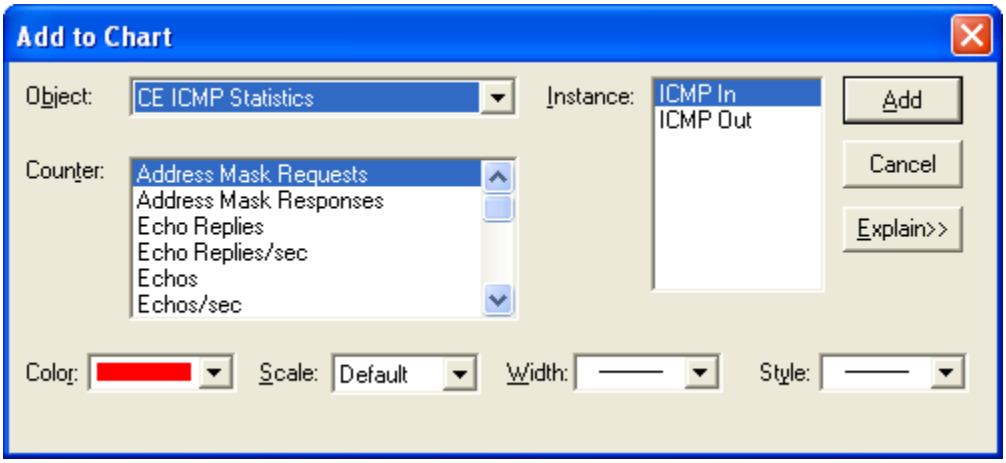
- Select **Windows CE Default Device**
- Click **OK**



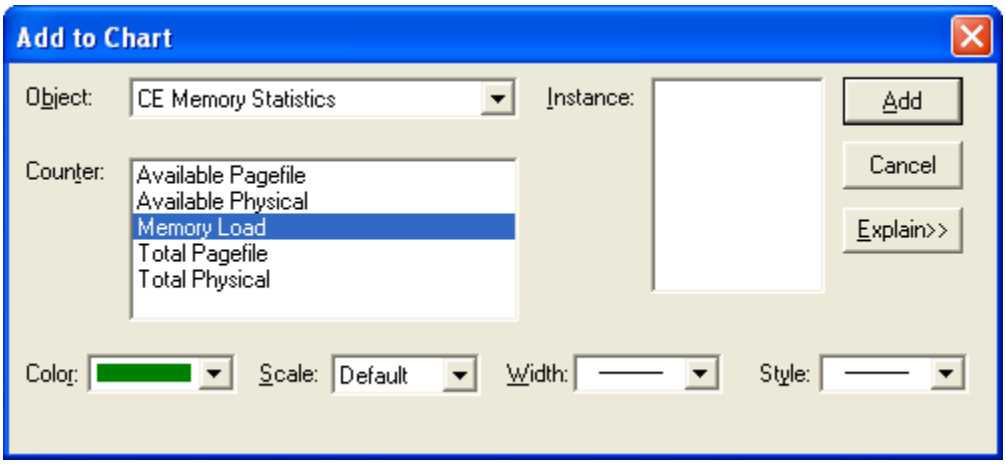
Once connected we will need to select which performance items we wish to monitor, in this case we will be monitoring memory load.

- Select **Edit** | **Add to Chart** (or CTRL+I)

The Default objects displayed are CE ICMP Statistics:



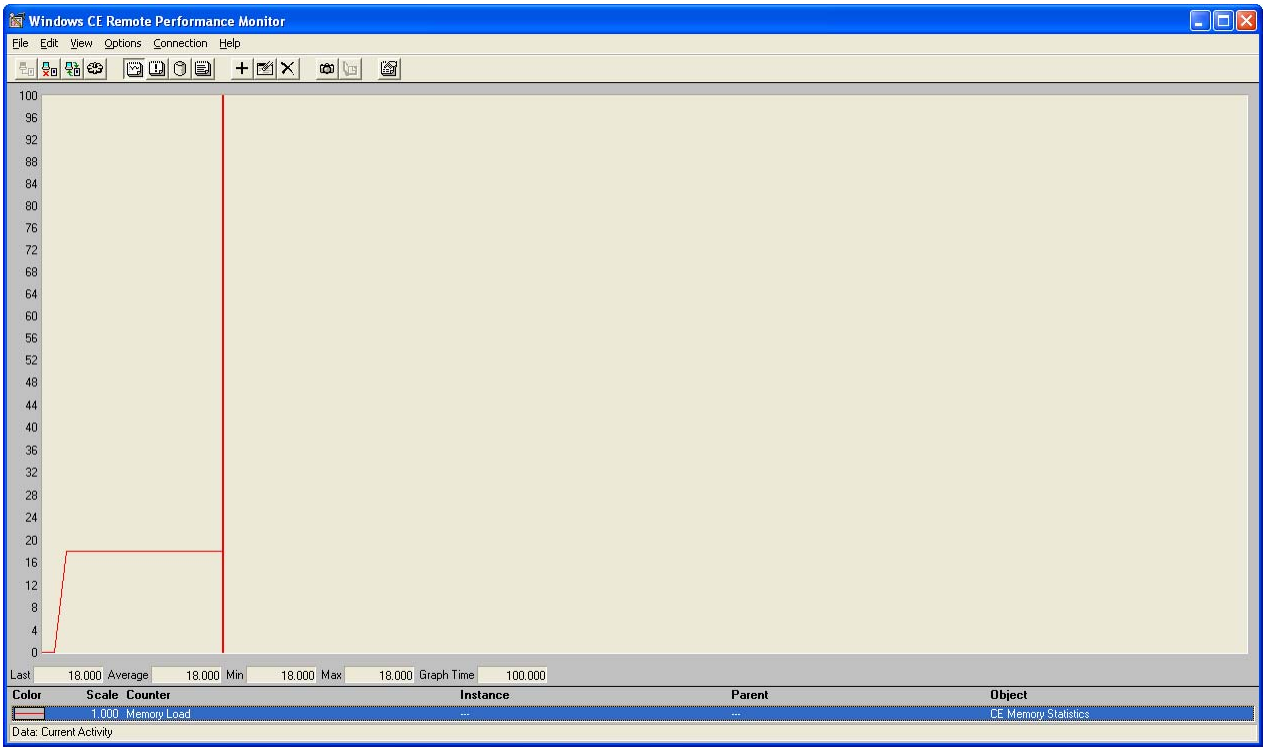
- Scroll the **Object** ComboBox to **CE Memory Statistics**
- Scroll the **Counter** ComboBox to **Memory Load**



- Click **Add**
- Click **Done**

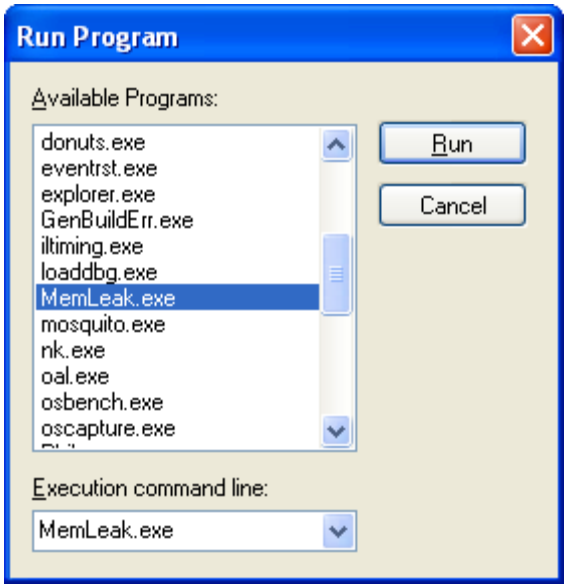
Performance Monitor will now display the current memory load, which should be flat. We will now run the **MemLeak** program which leaks memory and will show the leak on the Performance Monitor chart.

You can see from the Remote Display Monitor image below that the o/s is running at about 18% memory load.



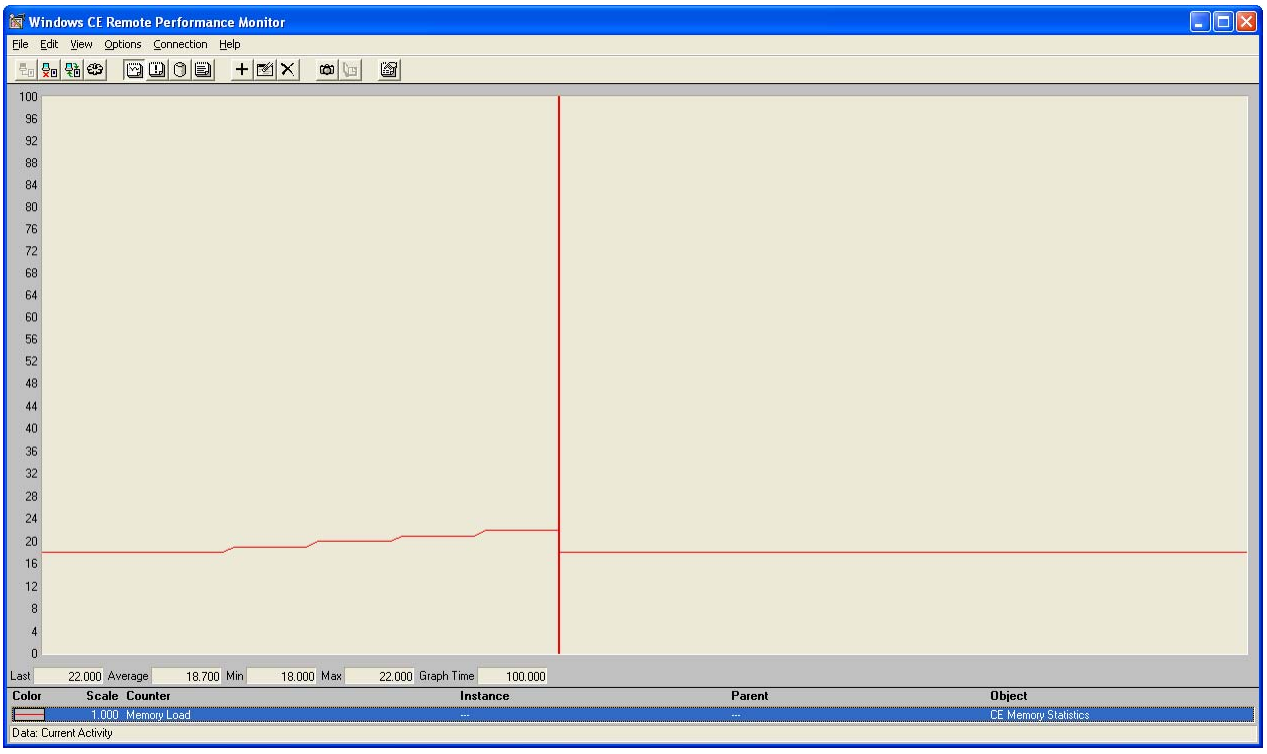
- Switch back to Platform Builder
- Select **Target** | **Run Programs…**

You will be prompted with a dialog showing all of the executable programs which are in the build directory for the workspace.



- Locate and select **MemLeak.exe** from the list of Available programs
- Click **Run**
- Switch back to Performance Monitor

You will notice that the graph clearly shows a change in the memory load in the system – we can also obtain memory load information directly from our application using the GlobalMemoryStatus API.



We're done with Performance Monitor so this can now be closed.

- Select **File** | **Exit**

With the Remote Performance Monitor tool, you can observe the behavior of performance objects such as CPUs, threads, processes, and system memory. Each performance object has an associated amount of performance counters that provide information about device usage, queue lengths, and delays, and information used to measure throughput and internal congestion. Remote Performance Monitor provides the ability to track current activity on a target device and the ability to view data from a log file.

9.2 Enabling Debug Zones in the MemLeak Application

Now that our **MemLeak** application is running, we can explore debug zones. Zones are useful in allowing a developer to dynamically alter the verbosity of debug message output to the kernel debugger. This is helpful when trying to narrow in on the source of a problem.

Debug zones and the Windows CE Console Debug Shell tool (Cesh.exe) provide the ability to selectively turn the debugging message output from your code on and off by using macros. This allows you to trace execution of your code without halting the OS. Tracing is a simple and non-intrusive way of catching problems in your code without causing the OS to stop responding. Debug Zones can be enabled either through Target Control, or through the Platform Builder IDE.

Debug zones are implemented by declaring a DBGPARAM structure in your source code. DBRPARAM is defined in dbgapi.h. DBGPARAM contains three elements – we can see this through the structure definition from dbgapi.h

```
typedef struct _DBGPARAM {

  WCHAR     lpszName[32];     // @field Name of module

  WCHAR   rglpszZones[16][32]; // @field names of zones for first 16
bits

  ULONG   ulZoneMask;          // @field Current zone Mask

} DBGPARAM, *LPDBGPARAM;
```

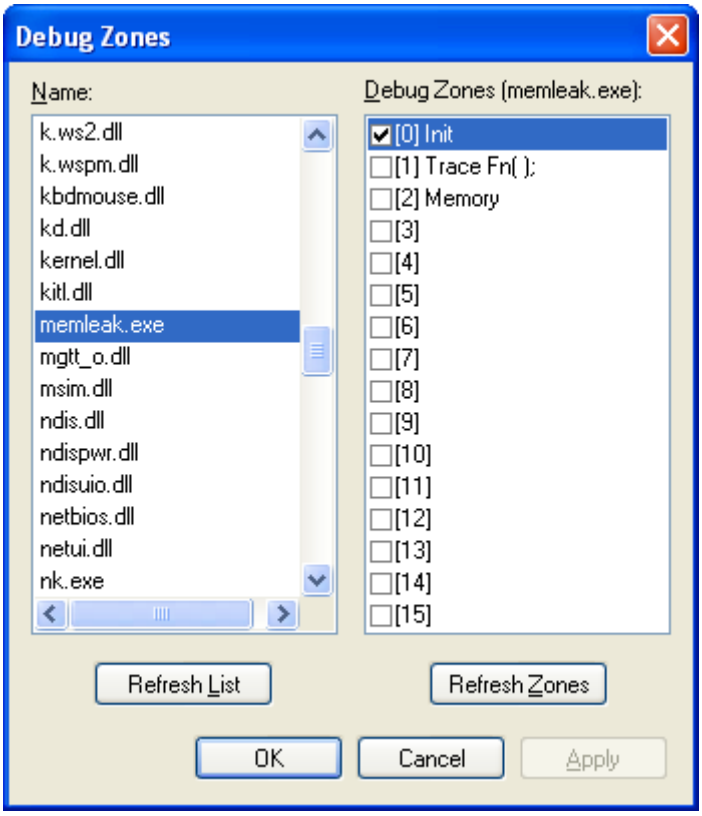
The following code example shows how the **DBGPARAM** structure is defined in memleak – in this case we're only defining three zones, Initialization, Function Tracing, and memory allocation/free – you can also see that we're defining that initialization is enabled as the default zone, we could of course add tracing for any/all of the other zones.

```
DBGPARAM dpCurSettings = {

    TEXT("MemLeak"), {

        TEXT("Init"),TEXT("Trace Fn( );"),TEXT("Memory"),TEXT(""),

        TEXT(""),TEXT(""),TEXT(""),TEXT(""),

        TEXT(""),TEXT(""),TEXT(""),TEXT(""),

        TEXT(""),TEXT(""),TEXT(""),TEXT("")},

    // By default, turn on the zones for init and errors.

    ZONEMASK_INIT

};
```

The **MemLeak** sample, like most of the OS, is built to include debug zone information. We can see this by opening the debug zone dialog in Platform Builder.

- Select **Target** and choose **CE Debug Zones** – the zone dialog will appear

- Click on **MemLeak.exe** in the Name list and note the zone information dynamically updates in the Debug Zones list.



By way of demonstrating a zone, we'll enable **Trace Fn( )** messages – this will enable debug output messages when the application allocates and releases memory.

- Click the box next to **Trace Fn( )** in the Debug Zones list.
- Click **OK**

Here's the output from the MemLeak application, we can see the entry to AllocateMemory( ) and the allocation of 16384 * sizeof(TCHAR), we can also see the exit from Allocate memory – also, we can see the entry point to FreeMemory( ) and the exit from FreeMemory( ) function.

```
1700858 PID:5c5000e TID:5e2000e ----------------------------------------
----

1700858 PID:5c5000e TID:5e2000e Enter - AllocateMemory( ) Function

1700963 PID:5c5000e TID:5e2000e Leave - AllocateMemory( ) Function

1700963 PID:5c5000e TID:5e2000e Enter - UseMemory( ) Function

1700963 PID:5c5000e TID:5e2000e Leave - UseMemory( ) Function

1700963 PID:5c5000e TID:5e2000e Enter - FreeMemory( ) Function

1700964 PID:5c5000e TID:5e2000e Leave - FreeMemory( ) Function

1701465 PID:5c5000e TID:5e2000e ----------------------------------------
----
```

This simply shows the flow of the application but doesn't provide much additional debug information about memory allocation. We can enable the second debug zone to obtain more information from the application.

Next, we will enable Memory Tracing.

- Select **Target** and choose **CE Debug Zones** – the zone dialog will appear
- Click on **MemLeak.exe** in the Name list and note the zone information dynamically updates in the Debug Zones list.
- Click the box next to **Memory** in the Debug Zones list.
- Click **OK**

Here's a sample of the updated debug information from the memleak application. You can see in this instance, the application had already brought the devices memory usage to 60%. At this memory load, the application stops allocating memory.

```
1999153 PID:5c5000e TID:5e2000e ----------------------------------------
----

1999153 PID:5c5000e TID:5e2000e Enter - AllocateMemory( ) Function

1999153 PID:5c5000e TID:5e2000e Check GlobalMemoryStatus( )

1999263 PID:5c5000e TID:5e2000e Memory Load 60%

1999263 PID:5c5000e TID:5e2000e Memory Load too high - not allocating
memory

1999263 PID:5c5000e TID:5e2000e Leave - AllocateMemory( ) Function

1999264 PID:5c5000e TID:5e2000e Enter - UseMemory( ) Function

1999265 PID:5c5000e TID:5e2000e Do Something Interesting here.

1999266 PID:5c5000e TID:5e2000e Leave - UseMemory( ) Function

1999266 PID:5c5000e TID:5e2000e Enter - FreeMemory( ) Function

1999266 PID:5c5000e TID:5e2000e Free Pointer 0x0

1999267 PID:5c5000e TID:5e2000e Leave - FreeMemory( ) Function

1999768 PID:5c5000e TID:5e2000e ----------------------------------------
----
```

We can see that the Free memory function is trying to free a pointer value of zero, hence the memory leak.

9.3 Kernel Tracker

Using the Remote Kernel Tracker you can view thread interactions, internal dependencies, and system state information, the tool shows all processes and threads in the system and when these processes and threads are created, run, or stopped. It also shows when processes and threads are sleeping. In addition to showing the running states of processes and threads, the Remote Kernel Tracker tool displays system events. These system events are mapped onto the thread that was executing at the time they occurred. The tool also shows system interrupts.

Kernel Tracker is an essential tool for tracking real-time events – but how can Kernel Tracker be used to track memory? – The answer is simple, through the use of a call to the CELOG( ) API – let's take a look at some of the source code to the memleak application.


```
MEMORYSTATUS g_MemStatus;

memset(&g_MemStatus,0x00,sizeof(g_MemStatus));

g_MemStatus.dwLength=sizeof(g_MemStatus);

GlobalMemoryStatus(&g_MemStatus);

DEBUGMSG (ZONE_MEMORY, (TEXT("Memory Load
%d%%\n"),g_MemStatus.dwMemoryLoad));

CELOGDATA(TRUE, CELID_RAW_LONG, &g_MemStatus.dwMemoryLoad, (WORD)
(sizeof(DWORD)), 1, CELZONE_MISC);
```

In the above code we obtain the current memory load by calling GlobalMemoryStatus( ) – we then call CELOGDATA using the memory load as one of the parameters. In this case we're outputting memory load whenever we allocate memory, of course we could output any useful information, including handles to events, or threads, memory allocation and free etc…

Let's see this in practice.

- In Visual Studio, select **Target** | **Remote Tools** | **Kernel Tracker**

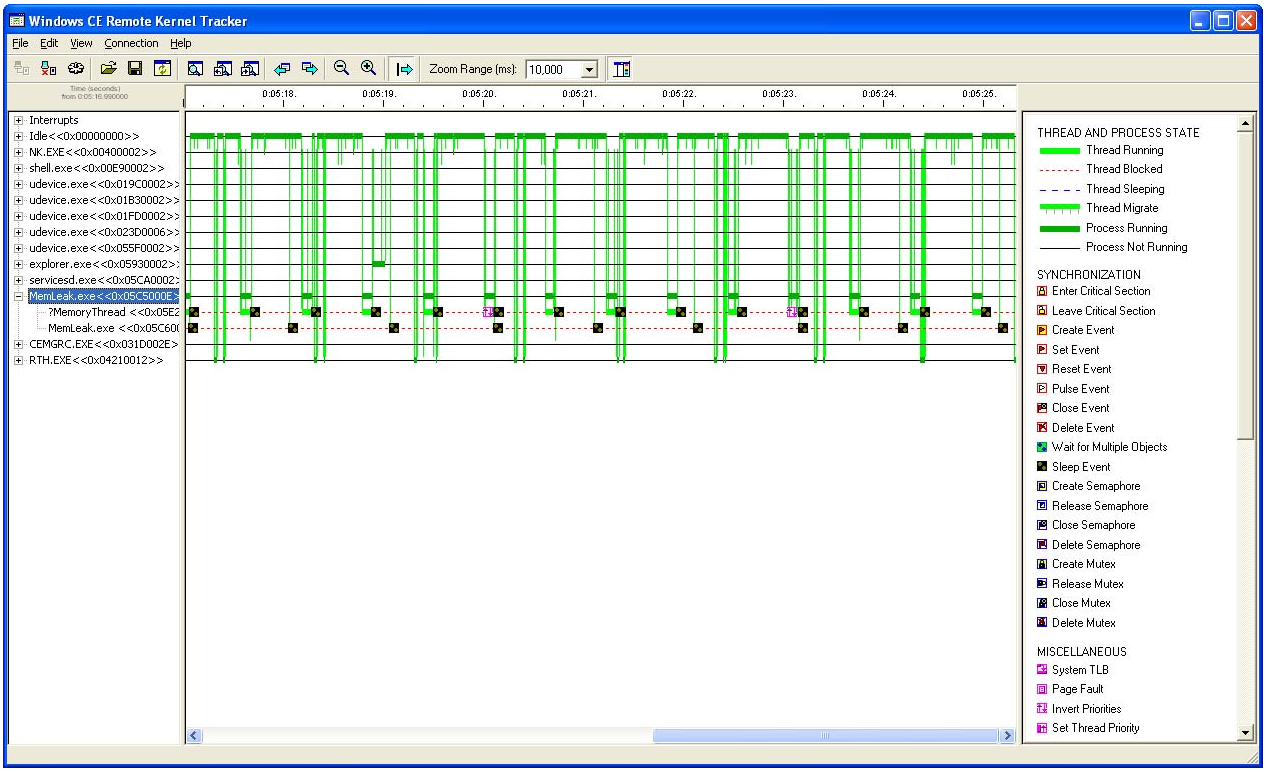
You will be prompted for the device to connect to.

- Select **Windows CE Default Device**
- Click **OK**

Kernel Tracker will download a number of device side components to the i.MX31; once these have been downloaded, Kernel Tracker will display all running processes, and interrupts.

- Expand the **MemLeak.exe** item within Kernel Tracker

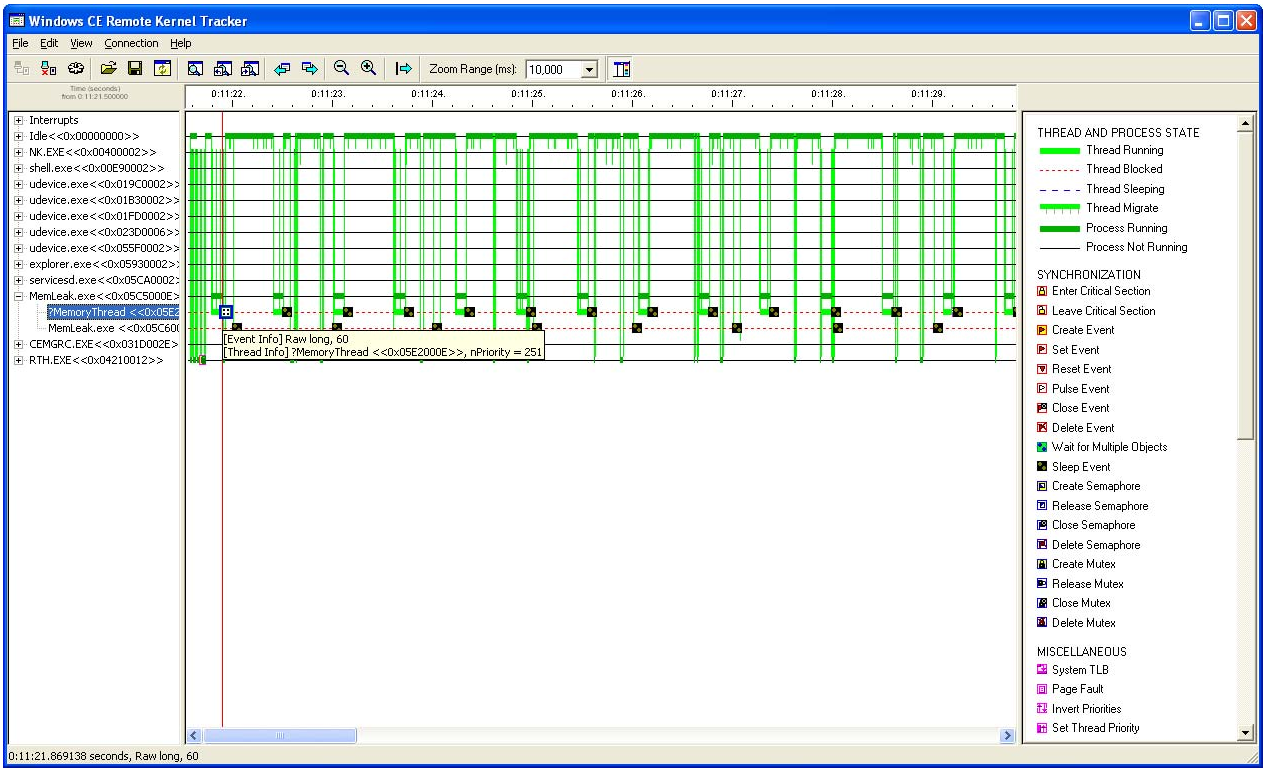
You will notice two threads being displayed, one being the application thread (identified as memleak.exe), and the second thread (identified as Thread) which is checking memory load every 500ms, and allocating 16384*TCHAR if memory load is below 60%.



- Right Click on the **?MemoryThread** (part of the memleak application)
- Select **Find Next Event on Thread**

Kernel Tracker will 'Jump' to the next event on the selected thread – you will notice in the image below that there's a white square with four dots, this is a custom (or CELOGDATA) item, we can hover our mouse over this item to get a 'data tip' this will display the information we output from our code).

- Hover your mouse over the CELOGDATA item – view the data tip

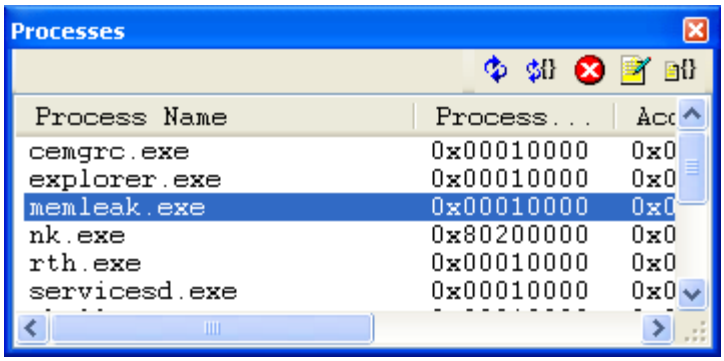


You will notice that the data tip shown below has [event info] Raw long, 60 – this shows that memory load was at 60% at the time we called CELOGDATA (of course this value may be different on your system).

- We're done with kernel tracker so you can close it
- If prompted to save data from Kernel Tracker, select No

We also need to stop the MemLeak application, there are a number of ways to achieve this, either through the Windows CE Target Control Window, or through the Platform Builder UI directly.

- Select **Debug** | **Windows** | **Processes**
- Locate and select **Memleak**



- Click the **RED** close process button
- Confirm "YES" to close the process

10 Windows CE Remote Tools

10.1 Using the Remote File Viewer

In this exercise you will use the Remote File Viewer to transfer files between your development workstation and a Windows Embedded CE 6.0 target device.

- Your Hardware should still be running from the previous section. If not, restart it.

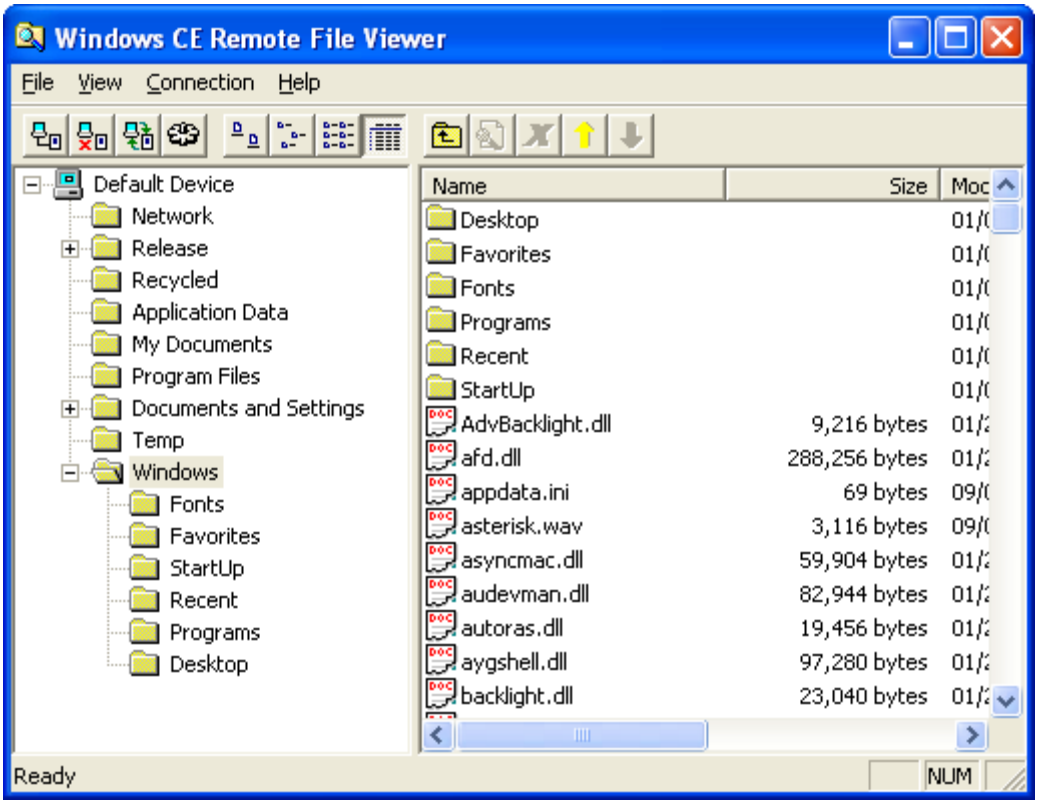
We will now take a look at the Remote File View tool.

- Select **Target** | **Remote Tools** | **File Viewer** from the Visual Studio menu. The **Select a Windows CE Device** dialog will appear.
- Expand the **Windows CE Default Platform** node and select **Default Device**. Click **OK**. Visual Studio will begin transferring the required files for the Remote File Viewer to the Hardware

You may get a dialog box asking for the location of an executable. This is because the kernel debugger running in the Hardware is attempting to monitor all processes that run on the device, including the remote tools. This cannot be done because the debugging information is not available for these tools, and we can safely ignore the request. If this happens, simply select **Don't display this dialog again** and click cancel. Note that this issue only occurs because we have left the kernel debugger running inside our OS run-time.

First we will use the tool to explore the device's file system.

- Expand the **Default Device** node in the left hand pane and select the **Windows** directory. The right hand pane shows a list of the files in the Windows directory on the target device.
- Select **View** | **Details** from the Remote File Viewer menu to see the details of each file in the folder.



Now we can try to copy a file from the device to your workstation.

- Select **logicpd.bmp** in the right hand pane. We will copy this file from the Hardware to the PC.
- Select **File** | **Import File** from the Remote File Viewer menu.
- Save the file to a convenient folder on your development workstation desktop.
- Observe that the file has been transferred to the specified directory.

Below is how you copy a file to your device from your workstation.

- Select the **Desktop** folder in the right hand pane of the **Remote File Viewer**. Expand the Windows folder, if necessary, to find the Desktop folder. Select **File** | **Export File** from the Remote File Viewer menu.
- In the **Export File** dialog select a file from the development workstation and click Open. The file will be transferred from the development workstation to the Hardware. You should be able to see the file on your CE desktop.
- Close the **Remote File Viewer** application.

10.2 Using the Remote Registry Editor

In this exercise, you will use the Windows CE Remote Registry Editor to examine and modify the registry on the target device.

First we will start the Remote Registry Editor tool.

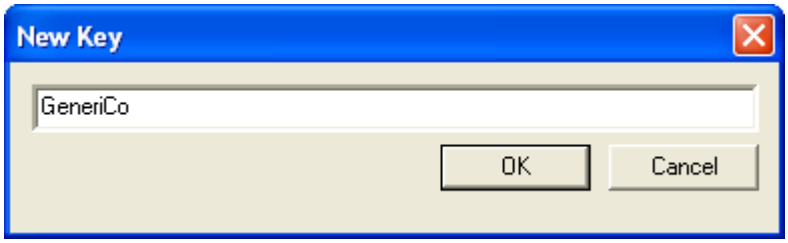
- Select **Target** | **Remote Tools** | **Registry Editor** from the Visual Studio menu.
- Click **OK** to accept the **Default Device** connection.

- Wait a few seconds for the connection to be established and for required files to be transferred to the device.

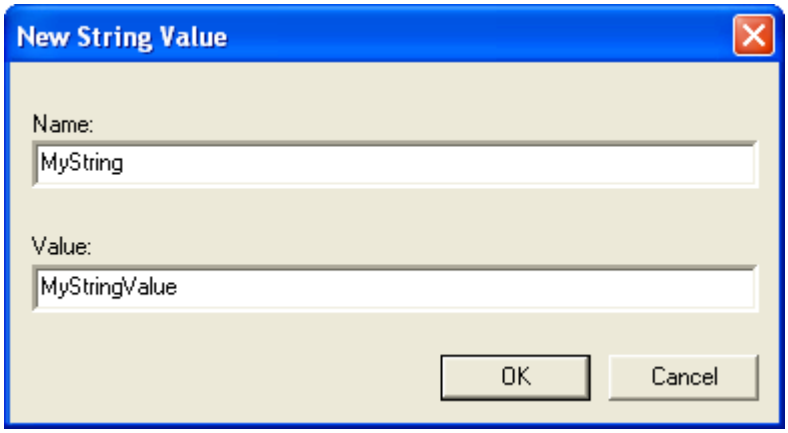
The Remote Registry Editor also displays the registry of your development workstation under the My Computer tree in the left hand pane. This tree will be available even if you are not connected to the target device. Make sure you don't get confused about which registry you are viewing and/or editing.

Now we can explore the device's registry and modify a registry setting.

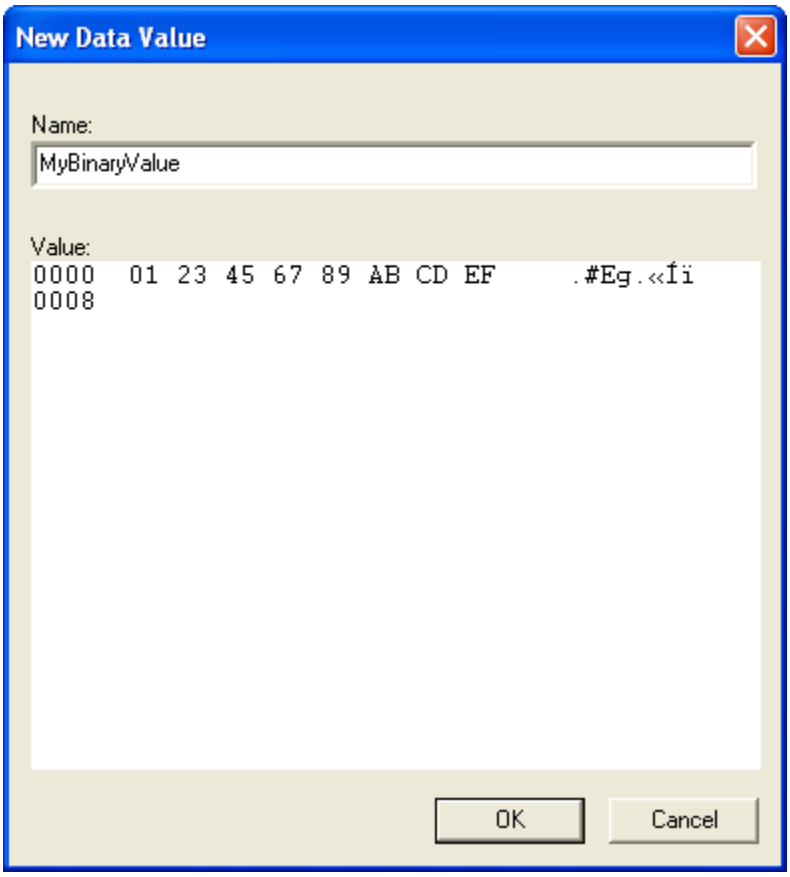
- In the left-hand pane of the Windows CE Remote Registry Editor, expand **Default Device**
- Then expand **HKEY_LOCAL_MACHINE** and select **SOFTWARE**
- Right click **SOFTWARE** and select **New Key**
- Enter in **GeneriCo** and click **OK**



- Right click on the new **GeneriCo** key and select **New** | **String Value**
- Create the string value with the name **MyString**, and the value **MyStringValue**



- Click **OK** to create the value
- Right click on the GeneriCo key and select **New** | **Binary Value**. This will bring up the **New Data Value** dialog
- Create a binary value with the name **MyBinaryValue**. In the **Value** box, type **0123456789abcdef** and click **OK**. Notice that the editor groups the entry into bytes, and adds an ASCII translation of the values in the right column of the value box



You can also edit the ASCII translation area of this dialog box instead of typing in binary values. There is no visible differentiator between the two areas in the dialog, just click your mouse in the far right hand area if you wish to enter ASCII text.

- Right-click on **MyBinaryValue** and select **Delete**
- Confirm your intention to delete by clicking **Yes**
- In the left-hand pane, right click on **GeneriCo** and select **Rename**
- Type **GeneriCo2** and press the **Enter** key

Now let's save the registry to the workstation.

- In the left-hand pane of the Remote Registry Editor, highlight the **[HKLM\SOFTWARE\GeneriCo2]** key
- Select **Registry** | **Export Registry File** from the Remote Registry Editor menu
- Save the file to your desktop as **MyDeviceRegKey.txt**
- Open the **MyDeviceRegKey.txt** file using a text editor
- Observe that the format of the Windows CE registry file is identical to that used in other versions of Microsoft Windows
- Close and delete **MyDeviceRegKey.txt**
- Close the **Remote Registry Editor**

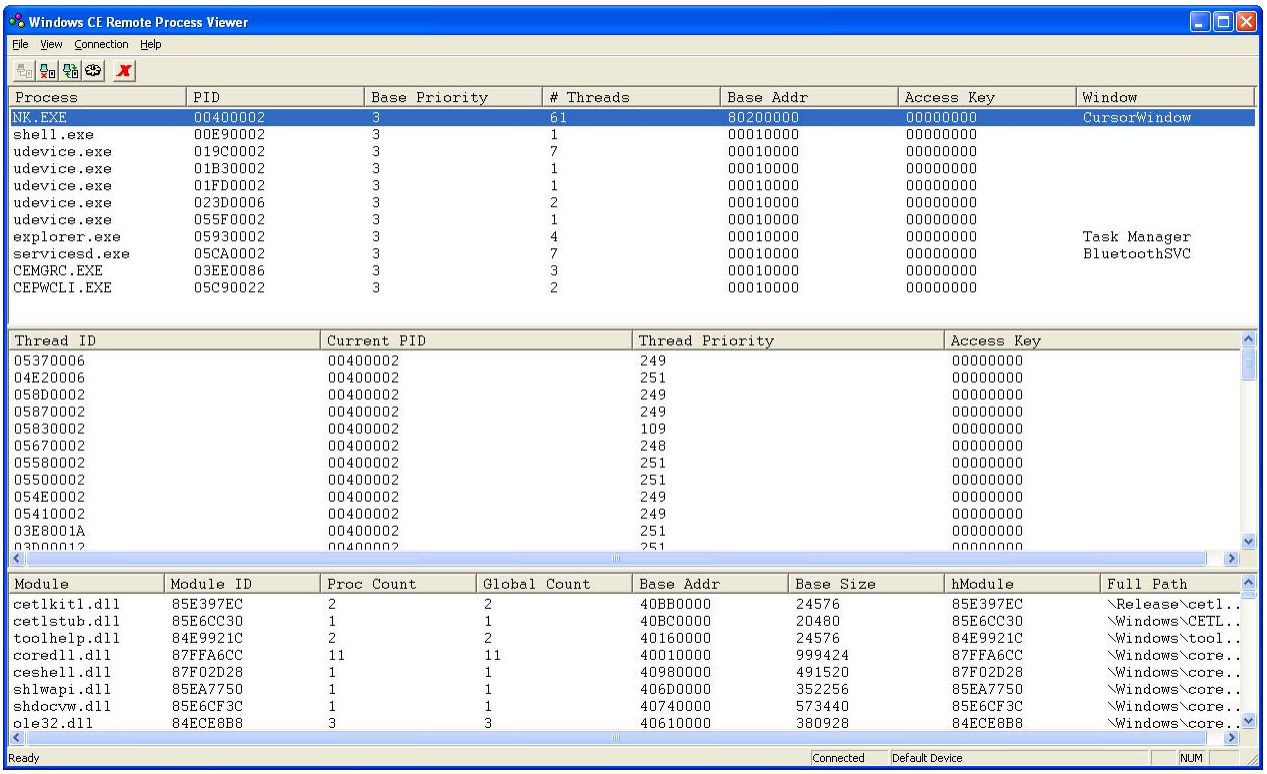
10.3 Using the Remote Process Viewer

In this exercise, you will use the Windows CE Remote Process Viewer to examine the processes and threads running on a Windows CE target device.

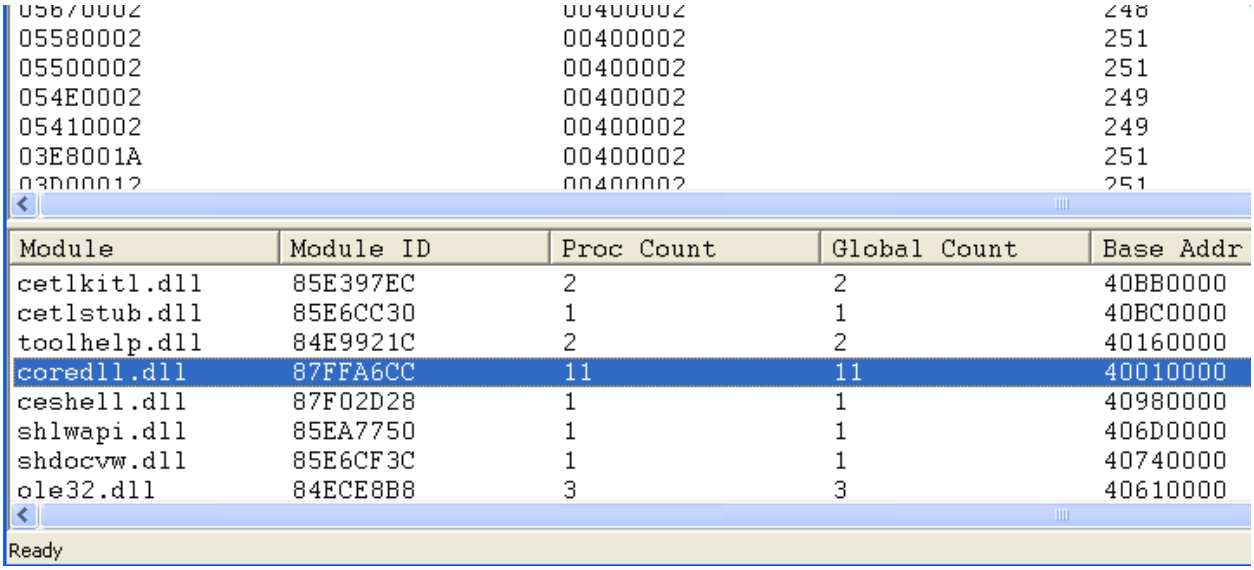
- Your hardware should still be running from the previous section. If not, restart it

First we will launch the Remote Process Viewer tool and explore what threads are running.

- Select **Target** | **Remote Tools** | **Process Viewer** from the Visual Studio menu
- Click **OK** to use the **Default Device** connection. The Windows CE Remote Process Viewer tool will load
- Examine the list of active **processes** in the upper pane



- Click on **NK.EXE** in the Process pane. Thread details are presented in the center pane
- The DLLs loaded by a process are listed in the lower 'Module' pane. Note the base address of **coredll.dll**



05580002	00400002	251
05500002	00400002	251
054E0002	00400002	249
05410002	00400002	249
03E8001A	00400002	251

Module	Module ID	Proc Count	Global Count	Base Addr
cetlkitl.dll	85E397EC	2	2	40BB0000
cetlstub.dll	85E6CC30	1	1	40BC0000
toolhelp.dll	84E9921C	2	2	40160000
coredll.dll	87FFA6CC	11	11	40010000
ceshell.dll	87F02D28	1	1	40980000
shlwapi.dll	85EA7750	1	1	406D0000
shdocvw.dll	85E6CF3C	1	1	40740000
ole32.dll	84ECE8B8	3	3	40610000

Ready

- Click on **shell.exe** in the Process pane. This process has one thread and loads two modules. Note that **coredll.dll** is loaded at the same base address in **shell.exe** as it was in **NK.EXE**
- Click on **explorer.exe** in the Process pane. This process has many threads and many loaded modules. Scroll to the bottom of the module list. Note that **coredll.dll** is loaded at the same base address as in the other processes
- Close the **Remote Process Viewer**
- Select **Target** | **Detach Device** from the Visual Studio menu and close the Hardware window

11 Summary

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

- Customized the platform by adding applications
- Enabled Profiling and Kernel Tracking
- Built an O/S image
- Downloaded an operating system to the i.MX31 device
- Worked with Remote Performance Monitor
- Worked with Debug Zones
- Worked with Remote Kernel Tracker
- Used various Remote Tools

This lab provided a hands on overview of the debug tools which can be used to track leaks within a Windows Embedded CE based application or driver.