Building and Testing Your Device Driver

Windows Embedded Compact 7 Technical Article

Writers: Michael Stahl, Jina Chan

Technical Reviewers: KS Huang and Michael Svob

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

Windows Embedded Compact 7 includes a sample device driver that you can use to begin writing your own driver. This article explains how to add the sample driver to your board support package (BSP) and then modify your BSP configuration files to include the driver as part of your run-time image. You learn how to dynamically load and test your driver by using console applications and the Windows Embedded Compact Test Kit (CTK). You also learn about debugging your driver by using some of the debugging tools that are available as part of Windows Embedded Compact 7.

Introduction

This article describes the steps to copy, build, test, and debug a simple device driver. You'll work with source code that is included as part of Windows Embedded Compact 7.

A companion article, *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236), suggests a process for planning and implementing your device driver project. That article also explains the most important functionality of the Windows Embedded Compact 7 device driver architecture. *Implementing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210237) helps you get started implementing your device driver. Before planning and implementing a device driver project, you must be familiar with Windows Embedded Compact 7 development, Platform Builder, and device driver development concepts such as interrupt handling and direct memory access.

Before you begin driver development, you must also have an OS design and be familiar with how to load the run-time image onto a target device. For a quick guide about OS design, see *Developing an Operating System Design* (http://go.microsoft.com/fwlink/?LinkID=210187).

You may have a physical device to deploy to, but you might want to test your device driver before you deploy it to your hardware. Windows Embedded Compact 7 provides a way to test your code without hardware by using a virtual CEPC. For more information about loading a run-time image by using a virtual CEPC, see *Getting Started with Virtual CEPC* (http://go.microsoft.com/fwlink/?LinkId=190470).

The steps in this document were verified by using a cloned virtual PC board support package (BSP) and deployed by using a virtual CEPC.

Adding and Building a Device Driver

Adding the sample stream driver to your BSP requires that you create a new driver subfolder, copy existing source and configuration files, and modify your BSP configuration files. Then you can build your driver as part of the run-time image.

The following sections include information for you to add and build the stream driver as part of your run-time image. Review each section for detailed steps.

- Adding the stream driver sample files to your BSP.
- Updating the driver source code.
- Updating the DIRS file.
- Updating the stream driver registry file.
- Updating the platform registry file.
- Updating the platform binary image builder file.
- Building the driver.
- Setting alternate release directories.
- Using Dumpbin.exe to review driver functions.

When finished, you have a shell for a simple monolithic stream driver that is integrated as part of your BSP. You can then add code to the empty functions that are provided in the driver shell that add support specifically for your device.

Adding the Stream Driver Files to Your BSP

The files for each driver in your BSP are located in unique subfolders under the following path:

C:\WINCE700\Platform\<Your BSP>\Src\Drivers

Before you can add new driver files to your BSP, you must create a new folder for your driver. The generic sample driver that is included with Windows Embedded Compact 7 is called “Streamdriver” and can be an initial template for your driver.

To add the stream driver files to your BSP

1. Create a new subfolder in your BSP driver folder:
   C:\WINCE700\Platform\<Your BSP>\Src\Drivers\Streamdriver
2. Copy the sample driver files to your Streamdriver folder from the following location:
   C:\WINCE700\Platform\BSPTemplate\Src\Drivers\Streamdriver

When you finish, the following files are in your stream driver folder:

```
Makefile
Readme.html
Sources
Streamdriver.bib
Streamdriver.cpp
Streamdriver.def
Streamdriver.reg
```

Updating the Driver Source Code

Before you run tests on the sample stream driver, customize the driver source code to return useful values.

To update the driver source code

1. In Solution Explorer, navigate to the following path:
   C:\WINCE700\Platform\<Your BSP>\Src\Drivers\Streamdriver
2. Expand the Source files node, and then open the Streamdriver.cpp file.
3. Modify the functions **SDT_Init** and **SDT_Open** to return values other than 0 (zero is the default value and indicates failure):

```
extern "C" DWORD SDT_Init(LPCTSTR pContext, DWORD dwBusContext)
{
  // Fill in initialization code here
  return 1;
}

extern "C" DWORD SDT_Open(
  DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode
```

```
)
{
  // Fill in open code here
  return 1;
```

You can now use this sample driver for testing, or you can modify it further to create a fully-featured device driver.

Updating the DIRS File

For the build process to include the source files for your driver, you must update the DIRS file in the driver directory. The build process uses the information that is specified in this file to determine which subdirectories to include in the build.

To update the DIRS file

1. Go to the C:\WINCE700\Platform\<Your BSP>\Src\Drivers folder.
2. Open the DIRS file by using a text editor such as Notepad.

   Note The DIRS file does not have a file name extension.
3. Add a line to the file to include the stream driver folder followed by a backslash, as shown.

```
DIRS= \
      Streamdriver \
      wavedev2_sb16 \
      ndis_dc21x4 \
      keybd \
      isr_vpcmouse \
```

The list of driver folders included in your DIRS file may vary, depending on the functionality you selected during the process of designing your OS.

Updating the Stream Driver Registry File

By default, the Streamdriver.reg file includes the following registry subkey for its settings:

HKEY_LOCAL_MACHINE\Drivers\BuiltIn\StreamDriver

When Device Manager loads, it enumerates and loads each of the drivers that are located under the **HKEY_LOCAL_MACHINE\Drivers\BuiltIn** path. While you are developing, testing, and debugging your driver, the process is much easier when the driver is loaded dynamically from your test code. To prevent Device Manager from automatically loading your driver, you must change the path that is specified in the Streamdriver.reg file.

To change the path that is specified in the stream driver registry file

1. Go to the C:\WINCE700\Platform\<Your BSP>\Src\Drivers\Streamdriver folder.
2. Open the Streamdriver.reg file by using a text editor such as Notepad.
3. Change the registry subkey, as shown.

HKEY_LOCAL_MACHINE\Drivers\StreamDriver

Because the stream driver information has been removed from the BuiltIn folder, Device Manager can no longer load it at boot time.

After your driver has been tested and debugged, change the path back to its original value so that Device Manager can load the driver as it usually does.

For more information about the registry settings and Device Manager, see *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236).

Updating the Platform Registry File

Each BSP has a registry file (Platform.reg) that defines the hardware registry settings for the target device. In some cases, the hardware registry settings are defined directly within the Platform.reg file itself. In the case of the stream driver, a separate registry file (Streamdriver.reg) is provided for you. For your sample stream driver to be configured correctly with your BSP, you must update the platform registry file to include the stream driver registry information.

To include the stream driver registry settings in your platform registry

1. Go to the C:\WINCE700\Platform\<Your BSP>\Files folder.
2. Open the Platform.reg file by using a text editor such as Notepad.
3. Add a line that includes the stream driver registry file. Add the line to the beginning of the Platform.reg file, as the following code shows.

```
#include "$(_winceroot)\platform\<Your BSP>\src\drivers\streamdriver\streamdriver.reg"
```

Updating the Platform Binary Image Builder File

Each BSP includes a binary image builder file (Platform.bib) that defines which modules and files are included in the run-time image. You need to add an entry to this file so that your driver is included when the run-time image is loaded onto the target device.

To update the Platform.bib file

1. Go to the C:\WINCE700\Platform\<Your BSP>\Files folder.
2. Open the Platform.bib file by using a text editor such as Notepad.
3. Add a line to the **MODULES** section of the file that includes the sample driver DLL, as shown.

```
streamdriver.dll    $(_FLATRELEASEDIR)\streamdriver.dll        NK SHK
```

For more information about the binary image builder flags and format, see *Binary Image Builder (.bib) File* (http://go.microsoft.com/fwlink/?LinkID=210868) in Windows Embedded Compact 7 Documentation.

Building the Driver

After you have the new driver files in place and have modified the previously mentioned files, you can build your driver.

To build your driver by using Platform Builder

1. In Visual Studio, on the **File** menu click **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, select the solution file for your BSP.
3. In the **Solution Explorer** pane, expand the tree nodes as follows: **C:/WINCE700**, **platform**, **<Your BSP>**, **src**, **drivers**, **streamdriver**.
4. Right-click the **streamdriver** node, and then click **Build** from the shortcut menu.

   Note By default, the stream driver subproject is excluded from the build. To include the sample driver as part of the BSP build project, right-click the **streamdriver** node, and then click **Include in Build**.

Setting Alternate Release Directories

When you deploy the OS image to your device or when the Windows Embedded Compact Test Kit (CTK) is connecting to your device, a Platform Builder dialog box may appear that requests you specify the path of certain files. To prevent this dialog box from appearing, set the alternate release directories.

To set the alternate release directories

1. Click the **Target** menu, and then click **Alternate Release Directories**.
2. In the dialog box that appears, click the **Add new directory** button, and then add the directories that are shown in the following table.

Table 1: Paths for Platform Builder

File	Path
Notify.dll	C:\Wince700\OSDesigns\<My OS Name>\RelDir\VirtualPC_x86_<Debug or Release>
Cetkdev.exe	C:\Program Files\WindowsEmbeddedCompact7TestKit\Device\Target\x86
Mscoree.dll	C:\Windows\System32
Tux.exe	C:\Program Files\WindowsEmbeddedCompact7TestKit\Harnesses\Target\x86
Drivertuxtest.dll	C:\WINCE700\OSDesigns\<Your BSP>\DriverTUXTest\Obj\x86\Debug

Using Dumpbin.exe to Review Driver Functions

Dumpbin.exe is a command-line utility that is included with Windows Embedded Compact 7. You can use it to review the functions that are exposed by your driver DLL. By reviewing output that is generated by the Dumpbin tool, you can verify that you are exporting the complete set of required functions for your driver.

To examine exposed functions from the driver

1. In Platform Builder, open the **Build** menu, and then click **Open Release Directory in Build Window**.
2. At the command prompt, type **dumpbin /exports streamdriver.dll**
3. When you finish reviewing the output, close the command prompt window by typing **Exit**.

The following output was generated by the Dumpbin tool when the **exports** flag was used.

```
ordinal hint RVA      name

      1    0 000010E6 SDT_Close = _SDT_Close
      2    1 000010DF SDT_Deinit = _SDT_Deinit
      3    2 000010FB SDT_IOControl = _SDT_IOControl
      4    3 000010DC SDT_Init = _SDT_Init
      5    4 000010E3 SDT_Open = _SDT_Open
      6    5 000010EA SDT_PreClose = _SDT_PreClose
      7    6 000010EE SDT_PreDeinit = _SDT_PreDeinit
      8    7 000010F2 SDT_Read = _SDT_Read
      9    8 000010F8 SDT_Seek = _SDT_Seek
     10    9 000010F5 SDT_Write = _SDT_Write
```

For more information about the Dumpbin tool, see *DUMPBIN Tool* (http://go.microsoft.com/fwlink/?LinkID=210871) in Windows Embedded Compact 7 Documentation.

Testing Your Device Driver

There are several paths you can take to begin testing your device driver. A simple first test that you can perform is to create a console application that loads and calls your driver. Another way to test your driver is by using the Windows Embedded Compact Test Kit (CTK) to make similar calls, but as part of a test suite. Writing a console application is a quick way to get started with simple tests, and using the CTK is more helpful for regression and stress testing.

This section details testing procedures using a console application and using the CTK, and other tools that can help you during the testing process.

Testing Your Device Driver Using a Console Application

Use a console application to quickly perform a unit test of a device driver that is under development. This section provides the basic steps to create a console test application; to use the proper call sequence to load and access the driver; and to show the relationship between calls that are made from the console application to the driver itself.

You can simplify your testing process by dynamically loading the device driver through the console application, instead of by permitting Device Manager to load the driver for

you at boot time. By following this practice, you can preclude some of the difficulty that is associated with testing and debugging your driver, particularly with driver initialization. For more information about the registry settings that are required to manually load your driver, see *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236).

Note Testing with a console application was chosen here for simplicity. You can also test your driver by using other application types.

Creating Your Console Application

Platform Builder provides an easy-to-use Subproject wizard that helps you quickly create applications for your OS design. The resulting subproject contains a C++ source file that serves as the starting point for your test application.

After creating the console application, building the OS design, and loading the run-time image to your device, you can run this application on the device to test your driver.

To create a console application

1. Open your previously created OS design in **Visual Studio**.
2. In the **Solution Explorer** window, right-click the **Subprojects** folder, and then click **Add New Subproject** from the shortcut menu.
3. In the **Subproject Wizard** window, select **WCE Console Application** from the **Available templates** list.
4. In the **Subproject name** field, type **StreamDriverTest** for your test application, and then click **Next**.
5. Select **A simple console application**, and then click **Finish**.

Your new subproject StreamDriverTest appears under the **Subprojects** folder in **Solution Explorer**. To build the new project, right-click the subproject, and then click **Build**.

Note If you later want to remove the subproject, right-click the subproject, and then click **Remove**. Removing a subproject does not delete the subproject source code; the subproject is only removed from the OS design. To delete the subproject source code, you must manually navigate to the source location and delete the files.

Expanding the subproject node reveals the **Source files** folder. The folder contains a single file, StreamDriverTest.cpp, which contains the following code.

```
#include "stdafx.h"

int _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
      return 0;
}
```

You can use this code to begin writing your device driver tests.

Calling Driver Functions by Using Device Manager

As the *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236) article explains, applications do not directly call the device driver exported functions. Instead, calls that are made within the application cause Device Manager to call the device driver for you.

Before you can test your driver, you must call **ActivateDeviceEx** (or **ActivateDevice**). This call causes Device Manager to load your driver DLL. After the driver is successfully loaded, you can open the driver by calling **CreateFile**.

The following code example shows how to load, open, close, and deactivate the driver.

```
#include "stdafx.h"

int _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
      HANDLE      hActiveDriver = NULL;
      HANDLE      hDriver = NULL;
      bool        bReturn = false;

      // Ask Device Manager to load the driver
      hActiveDriver = ActivateDeviceEx(L"\\Drivers\\streamdriver", NULL,
0, NULL);
      if (hActiveDriver == INVALID_HANDLE_VALUE)
      {
            ERRORMSG(1, (TEXT("Unable to load stream driver.")));
            return -1;
      }

      // Open the driver
      hDriver = CreateFile (L"SDT1:",
                            GENERIC_READ| GENERIC_WRITE,
                            FILE_SHARE_READ | FILE_SHARE_WRITE,
                            NULL,
                            OPEN_EXISTING,
                            FILE_ATTRIBUTE_NORMAL,
                            NULL);
      if (hDriver == INVALID_HANDLE_VALUE)
      {
            ERRORMSG(1, (TEXT("Unable to open stream driver.")));
            return 0;
      }
```

```
	// Add test code here

	// Close the driver
	if (hDriver != INVALID_HANDLE_VALUE)
	{
		bReturn = CloseHandle(hDriver);
		if (bReturn == FALSE)
		{
			ERRORMSG(1, (TEXT("Unable to close stream driver.")));
		}
	}

	// Ask Device Manager to unload the driver
	if (hActiveDriver != INVALID_HANDLE_VALUE)
	{
		bReturn = DeactivateDevice(hActiveDriver);
		if (bReturn == FALSE)
		{
			ERRORMSG(1, (TEXT("Unable to unload stream driver.")));
		}
	}
	return 0;
}
```

Running Your Console Application

After you add your test code and rebuild and deploy the operating system, you can then begin testing by running your console application on the target device.

> **Note** The first time you run your console application, you may want to set breakpoints into the initialization source code for both the console application and the stream driver. Setting breakpoints helps you verify that Device Manager is loading the driver as expected when the console application calls ActivateDeviceEx.

To run your console application

1. On the target device or virtual CEPC, click the **Start** menu, and then click **Run**.
2. Type **StreamDriverTest** and then click **OK**.

This procedure launches your test application and halts Platform Builder on any previously set breakpoints.

Mapping Between Console Functions and Driver Functions

Because calls that are made from the console application are routed through Device Manager, it is important to understand how application calls and driver functions correspond to one another.

The following table shows the driver functions that are invoked by Device Manager when core function calls are made from the console application.

Table 2: Console and Driver Function Mapping

Core function	Driver function
ActivateDeviceEx (or **ActivateDevice**)	**DLLMain** **xxx_Init**
CreateFile	**xxx_Open**
WriteFile	**xxx_Write**
ReadFile	**xxx_Read**
SetFilePointer	**xxx_Seek**
DeviceIoControl	**xxx_IOControl**
NA	**xxx_Cancel**
CloseHandle	**xxx_PreClose** **xxx_Close**
DeactivateDevice	**xxx_PreDeinit** **xxx_Deinit**
NA	**xxx_PowerDown**
NA	**xxx_PowerUp**

Notice that calling ActivateDeviceEx, CloseHandle, or DeactivateDevice from the console application causes Device Manager to make more than one call to the driver. When debugging for the first time, you might set breakpoints on each of these driver functions in order to trace the code.

The driver functions are listed here in the order that they are called. For example, when you call ActivateDeviceEx, DLLMain is called followed by xxx_Init.

When ActivateDeviceEx is called, Device Manager also calls xxx_Open, xxx_IOControl, xxx_PreClose, and xxx_Close. This function sequence gives Device Manager an early opportunity to query the device driver for power capabilities. If the device is turned off

prior to other calls being made to the driver, Device Manager can make appropriate power management calls to the driver.

> **Note** Microsoft recommends that you use the xxx_IOControl function instead of xxx_PowerUp and xxx_PowerDown to implement power management functionality, including suspend and resume functionality.

For more information about the relationship between driver functions and Device Manager, see *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236).

Testing Your Device Driver Using the CTK

Testing a driver by using the Windows Embedded Compact Test Kit (CTK) is slightly more complicated than testing by using a simple console application. However, using the CTK can help you build and manage large test suites for regression and stress testing. Instead of adding a console application to your OS design and running that application on the target device, you add a test DLL to your OS design that is called by the CTK.

For more information about using the CTK, see *Windows Embedded CTK User Guide* (http://go.microsoft.com/fwlink/?LinkID=210189).

Creating Your Test DLL for the CTK

Creating a test DLL for the CTK is similar to creating a console application; both are subprojects of your OS design.

To create a CTK test DLL

1. Open your previously created OS design in **Visual Studio**.
2. In the **Solution Explorer** window, right-click the **Subprojects** folder, and then click **Add New Subproject** from the shortcut menu.
3. In the **Subproject Wizard** window, select **WCE TUX Dynamic-Link Library** from the **Available templates** list.
4. In the **Subproject name** field, type **DriverTUXTest** for your test application, and then click **Next**.
5. Click **Finish**.

Your new subproject DriverTUXTest appears under the **Subprojects** folder in **Solution Explorer**.

If you expand the subproject node, you find the **Source files** folder. The folder contains three files: DriverTUXTest.cpp, Globals.cpp, and Test.cpp. Your test must be added to Test.cpp, which contains the following code.

```
#include "main.h"
#include "globals.h"

TESTPROCAPI TestProc(UINT uMsg, TPPARAM tpParam, LPFUNCTION_TABLE_ENTRY
lpFTE)
{
    // The shell doesn't necessarily want us to execute the test.
```

```
    // Make sure first.
    if(uMsg != TPM_EXECUTE)
    {
        return TPR_NOT_HANDLED;
    }

    // TODO: Replace the following line with your own test code here.
    // Also, change the return value from TPR_SKIP to the appropriate
    // code.
    g_pKato->Log(LOG_COMMENT, TEXT("This test is not yet implemented."));

    return TPR_SKIP;
}
```

Modify the Test.cpp file to include calls to **ActivateDeviceEx**, **CreateFile**, **CloseHandle**, and **DeactivateDevice**, just as you did in the console driver. You can then add any additional calls that are specifically required for testing the driver, such as **WriteFile**, **ReadFile**, and **DeviceIoControl**. For example, in the Test.cpp file, replace the code lines after the “TODO” comment with the following code.

```
    g_pKato->Log(LOG_COMMENT, TEXT("Stream driver TUX test starting"));

      HANDLE      hActiveDriver = NULL;
      HANDLE      hDriver = NULL;
      bool        bReturn = false;

      hActiveDriver = ActivateDeviceEx(L"\\Drivers\\Streamdriver", NULL,
0, NULL);
      if (hActiveDriver == INVALID_HANDLE_VALUE)
      {
            g_pKato->Log(LOG_COMMENT, TEXT("Unable to load stream
driver."));
            return TPR_FAIL;
      }

      // Open the driver
      hDriver = CreateFile (L"SDT1:",
                            GENERIC_READ| GENERIC_WRITE,
                            FILE_SHARE_READ | FILE_SHARE_WRITE,
                            NULL,
                            OPEN_EXISTING,
                            FILE_ATTRIBUTE_NORMAL,
```

```
                                NULL);
      if (hDriver == INVALID_HANDLE_VALUE)
      {
            g_pKato->Log(LOG_COMMENT, TEXT("Unable to open stream
driver."));
            return TPR_FAIL;
      }

      // Add test code here

      // Close the driver
      if (hDriver != INVALID_HANDLE_VALUE)
      {
            bReturn = CloseHandle(hDriver);
            if (bReturn == FALSE)
            {
                  g_pKato->Log(LOG_COMMENT, TEXT("Unable to close stream
driver."));
            }
      }

      // Ask the Device Manager to unload the driver
      if (hActiveDriver != INVALID_HANDLE_VALUE)
      {
            bReturn = DeactivateDevice(hActiveDriver);
            if (bReturn == FALSE)
            {
                  g_pKato->Log(LOG_COMMENT, TEXT("Unable to unload stream
driver."));
            }
      }

      return TPR_PASS;
```

To build the new project DriverTUXTest, right-click the subproject in **Solution Explorer**, and then click **Build**.

Adding Your Test DLL to the CTK

After you successfully build your test DLL, you can then add it to the CTK.

To add a test DLL to the CTK

1. Open the **Start** menu, and run the Windows Embedded Compact Test Kit (CTK).

2. In the **Getting Started** pane, select **Create a Custom Test Pass Template**.
3. On the **Test Pass Templates** tab, select the **My Templates** node, and then click **New**.
4. Rename the new template **MyDriverTestTemplate**, and then click **Done**.
5. In the **Getting Started** pane, select **Create a Custom Test Case**.
6. Click the **New Category** button, and then rename the new category **My Category**.
7. Click the **New Test** button, and then rename the new test **MyDeviceDriverTest**.
8. Verify that the value for **Category** is **My Category**.
9. In the **Run type** list, select **Fully Automated**.
10. In the **Test harness** list, select **Tux**.
11. In the **Supported architectures** list box, select **x86 (Device)**, which opens the **Add/Remove Test Files** dialog box.
12. Click the **Browse** button, navigate to the DriverTUXTest.dll file, select it, and then click **OK**.
13. On the **Test Case Information** tab, click **Save**, and then click **Done**.

Running Your Test from Within the CTK

To execute your DLL test from within the CTK, you must have an active device connection available. Before getting started with the next steps, ensure that you:

- Have a complete build of your OS image.
- Have loaded your run-time image on your target device or virtual CEPC.

After your debugging session is established, you are ready to run your test from the CTK.

To run your test from within the CTK

1. Open the **Start** menu, and then run the Windows Embedded Compact Test Kit (CTK).
2. In the **Getting Started** pane, select **Connect to Device**.
3. When the **Select a Device** dialog box appears, under the **Platform Builder** node, select the name of your device, such as **vCEPC**, and then click **OK**.

   Note The **Connecting to device** dialog box appears briefly while the CTK attempts to establish a connection.

4. On the **CTK** toolbar, click **Remove Test Pass from Selected Connection**.

   Note For development purposes, it is only necessary to include a single test as part of the test pass.

5. In the **Test Case Explorer** pane, select the **Test Case Explorer** tab.
6. Expand the **My Category** node that you created in step 6 of Adding Your Test DLL to the CTK.
7. Right-click **MyDeviceDriverTest**, and then click **Add to Current Test Pass** from the shortcut menu.

   Note You must now have a single test selected for your test run.

8. To save your test pass, click the **File** menu, and then click **Save**.
9. On the CTK toolbar, click **Run Test Pass**.

The CTK then executes your test, and at any breakpoints that you set in your source code for either the driver or the test itself, the CTK halts execution so that you can trace through your source code. You can verify that your driver is loading correctly and that your driver functions are executing as expected.

Debugging Your Device Driver

Several tools are available to help you debug your device driver. The following sections provide a brief overview of these tools.

Using Debug Zones

Debug zones help you to dynamically control categories of debugging output that are reported by your device driver. These zones are essentially groups of messages that you can enable and disable while you debug. Debug zones can help you to quickly and precisely identify where your driver may be having unexpected problems.

To use debug zones, you must define them for your driver and then register your driver with the kernel subsystem so that the zones may be toggled externally during testing. You define the zones by using a special global variable, *dpCurSettings*. The sample stream driver source already includes a definition for this variable, as shown.

```
#ifdef DEBUG
DBGPARAM dpCurSettings =
{
  TEXT("SDTDriver"),
  {
    TEXT("Init"),   TEXT("Error"),  TEXT("Warning"), TEXT("Info"),
    TEXT("Power"),  TEXT("Zone6"),  TEXT("Zone7"),   TEXT("Zone8"),
    TEXT("Zone9"),  TEXT("Zone10"), TEXT("Zone11"),  TEXT("Zone12"),
    TEXT("Zone13"), TEXT("Zone14"), TEXT("Zone15"),  TEXT("Zone16")
  },
  1 | 1 << 1 | 1 << 2
};
#endif
```

You define dpCurSettings in three parts: the module name, the zone names, and the bit values that indicate which zones are initially active. You may define up to 16 individual zones for your device driver. In the case of the stream driver example, the first three zones are enabled by setting the first three bits inline. Alternatively, you can define your flags by using a DEBUGMASK macro, as shown.

```
#ifdef DEBUG
#define DEBUGMASK(bit) (1 << (bit))
#define MASK_INIT    DEBUGMASK(0)
#define MASK_ERROR   DEBUGMASK(1)
#define MASK_WARNING DEBUGMASK(2)
#define MASK_INFO    DEBUGMASK(3)
```

```
#define MASK_POWER   DEBUGMASK(4)

#define ZONE_INIT    DEBUGZONE(0)
#define ZONE_ERROR   DEBUGZONE(1)
#define ZONE_WARNING DEBUGZONE(2)
#define ZONE_INFO    DEBUGZONE(3)
#define ZONE_POWER   DEBUGZONE(4)

DBGPARAM dpCurSettings =
{
  // Name of the debug module
  TEXT("SDTDriver"),
  {
    TEXT("Init"),   TEXT("Error"),  TEXT("Warning"), TEXT("Info"),
    TEXT("Power"),  TEXT("Zone6"),  TEXT("Zone7"),   TEXT("Zone8"),
    TEXT("Zone9"),  TEXT("Zone10"), TEXT("Zone11"),  TEXT("Zone12"),
    TEXT("Zone13"), TEXT("Zone14"), TEXT("Zone15"),  TEXT("Zone16")
  },
  MASK_INIT | MASK_ERROR | MASK_WARNING
};
#endif
```

After you define the zones, you must register your driver when the driver is loaded. The sample stream driver provides an example of registering the driver by calling DEBUGREGISTER when the driver is loaded.

```
extern "C" BOOL WINAPI DllMain(
  HANDLE hinstDLL, DWORD dwReason, LPVOID lpvReserved
)
{
  switch(dwReason)
  {
    case DLL_PROCESS_ATTACH:
      DEBUGREGISTER((HINSTANCE) hinstDLL);
      DisableThreadLibraryCalls ((HMODULE) hinstDLL);
      break;

    case DLL_PROCESS_DETACH:
      break;
  }

  return TRUE;
```

```
}
```

After the driver is registered, you can add debug messages throughout your driver code, as needed.

```
extern "C" DWORD SDT_Open(
  DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode
)
{
      DEBUGMSG(ZONE_INFO, (TEXT("SDT_Open\r\n")));
      return 1;
}
```

Setting Debug Zones Using Platform Builder

You can also use Platform Builder to debug your driver and to set the debug zones that are currently active. Initially, the active debug zones are defined by you in the dpCurSettings variable; however, you can adjust the active zones as needed to isolate the debug information that you currently need.

To set the current debug zones

1. Load Streamdriver.dll as part of a Platform Builder debugging session, either with your test application or at device boot time.
2. In Platform Builder, click the **Target** menu, and then click **CE Debug Zones**.
3. When the Debug Zones dialog box finishes initializing the list of loaded libraries, select Streamdriver.dll from the list.
4. Enable or disable the active debug zones by selecting them in the **Debug Zones** list.
5. Click **Apply** to accept the changes; or click **OK** to accept the changes and close the dialog box.

Setting Debug Zones Programmatically

At any time, you can also programmatically change the active debug zones in your code by setting dpCurSettings.ulZoneMask. For example, if you want to output only information, set the debug mask as follows.

```
dpCurSettings.ulZoneMask = MASK_INFO;
```

Removing Debug Zone Information for Release

When your driver is ready for release, compile the source code by using the environment variable "WINCESHIP=1", which causes the dpCurSettings variable to be excluded from the library. When you use WINCESHIP=1 to compile, no debug zone output is generated.

For more information about debug zones, see *Debug Messages and Debug Zones in Windows CE* (http://go.microsoft.com/fwlink/?LinkID=210872).

Levels of Debugging Support

When you create an OS design, the integrated development environment (IDE) creates a Debug configuration and a Release configuration of the OS design, and then sets build options for each configuration.

The level of support for debugging in a default Debug configuration differs from the level of support for debugging in a default Release configuration. The choice of build options in a configuration determines the level of debugging support in the run-time image that you build. Like the OS design as a whole, these levels of debugging support also apply to your driver.

The following table describes the levels of debugging support for the default configurations that are provided by Platform Builder.

Table 3: Debug and Release Build Support

Configuration	Description
Debug	• Uses .lib files from %_WINCEROOT%\Public\Common\Oak\Lib\<CPU>\Debug. • Places object files in directories that are named Debug. • Uses Microsoft format to provide full symbolic debugging information. • Provides the ability to turn debug zones on and off. • Does not provide optimization, which generally makes debugging more difficult. • Sets the environment variable WINCEDEBUG=debug.
Release	• Uses .lib files from %_WINCEROOT%\Public\Common\Oak\Lib\<CPU>\Retail. • Places object files in directories named Retail. • Provides no symbolic debugging information. • Optimizes for speed. • Has a smaller run-time image than the run-time image that is built from the Debug configuration of the same OS design. • Sets the environment variable WINCEDEBUG=retail.

Building a Run-Time Image from a Debug Configuration

A Debug configuration provides complete symbolic debugging information and complete debugger access to processes, threads, and modules.

To build a run-time image from a Debug configuration

1. On the **Project** menu, click **Properties**.
2. In the navigation pane, expand **Configuration Properties**, and then click **Build Options**.

3. In the **Configuration** box, select a Debug configuration.
4. In the **Build Options** window, set **Enable kernel debugger** to **No (IMGNODEBUGGER=1)**.
5. Click the **Configuration Manager** button.
6. In the **Configuration Manager** dialog box, verify that the active configuration set for your OS design is a Debug configuration, and then close the **Configuration Manager** dialog box.
7. Click **OK** to close both the **Properties** and **Property Pages** dialog boxes.
8. From the **Build** menu, select **Advanced Build Commands**, and then select **Sysgen (blddemo –q)**.

Building a Run-Time Image from a Release Configuration

A Release configuration is optimized for speed and has a smaller run-time image than a Debug configuration.

To build a run-time image from a Release configuration

1. On the **Project** menu, click **Properties**.
2. In the navigation pane, expand **Configuration Properties**, and then click **Build Options**.
3. In the **Configuration** box, select a Release configuration.
4. In the **Build Options** window, set **Enable kernel debugger** to **No (IMGNODEBUGGER=1)**.
5. Click the **Configuration Manager** button.
6. In the **Configuration Manager** dialog box, verify that the active solution configuration set for your OS design is a Release configuration, and then close the **Configuration Manager** dialog box.
7. Click **OK** to close both the **Properties** and **Property Pages** dialog boxes.
8. From the **Build** menu, click **Advanced Build Commands**, and then select **Sysgen**.

Using the Kernel Debugger

The kernel debugger integrates functionality to configure a connection to a target device and then to download a run-time image to the device. Platform Builder provides platform settings that enable kernel debugging during the process of building a run-time image. When you enable kernel debugging, Platform Builder includes the debugging stub, KdStub, in the run-time image.

When enabled, the kernel debugger runs independently as a service under the Core Connectivity infrastructure. The debugger starts automatically for your run-time image and continues to run until you stop it.

Using the Kernel Debugger with a Run-Time Image Built from a Release Configuration

You can use the kernel debugger together with a run-time image that is built from the Release configuration that Platform Builder provides. However, a run-time image that is

built from a Release configuration provides more limited debugging because of optimization. This debugging combination may not provide the level of debugging support you require.

If you want complete symbolic debugging information that includes full debugger access to processes, threads and modules, build a run-time image from the Debug configuration that Platform Builder provides.

To use the kernel debugger with a run-time image built from a Release configuration

1. On the Project menu, click **Properties**.
2. In the navigation pane, expand **Configuration Properties**, and then click **Build Options**.
3. In the **Build Options** window, set **Enable kernel debugger** to **No (IMGNODEBUGGER=1)**.
4. Click **OK** to close the **Properties** dialog box.

When you apply these settings and then build your run-time image, support for the kernel debugger is built into the run-time image.

Conclusion

This article describes how to integrate your own BSP with the monolithic stream driver that is included with Windows Embedded Compact 7. The article also describes how to include the device driver as part of your run-time image and dynamically load the device driver by using your own test code. Finally, this article described how to debug your driver by using some of the debugging tools and techniques that are available as part of Windows Embedded Compact 7.

A companion article, *Planning Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210236), helps you plan and implement the initial phases of your device driver project while it explains the important functionality of the Windows Embedded Compact 7 device driver architecture. *Implementing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210237) helps you get started implementing your device driver.

Additional Resources

- Windows Embedded website (http://go.microsoft.com/fwlink/?LinkID=203338)
- *Developing an Operating System Design* (http://go.microsoft.com/fwlink/?LinkID=210187)
- *Getting Started with Virtual CEPC* (http://go.microsoft.com/fwlink/?LinkId=190470)

Copyright