Module 4: Implementing a Boot Loader

Contents

Overview 1
Role of the Boot Loader 2
Boot Options 3
Implementing a Boot Loader 11
Sample Boot Loader Requirements 31
Building a Boot Loader 33
Debugging a Boot Loader 35
X86 Boot Options 36
Boot Loader in Manufacturing 39
Review 40
In this module, you will learn the role of the boot loader in loading image onto a Windows CE development platform. You will also learn about the different technologies used with boot loaders. You will create a boot loader from a sample code, build a boot loader in the integrated development environment (IDE), and then learn the different techniques of debugging it.

After completing this module, you will be able to:

- Explain the role of a boot loader.
- Describe the boot options that are available in a boot loader.
- Implement a boot loader.
- Describe the use of the sample code provided to build a custom boot loader.
- Build a boot loader.
- Debug a boot loader.
- Describe how the Microsoft provided loadcepe boot loading technology works on x86 BIOS based platforms.
- Transit boot loader from the development stage into the manufacturing stage.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft® PowerPoint® file 2535A_04.ppt

Preparation Tasks

To prepare for this module:

- Read all of the materials for this module.
- Read all instructor notes and margin notes for this module

Module Strategy

Use the following strategy to present this module:

- Role of the Boot Loader
  
  Students who are unfamiliar with embedded design techniques can use this section to understand how images are loaded onto Windows CE development platforms.

- Boot Options
Use this section to describe all of the different technologies that are used with boot loaders. Describe how boot loaders are controlled, how are they downloaded, and how they write images.

- **Implementing a Boot Loader**
  Use this section to teach students how to create a boot loader from sample code. Platform Builder provides numerous samples of boot loaders, which students can utilize.

- **Sample Boot Loader Requirements**
  Use this topic to inform students of a general list of requirements for building a boot loader in Windows CE .NET. This list includes both software and hardware requirements.

- **Building a Boot Loader**
  Inform students that often, boot loaders are not built within the framework of the IDE. While it is possible to build a boot loader in the IDE, it is often easier to do it from a command prompt window.

- **Debugging a Boot Loader**
  A boot loader is a standalone executable. There is no operating system kernel to aid in debugging at this point. Students learn here the techniques for debugging a boot loader.

- **x86 Boot Options**
  There is an out of the box solution for downloading images onto x86 BIOS based platforms in Windows CE. In this section, students learn to use that solution.

- **Boot Loader in Manufacturing**
  It is often easiest to use your boot loader as part of your release product. Use this section to explain some possible mechanisms for transitioning a boot loader from the development environment into the manufacturing and release phase of your Platform's life cycle.
Overview

In this module, you will learn the role of the boot loader in loading image onto a Windows CE development platform. You will also learn about the different technologies used with boot loaders. You will create a boot loader from a sample code, build a boot loader in the integrated development environment (IDE), and then learn the different techniques of debugging it.

After completing this module, you will be able to:

- Explain the role of a boot loader.
- Describe the boot options that are available in a boot loader.
- Implement a boot loader.
- Describe the use of the sample code provided to build a custom boot loader.
- Build a boot loader.
- Debug a boot loader.
- Describe how the Microsoft provided loadcepc boot loading technology works on x86 BIOS based platforms.
- Transit boot loader from the development stage into the manufacturing stage.
Role of the Boot Loader

- Initializes the target device
  - Memory and interrupt controller
  - Setting up the clocks and MMU

- Controls the boot process
  - Directly boot an existing flash RAM image
  - Clear RAM before downloading
  - Memory read/write test

- Downloads the Windows CE image to RAM or flash RAM using:
  - Parallel port
  - Ethernet port

The boot loader is a target-specific program that manages the boot process. The boot loader initializes the target platform and allows you to download a Microsoft Windows CE .NET image from a development workstation to a target platform. Depending on your platform, you can download an image to RAM or to FLASH using either a parallel or Ethernet port. Once the image is loaded you can use the boot loader to monitor and debug the target device. Because the boot loader is only used in development, you do not need to include it with your Windows CE .NET Platform.

- The Ethernet port is generally preferred because of its speed. If you do not have an Ethernet connection on your board, or if you have small images, you can use the parallel port.

You can add other functionality to your boot loader, such as passing control directly to a Windows CE .NET image in flash RAM, or erasing the target platform RAM before downloading or performing memory read/write test.

The boot loader program is executed when you power on your platform. It is programmed into permanent memory, such as flash memory or ROM.

A boot loader is very useful in the development phase. It allows the developers to quickly download the operating system to debug.

- A new feature in Windows CE .NET is a virtual miniport driver called vMINI. This allows you to use the Ethernet port not only for boot loader monitoring and debugging, but also makes your network card available for retail activities. vMINI multiplexes the communication on the physical media so that both retail and debug activities can occur on the same port.
There are several options that you can implement in a boot loader. This section covers some of these options, such as boot loader communications, serial menu, and user initiated options.
The illustration on the slide shows the most common way to implement boot loader communications on a reverence platform.

The developer connects HyperTerminal on his development workstation to send serial commands back and forth to control the boot loader on the target device.

The Ethernet connection is used to download the image. It also provides the debug transport for debugging the target platform.
Serial Menu

**Topic Objective**
To describe the Serial connection used to control the boot loader.

**Lead-in**
The most common way developers implement boot loader control is through a serial terminal connection.

- Hyperterminal communicates using a plain text menu with the device using a serial port
- Bootloader on device executes requested functions such as:
  - Downloading from parallel or Ethernet
  - Booting from flash (if operating system is programmed already in FLASH)
  - Running a RAM test
  - Reading and writing from I/O ports
- User can modify this menu and add additional functions

The boot loader has several boot options. One option is designing a serial menu as a method to control the target boot process. This is done using a serial port connection.

A developer uses a HyperTerminal program on a development workstation to communicate with the boot loader using the serial ports. The boot loader receives character data from the development workstation, processes it and then performs the required boot loader function. Such functions include downloading from parallel or Ethernet, booting from flash (if operating system is programmed already in FLASH), running a RAM test, and reading and writing from I/O ports. A typical serial menu looks like:

```
*************************
Generic Boot loader Version 1.0
*************************
Select option:
D – Download Image
M – Memory Test
B – Start Windows CE .NET
E – Download via Ethernet
P – Download via Parallel
A – Dial-up Boot
=> (enter your selection here)
```

The user can modify this menu and add additional functions as needed. Remember the Windows CE .NET operating system is not running at this time, only the boot loader code is running. The user can also modify this menu to time-out and boot the operating system after a certain interval so that the user does not need to intervene to start Windows CE .NET.
User Initiated Options

You can modify your loader to perform various tasks, such as test memory, program flash, and boot from flash.

Memory Test
It is important to verify that you have register access in your platform. This allows you to test the low level functionality of the board. Here is an example of register access for an x86 platform

```c
UCHAR __inline READ_PORT_UCHAR(PUCHAR port) {
    return _inp((USHORT)port);
}

VOID __inline WRITE_PORT_UCHAR(PUCHAR port, UCHAR value) {
    _outp((USHORT)port, (value));
}
```

For other platforms other than x86, you use

```c
UCHAR __inline READ_PORT_UCHAR(PUCHAR port) {
    return *(volatile unsigned char * const)port;
}

VOID __inline WRITE_PORT_UCHAR(PUCHAR port, UCHAR value) {
    *(volatile unsigned char * const)port = value;
}
```
Dump/Fill Memory
This test is useful to check if certain memory locations are readable and writeable on your target platform. The user can test memory location in an 8bit, 16bit, 32bits chunks.

For example, the following code sample shows how you can you access memory in 32bit chunks:

```c
ULONG __inline READ_REGISTER_ULONG(PULONG Register)
{
    return *(volatile unsigned long * const)Register;
}

VOID __inline WRITE_REGISTER_ULONG(PULONG Register, ULONG value)
{
    *(volatile unsigned long * const)Register = value;
}
```

Program FLASH:
A loader can also be used to flash the Windows CE .NET image to FLASH. Usually the image is downloaded to RAM and then copied to FLASH.

Boot from FLASH:
You can also set your loader to boot from FLASH directly. This can be done by having the boot loader read a HW switch. Based the HW switch reading, the loader can boot the image from FLASH or go back to the menu.
Remote Update Boot Loader

One boot loader technology to consider is one which updates Windows CE .NET images from a development workstation, either locally or remotely. This technology uses a tiny Windows CE .NET operating system that downloads a new Windows CE .NET image in the flash memory. It can download this image from a remote FTP server (by automatic and user-initiated download) or from the parallel port (by factory download).

Use this loader technology to perform three different types of updates:

- **User-initiated update**
  A user can update a Windows CE .NET image intentionally while the target platform is running (using a hardware signal or a custom application). This update obtains a new Windows CE .NET image using a modem connection.

- **Automatic update**
  A target platform can download a Windows CE .NET image. This happens when the operating system is absent or corrupted, or when the system logically detects that a download is required. This update obtains a new Windows CE .NET image using a modem connection.

- **Factory update**
  The factory that maintains the target platform can also download a new Windows CE .NET image by using the parallel port.
Remote Update Boot Loader Booting Process

The illustration in the slide shows a specific implementation of a user initiated updated loader, which has two components:

- **A boot loader**
  
  This code is executed at boot. Depending on the logic implemented by the OEM, it can either jump to an existing Windows CE .NET image or boot a tiny operating system.

- **Tiny operating system**
  
  The code is a subset of the Windows CE .NET operating system that downloads a new Windows CE .NET image in the flash memory. It can download this image from a remote FTP server (by automatic and user-initiated download) or from the parallel port (by factory download).

At the boot, the CPU branches into the boot loader, which carries out logical tests to determine if it must boot the Windows CE .NET operating system or the tiny download operating system.

- **If the boot loader has to boot the Window CE .NET operating system**, it jumps directly to its starting point. It then runs the Window CE .NET operating system. At any moment, it can prompt the user to download a new Window CE image. If the user accepts, the operating system sets a download flag in flash and reboots the system to pass control back to the boot loader.

- **If the boot loader has to boot the tiny download operating system**, it determines the download type (modem or parallel) before hand and writes this type in RAM. Then it loads the tiny download operating system in RAM and executes it.

At the beginning of its execution, the tiny download operating system reads the download type and initializes the transport connection (modem or parallel). Then it begins the download of the image and writes the image in flash memory.
At the end of the download, it disconnects the transport connection and finalizes flash writing.

If the download is carried out correctly, it clears the download flag and reboots (passes the control to the boot loader). If there was a problem with the download, it reboots without clearing the download flag. Because this flag is not cleared, the boot loader again executes the tiny download operating system that resumes the download that failed.
Implementing a Boot Loader

Topic Objective
To explain the implementation details of a boot loader.

Lead-in
In this section, you will learn to identify the main components of a boot loader.

- Components of a Boot Loader
- Boot Loader Control Flow Functions
- Implementing the Startup Code
- Implementing the Main Code
- Implementing the Image Download Code
- Implementing the Parallel Port I/O Code
- Implementing the Ethernet Port I/O Code
- Implementing the Debug Serial I/O Code
- Implementing the Flash Write Code

This section focuses on the code necessary to implement a ROM based boot loader. This method applies to general Windows CE .NET platforms that are not BIOS based.
Components of a Boot Loader

**Topic Objective**
To describe the various components of a boot loader.

**Lead-in**
In this section, you will learn about the OEM startup, Main code, Image download code - serial, parallel, and Ethernet, Debug serial I/O code, Flash write code.

<table>
<thead>
<tr>
<th>OEM startup code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Is the first code executed in the boot loader</td>
</tr>
<tr>
<td>- Initializes the memory registers to enable memory access, set the CPU frequency, and initializes the cache</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- OEM platform initialization code</td>
</tr>
<tr>
<td>- Image download code</td>
</tr>
<tr>
<td>- Parallel port I/O code</td>
</tr>
<tr>
<td>- Ethernet port I/O code</td>
</tr>
<tr>
<td>- Debug serial I/O code</td>
</tr>
<tr>
<td>- Flash write code (optional)</td>
</tr>
<tr>
<td>- Firmware monitor (optional)</td>
</tr>
</tbody>
</table>

The boot loader consists of the two main code sections, OEM startup code and Main code.

**OEM startup code**

The OEM startup code is the first code executed in the boot loader. It initializes the memory registers to enable memory access, set the CPU frequency, and initializes the cache. It can perform any additional initialization needed for your boot loader. It then jumps to the main entry point implemented in `platform\<PLATNAME>\eboot\main.c`. This startup code is typically implemented in assembly.

**Main code**

This is part of the BLCOMMON code. This code is the main C code that manages the execution of the boot loader. It can receive user commands from the serial port or check hardware switches to determine what action it needs to perform. The Main code can also display a splash screen on the targeted platform monitor, for example boot loader version, firmware information, and company logo.

Main code contains the other code sections:

- **Image download code**

  The image download code allows an image to be downloaded through the:

  - Parallel port I/O code

    These are routines that read and write data over the connection between the target device parallel port and the development workstation parallel port. See `\kernel\hal\mdppfs.c` for implementation details. The parallel port image download code communicates with the Platform Builder to receive a Windows CE .NET image from the development workstation using the parallel port.

  - Ethernet port I/O code
There are routines that read and write data over the connection between the target device Ethernet port and the development workstation. See \WINCE400\public\common\oak\DRIVERS\ETHDBG\EBOOT for implementation details. The Ethernet download code communicates with the Platform Builder to receive a Windows CE .NET image from the development workstation using the Ethernet port.

- Debug serial I/O code

  The debug serial I/O code is a set of functions that allow you to read and write from the serial port. This code can be used by the Main code to receive user commands or send data through a serial terminal connection on the development workstation or the output window of Platform builder.

- Flash write code (optional)

  The flash write code is a set of functions that write data to a flash memory. You implement these functions only if your boot loader can load a Windows CE .NET image on a flash memory.

- Firmware Monitor (optional)

  These are routines to provide debugging support, such as dumping memory or disassemble code. Once the boot loader is functioning properly, you can perform basic debugging by sending firmware monitor commands to the debug serial monitor port.
In this section, we will look at the functions that are called during the boot loader execution.

**Important**  The BLCOMMON code is provided by Microsoft and is rarely changed. The functions in Download and Flash, are provided in sample form as well. However, the functions may need to be modified for your particular platform.

At power-on reset, the CPU first executes the **StartUp** function, which is typically written in assembly code. The Startup code mainly sets up memory access and initializes the cache. Then the StartUp functions jump into the main code (BootLoaderMain). BootLoaderMain is inside the BLCOMMON framework. This common infrastructure is implemented as a library BLCOMMON.LIB that is linked with the platform specific boot loader code. The BLCOMMON code is located in `\public\common\oak\drivers\ethdbg\blcommon`. It basically provides the standard control flow for any boot loader to use.

The purpose of each function is as follows:

- **OEMDebugInit**
  This function initializes the debug transport (usually a debug serial port).

- **OEMPlatformInit**
  This function performs any needed platform initialization for debug.

- **OEMPreadLoad**
  This function performs any pre download work. It can be customized to prompt for user feedback. Examples of this are prompting the user for a static IP address or jumping directly to a resident flash image.

- **DownloadImage**
  This function downloads the operating system image to RAM or FLASH.
OEMLaunch

The function starts the operating system and never returns.

Here is an example on how debug serial is implemented. See \WINCE400\PLATFORM\CEPC\SBOOT\main.c for more coding examples.

BOOL OEMDebugInit(void) {
    // Initialize our debug UART.
    OEMInitDebugSerial();
    return(TRUE);
}

void OEMInitDebugSerial(void)
{
    WRITE_PORT_UCHAR(IoPortBase+comLineControl, 0x80);   // Access Baud Divisor
    WRITE_PORT_UCHAR(IoPortBase+comDivisorLow, 0x03);    // 38400
    WRITE_PORT_UCHAR(IoPortBase+comDivisorHigh, 0x00);
    WRITE_PORT_UCHAR(IoPortBase+comFIFOControl, 0x01);   // Enable FIFO if present
    WRITE_PORT_UCHAR(IoPortBase+comLineControl, 0x03);   // 8 bit, no parity
    WRITE_PORT_UCHAR(IoPortBase+comIntEnable, 0x00);     // No interrupts, polled
    WRITE_PORT_UCHAR(IoPortBase+comModemControl, 0x03);  // Assert DTR, RTS
}

The download functions are called by DownloadImage. They are OEMReadData, OEMShowProgress, and OEMMapMemAddr. The flash routines are OEMIsFlashAddr, OEMWriteFlash, OEMStartEraseFlash, OEMFinishEraseFlash, and OEMContinueEraseFlash. The purpose of these functions is described below:

- OEMReadData
  This function reads data from the transport provided during the download process.

- OEMMapMemAddr
  This function maps the memory between the NK and boot loader’s view of memory.

- OEMShowProgress
  This function lets the users know that the download is in progress. You can use the LED for this.

- OEMIsFlashAddr
  This function returns TRUE if address is a flash address, else FALSE if not.

- OEMFinishEraseFlash
  This function is called when the loader is about to write the image to flash.

- OEMWriteFlash
  This function writes the image to flash.

- OEMContinueEraseFlash
This function is called when downloading the image. You can check the flash erasing progress and recover when you get an error.

- **OEMStartEraseFlash**

  This function is called as soon as the boot loader knows how much flash memory is needed based on the size of the image.

Here is an coding example of **OEMReadData**. This example was taken from \WINCE400\PLATFORM\CEPC\EBOOT\main.c.

```c
// callback to receive data from transport
BOOL OEMReadData (DWORD cbData, LPBYTE pbData)
{
    BOOL fRetVal;
    int cPercent;
    if (fRetVal = EbootEtherReadData (cbData, pbData)) {
        g_dwOffset += cbData;
        if (g_dwLength) {
            cPercent = g_dwOffset * 100 / g_dwLength;
            DrawProgress (cPercent, FALSE);
        }
    }
    return fRetVal;
}
```

To find more Flash routine examples, see \WINCE400\PLATFORM\SA11X0BD\EBOOT\EBOOTLIB\flash.C and \WINCE400\PLATFORM\CEPC\EBOOT\main.c.

---

**Note** To find more Flash routine examples, see \WINCE400\PLATFORM\SA11X0BD\EBOOT\EBOOTLIB\flash.C and \WINCE400\PLATFORM\CEPC\EBOOT\main.c.
Implementing the Startup Code

The OEM Startup code is implemented in the StartUp function. This function is almost the same as its OAL counterpart, except that it initializes the cache. You can reuse the OAL function to implement this boot loader function. An example of this sharing of code is shown in \WINCE400\PLATFORM\SA11X0BD\EBOOT\EBOOTLIB\ARM\FWP2.S.

The example shown above of the StartUp function is simplistic. This example came from a CEPC where most of the platform initialization is already done by the x86 BIOS. For non-x86 platforms, you add your target specific code initialization here. In this example, StartUp jumps to _BootUp which then calls out to _BootLoaderMain. For a more detailed example of CPU StartUp, explore one of the non-X86 platform BSP's.
Implementing the Main Code

In BootloaderMain, this is where the main control of the boot loader is done. The code example shown in the slide is common code (BLCOMMON), which all boot loaders can use. You can make modifications to this function but keep in mind that this is common code and all platforms can use this. This function can also check hardware switches to determine what download options to automatically implement.

First, you need to initialize your debug transport. This is done in OEMDebugInit(). Next, you need to initialize the platform which is done in OEMPlatformInit(). In that function, you need to initialize the interrupt timer, clocks, and other needed peripherals. Then OEMPreDownload() is called. This is where you can customize your menu to prompt for user details and add additional code, such as display a splash screen. The returned menu commands prompt the download to begin by calling DownloadImage().

DownloadImage() is part of the BLCOMMON common code. It makes calls to OEMReadData, OEMShowProgress, OEMMapMemAddr, OEMIsFlashAddr, OEMStartEraseFlash, OEMContinueEraseFlash, OEMFinishEraseFlash, and OEMWriteFlash. If you have no flash on your platform, you need to implement OEMReadData at minimum. The others are technically optional.

Once the image is downloaded, you can add code to copy the image to FLASH. Upon return from DownloadImage(), you call OEMLaunch() to launch the image. Once the image is launched, Windows CE .NET takes control of RAM and reclaim the area where the boot loader was shadowed. Therefore, the boot loader is no longer active.
Implementing the Image Download Code

**Topic Objective**
To implement the code for image download.

**Lead-in**
In this section, you will learn how to implement the code for image download.

The communication between the target platform and the development workstation must first be initialized. To do this, the image download code sends a BOOTME packet to Platform Builder. Platform Builder receives this packet and acknowledges it by sending a synchronization packet, followed by the start address and the length of the image. The image download code reads this information and then communication transport is established.

Information about this BOOTME packet is located in `\WINCE400\public\common\oak\INC`.

**Delivery Tip**
This sample is also truncated code.

**Note**
If you are downloading nk.bin to RAM, do not overwrite your loader’s RAM area. Verify the bib file settings for your loader to the settings for building your nk.bin and ensure that there is no overlap.
typedef struct _EDBG_BOOTME_DATA
{
    UCHAR  VersionMajor;   // Bootloader version
    UCHAR  VersionMinor;   // Bootloader version
    USHORT MACAddr[3];     // Ether address of device
    (net byte order)
    DWORD  IPAddr;         // IP address of device (net byte order)
    UCHAR  PlatformId[EDBG_MAX_DEV_NAMELEN+1];   // Platform Id string (NULL terminated)
    UCHAR  DeviceName[EDBG_MAX_DEV_NAMELEN+1];   // Device name string (NULL terminated). Should include
    // platform and number based on Ether address (e.g. Odo42, CEPCLS2346, etc)
    UCHAR  CPUId;           // CPU identifier (upper nibble = type)
    // The following fields were added in CE 3.0 Platform Builder release
    UCHAR  uBootmeVer;      // BOOTME Version. Must be in the range 2 -> EDBG_CURRENT_BOOTME_VERSION, or
    // remaining fields will be ignored by Eshell and defaults will be used.
#define EDBG_BOOTFLAG_FORCE_DOWNLOAD 0x0001 // always download image
    DWORD  dwBootFlags;     // Boot Flags
    USHORT wDownloadPort;  // Download Port (net byte order) (0 -> EDBG_DOWNLOAD_PORT)
    USHORT wSvcPort;       // Service Port (net byte order) (0 -> EDBG_SVC_PORT)
} EDBG_BOOTME_DATA, *PEDBG_BOOTME_DATA;

OEMReadData is called to read data from the transport. This transport could be serial, parallel, or Ethernet depending on what you choose to implement.

Note To learn how to implement a serial version of OEMReadData, see \WINCE400\PLATFORM\CEPC\SBOOT\main.c and to learn how to implement an Ethernet version, see \WINCE400\PLATFORM\CEPC\EBOOT\main.c.

DownloadImage then reads in the header of the file to check if it is a *.BIN file. The header should contain B000FF. Next the image start and the image length values are read. These are values embedded in the *.BIN file. The *.BIN format is described in the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync bytes (optional)</td>
<td>7</td>
<td>Byte 0 is B, indicating a .bin file format. Bytes 1-6 are reserved and set to 0, 0, 0, F, F, \n.</td>
</tr>
<tr>
<td>Image header, consisting of:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Image address</td>
<td>4</td>
<td>Physical starting address of the image.</td>
</tr>
<tr>
<td>Image length</td>
<td>4</td>
<td>Physical length, in bytes, of the image.</td>
</tr>
</tbody>
</table>
Record address 4 Physical starting address of data record. If this value is zero, the record address is the end of the file, and record length contains the starting address of the image.

Record length 4 Length of record data, in bytes.

Record checksum 4 Signed 32-bit sum of record data bytes.

Record data Record length Record data.

Next a function pointer is set up for the OEMVerifyMemory function. This basically verifies memory before the write. This function is optional.

Then the DownloadImage checks if the address is a valid flash address. If it is, then it starts erasing flash. If not, then you have the option to download to RAM at this time (code is not implemented).

The rest of the code performs the downloading to flash.

OEMFinishEraseFlash is called when DownloadImage is about to write the image to flash. You should make sure that flash is erased completely before writing to it. Next the OEMWriteFlash is called to program flash.

Note For more information on implementing the Flash Write code, see topic “Implementing the Flash Write Code” in PowerBuilder Help.
Implementing the Parallel Port I/O Code

**Topic Objective**
To implement the parallel port I/O code.

**Lead-in**
In the section, you will learn how to implement the parallel port I/O code.

- Supports communication between target platform and CESH
- The two handshaking modes
  - Client-side parallel interface
  - Host-side parallel interface
- Functions that are implemented:
  - OEMParallelPortGetByte
  - OEMParallelPortSendByte
  - OEMParallelPortGetStatus

The parallel port I/O code is used to support the data exchange between the target platform and the debug tool shell (Cesh.exe) through the parallel port. The target platform and the debug tools shell can communicate in two different handshaking modes, depending on the parallel I/O chip set of your platform. These handshaking modes are:

- Client-side parallel interface
  - Client-side parallel interface uses a standard parallel cable.
- Host-side parallel interface
  - Host-side parallel interface uses a null parallel cable.

**Note**
You can find the parallel port reference (pin assignment) for both the modes in the Platform Builder documentation.

The parallel port I/O code is composed of the following functions:

- **OEMParallelPortGetByte**
  - The OEMParallelPortGetByte function gets a byte from the development workstation through the parallel port.

- **OEMParallelPortSendByte**
  - The OEMParallelPortSendByte function writes a byte to the development workstation through the parallel port.

- **OEMParallelPortGetStatus**
  - The OEMParallelPortGetStatus function gets the status of the parallel port (available or unavailable).

**Note**
All OEMxxx functions can be named according to the requirements of the developer. These functions are not linked with the Microsoft-provided
libraries, so they do not need special names. For ease of use, the functions are prefixed with the OEM because these functions are also implemented in OAL with those file names. You can reuse them when implementing the boot loader.

Here are a sample code taken from \WINCE400\PLATFORM\CEPC\KERNEL\HAL\MDPPFS.C

```c
int OEMParallelPortGetByte(void)
{
    BYTE    value;
    
    if ( NoPPFS ) {
        return (-1);
    }

    Retry:
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_READ | PAR_CTRL_STROBE);
    
    if ( !WaitForStatus(PAR_STAT_NACK, PAR_STAT_NACK) ) {
        return (-1);
    }

    value = READ_PORT_UCHAR(IoPortBase + PAR_PORT_DATA);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_READ | PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    
    if ( !WaitForStatus(PAR_STAT_NACK, 0) ) {
        return (-1);
    }

    if ( bLastOpWasWrite && value == 0x1A ) {
        // Periodically after the first character we receive after a write is the last byte sent of the previous write.
        // For now we will ignore it
        LOG_ENTRY(LOG_ENTRY_EVENT | LOG_EVENT_SKIP_RECEIVE);
        bLastOpWasWrite = FALSE;
    goto Retry;
    }
    bLastOpWasWrite = FALSE;
    return (value);
}
```
VOID OEMParallelPortSendByte(BYTE chData)
{
    if ( NoPPFS )
        return;
    if ( !WaitForStatus(PAR_STAT_NBUSY, 0) ) {
        return;
    }

    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL,
            PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_DATA, chData);

    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL,
            PAR_CTRL_AUTOFEED);
    bLastOpWasWrite = TRUE;
}

int OEMParallelPortGetStatus(void)
{
    DWORD dwStatus, dwData;
    dwStatus = PAR_AUTOEN | PAR_SELECT | PAR_NFAULT |
            PAR_INTR_MASK;
    *(PVDWORD)(PAR_CONTROL_REG) = dwStatus;
    dwStatus = *(PVDWORD)(PAR_CONTROL_REG);
    dwData = *(PVDWORD)(PAR_DATA_REG);
    if((!((dwStatus & PAR_INTR)))
        return -1;
    *(PVDWORD)(PAR_CONTROL_REG) = (PAR_BUSY | PAR_NFAULT |
            PAR_AUTOEN | PAR_SELECT);
    return (dwData & 0xff);
}
Implementing the Ethernet Port I/O Code

The Ethernet port I/O code works with the Platform Builder integrated download tools to download a Windows CE .NET image.

The Ethernet download process is as follows:

1. First, the boot loader must assign an IP address for the target platform. Depending on the user network, this address can be a static IP address or an address given by a dynamic host configuration protocol (DHCP) server.

2. When the IP address is validated, the boot loader sends a broadcast BOOTME packet over the network. This broadcast BOOTME packet indicates to all of the Platform Builder Download connected to the network that the target platform is waiting for a download. The BOOTME packet contains:
   - A BOOTME code.
   - The IP address of the target platform.
   - A display string (the string that identifies the platform, for example, CEPC1234).

3. Platform Builder answers the BOOTME packet and then initializes the download.

4. The download is carried out by a trivial file transfer protocol (TFTP).
   - At the end of the download, Platform Builder sends the target platform a JUMPING packet notifying the boot loader to jump to the downloaded image.

Note  An Ethernet download code is supplied with Platform Builder. When you implement an Ethernet download code for your boot loader, you can take this code and adapt it for your platform. Look under \.\Public\Common\oak\drivers\Ethdbg
When implementing the Ethernet download code, you use OEM functions such as `OEMEthGetFrame` and `OEMEthSendFrame`. These functions already exist in the OAL, so you can reuse them in the boot loader.
Implementing the Debug Serial I/O Code

**Topic Objective**
To explain the functions to be implemented in the debug serial I/O code.

**Lead-in**
In this section, you will learn how to implement the serial debug I/O code.

- Debug serial I/O code allows the boot loader to send and receive data from the development workstation through the serial port.

- **Functions to be implemented:**
  - OEMInitDebugSerial initializes the serial port
  - OEMReadDebugByte reads a byte on the debug serial port
  - OEMWriteDebugByte outputs a byte on the debug serial port

The debug serial I/O code allows the boot loader to send and receive data from the development workstation using the serial port.

To implement the debug serial I/O code, you must implement the following functions:

- **OEMInitDebugSerial**
  Initializes the serial port (speed, parity, stop bit length).

- **OEMReadDebugByte**
  Reads a byte on the debug serial port.

- **OEMWriteDebugByte**
  Outputs a byte on the debug serial port.

**Note**
These functions are also implemented in the OAL. You can reuse them to implement the boot loader.
Implementing the Flash Write Code

If you want your boot loader to support writing an image in flash, you must implement the flash write code. The flash write code is used by the image download code to write in flash RAM bytes that are read from either the parallel or Ethernet port.

The flash write code is composed of the following functions:

- **OEMIsFlashAddr**
  Determines a valid flash address.

- **OEMStartEraseFlash**
  Erases flash.

- **OEMContinueEraseFlash**
  Continues to erase flash.

- **OEMFinishEraseFlash**
  Does any post-write cleanup in flash memory.

- **OEMWriteFlash**
  Writes bytes to the flash memory.

Here is a sample of **OEMIsFlashAddr** and **OEMStartEraseFlash**:
BOOL OEMIsFlashAddr( DWORD dwPhysStart ) {
    BOOL retval;
    if ((dwPhysStart & 0x0FFFFFFFUL) < (FLASH_ADDR_START & 0x0FFFFFFFUL) ||
        (dwPhysStart & 0x0FFFFFFFUL) >= (FLASH_ADDR_END & 0x0FFFFFFFUL))
        retval = FALSE;
    else
        retval = TRUE;
    return retval;
}

BOOL OEMStartEraseFlash (DWORD dwStartAddr, DWORD dwLength) {
    ULONG i = 0;
    ULONG nNumBlocks = 0;
    //
    // Make sure the start and end addresses are in flash.
    //
    if (!OEMIsFlashAddr(dwStartAddr) ||
        !OEMIsFlashAddr(dwStartAddr + dwLength - 1))
        { EdbgOutputDebugString("ERROR: OEMStartEraseFlash - not a flash address (0x%x or 0x%x).
\" dwStartAddr,
            (dwStartAddr + dwLength - 1));
            return(FALSE);
        }
    //
    // Make sure start address is block-aligned.
    //
    if (dwStartAddr % FLASH_BLOCK_SIZE)
        { EdbgOutputDebugString("ERROR: OEMStartEraseFlash - start address isn't block aligned (0x%\n\\", dwStartAddr); return(FALSE);
        }
    if (dwLength & 0x03)
        { EdbgOutputDebugString("ERROR: OEMStartEraseFlash - length isn't an integral number of longwords (0x%\n\\", dwLength);
            return(FALSE);
        }
    //
    // Clear the FLASH cache, where we'll be storing the image during download.
    //
    // memset((void *)FCACHE, 0, dwLength);
    //
    // Determine affected flash blocks.
    //
gnStartBlock = (dwStartAddr - FLASH_BASE) / FLASH_BLOCK_SIZE;
gnEndBlock = ((dwStartAddr + dwLength + (FLASH_BLOCK_SIZE - 1) - FLASH_BASE) / FLASH_BLOCK_SIZE);
gnBlocks = (int)(gnEndBlock - gnStartBlock);
gnBlockCount = gnStartBlock;

EdbgOutputDebugString("Erasing flash blocks: start block = %d end block = %d\r\n", gnStartBlock, gnEndBlock);

return(TRUE);
}

Additional examples can be found at \WINCE400\PLATFORM\<platform name>\EBOOT\flash.c
Sample Boot Loader Requirements

There are certain boot loader features that are required for a board support package (BSP) in Platform Builder.

These requirements ensure that all boot loaders are consistent across BSPs in Platform Builder. You should note that the x86-based platforms can be exempt from some of these conditions, unless the boot loader supports flashing the operating system image to RAM and linear flash.

The required boot loader features are:

- A BSP must have a mode to run the Windows CE boot loader as the default.
- No user input should be required to download the image.
- Boot loader must reside in flash and must use BLCOMMON.
- Boot loader must provide the ability to flash and update.
- Boot loader must provide the ability to download an image to linear flash and RAM.
- The boot loader should work with all Platform Builder switches.
- There should be a fallback mechanism.
- The boot loader can use any transport to download images.

Hardware considerations for target device:

- Allow ROM or flash memory containing the boot loader to be replaceable.
- Provide a hardware switch.
- Provide ample RAM and flash memory to support debugging.
- Provide debug LEDs on a target device.

Two options for downloading the boot loader to the target device are: download it into a flash memory device and burn the boot loader image onto a ROM chip.
process is quick and reliable. All the sample boot loaders in the Platform Builder BSPs default to using the Ethernet port for downloading operating system images, but some have support for the parallel port.

The hardware design of the target influences how it is used and how effective it is as a development and test tool.

The following list describes hardware considerations for the target device:

- Allow ROM or flash memory containing the boot loader to be replaceable.
- Provide a hardware switch to select booting directly to an operating system image in flash memory or to select booting the boot loader.
  - If a switch is not implemented to bypass the boot loader and the device boots directly to the operating system image, the boot loader must support power off and resume, and must not use any RAM that is used by the operating system image. When it is not possible to bypass the boot loader, the boot loader may corrupt the RAM that is used by an operating system image, resulting in diagnostic problems.
- Provide ample RAM and flash memory to support debugging.
- Provide debug LEDs on a target device during the early stages of development to facilitate debugging of the initial startup code.

The sample boot loader and target device are developed to use Ethernet and the parallel port for downloading operating system images. However, you can use alternative methods for downloading operating system images if the operating system images are installed in a quick and reliable manner.

There are two options for downloading the boot loader to the target device:

- Download the boot loader into a flash memory device and then insert the flash memory device into the target device.
- Burn the boot loader image onto a ROM chip and then insert the chip into the target device.

On target devices that do not cache the ROM-based boot loader code, you should configure the boot loader to move into RAM so that it runs faster, provided the RAM is cache-backed. To configure the boot loader to move into RAM, set the addresses of all variables used in the process to the destination addresses in RAM.
Building a Boot Loader

**Topic Objective**
To build the boot loader.

**Lead-in**
Building a boot loader can be done in the IDE; however it is sometimes easier to bend the rules a little.

- On the Build menu, click Open Build Release Directory
- Enable the built features to be copied to the `%_FLATRELEASEDIR%` directory
  - Set WINCEREL=1
- Navigate to the boot loader directory to modify the binary image builder file (.bib), `Boot.bib`
  - `cd %_TGTPLATROOT%\Eboot`
- Build Eboot.bin
  - Build -cfs

The following example builds the default sample included with Platform Builder. This builds Eboot.bin, which is the Ethernet download program for DOS/BIOS downloads.

From your workspace, on the Build menu, click Open Build Release Directory. The command prompt window allows you to build selected components, but should not be used to attempt full Platform builds.

Type the following command line to enable the built features to be copied to the `%_FLATRELEASEDIR%` directory.

```
Set WINCEREL=1
```

Modify the binary image builder file (.bib), `Boot.bib`, located in `%_TGTPLATROOT%\Eboot\buildbin\%_TGTCPUTYPE%`, to match your platform configuration.

`Boot.bib` is a text file that specifies the size, type, and location of the memory on the target device where you store the boot loader.

**Example**

**Note** For more information on `Boot.bib` file, see “Editing the Boot Loader Image Builder File” in online help.

Type the following command line.

```
cd %_TGTPLATROOT%\Eboot
Build -cfs
```

**Note** When building a general boot loader, you have to pass the executable through Romimage to create the binary loaded onto the reference platform.
Module 4: Implementing a Boot Loader

Debugging a Boot Loader

<table>
<thead>
<tr>
<th>Topic Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>To explain several ways to debug a boot loader.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lead-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this section, you will learn useful methods of debugging a bootloader.</td>
</tr>
</tbody>
</table>

- **Guidelines for debugging a boot loader**
  - Make sure you are building the loader correctly
  - Make sure you flash the boot loader to the correct location in ROM
  - Use LEDs
  - Initialize the debug serial for serial debug output
  - Use a Port 80 I/O card
  - Use GPIO pins to toggle

Debugging a boot loader is one of the first tasks you encounter during porting Window CE .NET to a new platform. Here are some helpful tips for debugging a boot loader.

- Make sure you are building the loader correctly. Review the *.BIB file and make sure the CONFIG and MEMORY sections contain the correct values. These values define how the boot loader is loaded into memory. Be careful not to have the RAM section of the boot loader overwrite the CODE section during execution.

- Make sure you flash the boot loader to the correct location in ROM. For example, in x86 architectures the boot loader is flash starting in high memory. In other ARM based platforms, the bootloader is flash at 0x0.

- Use the LED as an indicator of where you are in the boot loader code. This is synonymous with debug print statements.

- You can initialize the debug serial for serial debug output.

- You can use a Port 80 I/O card. These cards come in ISA or PCI version. These cards are useful if your target platform support an ISA/PCI bus.

- You can use GPIO pins to toggle as another indication of telling where you are in the bootloader code.

A JTAG debug interface unit enables the software tools to debug your target platform using the processors’ JTAG pins with ICE-like functionality (breakpoints, watchpoints, single stepping).

Some JTAG debug tools have features such as fast code download using Ethernet, execution of code from the targets’ real ROM address space, and support of unlimited number of breakpoints in the ROM address space. For most applications, these tools do not require an intrusive ROM monitor that is typically required by traditional ROM emulators. All of the target CPU run control and debug features are accomplished using the JTAG interface.
X86 Boot Options

x86 platforms have two possible booting options:

- BIOS boot using a DOS program loadcepc.exe to load an image.
- Replace BIOS with a standard CE boot loader.

LOADCEPC.EXE

The Load CEPC tool (loadcepc.exe) is a (DOS) tool that you can use to load and boot a Windows CE image on a CEPC. You can also use loadcepc.exe to copy files from a development workstation to your CEPC. Loadcepc.exe can be used standalone or in an autoexec.bat file. All you need to do is make sure the arguments to loadcepc.exe are correct. Loadcepc.exe uses the arguments to determine how to connect the CEPC device by using parallel, serial, or Ethernet. Here is the loadcepc.exe usage:

```
```

Arguments:

- /B
  Baud Rate. Default is 19200
- /C
  Com port. Default is COM1
- /D
  Display resolution
- /E
  Ethernet debug port
- /H
Help

- /L
  Display mode settings
- /P
  Use Parallel port
- /Q
  Use Serial port
- /V
  Verbose mode
- Inputfile
  File to be downloaded from development workstation to target device

Here is an example that shows how to specify a 768x576 display area with 8bp color and a 1024x768 physical display mode:

```
Loadcepc /b:38400 /v /e:1:300:5 /l:768x576x8:1024x768 eboot.bin
```

You can use loadcepc as a file copy tool. Here is the usage:

```
Loadcepc [\[-s\] [:] Pattern] [\[-g\] \[-r\] \[-u\] [:] Pattern] [Destination]
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s</td>
<td>Shows a directory of files matching Pattern.</td>
</tr>
<tr>
<td>-g</td>
<td>Get files matching Pattern and copies the files to the optional Destination.</td>
</tr>
<tr>
<td>-r</td>
<td>Refreshes files matching Pattern that already exist in Destination.</td>
</tr>
</tbody>
</table>

Pattern  Specifies the Windows CE .NET file path.
Destination Updates Destination on the target device that matches Pattern with files from the development workstation.

**Note** Loadcepc.exe only supports using the parallel connection to download operating system image binary files. When you use loadcepc.exe with eboot.bin you can use the Ethernet connection to download these files; however, eboot.bin does not support the –s, -g, -r, -u parameters.

**BIOS Replacements:**

You can use the x86 ROM-based boot loader as an alternative to loadcepc.exe. The x86 ROM-based loader is designed such that you do not need the BIOS or DOS Services. By default, this loader replaces the system BIOS (in flash/ROM) and initializes the hardware before providing operating system image loading services.

You can also use the BIOS to launch a loader stored on a FAT drive and called by the master boot record. This loader in turn controls the booting of a Windows
CE image. This technology is similar to a regular DOS boot. Instead of launching IO.sys from DOS, the FAT drive is calling your loader.
Boot Loader in Manufacturing

- **Listen to debug port**
  - Instead of issuing a menu, the function should listen to the debug serial port
  - If a signal is detected, jump automatically to the download

- **Check for physical condition**
  - Instead of using the debug port, the code should check some physical settings on the board

- **Check for Compact Flash card**
  - If a compact flash card is present, and an image is detected there, automatically transfer it to onboard Flash and boot

The following are some examples of altering the Main function to move your boot loader from the development stage into the manufacturing stage of your Platform’s life cycle:

- Instead of issuing a menu, the function should listen to the debug serial port. If a signal is detected, jump automatically to the download.

- Instead of using the debug port, the code should check some physical settings on the board. This could be a dip switch, hidden push button, sequence of keys, or other physical settings. If the monitor detects this condition, automatically jump to download. Otherwise, boot the existing Flash image.

- Have the monitor check the pcard slot. If a compact flash card is present, and an image is detected there, automatically transfer it to onboard Flash and boot.

**Note**  It is important that your device user never accidentally enters the download state. Careful consideration should be given to your Platform’s use and users to determine the appropriate security level for your device’s image.
1. What is the role of a boot loader?
   To download a Windows CE .NET image from the development workstation to the target platform by using serial, parallel, or Ethernet port.

2. What is the first function executed by the boot loader?
   The StartUp function

3. Which are the two solutions that allow a user to switch between various boot loader functions?
   A hardware solution with the hardware switches or a software solution implementing a menu through the serial port.

4. Which extended functions do you have to supply with the boot loader if you want to implement a firmware monitor?
   The serial port functions to receive and send data through the parallel port (OEMInitDebugSerial, OEMReadDebugByte, and OEMWriteDebugByte).

5. Which are the special functions that you implement to enable a boot loader to download in a flash RAM?
The flash write functions (OEMIsFlashAddr, OEMStartEraseFlash, OEMContinueEraseFlash, OEMFinishEraseFlash, OEMWriteFlash, and OEMFlashWriteStatus)

6. What is the difference between the OAL StartUp code and the boot loader StartUp code?
   
The boot loader StartUp code initializes the cache, not the OAL StartUp code. The OAL StartUp code jumps to KernelStart while the boot loader StartUp code jumps to main.

7. What are the three types of remote image updates you can have?
   