3

Platform Builder Basics

The goal for Windows CE is to be able to run in a variety of devices and appliances. If you stop and think for a moment, you will realize that the world of consumer electronics is home to an array of devices that have wildly fluctuating hardware architectures. It would be hazardous to make assumptions about the processor, processor speed, available RAM, and storage medium on any given device. Can a tiny handheld digital telephone make room for a Pentium processor? Does it make sense for a refrigerator to have a hard disk? What's an operating system to do?

Microsoft has attempted to resolve these issues by making Windows CE a component-based operating system. A few of the basic OS functions are mandatory and must be supported by any hardware device that intends to use Windows CE. Everything else is up for grabs; you can add it to your operating system, or you can leave it out.

What exactly are these components? A **component** is a particular functionality that can be integrated into or left out of Windows CE. Components are available in the form of drivers, static or dynamic libraries, and executables. Windows CE can be built with selected components that are appropriate to the platform being developed. The tool that is used to build modified versions of Windows CE is called Platform Builder. Platform Builder is a set of CDs that contain utilities, files, and the components of Windows CE. Components can be platform dependent or platform independent. An example of a platform-dependent component is the kernel for an x86 PC. A component that allows the user to calibrate the touch panel and saves the coordinates is an example of a component that is platform independent. In addition, a component usually has a distinct resulting build target—either an executable or an object module library.

Let's look at the concept of a module. Several components can be linked together to form a module. A **module** is an executable or library file that performs a set of well-defined operations and exports a well-defined API. A module is divided into components according to areas of functionality, although such division is historically restricted by how the code was written. Microsoft has done its best to separate the modules into components in such a way that OEMs can select only the functionality they need for their platform. On value-priced consumer devices with multiple form factors, this is a critical feature. However, not all modules can be broken down into components.

Exploring Components in Platform Builder

Now that we have an idea of what components are, let's look at the real situation in the Platform Builder IDE. If you start Platform Builder and wait for it to open its main window, you will see all components collected in a **Catalog** window (Figure 3.1).

Platform Builder Catalog

The **Catalog** window in Platform Builder lists all the components that are available for inclusion in your build of the operating system. To create a custom version of Windows CE, also referred to as a build, you must create a **project** in Platform Builder. Note that Platform Builder terminology dictates that a component in the **Catalog** window is referred to as an **implementation**. When the implementation is included in a project, it becomes a component. An implementation is depicted by the icon shown in Figure 3.2. Implementations are grouped together by functionality into categories, or **types**. Types are depicted by the icon shown in Figure 3.3. An example of a type is `display`, which groups together available display drivers in a platform.

The types can be traversed and viewed just like a folder hierarchy. The implementations contained in the types and folders of the default catalog (see Figure 3.1) opened by Platform

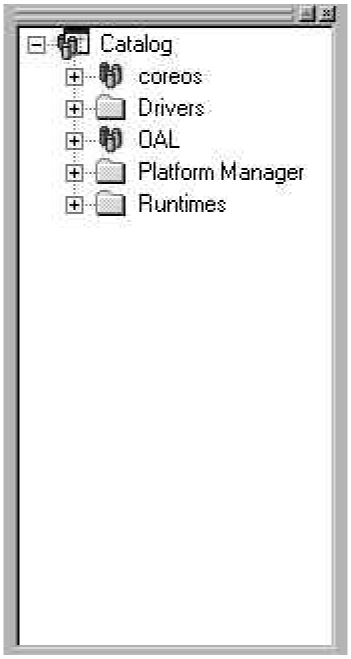


Figure 3.1 Catalog window in Platform Builder

Figure 3.2 Implementation icon

Figure 3.3 Type icon

Builder are presented in Table 3.1. You can extend the default catalog view by adding components of your own. We'll say more about that later in the chapter.

Table 3.1 Types and Implementations in the Default Catalog

Folder	Configuration	Type	Implementation	Description
`coreos`				A folder that contains sample configurations of the operating system.
	`IESAMPLE`			A sample configuration that includes almost all available components and adds the Internet Explorer Web browser control. Full localization is supported, and the Input Method Manager (IMM) is also included.
	`MAXALL`			A sample configuration that includes almost all available components, including the shell and Pocket applications.
	`MINCOMM`			A sample configuration that includes a minimal set of components and adds serial communications and networking.
	`MINGDI`			A sample configuration that includes a minimal set of components that can support the Graphical Device Interface (GDI). No window support is provided, but you can use GDI calls to create a minimal user interface if required.
	`MININPUT`			A sample configuration that includes a minimal set of components that can support user input via the keyboard. A display

(continued)

Table 3.1 *Continued.*

Folder	Configuration	Type	Implementation	Description
				driver is included, but GDI is not supported.
	MINKERN			A sample configuration that contains the operating system kernel and a "Hello World!" application that outputs text to the debug serial port. This is a good first configuration to use when you're booting a platform with Windows CE.
	MINSHELL			A sample configuration that is very much like MAXALL without the Pocket applications.
	MINWMGR			A sample configuration that includes components that can support the window manager. Full networking support is included.
Drivers				A folder that contains platform-specific device drivers.
	CEPC			A folder that contains device drivers for the CEPC platform, which supports the x86 family of processors.
		display		Display drivers type.
			Ddi_ct	Display driver for Chips & Technologies CT6555x chip set.
			Ddi_364	Display driver for the S3 Trio64 chip set.
			Ddi_s3v	Display driver for the S3 ViRGE chip set.
			Ddi_vga8	A simple VGA display driver for eight-bits-per-pixel displays. A good first choice when testing a display adapter on a CEPC or any other platform that supports a VGA-compatible display.
		kbdms		Keyboard and mouse drivers type.
			Kbdmsengus1	Driver for U.S. English keyboards.

Table 3.1 *Continued.*

Folder	Configuration	Type	Implementation	Description
			Kbdmsjpn1, Kbdmsjpn2	Drivers for Japanese keyboards.
		nscirda		Infrared driver.
		ohci		Universal Serial Bus (USB) host controller interface driver.
		pc_ddk		Hardware Abstraction Library (HAL).
			Ddk_bus	Implementation of routines to abstract bus I/O (input/output).
			Ddk_map	Implementation of routines to abstract memory I/O.
		pcmcia		PCMCIA driver.
		serial		Serial port driver.
		sermouse		Serial mouse driver.
		wavedev		SoundBlaster AWE64 PNP ISA driver.
		eboot		Ethernet debugging library and helper routines for creating an Ethernet boot loader.
	ODO			Device drivers for the Hitachi D9000 (Odo) platform, which supports multiple processors.
OAL				OEM Abstraction Layer type. This is a platform-specific layer of code that is created by the OEM.
	CEPC			OAL for the CEPC.
	ODO			OAL for the D9000.
Platform Manager				A folder that contains Platform Manager client components. These components are used to provide a communication channel between Platform Builder and the operating system being developed on the platform.
	Cemgrc			Platform Manager client. This component manages high-level communication between the Platform Builder (cemgr.exe) and the operating system running on the platform.

(continued)

Table 3.1 *Continued.*

Folder	Configuration	Type	Implementation	Description
	Transport			Transport component type. A transport component is the protocol to be used between Platform Builder and the Platform Manager client.
		pm_ppp		PPP protocol used as transport.
		pm_tcpip		TCP/IP protocol used as transport.
		pm_cesrv		Windows CE services transport.
Runtimes				A folder that contains runtime environments for Windows CE application development.
	Adoce			A configuration that contains ActiveX data objects for Windows CE.
	VB			A configuration that contains components for Visual Basic runtime support.
		vbeng		Visual Basic runtime engine.
		vbforms		Support for forms.
		controls		Visual Basic controls.
			MSCEComDlg	Common dialog control.
			MSCEComm	Support for common controls.
			MSCECommandBar	Command bar control.
			MSCEFile	File I/O control.
			MSCEGrid	Grid control.
			MSCEImage	Image control.
			MSCEImageList	Image list control.
			MSCEListView	List view control.
			MSCEPicture	Picture control.
			MSCETabStrip	Tab control.
			MSCETreeView	Tree view control.
			MSCEWinSock	A control that supports the Windows Socket API.
	VC			A folder that contains components for Visual C++ runtime support.

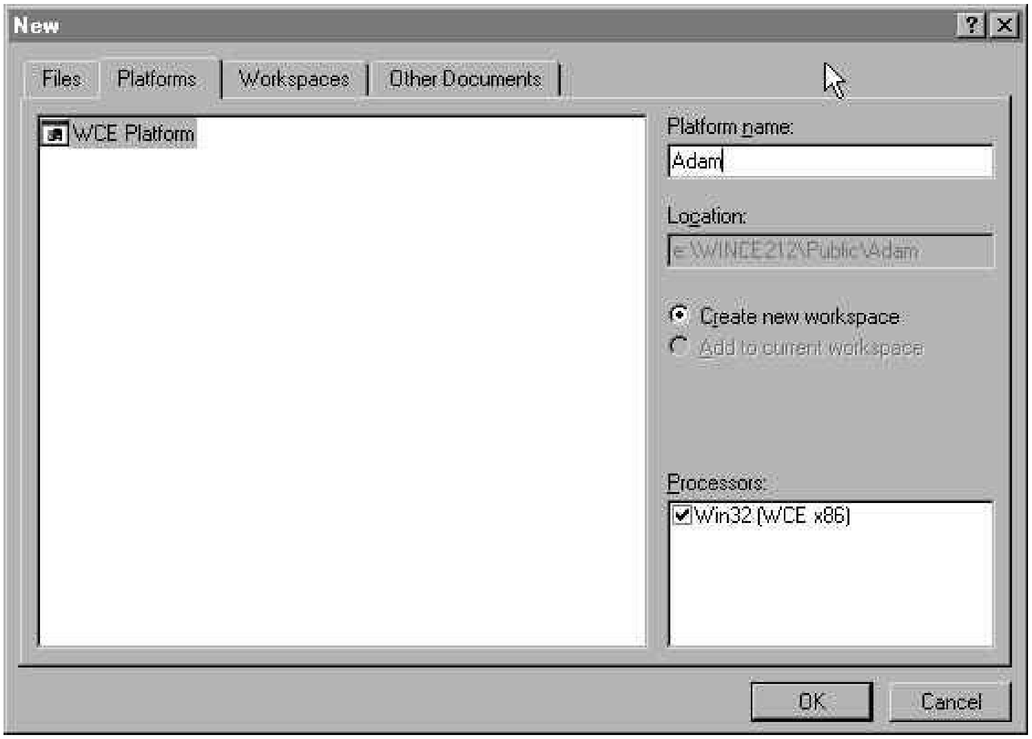
Table 3.1 *Continued.*

Folder	Configuration	Type	Implementation	Description
		`mfc`		Runtime component for Microsoft Foundation Classes.
		`atl`		Runtime component for Active Template Library.

Now that we have these components in a catalog, what can we do with them? Components are reusable objects and can be added to any project. The best way to illustrate the use of the catalog is to create a sample project. As we discussed in Chapter 2, the best place to start is to run Windows CE on a PC. For our first Windows CE build, we'll create a special build of the CEPC based on `MAXALL` called Adam. A special build of Windows CE is referred to as a **platform**.

Creating a New Platform with the Platform Wizard

To create a new platform, select the **File | New** menu item in the Platform Builder IDE. This selection invokes the **New Platform** window of the Platform Wizard (Figure 3.4).



Platform name: Adam

Location: e:\WINCE212\Public\Adam

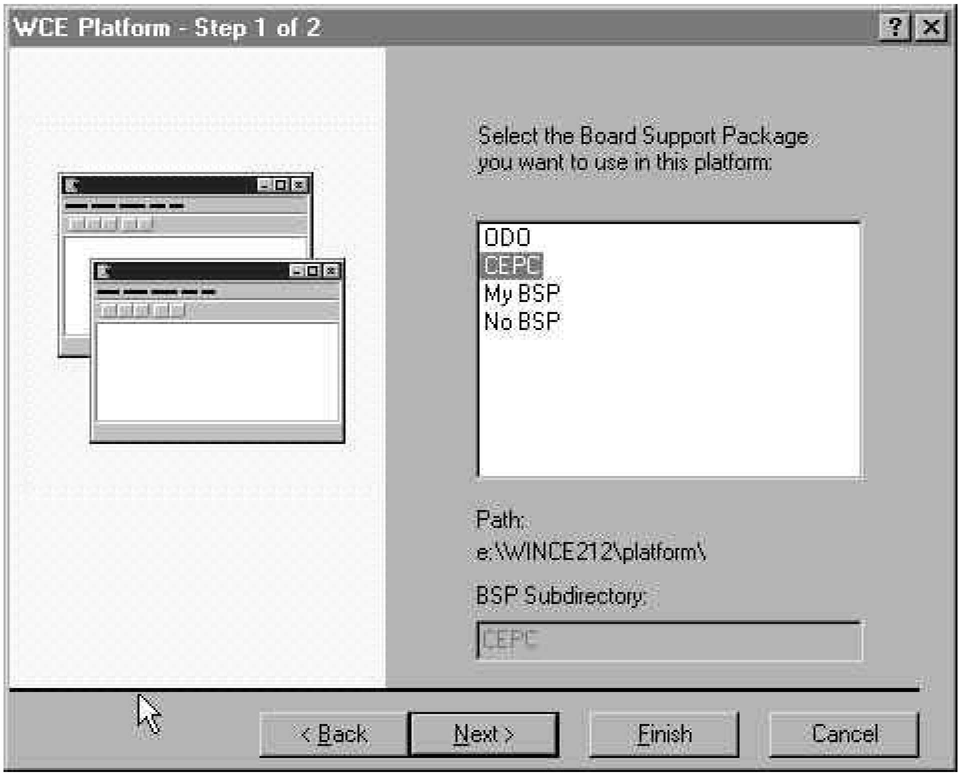
Processors: ☑ Win32 (WCE x86)

Figure 3.4 Platform Wizard

Platform Wizard gives you just one choice—a Windows CE (WCE) platform. When you type in a platform name, the wizard automatically constructs the location folder for the platform. This location is under the subdirectory `PUBLIC` in the `Platform Builder` folder. In theory you cannot modify this location. However, as we plumb the depths of the build process in Chapter 10, you will see that there is nothing magical about this folder location; it can easily be changed, although there may be no compelling reason to do so. The **Processors** list box contains choices for processor types supported by the Platform Wizard. In Figure 3.4, the only available choice is Win32 for Windows CE on an x86 processor. Additional processor types are added to this window if you select the processor choices when you install Platform Builder on your workstation.

Clicking on **OK** in the **New Platform** window activates the wizard by opening the first of two dialog boxes. This first dialog box gives you an opportunity to select a board support package for the platform being created (Figure 3.5). A **board support package** (BSP) is a set of basic, hardware-dependent components that have been created for a particular platform. These components support a particular processor type and hardware configuration. Platform Builder includes two preconfigured BSPs. The CEPC BSP is the package for an x86 PC. The Odo BSP is for the Hitachi D9000 platform. We will select **CEPC** (see Figure 3.5) because this BSP runs on an x86 PC.

We could have made two other choices. The option **My BSP** allows you to create your own BSP. This option is useful when you are working with a board that is not supported by Windows


WCE Platform - Step 1 of 2

Select the Board Support Package you want to use in this platform:

ODO
CEPC
My BSP
No BSP

Path:
e:\WINCE212\platform\

BSP Subdirectory:
CEPC

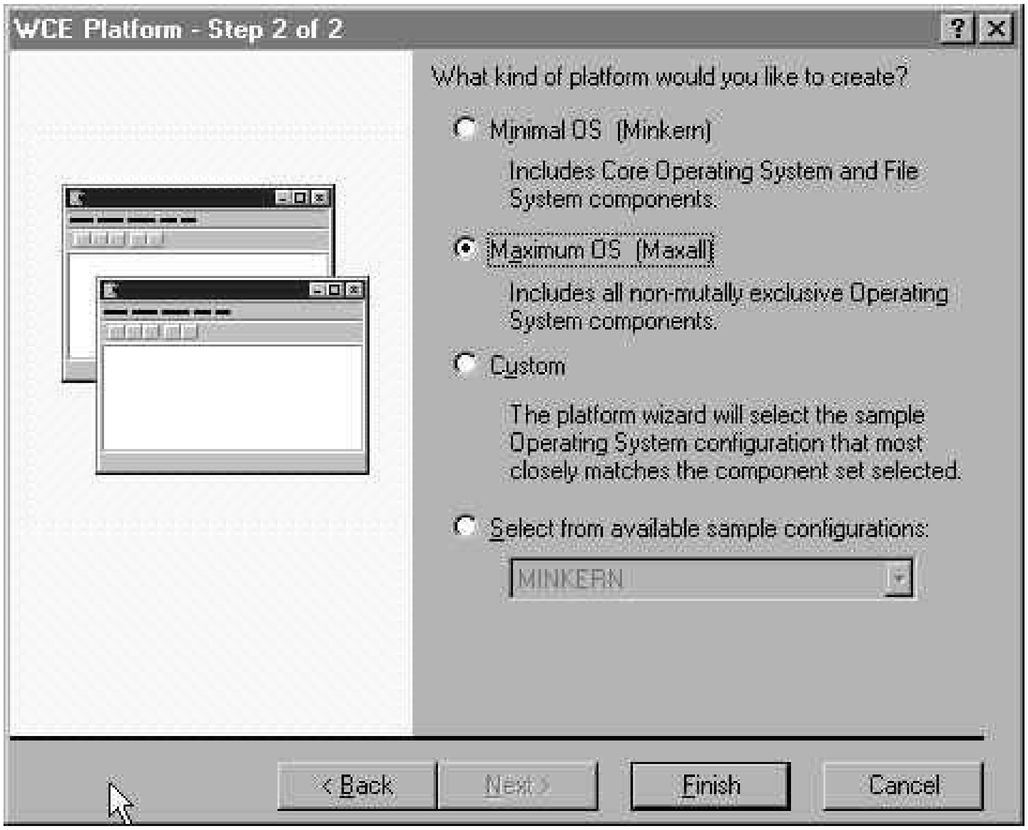
Figure 3.5 Selecting a BSP for your project

CE out of the box. After you create your own BSP, the board support list box in this dialog will list your BSP by name.

Selecting **No BSP** allows you to create a platform without a board support package. In this case, you have to create a BSP and add it to your platform later, which you do using the **My BSP** choice just described. We will cover this aspect in detail later in the chapter.

Note that the wizard automatically fills in the path of the BSP. BSPs are generally found under the PLATFORM subdirectory of Platform Builder.

The second and final dialog box allows you to select the type of platform you are creating (Figure 3.6). In essence, you are selecting one of the sample configurations that come in Platform Builder and that were listed under the coreos type described in Table 3.1. You can modify this choice in finer detail after the platform has been created. However, selecting the best option here will minimize the effort you spend later in fine-tuning the components in your platform. The default choice is MAXALL. Clicking on the **Finish** button brings up one final dialog box, which informs you of the choices you have made so far. When you have confirmed your choices, Platform Builder proceeds to build the new platform.


WCE Platform - Step 2 of 2

What kind of platform would you like to create?

- ○ Minimal OS (Minkern)
  Includes Core Operating System and File System components.
- [✓ ● Maximum OS (Maxall)]
  Includes all non-mutally exclusive Operating System components.
- ○ Custom
  The platform wizard will select the sample Operating System configuration that most closely matches the component set selected.
- ○ Select from available sample configurations:
  MINKERN

Figure 3.6 Selecting a platform for your project

Building and Executing the Platform

Once Adam is created, Platform Builder shows the components included in the project in its **Workspace** window (Figure 3.7). The **Workspace** window has two tabs: **Components** and **Parameters**. The component view shows that several essential components have been included in Adam for us by the Project Wizard. Of particular interest is the component called `MAXALL`.

Recall that we selected `Maximum OS (MAXALL)` by default as the kind of platform we wanted to create. `MAXALL` has several subfolders, each of which corresponds to a module (see Table 3.2). Recall that a module is a set of components that have a common basis of functionality. By including a module, you can pull in all or selected components of related functionality into your project. The modules included in Adam are inherited from `MAXALL`.

Building Adam will build all the components and modules that are part of the platform. Select **Build | Build Platform . . .** from the menu to build the platform. The Platform Builder IDE shows the results of the build in the **Output** window under the **Build** tab. You will notice a series of messages that correspond to important sequences undertaken by the CE build process. Since we will dissect these sequences in Chapter 10, let's look at them just briefly for now.

The first message you see is *Building Platform header files. . . .* The process of building platform header files is also called **building the system**. This phase is responsible for building the Windows CE components and modules that have been included in your platform. During this process, header files specific to your platform are generated. When creating code for your platform, you must include these header files because they will contain only information that is relevant to components that are part of your platform. These header files preclude the possibility that

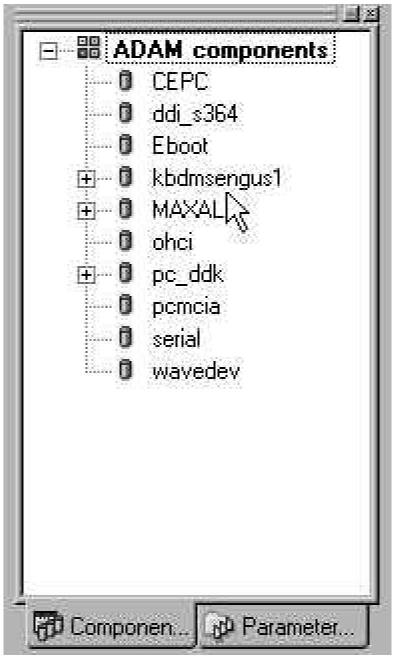


Figure 3.7 The new project's **Workspace** window

Table 3.2 Modules Inherited from MAXALL

Module	Description	Examples
`CE_MODULES`	Operating system components	`filesys` (the file system), GWES, `tcpstk` (TCP/IP), and so on
`IE_MODULES`	Internet Explorer components	`wininet` (the WinInet API)
`WCEAPPS_MODULES`	Windows CE application components	`msgstore` (the message store), `office` (Pocket Office applications)
`WCESHELL_MODULES`	Windows CE shell components	`ctlpnl` (control panel), `explorer`, `webview` (an HTML Web control)

a platform-specific module will compile and link but fail at runtime because of a failure to locate the component. What follows is a series of messages emitted by Cebuild and its helper utilities.

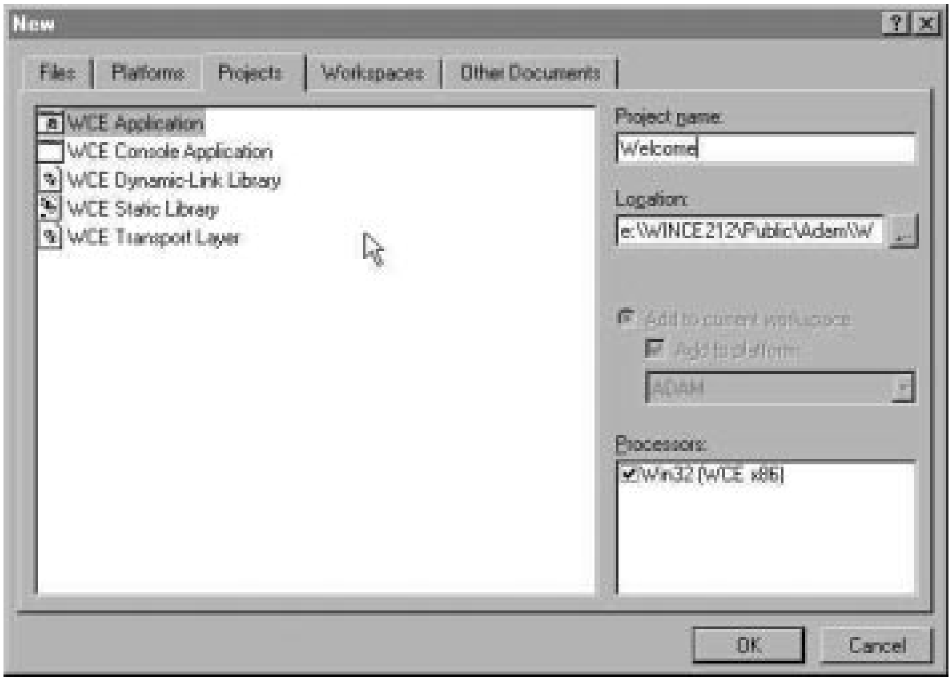
The next step in the build process is accompanied by the message *Building.* . . . This message marks the process of compiling and linking source code from the BSP and platform-specific drivers, libraries, and applications. The verbiage that follows this message comes from the build process as it attempts to build the libraries, drivers, and applications in the platform.

The message *Copying Platform header files* . . . is displayed to indicate the phase in which all the binaries and configuration files generated by the previous stages of the build are copied into a folder, which acts as a repository for the final phase. The final phase takes all the files copied into the repository and creates the image of the Windows CE operating system. This image will conform to the specifications laid out by your platform. It includes the components you specified and will support the processor and board you selected via the wizard. This final phase is preceded by the message *Creating kernel image.* . . . On completion of a successful build, you should see the message *Adam – 0 error(s), 0 warning(s).* We are now ready to create an application!

Creating Applications for Your Platform

A famous software developer once said, "With every platform must come applications." OK, we made that up. However, the fact is obvious: After you're done developing the OAL, drivers, and related libraries for your platform, you will have to turn your attention to delivering applications on your platform that your customers can use to accomplish tasks. Let's start by creating an application called Welcome for our platform Adam.

We create an application using a process similar to the one we use to create a new platform. Select **File | New** from the menu, which eventually launches Project Wizard (Figure 3.8). From that screen, follow the steps we've already outlined for creating a new platform.



WCE Application
WCE Console Application
WCE Dynamic-Link Library
WCE Static Library
WCE Transport Layer

Project name: Welcome

Location: e:\WINCE212\Public\Adam\W

Add to current workspace

Add to platform

ADAM

Processors: ✓ Win32 (WCE x86)

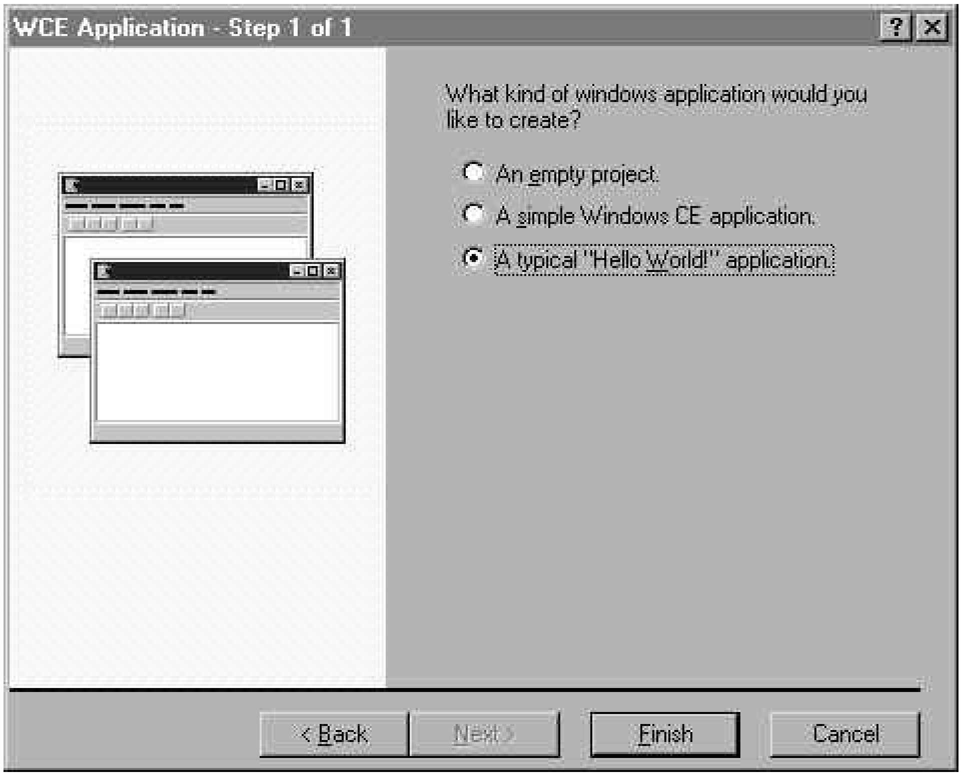
Figure 3.8 Project Wizard

Windows CE Project Wizard

Since a platform already exists in your current workspace, Platform Builder automatically adds a **Projects** tab to the initial **New** dialog box (Figure 3.8). The **Projects** tab allows you to select the type of application you would like to create. You can choose to create a Windows CE executable, a console executable (no windowing support), a Windows CE dynamic link library (DLL), a static library, or a transport layer (which is a DLL with special entry points). For our sample Welcome application, we will choose to create a Windows CE executable. Note that when you type in the name of the application you're creating, the **Location** field is updated to point to a subdirectory under the platform directory for Adam.

Clicking on **OK** launches the Windows CE (WCE) Application Wizard. This simple wizard displays a single dialog box (Figure 3.9). It allows you to select a template for your application. The choice **An empty project** simply allows you to insert a project into your workspace that doesn't contain any files. This option is useful when you have an existing application that you would like to integrate into your platform. In this case you would then insert the files of the existing application into the empty project created by the wizard.

The second option, **A simple Windows CE application**, creates a project with files that compile into an application that has a `winmain` entry point. Windows applications are started at `winmain`. However, the wizard-created `winmain` does nothing except return immediately. This option allows you to insert code into the entry point and start building your application.


WCE Application - Step 1 of 1

What kind of windows application would you like to create?

- An empty project.
- A simple Windows CE application.
- [✓ A typical "Hello World!" application.]

Figure 3.9 Selecting the type of application to create

Finally, the option **A typical “Hello World!” application** creates a complete application that displays “Hello World!” in a dialog box. We’ll choose this option so that we have a complete sample application to run on our platform.

When you click on the **Finish** button, the wizard shows you a single dialog box displaying the type of application it will generate. The application Welcome is then inserted into your project workspace.

Applications are always inserted in Platform Builder’s project workspace. Platform Builder now has two views: a platform view, which contains Adam, and a project view, which contains Welcome. The platform view contains a view of the platform under development. The project view displays applications being developed for the platform. You can toggle between the project view and the platform view from the Platform Builder toolbar using the buttons shown in Figure 3.10. These buttons are mutually exclusive. When one is depressed the other is not, and vice versa; you can view only either your platform or your project at any given time. This separation is provided to facilitate the development of both platform software and applications for that platform.

Figure 3.10 Switching between platform and project views

Building the Application

To build `welcome.exe`, you must select the menu item **Build | Build Welcome.exe**. Platform Builder first attempts to build the platform. If the platform builds successfully, then Platform Builder builds `welcome.exe`. You can make this selection also by pressing the **F7** key. Before attempting to build `welcome.exe`, the build process checks to see if all the platform header files have been generated since the last time the project was modified. Recall that the platform build process has generated header files on the basis of the selected configuration. These header files are to be used by any applications or drivers that must run on the platform. This dependency is built into the project file generated for `welcome.exe`.

Another dependency added explicitly to the IDE when the project is generated is for the platform build. The platform build for Adam will now attempt to build `welcome.exe` after the platform build completes successfully. You can check this new dependency by selecting **Platform | Dependencies...** .

Successfully building `welcome.exe`, however, does not automatically include it in the operating system image that will be uploaded to the target platform. The steps required to do so must be performed manually. Alternatively, you can set up Platform Builder to perform these for you automatically. We will take the latter approach.

Testing the Application during Rapid Development

While still developing the application, we will make use of an interesting CESH feature to shorten our development and debugging cycle. CESH transfers an image from the workstation to the target and then continues to monitor the kernel on the target using an undocumented API. If the kernel attempts to load a module that does not exist in the operating system image, it asks CESH for a copy of the module. If CESH finds the module in its working directory, it uploads a copy of the module to the target platform, where the kernel copies it into RAM. This process causes the module to be executed on the target platform. Once the module is done executing, it is unloaded from RAM and its copy is discarded.

This feature allows us to create an application and simply copy it into the release directory without including it in the operating system image. Thus, when we modify the application, we simply copy a new version to the release directory and execute the application on the target platform. On execution, the kernel asks CESH to load the module because it is unable to locate it in the image. CESH obliges by sending the kernel a fresh copy of the module. Once the application is terminated, we can repeat the entire cycle with a new copy of the module.

If we chose to include the application in the operating system image, we would have to create a new copy of the image for every change in the application. The new copy would then have to be uploaded to the target. The entire image of the operating system takes considerably longer than just a single executable to upload to the target.

After a successful build, the build process for the project copies the output file into the release directory for the platform. The variable `_FLATRELEASEDIR`, defined by the Windows CE build process, contains the value of the release directory. The variable is set up for each platform, and for Adam it is set to `\WINCE212\PUBLIC\ADAM\RELDIR\X86_DEBUG` for the debug build of the platform.

Once we have our custom build step in place, `welcome.exe` will be copied, after every successful build, into the release directory. From here, it will initially be uploaded *on demand* by

CESH to the target platform. Later, we will integrate it directly into the operating system image. First, though, we need to figure out how to upload the operating system to the target platform and run it there.

Running Windows CE on a CEPC

Running Windows CE is not as straightforward as creating a new platform and project. This task requires two separate operations. The final build of Windows CE, also referred to as the **image**, must be uploaded to the target platform first. Next, the kernel debugger built into the Platform Builder IDE must connect with the debug kernel on the target. There are several ways to upload an image and debug it. For starters, we will focus on uploading via the parallel port and debugging over the serial port. These methods are supported on all versions of Windows CE, including those that use the early version of Platform Builder called the Embedded Development Kit (EDK).

How does one upload the image to that target platform? You'll need a bidirectional enhanced parallel port (EPP) on both sides of the equation: the workstation and the platform. It is a good idea to make sure that the parallel port is in EPP mode by checking the settings in the system BIOS. The parallel port cable itself must be tweaked a bit. First you'll need a male DB25 connector at both ends. Next, the cable itself must be wired as shown in Table 3.3. You can order such a cable from a variety of vendors, but it's not difficult to make one yourself. Start with a regular parallel port cable and add an expansion box with the appropriate pins soldered to meet the requirements in Table 3.3. If you don't have an EPP-compatible port on the workstation or the CEPC target, you can buy a plug-in ISA card that supports an EPP-compatible port. The EPP-compatible port is jumper configurable on these cards. Be sure to avoid conflicts with your regular parallel port in the workstation or CEPC.

The serial connection used to debug the kernel is fortunately quite straightforward, simply requiring a serial cable. Once both connections have been set up, you are ready to upload.

An upload is carried out via the Windows CE Debug Shell Tool (CESH). CESH is capable of uploading images to the target via several transports, including serial and Ethernet. For the exercise we are conducting, we have already decided to use CESH's parallel port capabilities.

To use CESH to upload the image, we must first start a command prompt. Platform Builder provides a convenient way to create a command prompt and change directory to the release directory for the platform under development with a few simple mouse clicks. From the **Build** menu, select **Open Build Release Directory**. This selection will launch a command prompt in the release directory of the platform. The release directory of a platform refers to the repository for the final phase, `\WINCE212\PUBLIC\ADAM\RELDIR\X86_DEBUG`. This directory contains the final Windows CE image to be uploaded to the target. The image is called `nk.bin` and must be sent to the target in its entirety. This is accomplished by the following command:

```
Cesh -p CEPC Nk.bin
```

Table 3.3 CESH Parallel Port Cable Pin Connections

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Pin	10	2	3	4	5	6	7	8	9	1	14	16	17	11	—	12	13	18	19	20	21	22	23	24	25

The `-p` option of CESH indicates the profile that CESH must use. In this case, the profile is called CEPC. Profiles for CESH are set up in the registry by the Platform Builder installation procedure. The CEPC profile urges CESH to use the parallel port with some predefined settings that configure the parallel port protocol. When using this profile, CESH starts passing `nk.bin`, chunk by chunk, to the parallel port. The parallel port base address and interrupt level are assumed to be the default values for LPT1 (base address `0x378` and IRQ 7). Should this change on your workstation, you must edit the CEPC profile directly to reflect the changed parallel port settings. To change these settings, you must change the registry values shown in Listing 3.1.

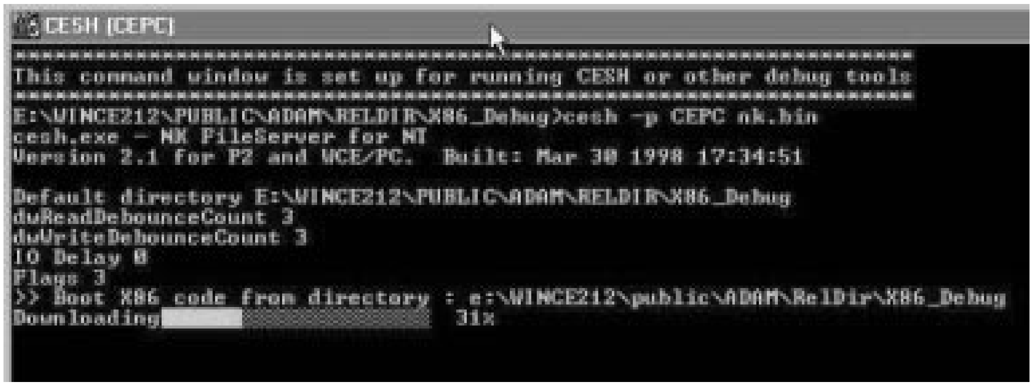
Listing 3.1 CESH registry settings

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ppsh\CEPC:
InterruptLevel  = REG_DWORD 7
InterruptMode   = REG_DWORD 1
InterruptVector = REG_DWORD 7
PortAddress     = REG_DWORD 0x378
```

On a CEPC, at the other end, a tool called Loadcepc communicates with CESH to receive pieces of the operating system image and then loads and boots the operating system. You must start by creating a bootable floppy, preferably with MS-DOS 6.22, and put `loadcepc.exe` on it. This floppy disk should be used to boot the CEPC. Loadcepc is available in Platform Builder, and its source code is in the CEPC platform directory. Once the two utilities start to communicate, a progress bar indicates the portion of the image that has been transferred successfully between the workstation and the CEPC (Figure 3.11). After the entire image has been uploaded, Windows CE will boot on the CEPC.

Kernel Debugging

The Platform Builder IDE has a built-in kernel debugger. You can launch this debugger by selecting the **Build | Start Debug | Go** menu item or by hitting the **F5** key. The kernel debugger displays its output in the IDE output window under the **Debug** tab. You can view the output window by selecting the **View | Output** menu item. Then select the **Debug** tab.



```
*****************************************************************
This command window is set up for running CESH or other debug tools
*****************************************************************
E:\WINCE212\PUBLIC\ADAM\RELDIR\X86_Debug>cesh -p CEPC nk.bin
cesh.exe - NK FileServer for NT
Version 2.1 for P2 and WCE/PC.  Built: Mar 30 1998 17:34:51

Default directory E:\WINCE212\PUBLIC\ADAM\RELDIR\X86_Debug
dwReadDebounceCount 3
dwWriteDebounceCount 3
IO Delay 0
Flags 3
>> Boot X86 code from directory : e:\WINCE212\public\ADAM\RelDir\X86_Debug
Downloading                     31%
```

Figure 3.11 Uploading an image using CESH

Like downloading, debugging can occur over an Ethernet connection. It can also occur over the serial port. We'll choose the latter because it is easy to set up and use. A simple serial cable (not null modem) will do the trick. You can improve the debugger throughput by increasing the speed at which the debugger and target platform communicate over the serial port. On the workstation, select **Build | Debugger | Remote Connection** to view the **Remote Connection** dialog box. In the **Connection** list box, you must select **Kernel Debugger Port** and then click on **Settings...** (see Figure 3.12).

At the other end, the change in the default speed of the serial connection must be passed on Loadcepc. This is accomplished with the `/B:115200` option. In addition, if you are using the S3 display drivers that are bundled with the CEPC, you must instruct Loadcepc to initialize the display adapter in 640_480 mode. This is done via the `/D:2` option. Finally, Loadcepc must be told to expect the image to come down the parallel port, with the `/P` option.

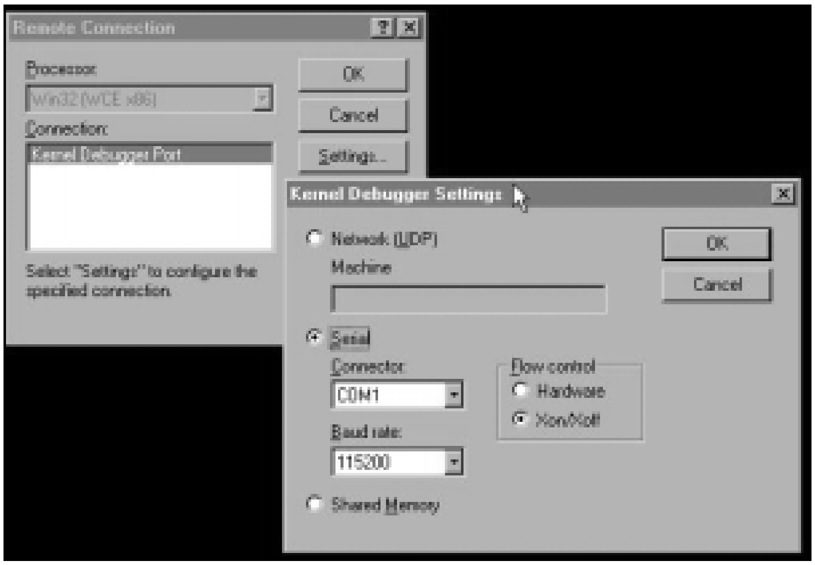
```
Loadcepc /B:115200 /D:2 /P
```

You can request Loadcepc to use the serial port to download the binary image, provided that you set up CESH to do the same on the workstation. In this case, you specify the `/Q` option. The COM port to be used by Loadcepc is identified with an additional option, `/C:{1 | 2}`.

After you initiate the kernel debugger, the first messages in the debug window announce the handshaking between the debugger and the target:

Kernel debugger waiting to connect on com1 at 115200 baud
Host and target systems resynchronizing . . .

When the debugger is started, you may get the erroneous error message shown in Figure 3.13. You can safely ignore this message. The IDE is simply trying to find an updated copy of



Processor: Win32 (WCE x86)
Connection: Kernel Debugger Port
Select "Settings" to configure the specified connection.

Network (UDP)
Machine:
[✓ Serial]
Connector: COM1
Baud rate: 115200
Flow control: Hardware [✓ Xon/Xoff]
Shared Memory

Figure 3.12 Setting the kernel debugger port's speed

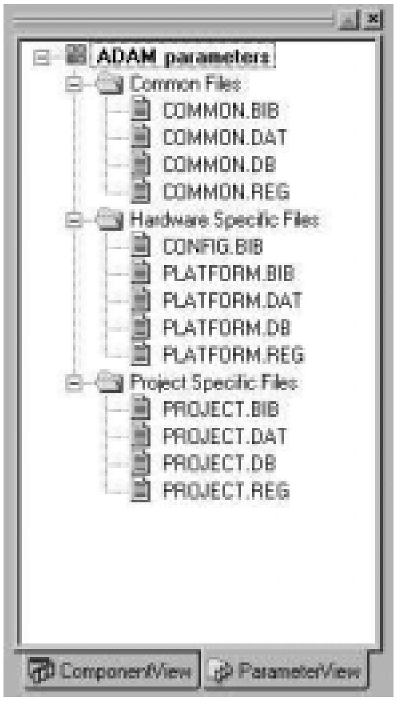

ADAM parameters
Common Files
COMMON.BIB
COMMON.DAT
COMMON.DB
COMMON.REG
Hardware Specific Files
CONFIG.BIB
PLATFORM.BIB
PLATFORM.DAT
PLATFORM.DB
PLATFORM.REG
Project Specific Files
PROJECT.BIB
PROJECT.DAT
PROJECT.DB
PROJECT.REG

Figure 3.13 Error message when the debugger is started

`welcome.exe` in the wrong release directory, a throwback from the days of the Embedded Development Kit (EDK).

Let's review what we've done so far:

1. Created a new platform, Adam, based on the `MAXALL` configuration
2. Created a simple new application for Adam called `welcome.exe` that displays a "Hello World!" message in a dialog box
3. Connected the workstation and CEPC for downloading over the parallel port and debugging over the serial port
4. Downloaded the operating system image to CEPC and watched it boot on the CEPC

After the last step is completed, you should see a Handheld PC–like shell pop up, complete with a Windows CE task bar. Let's take our configuration for a spin. Select the **Start** button and then click on **Run...** . In the **Run** dialog box, type in "welcome.exe" and then click on **OK**. The shell will attempt to execute `welcome.exe`, a request that will be translated in the inner layers of the kernel into a request from CESH to load `welcome.exe` from the release directory on the workstation:

```
KernelLoader: Using PPFS to load file welcome.exe
```

PPFS stands for Portable Parallel File System, which supports loading on demand. On the target platform, you should see `welcome.exe` running and greeting the world in a dialog box. You can close `welcome.exe` by right-clicking on its icon in the task bar and selecting **Close**.

You can try out our rapid development technique by stopping the debugger, making a change to the string displayed by Welcome, and then building the project, restarting the debugger and executing `welcome.exe` from the **Run** dialog box on the target.

Debugging Capabilities of CESH

CESH is more than just a downloading utility. It is called the Windows CE Debug Shell Tool for a reason. After the download is over, CESH gives you a command prompt. You can type in a variety of commands at the prompt to get useful information from the kernel running on the target platform. Typing "?" retrieves a list of all the commands supported by CESH (see Table 3.4).

Table 3.4 Commands Supported by CESH

Command	Options	Function
`break`		Stops the kernel at the current line of execution. This command can be used to halt the kernel and set a new breakpoint.
`dd`	`addr [<size>]`	Displays the contents of `addr`. The optional size argument tells the command how many bytes to display.
`df`	`filename addr [<size>]`	Writes contents of `addr` to the file specified by `filename`. The optional size argument tells the command how many bytes to display.
`dis`		Tells the virtual memory manager to mark all discardable memory as available.
`gi`	`proc`	Lists all the processes running on the target. Processes are created from executables by the operating system.
	`thrd`	Lists all the threads running on the target. A process may have multiple threads.
	`mod`	Lists all the modules loaded on the target. Modules are DLLs that are loaded by executables. Device drivers are DLLs loaded by `device.exe` (mostly); as such, they appear under the modules' listing.
	`all`	Lists processes, threads, and modules. This is the default value for the `gi` command.
`kp`	`pid`	Kills the process with process ID `pid`.
`mi`	`kernel`	Displays detailed information on memory used by the kernel.
	`full`	Displays memory maps used by all processes and modules in the system.
`run`	`filename`	Runs the file specified by `filename` in batch mode.
`s`	`process`	Starts a new process. This command is useful when you don't have a user interface on the target platform that can be used to start a process. The `s` command can also be used for rapid development because it supports load on demand via PPFS.
`zo`		Displays and modifies debug zones for a process or module. This command is discussed in more detail in Chapter 9.

After booting Adam successfully on a CEPC, you can run the `gi` command from the CESH command prompt and observe output similar to that shown in Listing 3.2. When executing `welcome.exe`, you will notice it listed as a process. The `gi` command prints out information that is used not just for debugging, but also for indexing other commands. For example, the process or module index, a number prefixed by either *P* or *M*, is passed to the `zo` command to identify the entity whose debug zone is being displayed or modified. Debug messages emitted by the kernel are prefixed with the address of the line that originated the message. You can map these addresses by looking at the `pModule` field of a module in the output of `gi`.

Listing 3.2 shows the processes running on the target platform. The file `nk.exe` is the Windows CE kernel process, `filesys.exe` manages the CE file system, the debug shell that communicates with CESH is `shell.exe`, `device.exe` loads and manages all the device drivers on the target platform, `gwes.exe` is responsible for creating and managing windows and messages, and finally, `explorer.exe` is the Handheld PC–like shell included in our image.

Listing 3.2 Running the `gi` command in CESH

```
Windows CE>gi
PROC: Name            hProcess: CurAKY :dwVMBase:CurZone
THRD: State :hCurThrd:hCurProc: CurAKY :Cp:Bp:CPU Time
 P00: NK.EXE          00ffefe2 00000001 02000000 00000100
 T    Blockd 00ffdefe 00ffefe2 00000001  3  3 00:00:00.000
 T    Blockd 00ffe012 00ffefe2 00000001  7  7 00:00:00.082
 T    Blockd 00ffe31e 00ffefe2 ffffffff  2  2 00:00:00.005
 T    Blockd 00ffef1a 00ffefe2 00000001  1  1 00:00:00.063
 P01: filesys.exe     00ffdaf6 00000002 04000000 00000000
 T    Blockd 00ffdb16 00ffdaf6 00000003  3  3 00:00:01.491
 P02: shell.exe       00ffbd42 00000004 06000000 00000001
 T    Runing c0ffc942 00ffbd42 ffffffff  1  1 00:00:02.965
 P03: device.exe      00ffb70a 00000008 08000000 00000000
 T    Sl/Blk 00eb5366 00ffb70a 00000009  2  2 00:00:00.081
 T    Blockd 00ff517e 00ffb70a 00000009  3  3 00:00:00.001
 T    Blockd 00ff551a 00ffb70a 00000009  3  3 00:00:00.004
 T    Sl/Blk 00ff7d56 00ffb70a 00000009  2  2 00:00:00.001
 T    Sl/Blk 20ff98ce 00ffb70a 00000009  2  2 00:00:00.132
 T    Blockd 20ff98ee 00ffb70a 00000009  2  2 00:00:00.000
 T    Blockd 00ffb72a 00ffb70a 00000009  3  3 00:00:03.026
 P04: gwes.exe        00fd626e 00000010 0a000000 00000040
 T    Blockd 00f60e02 00fd626e 00000011  3  3 00:00:00.029
 T    Blockd 00f6200a 00fd626e 00000011  3  3 00:00:00.014
 T    Blockd 00f6230e 00fd626e 00000011  1  1 00:00:00.000
 T    Blockd 00f625f2 00fd626e 00000011  1  1 00:00:00.004
 T    Blockd 00f62a56 00fd626e 00000011  1  1 00:00:00.193
 T    Blockd 00f62b6a 00fd626e 00000011  1  1 00:00:00.046
 T    Sl/Blk a0f74386 00fd626e 00000011  1  1 00:00:00.291
 T    Sl/Blk 00fd628e 00fd626e 00000011  3  3 00:00:01.029
 P05: explorer.exe    00f60c5a 00000020 0c000000 00000000
 T    Blockd 40ea6c02 00fd626e 00000031  3  3 00:00:01.769
 T    Blockd 00eb572a 00fd626e 00000031  3  3 00:00:04.446
 T    Blockd 00f60c7a 00f60c5a 00000021  3  3 00:00:00.430
 MOD: Name            pModule :dwInUSE :dwVMBase:CurZone
 M00: unimodem.dll    80eb5584 00000008 01160000 0000c000
```

```
  M01: TAPI.DLL        80eb5ec0 00000008 01180000 0000c000
  M02: SHLWAPI.dll     80f4e640 00000020 01060000 00000000
  M03: IECEEXT.dll     80f4e810 00000020 01090000 00000000
  M04: WININET.dll     80f4ed40 00000020 00f70000 0000c000
  M05: imgdecmp.DLL    80f4ef44 00000020 00ed0000 00000000
  M06: webview.dll     80f493ec 00000020 00e20000 00000000
  M07: commctrl.dll    80f49c28 00000020 01520000 00000000
  M08: CEShell.DLL     80f49ef8 00000020 00ef0000 00000000
  M09: OLEAUT32.dll    80f60358 00000020 01390000 00000000
  M10: ASForm.dll      80f627c8 00000020 00f50000 00000000
  M11: kbdmouse.dll    80f62d78 00000010 00d20000 00000000
  M12: DDI.DLL         80fd6b80 00000010 00d30000 00000003
  M13: Redir.dll       80ff50c0 00000008 01110000 0000c001
  M14: irdastk.dll     80ff6b78 00000008 011a0000 00008000
  M15: netbios.dll     80ff76ec 00000008 010f0000 0000c000
  M16: dhcp.dll        80ff7840 00000008 01220000 0000ffff
  M17: arp.dll         80ff7a20 00000008 01290000 00000001
  M18: tcpstk.dll      80ff8250 00000008 011d0000 00000001
  M19: ppp.dll         80ff8800 00000008 01340000 00000000
  M20: CXPORT.dll      80ff89a0 00000008 01330000 0000c000
  M21: AFD.Dll         80ff8c88 00000008 012a0000 00000000
  M22: ole32.dll       80ff8f88 00000028 013d0000 00000000
  M23: softkb.DLL      80ff9320 00000008 01500000 00000000
  M24: WINSOCK.dll     80ff95a0 00000028 01310000 0000c000
  M25: IRCOMM.DLL      80ff971c 00000008 01320000 0000c003
  M26: irsir.dll       80ff9e74 00000008 00ce0000 0000c000
  M27: NDIS.Dll        80ffa6bc 00000008 01270000 00000000
  M28: msfilter.dll    80ffa810 00000008 00cb0000 00000000
  M29: CEDDK.dll       80ffaef0 00000018 00da0000 00000000
  M30: Serial.Dll      80ffb020 00000008 00cf0000 00000000
  M31: toolhelp.dll    80ffbf68 00000004 015a0000 00000000
  M32: coredll.dll     80ffe690 0000003f 015e0000 00000000
 Windows CE>
```

Integrating New Components into the Image

We can use the rapid development scenario described in earlier sections until we finish all basic development of `welcome.exe`. We made changes to the Welcome application so that it displays a message greeting on startup. We also added a caption and a system menu so that the window can be conveniently closed. The text now appropriately welcomes users to the first Windows CE platform in this book. Welcome will be started every time Adam boots so that the welcome message is the first message users see. Let’s see how to add this somewhat concocted example to the image and automatically start Welcome on startup.

To add `welcome.exe` to the operating system image, we will need to modify a file in the parameter view of the platform workspace. Figure 3.14 shows the parameters view window for Adam. The files displayed in this window are used to specify how the operating system image will be built.

We’ll delve into the specifics of each file elsewhere throughout the book, so a brief overview will suffice for now. The parameter view is organized into a set of files in three different cate-

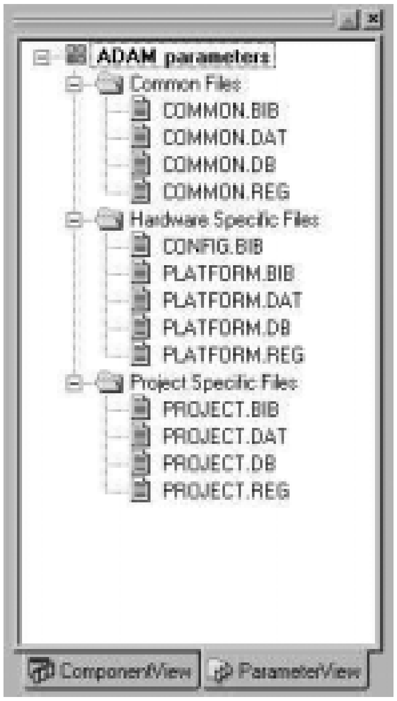

ADAM parameters
Common Files
COMMON.BIB
COMMON.DAT
COMMON.DB
COMMON.REG
Hardware Specific Files
CONFIG.BIB
PLATFORM.BIB
PLATFORM.DAT
PLATFORM.DB
PLATFORM.REG
Project Specific Files
PROJECT.BIB
PROJECT.DAT
PROJECT.DB
PROJECT.REG

Figure 3.14 Parameter view

gories. The option **Common Files** refers to files that are provided by Microsoft to specify how the operating system must be built with Microsoft-supplied components. You should not modify these files. Any modifications you make will affect *all* platforms and projects you create.

Different types of files contain different types of information:

- **BIB files** contain information about which files, executables, and libraries should be included in the operating system image.
- **DAT files** contain a directory map for the file system on Windows CE.
- **DB files** contain information on how the Windows CE database should be initialized.
- **REG files** contain a map of the system registry on startup.

The option **Hardware Specific Files** refers to files that specify configuration and initialization when you're building the platform.

Files specified by the option **Project Specific Files** contain additional information in each of these categories that is specific to the project being created. The word *project* in this case does not refer to Welcome, although that's what we've been telling you all along. Here the term *project* is a confusing throwback to the old days of the EDK. Under the terminology used then (and still reflected in the Platform Builder directory and file structure), CEPC is our platform and

Adam is our project. This clarifies the role of these categories. Platform-specific files typically contain information regarding platform-specific drivers and modules. Project-specific files contain information about platform-independent modules. All the files are used by the set of Windows CE build tools when the image is being created.

If you have been following along closely, you might have already identified the BIB file as the one we must modify to integrate `welcome.exe` in the final operating system image. The first question, of course, is, To which BIB file should we add `welcome.exe`? The answer is simple enough: Since `welcome.exe` is platform-independent, it should be added to the BIB file `project.bib` in the Project Specific Files category.

Listing 3.3 shows the line that needs to be added to the BIB file to integrate the executable in the image. This line instructs the appropriate build tool to add the file `$(_FLATRELEASEDIR)\welcome.exe` to the image, where it will be called `welcome.exe`. The file must be loaded into the `NK` section of memory in ROM and must be uncompressed (type `U`). Sections of memory are set up in the file `config.bib`. We'll say more about this later in the book. Leaving the executable uncompressed gives the operating system the option of running the program in place in ROM. This avoids the work that the loader would normally do of uncompressing a file from ROM into RAM before executing it. In addition to the time gained because the file does not need to be uncompressed, precious RAM is made available for other programs.

Listing 3.3 Adding `welcome.exe` to `project.bib`

```
MODULES
;  Name            Path                                          Memory Type
;  -------------   --------------------------------------------  -----------
   welcome.exe     $(_FLATRELEASEDIR)\Welcome.exe                NK     U
```

Now that Welcome is part of the operating system image, it will be transferred to the target platform by CESH. PPFS will no longer be required to load it on demand because the loader will find the file in the local file system on the target platform.

We still need to make Welcome execute automatically on startup. The kernel looks for a special registry entry after starting the file system to look for modules to load during startup. Since the REG file allows us to specify the contents of the system registry when the target platform is initialized, we can zero in on the platform-independent registry file `project.reg`. To automatically execute Welcome on startup, we must add the following lines to the registry initialization file `project.reg`:

```
[HKEY_LOCAL_MACHINE]\Init
"Launch80"="Welcome.exe"
"Depends80"=hex:1E,00
```

The `Launch80` registry entry tells the kernel to launch the program `welcome.exe`. The function performed by the `Depends80` line is not immediately obvious. This registry entry lists a hex number (`001E`) as its value. To discover the significance of this number, we must take a peek at the common registry file, `common.reg`. This file has its own `init` section that is used to specify programs to launch on startup. An excerpt of this file is shown in Listing 3.4.

Listing 3.4 The `init` registry key in `common.reg`

```
 [HKEY_LOCAL_MACHINE\init]
; @CESYSGEN IF CE_MODULES_SHELL
      "Launch10"="shell.exe"
; @CESYSGEN ENDIF
; @CESYSGEN IF CE_MODULES_DEVICE
      'Launch20"="device.exe"
; @CESYSGEN ENDIF
; @CESYSGEN IF CE_MODULES_GWES
IF NOGUI !
      "Launch30"="gwes.exe"
      "Depend30"=hex:14,00
ENDIF
; @CESYSGEN ENDIF
```

The hex number 001E corresponds to 30. The `Launch30` registry item in the listing is the module `gwes.exe`. The entries related to `gwes.exe` instruct the kernel to launch `gwes.exe` on startup. The `Depends80` line simply tells the kernel to wait until `gwes.exe` has been launched before launching `welcome.exe`, because `welcome.exe` uses the windows and message support made available by `gwes.exe`. Finally, the number 80 was chosen at random from among numbers higher than 30. This number indicates the sequence in which the registry entries must be processed. We need a number higher than that corresponding to the last entry in the `common.reg` file.

However, this dependency only synchronizes the launch sequence of different modules. It does not guarantee that `gwes.exe` will have fully initialized its services before `welcome.exe` is executed. To make sure that the window manager is available, Welcome must call `IsAPIReady` with its lone argument set to the constant `SH_WMGR`.

```
…
#include “windev.h”
…
While (!IsAPIReady (SH_WMGR))
       Sleep (1000);
```

The `IsAPIReady` call will return a value of `TRUE` if the API specified by the constant is available. If a value of `FALSE` is returned, the API is not yet ready for use and Welcome must wait in a loop for the API to initialize before it executes the remainder of its code.

Note the inclusion of the header file `windev.h`. This file contains the function prototype for `IsAPIReady`. Welcome loops and checks the value returned by this call to see if the window manager API is available. If the API is not available, Welcome sleeps for one second before checking again.

Now that we are ready with all the changes, we can test out the build by selecting the **Platform | Build | Build Platform** menu item after switching to the platform view. To confirm that `welcome.exe` has been included in the image, sift through the final output of the build process that lists all the files included in the image. The entry for `welcome.exe`, if included in the image, should look like Listing 3.5.

Z

Note:
To make sure that the compiler can find the file `windev.h`, you must add the include path in the `CESYSGEN` directory of Adam to the list of additional include directories in which the compiler may find included header files. `CESYSGEN` is the directory in which header files generated for the specific platform are placed. To add this directory to the include path for the compiler, you must click on the **Project | Settings...** menu item and then click on the **C/C++** tab. Next, select **Preprocessor** in the **Category:** combobox. In the **Additional include directories:** edit box, type in "$(_WINCEROOT)\PUBLIC\ADAM\CESYSGEN\OAK\INC". This directory path will be passed to the compiler with the `/I` flag, which instructs the compiler to search the specified directory for the include path after looking in the standard include paths.

Listing 3.5 Output from the build process

```
MODULES Section
Module       Section  Start     Length  psize   vsize   Filler
------------ -------- --------- ------- ------- ------- ------
…
welcome.exe  .text    80a6f000h    8192    5632    5508 o32_rva=1000h
welcome.exe  .rdata   80a71000h    4096     512     289 o32_rva=3000h
welcome.exe  .idata   80a72000h    4096    1024     565 o32_rva=5000h
welcome.exe  .rsrc    80a73000h    4096    2560    2254 o32_rva=6000h
…
welcome.exe  .data    8024ffdbh       0       0    2584 FILLER
…
welcome.exe  E32      8052b55ch     100                 FILLER
welcome.exe  O32      8052b5c0h     120                 FILLER
```

The next time you download the new image and boot the CEPC, Welcome should pop up as it did before on startup, welcoming the user to our new platform, Adam.

Customizing the Build Using Environment Variables

Once a Platform has been defined, you can use it to produce builds of the operating system that vary slightly in flavor without having to change the underlying definition of the platform. For example, you could choose to build Adam with a different display driver (either an S3 ViRGE or a Chips & Technologies CT65555) or choose to omit support for audio altogether. Such modifications to a platform are made dynamically through environment variables.

You can set environment variables in Platform Builder by selecting the **Platform | Settings...** menu item and then clicking on the **Environment** tab. Most, although not all, of the environment variables carry the prefix *IMG*. This prefix is short for *image*, which refers to the binary image of the operating system. *IMG* variables modify the way in which the image is built. The variables available in the **Environment** tab are listed in Table 3.5. To take effect, these variables must be defined. You can set up the definition by simply setting the variable to 1. Most variables work in a negative fashion. When set, they have the effect of *not* including a component or module. These variables have the word *NO* in their names and can be particularly confusing to use.

Table 3.5 Environment Variables Used by Platform Builder

Environment Variable	Usage	Description
FBBPP	Sets the number of bits per pixel on the display supported by the platform. This value is usually set to FB16BPP to indicate a display of 16 bits per pixel.	When FBBPP is set, the display drivers in the CEPC platform display drivers (S3 ViRGE, S3 Trio64, CT65555x, and so on) compile themselves for a pixel depth of 16; otherwise the default depth of 8 bits per pixel is used.
IME	Selects the type of Input Method Manager (IME) bundled with the platform. If the value of this variable is PIME, the Pocket IME version is added to the image. Pocket IME does not contain a user interface. If the value of this variable is TESTTIME, support that allows you to create your own IME is added to the image. The default value is IME98, which adds an IME with a user interface that mimics the full-blown Win32 API IME.	The value of IME is checked for the configuration file common.bib, and depending on its value, corresponding modules are added to the image.
IMGACMSAMPLES	Adds the sample codec driver (cegsm.dll) and the sample filter driver (msfilter.dll) to the image.	The value of IMGACMSAMPLES is checked in the project-specific configuration file project.bib. If the value is defined, the corresponding sample drivers are added to the image. However, the only project in Platform Builder 2.12 that supports this behavior is the MINSHELL project. Changing this value in our sample platform Adam will have no effect because it is derived from MAXALL.
IMGBIGFLASH	Configures the image to utilize a larger area of flash memory (an additional 8MB of flash is assumed).	IMGBIGFLASH is used in config.bib as a conditional to lay out the image to use the additional 8MB. This conditional is used only for the Odo platform. Hence, it doesn't affect Adam because Adam is derived from the CEPC platform.
IMGCOM2	Configures COM2 on the platform at 0x3E8.	IMGCOM2 is used in platform.reg, which initializes the system registry with platform-specific information.
IMGCOM3	Configures COM3 on the platform at 0x2E8.	IMGCOM3 is used in platform.reg, which initializes the system registry with platform-specific information.
IMGCOMMDEMOS	Adds two sample communication programs—ping.exe and ipconfig.exe—to the image.	IMGCOMMDEMOS is used in common.bib, which specifies the CE system files included in the operating system image.

Table 3.5 *Continued.*

Environment Variable	Usage	Description
IMGCTLPNL_G	Unknown	
IMGDUB	Leaves additional space in the image layout to include the CE dial-up boot loader (DUB). The DUB is a component that can be used to upgrade the operating system image.	IMGDUB is used in config.bib to make space for the DUB file in the image layout. It is used only in the config.bib file for the Odo platform; hence, it doesn't affect Adam because Adam is derived from CEPC.
IMGEBOOT	Adds support for Ethernet debugging by bundling the Ethernet boot loader in the image.	IMGEBOOT is used in config.bib to make space for Ethernet debugger modules. It is used only in the config.bib file for the Odo platform.
IMGFLASH	Lays out the image in flash memory as opposed to RAM.	IMGFLASH is used in config.bib to make space for Ethernet debugger modules. It is used only in the config.bib file for the Odo platform.
IMGICONPOSITIONS	Allows explicit positioning of icons on the desktop as opposed to automatic positioning set by the shell.	Not used.
IMGMOREAPPS	Allows additional applications to be added to the image.	Not used.
IMGMORERAM	Lays out the image to simulate less area for ROM and release it for use as RAM area.	IMGMORERAM is used in config.bib to allocate more space to the RAM section. Use this setting if your image is relatively small (less than 6MB). The total memory assumed in the system is 32MB.
IMGMOREROM	Lays out the image to simulate less area for R<K>AM <K>and release it for use as ROM area.	IMGMOREROM is used in config.bib to allocate more space to the ROM section. Set this variable if your image is large (more than 6MB but less than 20MB). The total memory assumed in the system is 32MB.
IMGMOREROM16	Used just like IMGMOREROM, but when total memory in the system is assumed to be 16MB.	
IMGNOBROWSER	Excludes Internet browser components in the operating system. Since Windows CE Help uses browser components, also disables Help. We are now in *NO* territory, so note that not defining this variable causes components	IMGNOBROWSER is used (1) in wceshell.bib to add the Pocket Internet Explorer files iexplore.exe, webview.dll, and imgdecmp.dll; (2) in wceshell.reg to specify browser proxy, start and search pages, and file associations; and (3) in wceapps.bib to

(continued)

Table 3.5 *Continued.*

Environment Variable	Usage	Description
	to be added. The same goes for the *NO* variables that follow.	ensure that browser components are bundled before Help is added.
`IMGNOCEDDK`	Excludes the CE DDK library in the image. The CE DDK library provides a processor-independent interface to the kernel, memory, and I/O to device drivers.	`IMGNOCEDDK` is used (1) in `common.bib` to include `ceddk.dll` in the image, and (2) in `platform.bib` for the CEPC platform to include the file `pc_ddk.dll` and rename it as `ceddk.dll`. The latter directive overrides the former when the image is being built.
`IMGNOCOMM`	Excludes all communications components in the image.	`IMGNOCOMM` is used in `common.bib` to include communication-specific components in the image.
`IMGNOCONN`	Excludes the components `repllog.exe` and `rapisrv.exe`, which communicate with Windows CE Services, which provide connectivity to a server (referred to as a *desktop*). An example of such an application is ActiveSync.	`IMGNOCONN` is used (1) in `common.bib` to include the components, and (2) in `common.reg`, where it is used to set up a registry entry that specifies the version of the modules. You must successfully negotiate this version number when you're communicating with a desktop.
`IMGNOCONSOLE`	Excludes the console support component `console.dll` and the console command language processor `cmd.exe` in the image.	`IMGNOCONSOLE` is used in `common.bib` to include the components.
`IMGNOCTLPNL`	Excludes the control panel and all applicable control panel applets in the image.	`IMGNOCTLPNL` is used in `wceshell.bib` to include the components. It is used in `wceshell.reg` to set up registry entries that describe the control panel color scheme and specify default settings for the various control panel applets.
`IMGNODEBUGGER`	Builds an image with a kernel debugger. Kernel debugging enables the kernel on the target to communicate with the kernel-debugging tool in Platform Builder. Extensive traces are also displayed by the kernel debugger.	`IMGNODEBUGGER` is used in `common.bib` to include either the kernel with debugging (`nk.exe`) or the kernel without debugging (`nknodbg.exe`) in the image.
`IMGNODRIVERS`	Excludes certain common drivers from the image.	`IMGNODRIVERS` is used in `common.bib` to include the parallel port and printer drivers (`prnport.dll`, `prnerr.dll`, `pcl.dll`), PC card ATA and IDE driver (`atadisk.dll`), PC card static RAM

Table 3.5 *Continued.*

Environment Variable	Usage	Description
		(SRAM) driver (`sramdisk.dll`), PC card linear flash driver (`trueffs.dll`), dual serial driver (`dualio.dll`) and waveform audio driver (`waveapi.dll`).
`IMGNOETHER`	Includes Ethernet support in the image.	`IMGNOETHER` is used in `common.bib` to include Address Resolution Protocol driver (`arp.dll`), NDIS driver (`ndis.dll`), NE2000-compatible card driver (`ne2000.dll`), Proxim RangeLAN PC card driver (`proxim.dll`), Xircom PC card driver (`xircce2.dll`), and Dynamic Host Configuration Protocol driver (`dhcp.dll`).
`IMGNOFILES`	Excludes certain files depending on context.	`IMGNOFILES` is used (1) in `wceshell.bib` to include desktop shortcuts (LNK files) and help files for shell components that have been included in the image, (2) in `project.bib` for the project `MAXALL` to include WAV files for system sounds, and (3) in `wceapps.bib` to include the shortcuts and help files for applications that have been included in the image.
`IMGNOFLTDDK`	See `IMGNOCEDDK`.	
`IMGNOHELP`	Excludes Windows CE Help in the operating system.	`IMGNOHELP` is used (1) in `wceshell.bib` to include help components (part of the shell), (2) in `wceshell.reg` to include registry entries that set up help file associations with the appropriate modules, (3) in `wceapps.bib` to include help for components included in the image, and (4) in `wceapps.reg` to include registry entries in a manner similar to `wceshell.reg`.
`IMGNOIDE`	Not used.	
`IMGNOIE`	Excludes Pocket Internet Explorer and support components.	`IMGNOIE` is used in `ie.bib` to exclude localization support for Internet Explorer (`mlang.dll`), `ieceext.dll`, `shlwapi.dll`, WinInet API (`wininet.dll`), URL and Moniker support (`urlmon.dll`), HTML support (`mshtml.dll`), HTML frames support (`shdocvw.dll`), limited XML support (`msxml.dll`), and `mmefx.dll`.

(continued)

Table 3.5 *Continued.*

Environment Variable	Usage	Description
IMGNOJAVA	Excludes Java support from the operating system.	IMGNOJAVA is used in common.bib to exclude the Java modules cejvm.dll, jview.dll, ce_awt.dll, ce_local.dll, ce_math.dll, ce_irda.dll, ce_zip.dll, ce_net.dll, jcls.dll, and verifier.dll.
IMGNOJSCRIPT	Excludes JavaScript support from the operating system.	IMGNOJSCRIPT is used (1) in common.bib to exclude jscript.dll, and (2) in common.reg to set up OLE IDs for the JScript component in the system registry.
IMGNOLOC	Not used. Localization support is not optional.	
IMGNOMAIL	Excludes Pocket Mail, Internet Message Access Protocol (IMAP), and Simple Mail Transport Protocol (SMTP) support.	IMGNOMAIL is used (1) in wceapps.bib to exclude pmail.exe, imap4.dll, smtp.dll, msgstore.dll, tnefutil.dll, mailutil.dll, labledit.dll, uicom.dll, and pimprint.dll; and (2) in wceapps.reg to set up registry entries for Pocket Mail.
IMGNOMLANG	Excludes localization support from Pocket Internet Explorer.	IMGNOMLANG is used in ie.bib to exclude mlang.dll.
IMGNOMSHTML	Excludes HTML support from the operating system.	IMGNOMSHTML is used in ie.bib to exclude mshtml.dll.
IMGNONETUI	Excludes the network user interface from the operating system. The network user interface allows manipulation of the configuration properties of the network via the Communication control panel applet.	IMGNONETUI is used in common.bib to exclude netui.dll.
IMGNOOLE32	Disables OLE support in the operating system.	IMGNOOLE32 is used in common.bib to exclude the OLE support components ole32.dll and oleaut32.dll.
IMGNOPCMCIA	Disables PC card support in the operating system. This entry doesn't exclude PC card support in the operating system. It includes it but disables it at runtime. To exclude PC card support, you must exclude the file pcmcia.dll from the image (see ODO_NOPCMCIA).	IMGNOPCMCIA is used in common.reg to exclude registry entries for all supported PC cards.

Table 3.5 *Continued.*

Environment Variable	Usage	Description
IMGNOPWORD	Excludes Pocket Word from the operating system.	IMGNOPWORD is used (1) in wceapps.bib to exclude office.dll, pwd_res.dll, pwwiff.dll, and pword.exe; and (2) in wceapps.reg to set up registry entries that define OLE IDs and file associations for Pocket Word.
IMGNOREDIR	Excludes network redirector support from the operating system.	IMGNOREDIR is used in common.bib to exclude the redirector components redir.dll and netbios.dll.
IMGNOSECURITY	Excludes security components from the operating system.	IMGNOSECURITY is used in common.bib to exclude the digital signature and data certificates (rsabase.dll) and the corresponding 128-bit version (rsaenh.dll).
IMGNOSERVERS	Excludes all servers from the operating system. If you plan to ship two different versions of your OS build (à la NT workstation and server), you can use this variable to switch between the versions.	IMGNOSERVERS is used in msmq.bib to exclude Microsoft Message Queue components msmqd.dll, netregd.dll, mqoa.dll, msmqadm.exe, and msmqrt.dll.
IMGNOSHDOCVW	Excludes HTML frames support from Pocket Internet Explorer.	IMGNOSHDOCVW is used in ie.bib to exclude shdocvw.dll.
IMGNOSHELL	Excludes the Windows CE shell and related components, shortcuts, and help files from the operating system. Excluding shell components automatically excludes the browser and CE Help.	IMGNOSHELL is used (1) in wceshell.bib to exclude the task manager (taskman.exe), asform.dll, the CE shell support component (ceshell.dll), and the explorer shell (explorer.exe); (2) in wceshell.reg to create registry entries that automatically launch the task manager and explorer shell on startup; (3) in wceapps.bib to exclude browser and help files if set; and (4) in wceapps.reg in the same way as in wceshell.reg.
IMGNOTXTSHELL	Excludes the CE shell from the operating system. The CE shell communicates with a designated desktop for debugging and synchronization services.	IMGNOTXTSHELL is used in common.bib to exclude the CE shell components cesh.dll and toolhelp.dll.
IMGNOURLMON	Excludes URL and Moniker support from the browser.	IMGNOURLMON is used in ie.bib to exclude urlmon.dll.
IMGNOWININET	Excludes WinInet API support from the operating system.	IMGNOWININET is used in ie.bib to exclude wininet.dll.

(continued)

Table 3.5 *Continued.*

Environment Variable	Usage	Description
IMGNSCFIR	Includes the National Security Council Fast Infrared driver.	IMGNSCFIR is used in platform.bib for the CEPC platform to include nscirda.dll (if not set, the regular IrDA driver irsir.dll is included), and (2) in platform.reg to set up registry entries for the appropriate IrDA driver included in the operating system.
IMGPROFILER	Builds a profile-enabled kernel in the operating system.	IMGPROFILER is used (1) in common.bib to include the profile-enabled kernel nkprof.exe in the image, and (2) in config.bib for the CEPC and Odo platform to instruct the OS build tool that profiling has been enabled in the kernel (PROFILE=ON).
IMGSTRICTLOC	Not used.	
IMGTINY	Builds a special bare-bones version of the operating system.	IMGTINY is used in common.bib, wceshell.bib, wceapps.bib, wceapps.bib, and the platform.bib files for the CEPC and Odo platforms.
IMGTINYFSRAM	Uses a (relatively) tiny percent of RAM for the file system. By default, CE uses the RAM for file system storage.	IMGTINYFSRAM is used in config.bib to set another variable, FSRAMPERCENT, to the hex value of 80. This number instructs CE to use only 50 percent of the *first* 1MB of RAM for the file system.
IMGUSB	Adds USB support to the operating system.	IMGUSB is used in platform.bib for the CEPC platform to include the USB support components Open Host Controller Interface driver (ohci.dll), USB driver (usbd.dll), and USB mouse driver (usbmouse.dll).
IMGUSEPROXY	Enables the use of a proxy server for HTTP.	IMGUSEPROXY is used in wceshell.reg to add lines to the system registry that instruct the browser to use a proxy server called itg-proxy for HTTP access on port 80. Edit these settings to configure your own proxy server by name if you turn on this variable.
INITNOCOMM	Disables the NDIS and auxiliary function driver (AFD) protocol manager at runtime.	INITNOCOMM is used in common.reg to disable the registry settings for the NDIS and AFD components. Note that this setting does not remove ndis.dll and afd.dll from the image. It allows the components to be part of the image but simply disables them at runtime.

Table 3.5 *Continued.*

Environment Variable	Usage	Description
SCHEDLOG	Includes the scheduler log functions in the operating system.	Scheduler log functions are implemented in `schedlog.dll` and are used as helper functions when thread and process logging are being implemented in the kernel. `SCHEDLOG` is used in the `Sources` file for the kernel modules in CEPC and Odo platforms to link the kernel with `schedlog.lib`. In the `Sources` file for the Hardware Abstraction Layer (HAL), this variable is used to pass `-D SCHEDLOG` to the compiler. This flag is used to conditionally add a HAL IOCTL (I/O control) code that enables scheduler logging.
TESTSIP	Includes the Software Input Panel (SIP) control panel applet in the operating system. The applet is used to configure the SIP, a keyboard implemented in software for devices that do not have a keyboard (e.g., the Palm-size PC).	`TESTSIP` is used (1) in `common.bib` to include the SIP control panel applet (`msim.dll`), and (2) in `common.reg` to specify default values for SIP configuration (manipulated by `msim.dll`).
WINCEPROFILE	Builds a version of the kernel that supports profiling.	`WINCEPROFILE` is used in the HAL `Sources` file for the CEPC platform to pass the `-D PROFILE` flag to the compiler. This flag is not really used by the HAL, since a kernel with profile information is always built (`nkprof.exe`). However, you can use conditional compiling around the constant `PROFILE` to add any profile-specific code to the HAL.

Environment variables also find use in customization of the base platform chosen to build a new platform. For example, we chose `MAXALL` to build Adam. Now we can modify the `MAXALL` configuration for use in Adam by changing the value of these environment variables appropriately.

These variables can also be put to good use during development and debugging. It may not be necessary for every developer on the project to include all components of the operating system. A developer working on writing a PC card driver for a bar code scanner may not need to include any of the Pocket applications like Explorer and Mail, the serial port driver, or communication components. Such choices can help reduce the size of the operating system image being built, which is instrumental in shorter download times to the hardware platform, resulting in a more rapid development cycle.

Of course, when the driver is finally ready, the developer must test it with the full build of the operating system that is expected to run on the hardware platform.

Extending the Platform Builder Catalog

The Platform Builder catalog isn't for components that ship with Windows CE. Components can be added to the catalog so that they become available as standard components for a given platform. When you're adding a component, the principal pieces of information you must supply are its name, a method to build the component, the group to which this component belongs (this could be a new group or an existing one in the catalog), and a unique ID. A multitude of other information, which we will discuss shortly, must also be supplied.

Let's start by introducing the unique ID. This ID must be unique across any component ever created for Windows CE. A globally unique identifier (GUID), also referred to as a universally unique identifier (UUID), is a 128-bit value that uniquely identifies a component. You can generate a unique number on demand by using the Microsoft utility `guidgen.exe`. You can then use this number to identify a component that must be added to the catalog. If a GUID is not supplied, Platform Builder generates one for the component when adding it to the catalog. However, providing a GUID for each component is recommended because it must be supplied when modifications are made to the component in the catalog.

Guidgen will generate a new GUID and allow you to copy it to Clipboard so that you can paste it into any other application. One of four formats can be selected. For a format suitable for our purposes, select **Registry Format**. Then click on the **Copy** button. Now the GUID can be pasted in via the editor being used to create the component that will be added to the catalog.

Components to be imported into a catalog must be specified by a special syntax and placed in a file with a *.cec* extension. Files with the *.cec* extension are called **component files**.

Component Files in Depth

Component files have a format for laying out information about a component. The best way to start is by example. Recall that we built an application called `welcome.exe` for our project Adam. This simple welcome application can be included as a standard component in the catalog. A CEC file must be used to describe this component before it can be imported into the catalog.

CECInfo Block

Every component file starts with the `CECInfo` block (Listing 3.6), a structure that contains information about the component file itself. Note that any text following the characters "//" up to the end of the line is considered to be a comment.

Listing 3.6 Sample component file header

```
CECInfo (
   Name(New.cec)
   CECVersion (3.00)
   // GUID() – left blank
   Vendor ("Windows CE Unlimited")
   Description ("A sample Cec file")
)
```

The fields of the CECInfo block are specified as follows:

- Name is an optional field that identifies the name of the component file.
- CECVersion is a mandatory field that can have a value of either 2.12 or 3.00. It identifies the version of Windows CE for which the component file was written.
- GUID is an optional field containing a number that uniquely identifies the component file. If it is left blank, a GUID will automatically be generated for the component file when it is used to import the component.
- Vendor is an optional field identifying the vendor that is distributing this component as part of the catalog.
- Description is an optional field that describes the component file.

ComponentType Block

The component itself is described by a ComponentType block. Consider the sample shown in Listing 3.7, which describes welcome.exe as a component.

Listing 3.7 Sample component file

```
ComponentType (
  Name( Welcome )
  GUID( {232FBCF4-72DD-4208-A40F-686A42FFE8B3} )
  Description( "Welcome application" )
  Group( "\Standard Applications" )
  Implementations(
    Implementation(
       Name(Welcome)
       GUID( {2A8D35B5-F6BC-485c-867B-8352826D27CF} )
       Description( "Welcome application" )
       Vendor("Windows CE Book")
       Date(05/05/2000)
       BuildMethods(
            BuildMethod(
                 Step( buildrel )
                 GUID( {A3CED1C5-617E-4065-A784-8551FA00A249} )
                 CPU( x86 )
                 InputFiles(  )
                 OutputFiles(  )
                 Action( "#COPY( "$(_PROJECTROOT\Welcome\Obj\Welcome.exe",
$(_FLATRELEASEDIR)" )))
                 Setting( '#CHM( "Welcome.chm" )' )
                 Setting( '#CHM( "Welcome.chi" )' )
                 Setting('#INPUT("Include Welcome", INCLUDE_WELCOME, 1, 0,
"")')
            )
       )
    )
  )
)
```

The fields of the `ComponentType` block are specified as follows:

- `Name` specifies the name of the component. The name of the component in Listing 3.7 is Welcome.
- `GUID` is an optional field containing a number that uniquely identifies the component. In this case we have assigned a GUID to the component so that we can specify it later to modify this component.
- `Description` is an optional field that describes this component. It is displayed when the properties for a component are viewed in Platform Builder.
- `Group` is an optional field that refers to the organization hierarchy displayed in the catalog. If you specify *\Standard Applications* in the sample, this component will be added to a new group in the catalog at the root level called Standard Applications. Welcome will be added as a component to this group. You can add a component to an existing component simply by specifying its name. Group names in a group hierarchy can be separated by a backslash (\). If no other option is specified, the component is added directly to the root of the catalog.
- `Vendor` identifies the vendor of the component. The value of this field is displayed when the properties for the component are viewed (right-click on the component in the catalog and select **Properties** from the pop-up menu).

Implementation Block

Each `ComponentType` block must have an embedded `Implementations` block (see Listing 3.7). The `Implementations` block can consist of one or more `Implementation` blocks that describe how the component has been implemented.

The fields of the `Implementation` block are specified as follows:

- `Name` is a mandatory field that identifies the `Implementation` block in other `Implementation` blocks.
- `GUID` contains the unique identifier for the `Implementation` block. This field is optional, but if specified, it must be unique for each block. An implementation may be referred to by name or by GUID. If this field is left blank, Platform Builder automatically generates and applies a GUID to this block.
- `Description` is an optional field that describes the implementation.
- `Vendor` is an optional field that identifies the vendor of the implementation and is optional.
- `Children` is an optional field that lists any children of an implementation. This field can be used to specify any dependent implementations. A child implementation must be described earlier in the component file. Implementations may be identified by name or GUID in this field.
- `Date` is an optional field that specifies the date of implementation in MM/DD/YY format. This date can be set to the date the component was built or to the date it was included in the catalog.

BuildMethods Block

The `BuildMethods` block, along with the `Name` field, is required in the `Implementation` block. The `BuildMethods` block is followed by one or more `BuildMethod` blocks, each of which specifies a method for building the component (see Listing 3.7).

The fields of the `BuildMethod` block are specified as follows:

- `Step` and `Action` are mandatory fields that form the heart of the `BuildMethod` block. Together these fields specify how this particular implementation of the component will be built. The following list gives the different keywords that can be specified in these fields. For a more thorough treatment of how each of these keywords operates, refer to Chapter 10. The command specified in the `Action` field depends on the keyword specified in the `Step` field (see Table 3.6).

  The `Action` field can contain the commands specified in the list that follows. In each case the entire command must be enclosed in quotation marks for it to be executed—that is, `Action ("<command>")`.

  - `#COPY("SrcPath", "TargetDir")` copies a file with a fully qualified path name into the target directory. Use this command to copy files during the system generation or build-release phases. For example, this command can be used to copy a component to the final target directory from which the operating system image is constructed.
  - `#ENV("Variable", "Value")` sets an environment variable to a specific value. For a more detailed explanation of how environment variables can affect a build, refer to the section titled Customizing the Build Using Environment Variables earlier in this chapter and to Chapter 10.

Table 3.6 Build Actions

`Step` **Keyword**	**Phase of Build**	`Action` **Command**
CESYSGEN	System generation phase. The Microsoft modules and third-party vendor components that make up the Windows CE system are combined to create a core operating system. Components supplied by the system integrator are added to this build to create the final image.	#COPY
BSP	Core build phase. During this phase of the build process, each component that is part of the Windows CE image is built individually.	#ENV
		#BUILD
		#CUSTOM
BUILDREL	Build-release phase. In this phase, all the output files are collected in a predetermined location in a mass copy operation.	#COPY
		#CUSTOM
MAKEIMG	Make-image phase. The final operating system image is built from the collected files.	#ENV

- `#BUILD(Dirs | Sources, "Directory")` tells the build process to build either a `Dirs` file or a `Sources` file to be found in the directory specified in `Directory`. `Dirs` and `Sources` files specify commands for building one or more components. A more thorough treatment of these files can be found in Chapter 11.
- `#BUILD(MAK, "Directory", "Makefile")` is an alternative flavor of the `BUILD` command that can be used to build a component with a custom makefile. A custom makefile would be used in lieu of a sources file for better control of the build process. This command can also be helpful in porting components to Windows CE, where you can use a fully tested makefile instead of converting it into a sources file. For a quick primer on how makefiles work, refer to Appendix B.
- `#CUSTOM("WorkingDirectory", "CustomCommand")` can be used to execute a command specified by `CustomCommand`. This command is executed from the directory specified by `WorkingDirectory`. It can be used to execute scripts like batch files that perform tasks that either cannot be performed by a makefile or would be extremely tedious to port to a makefile. Again, legacy components that are built by scripts can be accommodated by this command.

• `GUID` contains the unique identifier for the `Implementation` block. This field is optional, but if specified, it must be unique for each block. An implementation may be referred to by name or by GUID. If the field is left blank or not specified, Platform Builder automatically generates and applies a GUID to this block.

• `CPU` is a mandatory field that indicates if the implementation is CPU specific. Current CPU values that can be specified in this field are SH3, SH4, SA1100, ARM720, ARM720T, R3912, R4102, R4111.16, R4111.32, R4300, PPC403, PPC821, and x86 for Windows CE 3.0. This list, supported by Microsoft, may grow in the future as more processors are supported by Windows CE. Processors may also be dropped from this list. The value of the CPU field must be enclosed in quotation marks. The value `default` indicates that the implementation is for the default list of processors for the operating system. The default list is the list supported by Microsoft.

• `Setting` is an optional field that supports three different operations:

1. `#INPUT( "Sysgen setting", EnvironmentVariable, 1 | 0, InitialValue, BspValue)`. Each `BuildMethod` block is allowed to specify a setting during the system generation phase, referred to as `Cesysgen` or `Sysgen`. In the Platform Builder IDE, you can select or deselect `Sysgen` by selecting **Build** and then **Settings**, and finally clicking on the **Sysgen** tab in the **Platform Settings** dialog box.

   Each setting sets an environment variable. The `#INPUT` operation allows such a setting to be made visible in the **Sysgen** tab of the **Platform Settings** dialog box. The string *Sysgen setting* is displayed in the tab. The environment variable specified by `EnvironmentVariable` is either set or unset depending on its value: either 1 or 0. If the value is 1, then the environment variable will be set to `TRUE` when the setting is selected. A value of 0 specifies that the variable be set to `FALSE` when the setting is selected. Finally, `InitialValue` specifies the initial value of the environment variable and hence the default selection of the setting in the tab. `BspValue` is a string that is set to the name or GUID of the board support package that allows this setting.

2. `#OUTPUT(Output)` allows the selection of a particular module in the image. `Output` is usually an environment variable that is read by the `cesysgen.bat` file during the system generation phase (we'll give more details in Chapter 10).
3. `#CHM("HelpOrHelpIndexFile")` associates an HTML help file (.chm extension) or a help index file (.chi extension) with the component. When an SDK is exported, the component's help files specified by this operation are automatically included in the SDK by Platform Builder.

- `InputFiles` is an optional field that is used to specify a list of files, separated by spaces, required to perform the build for this component.
- `OutputFiles` is an optional field that is used to specify a list of files, separated by spaces, output by the build.

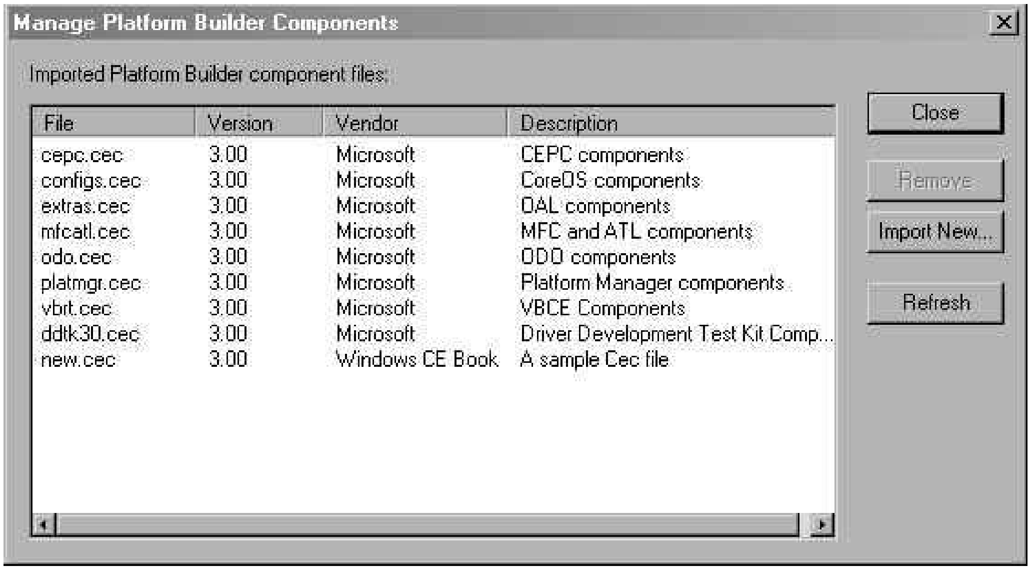
Adding a Component to the Catalog

One global catalog is used by Platform Builder to store components and can be reused across projects. To add a component to the catalog, you must create the component. Component files have a *.cec* extension. To import the component, select **Manage Platform Builder Components...** in the **File** menu. The resulting dialog box lists all the components that have already been imported (Figure 3.15). Click on **Import New...** and browse for the component file that has been created for the new component.

As an example, save Listing 3.6 to a file called `new.cec` and import it into the catalog. Welcome will show up as a member of the catalog under the folder `Standard Applications`. After Welcome has been added to the catalog (Figure 3.16), it is available for inclusion in all new platforms.

The dialog box to manage platform builder components is a front end to the `pbcec.exe` utility that comes with Platform Builder. Pbcec imports components into the catalog. To import a component, we call Pbcec with the component file name as its argument. For example, to import the Welcome component into the CEPC catalog via the command line, we would have invoked Pbcec in the following way:

```
Pbcec New.cec
```



Manage Platform Builder Components

Imported Platform Builder component files:

File	Version	Vendor	Description
cepc.cec	3.00	Microsoft	CEPC components
configs.cec	3.00	Microsoft	CoreOS components
extras.cec	3.00	Microsoft	OAL components
mfcatl.cec	3.00	Microsoft	MFC and ATL components
odo.cec	3.00	Microsoft	ODO components
platmgr.cec	3.00	Microsoft	Platform Manager components
vbrt.cec	3.00	Microsoft	VBCE Components
ddtk30.cec	3.00	Microsoft	Driver Development Test Kit Comp...
new.cec	3.00	Windows CE Book	A sample Cec file

Figure 3.15 Adding Welcome to the catalog

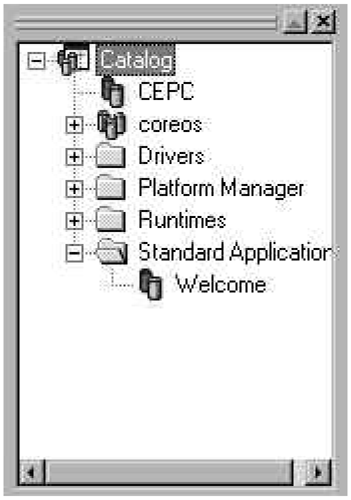


Figure 3.16 The catalog after Welcome has been added

Calling Pbcec with the `/list` argument lists the components in the catalog. A sample run yielded the following output:

```
Microsoft (R) Platform Builder 3.00 Catalog Utility
Copyright (C) Microsoft Corp 2000.  All rights reserved.
CEC File               Description
========               ===========
cepc.cec               CEPC components
```

```
configs.cec           CoreOS components
extras.cec            OAL components
mfcatl.cec            MFC and ATL components
odo.cec               Odo components
platmgr.cec           Platform Manager components
vbrt.cec              VBCE Components
ddtk30.cec            Driver Development Test Kit Components
new.cec               A sample Cec file
There are currently 9 cec files in the catalog.
```

When called with the `/r` option, Pbcec removes from the catalog the component that is specified in the file passed in as the Pbcec argument. The `/clean` option does exactly what it says: It clears out all components from the catalog.

Creating a New Board Support Package

Extending the catalog allows components to be used across platforms that use the same board support package. System integrators may want to create a new BSP for a new processor or for a significantly new platform for an existing processor. The key to creating a new BSP is to create a new BSP file. The BSP file contains a list of instructions for how to build the platform and which OEM components are required for the platform. Users of the BSP can then build the platform in the manner that is prescribed in the BSP. In other words, a BSP file contains directions for the build process, whereas component files define the components to be built.

Before a BSP can be created, all of its components must be imported into the catalog. Let's say we are creating a BSP called Appliances. Appliances will support the x86 CPU and will provide components typically used by kitchen appliances, such as coffee makers, refrigerators, and so on. We'll keep the Appliances BSP simple for illustration purposes.

The first step is to import all of the Appliances components. We take the CEPC component file and modify it for Appliances. Then we import it into the catalog. Before we do that, we adjust Appliances so that it has no display, no keyboard support, and no USB or infrared support because these are not needed for the type of BSP we are creating. All GUIDs in the file being copied must be created again via `guidgen.exe` because Platform Builder expects all of these new components to be exported into the catalog. Listing 3.8 shows the component file for Appliances after `cepc.cec` was copied and modified.

Listing 3.8 Component file for Appliances

```
//  appliances.cec - Appliances components

CECInfo (
    Name(Appliances)
    GUID({D1E60FE9-4370-4deb-B111-781D7CBAEA73})
    CECVersion(3.00)
    Vendor("Microsoft")
    Description("Appliances components")
)
```

```
//  type "OAL" and 2 implementations
//  Appliances
ComponentType (
    Name( OAL )
    GUID( {B3509B99-F1E4-11d2-85F6-004005365450} )
    Description( "OEM Adaptation Layer" )
    Implementations(

        Implementation(
            Name( Appliances )
            GUID( {E0A5CD0C-9D7D-4c4c-B4E7-17A6CACB3E3B} )
            Description( "Appliances OAL" )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {188258EE-AE2C-4013-8514-1D6EA325027C} )
                    CPU( "x86" )
                    Action( '#BUILD(DIRS, "$(_WINCEROOT)\platform\cepc\kernel")' )
                    Action( '#BUILD(DIRS, "$(_WINCEROOT)\platform\cepc\gwe")' )
                )
            )
        )
    )
)

//  child type "ddk_bus" and implementation
ComponentType (
    Name( ddk_bus )
    GUID( {4BB97298-47AC-43ef-BD2D-9E5B9FC3D1CA} )
    Description( "ddk_bus" )
    Implementations(
        Implementation(
            Name( ddk_bus )
            Description( "ddk_bus" )
            GUID( {C87FA2D3-8D13-49e4-91BD-A94C74DA6EE6} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {84A08ED9-87A1-466a-A268-F9DEA522D2C1} )
                    CPU( "x86" )
                    OutputFiles( ddk_bus.LIB )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\CEDDK\DDK_BUS")' )
                )
            )
        )
    )
)
```

```
//  child type "ddk_map" and implementation
ComponentType (
    Name( ddk_map )
    GUID( {8886E23C-797C-4b19-A871-843AA949B866} )
    Description( "ddk_map" )
    Implementations(
        Implementation(
            Name( ddk_map )
            Description( "ddk_map" )
            GUID( {02A46E4A-1CDB-4504-B99C-470C27157015} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {F07C875B-1D69-4cf0-A54B-D7486886627B} )
                    CPU( "x86" )
                    OutputFiles( ddk_map.LIB )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\CEDDK\DDK_MAP")' )
                )
            )
        )
    )
)


//  type "ceddk" and implementation
ComponentType (
    Name( ceddk )
    GUID( {8630294D-6B62-4422-9B72-2B2D29629AE5} )
    Description( "ceddk" )
    Group( "\Drivers\Appliances" )
    Implementations(
        Implementation(
            Name( ceddk )
            Description( "ceddk" )
            GUID( {BBD69EB7-AA85-440b-ADCC-8C8D1D8C34C0} )
            Children( ddk_bus ddk_map )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {D6907409-E7DB-4d98-8841-58AD8A25B75F} )
                    CPU( "x86" )
                    OutputFiles( pc_ddk.DLL )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\CEDDK\DLL")' )
                )

                BuildMethod(
                    Step( makeimg )
                    GUID( {C792A2F8-0419-49aa-8B1E-CEB62BA274F8} )
                    CPU( "x86" )
                    Action( '#ENV(IMGNOCEDDK, "")' )
                )
            )
        )
    )
)


//  type "serial" and 2 implementations
//  "serial", and "NewSerialMDD"
ComponentType (
    Name( serial )
    GUID ( {6401DC3D-E93A-4bfb-B58F-6818A0500E64} )
    Description( "serial" )
    Group( "\Drivers\Appliances" )
    Implementations(
        Implementation(
            Name( serial )
            Description( "serial" )
            GUID ( {81AA1070-3B88-4680-AD9D-E132F6773584} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {4DAB8E7F-365B-4f51-B905-EE82FA242D14} )
                    CPU( "x86" )
                    OutputFiles( serial.DLL )
                    Action( '#ENV(ODO_NOSERIAL, "")' )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\SERIAL.PDD")' )
                )

                BuildMethod(
                    Step( makeimg )
                    GUID( {8D6179A9-0BD4-4387-A664-5A8C5D946714} )
                    CPU( "x86" )
                    Action( '#ENV(ODO_NOSERIAL, "")' )
                    Action( '#ENV(NEW_SERIAL_MDD, "") ')
                )
            )
        )

        Implementation(
            Name( NewSerialMDD )
            Description( "New Serial MDD" )
            GUID ( {52B0F538-8E24-4a07-BDE5-76A7259608BF} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {2F65E925-A228-4875-841A-A1B456E2B23A} )
                    CPU( "x86" )
                    OutputFiles( com_card.dll com16550.dll )
                                   Action( '#ENV(ODO_NOSERIAL, "")' )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\COM_CARD")' )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\COM16550")' )
                )

                BuildMethod(
                    Step( makeimg )
                    GUID( {6CA564F3-178D-47ff-9D6B-D81378F1FC40} )
                    CPU( "x86" )
                    Action( '#ENV(ODO_NOSERIAL, "")' )
                    Action( '#ENV(NEW_SERIAL_MDD, 1)' )
                )
            )
        )
    )
)


//  type "wavedev" and implementation
ComponentType (
    Name( wavedev )
    GUID ( {E0B38875-FF8F-4685-99D4-9A591368609D} )
    Description( "wavedev" )
    Group( "\Drivers\Appliances" )
    Implementations(
        Implementation(
            Name( wavedev )
            Description( "wavedev" )
            GUID ( {0F132F1D-4AB8-4c57-853D-68935FDED1DF} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {A0E49B67-614C-47bf-80D0-48C7E25C664B} )
                    CPU( "x86" )
                    OutputFiles( wavedev.DLL )
                    Action( '#ENV(ODO_NOAUDIO, "")' )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\Drivers\WAVEDEV")' )
                )

                BuildMethod(
                    Step( makeimg )
                    GUID( {6DE00180-A0BD-4e74-87E7-4BD50AF5E158} )
                    CPU( "x86" )
                    Action( '#ENV(ODO_NOAUDIO, "")' )
                )
            )
        )
    )
)
```

```
//  type "EBOOT" and implementation
ComponentType (
    Name( EBOOT )
    GUID( {F02E3B9F-CD10-44a5-BED5-1EAE187A2AE6} )
    Description( "Appliances Eboot.bin" )
    Group( "\Drivers\Appliances" )
    Implementations(
        Implementation(
            Name( Eboot )
            Description( "Appliances Eboot.bin" )
            GUID( {086BA8F9-0788-4900-A48C-2D27F8AA8397} )

            BuildMethods(
                BuildMethod(
                    Step( BSP )
                    GUID( {C1450E38-C813-426f-BB98-6FDF40EC2216} )
                    CPU( "x86" )
                    Action( '#BUILD(SOURCES,
"$(_WINCEROOT)\platform\Appliances\EBOOT")' )
                )
            )
        )
    )
)
```

When the component file shown in Listing 3.8 is imported into the catalog, the catalog shows the addition of an OAL component, Appliances, and the drivers that are part of the Appliances BSP (Figure 3.17).

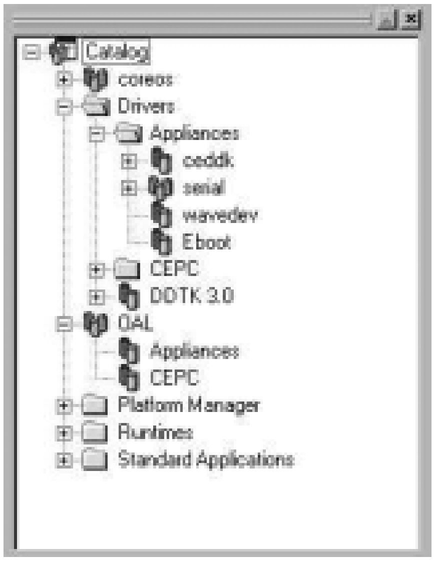


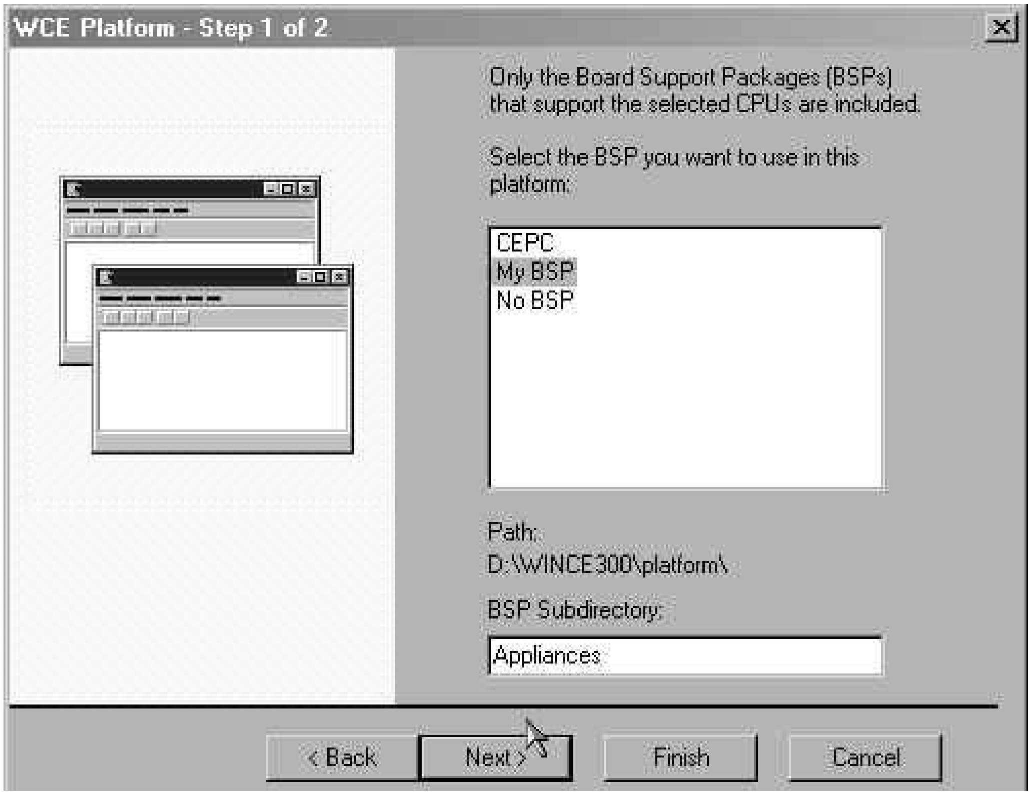
Figure 3.17 The catalog with imported Appliances components

Once the components for the BSP have been imported into the catalog, we need to create a new BSP. Platform Builder allows you to create a project with a placeholder for a BSP. To finish creating the BSP, you must expand the placeholder—a painstaking process that must be carried out by hand.

To create a placeholder for a BSP, click on the **File | New...** menu option in Platform Builder. Choose to create a WCE platform from the **Platforms** tab and type in the name of a sample platform when the WCE Platform Wizard launches its first dialog box. For example, to create a BSP for a category called Appliances, type in "Brewster." Brewster is a sample coffee maker that uses Windows CE. We will take a more intimate tour of Brewster later on in this book. For now, it makes a guest appearance for illustration purposes.

In the second dialog box (step 2) of the wizard, select **My BSP**. This option indicates to Platform Builder that you are trying to create a new board support package. Type in the name of the BSP subdirectory. In our example we chose the name *Appliances* (Figure 3.18).

You *must* create this directory before you can execute step 2. Platform Builder will not create it for you. Accepting the defaults in the ensuing wizard dialogs will lead to the creation of Brewster. Figure 3.19 shows the **Project** window after Brewster is created and loaded in Platform Builder.


WCE Platform - Step 1 of 2

Only the Board Support Packages (BSPs) that support the selected CPUs are included.

Select the BSP you want to use in this platform:

CEPC
My BSP
No BSP

Path:
D:\WINCE300\platform\

BSP Subdirectory:
Appliances

Figure 3.18 Creating a new BSP

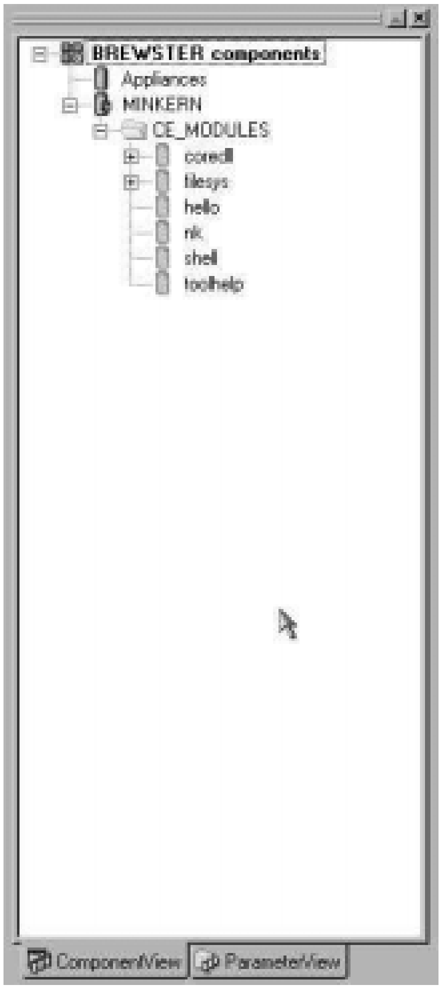


Figure 3.19 The Brewster platform used to create a new BSP

Note that Appliances shows up as an implementation in the project window. Right-clicking on it and selecting **Properties** will reveal that Platform Builder recognizes it as a BSP. However, there is nothing in the BSP yet. The directory created earlier is empty. The only trace that you have a new BSP is a file called `appliances.bsp` created in the Platform Builder IDE directory. For Windows CE 3.0, this directory is `\WINCE300\WINDOWS CE PLATFORM BUILDER\3.00\CEPB\BIN\IDE`. This file represents an empty BSP, and you must modify it by hand to complete the board support package. The `appliances.bsp` file is presented in Listing 3.9.

Listing 3.9 `Appliances.bsp`

```
//  *** Appliances ***

//  NOTE:  You will need to create an appliances.cec file
//  and import it into the catalog.  When you have done that,
//  you can delete the two #ADD_USER_OAL lines below.

//  *** Global components (for all the configs) ***

#ADD_USER_OAL_BUILD_METHOD ('#BUILD(dirs, "$(_WINCEROOT)\platform\
Appliances")')

#ADD_USER_OAL_COMPONENT ("{2367C526-2821-4CEF-94A2-7286D5152E6F}",
"Appliances")

//  *** CoreOS-specific components ***
#IF ("COREOS","MINKERN")
#ENDIF
#IF ("COREOS","MAXALL")
#ENDIF
#IF ("COREOS","MINCOMM")
#ENDIF
#IF ("COREOS","MINGDI")
#ENDIF
#IF ("COREOS","MININPUT")
#ENDIF
#IF ("COREOS","MINSHELL")
#ENDIF
#IF ('COREOS","MINWMGR")
#ENDIF
#IF ("COREOS","IESAMPLE")
#ENDIF
```

The BSP file contains placeholders for adding CoreOS-specific components. *CoreOS* refers to a configuration that represents a specific type of build of the operating system. Familiar configurations like `MINKERN`, `MAXALL`, and `MINCOMM` have already been added to this file. The directives required to build your custom platform should be inserted in this file as the final step to creating a board support package.

BSP File Directives

Before we add directives to the file, let's look at the types of directives that can be added to a BSP file. BSP file directives all start with the pound sign (#). Text that follows "//" is considered to be comment to the end of the line.

- `#ADD_COMPONENT_BY_GUID_TO_ROOT ("guid")`. This directive is used to add a component implementation to the BSP. The component is identified by its GUID. Recall that right-clicking on a component and selecting **Properties** from the pop-up menu will reveal its GUID. To add the serial port driver to the BSP, you would add the following line within a CoreOS conditional:

```
#ADD_COMPONENT_BY_GUID_TO_ROOT (“{35E70441-EEA9-11D2-A092-
0060085C1833}”)
```

- `#ADD_FOLDER_TO_ROOT ("Folder")`. You can organize the components by creating folders under the root. This directive is used to create a folder.
- `#ADD_COMPONENT_BY_GUID_TO_FOLDER ("guid", "Folder")`. You can add a component to a specific folder using this directive.
- `#ADD_FOLDER_TO_FOLDER ("SubFolder", "Folder")`. You can create deeper hierarchies by creating subfolders with this directive.
- `#ADD_TYPE_BY_GUID_TO_ROOT ("guid")`. You can add a component type to the root via this directive.
- `#ADD_TYPE_BY_GUID_TO_FOLDER ("guid","Folder")`. Using this directive, you can also add a component type under a folder, just as you would a component.
- `#ADD_ENV_VAR ("Variable","Value")`. You can define an environment variable and set it to a specific value using this directive. Environment variables are like settings for the platform. They are used to communicate values between different utilities that build the operating system (Chapter 10 shows exactly how this works).

Note the two directives in `appliances.bsp` (see Listing 3.9). These lines are added by Platform Builder to an empty BSP file to support the creation of a new BSP. After you have populated the BSP file with components using the directives discussed, you must delete these lines. The lines provide a placeholder for the empty BSP and specify how to build the BSP.

Notice also from the same listing that the BSP files support conditional directives. Conditionals are used in the sample listing to specify which CoreOS is being built.

Finishing the New BSP: Appliances

After the required directives have been added to the BSP file (Listing 3.10), all of the physical implementation of all the components that are part of this new BSP must be copied under the newly created BSP directory. This means copying source code, libraries, and so on. The CEPC and Odo directories are examples of what a BSP directory should look like.

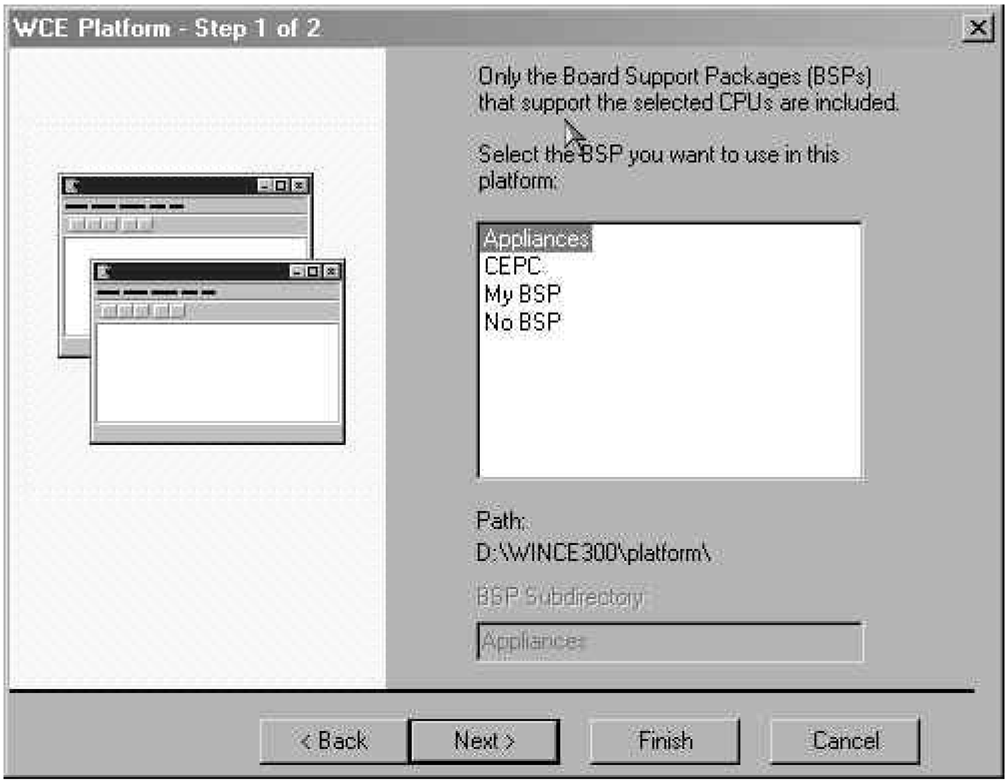
Listing 3.10 Modified `appliances.bsp`

```
//  *** Appliances ***

//  NOTE:  You will need to create an appliances.cec file
//  and import it into the catalog.  When you have done that,
//  you can delete the two #ADD_USER_OAL lines below.

//  *** Global components (for all the configs) ***
// #ADD_USER_OAL_BUILD_METHOD ('#BUILD(dirs, "$(_WINCEROOT)\platform\
Appliances")')

// #ADD_USER_OAL_COMPONENT ("{2367C526-2821-4CEF-94A2-7286D5152E6F}",
"Appliances")

//  Appliances

#ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B99-F1E4-11D2-85F6-004005365450}")

// EBOOT.BIN
#ADD_COMPONENT_BY_GUID_TO_ROOT ("{B4569ABC-F1E4-11D2-85F6-123405365450}")

//  *** CoreOS-specific components ***

//  *** MAXALL ***
#IF ("COREOS","MAXALL")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  ddi_flat
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{E2B049C8-F7DC-45d3-8204-
      0AA54FB4D4CC}")

      //  wavedev
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E7043C-EEA9-11D2-A092-
      0060085C1833}")

      //  serial
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E70441-EEA9-11D2-A092-
      0060085C1833}")

#ENDIF

//  *** MINKERN ***
#IF ("COREOS","MINKERN")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")
#ENDIF

//  *** IESAMPLE ***
#IF ("COREOS","IESAMPLE")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  ddi_flat
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{E2B049C8-F7DC-45d3-8204-
      0AA54FB4D4CC}")

      //  serial
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E70441-EEA9-11D2-A092-
      0060085C1833}")
```

```
#ENDIF

//  *** MINSHELL ***
#IF ("COREOS","MINSHELL")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  ddi_flat
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{E2B049C8-F7DC-45d3-8204-
      0AA54FB4D4CC}")

      //  serial
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E70441-EEA9-11D2-A092-
      0060085C1833}")

      //  wavedev
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E7043C-EEA9-11D2-A092-
      0060085C1833}")

#ENDIF

//  *** MINWMGR ***
#IF ("COREOS","MINWMGR")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  ddi_flat
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{E2B049C8-F7DC-45d3-8204-
      0AA54FB4D4CC}")

      //  wavedev
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E7043C-EEA9-11D2-A092-
      0060085C1833}")

      //  serial
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E70441-EEA9-11D2-A092-
      0060085C1833}")

#ENDIF

//  *** MINGDI ***
#IF (“COREOS”,"MINGDI")
```

```
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  ddi_flat
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{E2B049C8-F7DC-45d3-8204-
      0AA54FB4D4CC}")

      //  wavedev
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E7043C-EEA9-11D2-A092-
      0060085C1833}")

#ENDIF

//  *** MINCOMM ***
#IF ("COREOS","MINCOMM")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

      //  serial
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{35E70441-EEA9-11D2-A092-
      0060085C1833}")

#ENDIF

//  *** MININPUT ***
#IF ("COREOS","MININPUT")
      //  pc_ddk
      #ADD_COMPONENT_BY_GUID_TO_ROOT ("{B3509B75-F1E4-11D2-85F6-
      004005365450}")

#ENDIF

//  end of appliances.bsp file
```

We are now done creating a new BSP. The next time the WCE Platform Wizard is run, Appliances will appear as a choice in the **Select BSP** list box (Figure 3.20). Platform Builder filters the BSPs you can choose on the basis of the CPU you have chosen in the preceding dialog box. Since the Appliances BSP is valid only for x86, this CPU must be checked.



WCE Platform - Step 1 of 2

Only the Board Support Packages (BSPs) that support the selected CPUs are included.

Select the BSP you want to use in this platform:

Appliances
CEPC
My BSP
No BSP

Path:
D:\WINCE300\platform\

BSP Subdirectory

Appliances

Figure 3.20 The new BSP added to an existing list

Summary

The Windows CE Platform Builder is used to customize Windows CE and tailor the operating system to a particular hardware platform. Platform Builder comes with an IDE in the style of Visual C++ that provides wizards to help create a platform. A project can also be created that contains applications and platform-independent modules that will execute on the platform.

In this chapter we looked at how environment variables can be used to further customize a build for a particular platform. The Platform Builder can be extended, and a completely new board support package can be created. In Chapter 10 we will take a closer look at the fascinating process used by Platform Builder to build the Windows CE operating system.