CE Boot Framework

Writer: Wendy Giberson

Technical Reviewer: Karel Danihelka

Published: December 2011

Applies To: Windows Embedded Compact 7

Abstract

A boot loader initializes your device's hardware and then starts the OS. Board support packages in Windows Embedded Compact 7 include either a sample boot loader based on the BLCOMMON library like the one used in Windows Embedded CE 6.0 or a customizable production-quality boot loader based on the new CE Boot framework. This paper describes the CE Boot framework, including:

- The advantages of the CE Boot framework compared to the BLCOMMON library.
- The components of the CE Boot framework.
- Typical boot sequences showing the progress from power-on to starting the OS.
- The main functions, structures, and classes of the CE Boot code.
- The sample boot drivers that are included with Windows Embedded Compact 7.

This paper is intended for developers who are new to Windows Embedded Compact or for developers who are familiar with the BLCOMMON boot loader but want to use CE Boot for its extensible, layered architecture.

© 2011 Microsoft. All rights reserved.

Contents

Introduction 4

CE Boot Architecture 4

Boot Sequence States 6

CE Boot Code 9
- Core Code 9
  - Boot Loader Context 10
  - Flow 13
    - BootStart Function 15
    - BootMain Function 15
    - OEMBootInit Function 15
    - OEMBootLoad Function 16
    - OEMBootRun Function 16
    - OEMBootPowerOff Function 17
    - BootJumpTo Function 17
  - Memory Allocation 18
    - BootHeapInit Function 18
    - BootAlloc Function 18
    - BootFree Function 19
  - Memory Mapping 19
    - BootPAtoVA Function 19
    - BootVAtoPA Function 20
    - BootImageVAtoPA Function 20
- Boot Scenario Code 21
- Notification and Logging Code 22
  - Notification Code 22
    - OEMBootNotify Function 22
  - Logging Code 22
    - BootLog Function 23
- Boot Driver and Boot Driver Factory Code 23
  - Boot Driver Factory Code 24
    - OEMBootCreateDriver Function 24
  - Boot Driver Code 25
    - BootDriver_t Class 27
    - BootDriverIoCtl Function 28
    - BootDriverDeinit Function 29

© 2011 Microsoft. All rights reserved.

Boot Driver Examples 29
Boot Block 30
Terminal 31
Display 32
Download 32
File System 33
Transport 34

Conclusion 34

Additional Resources 35

© 2011 Microsoft. All rights reserved.

Introduction

A board support package (BSP) is the software that enables Windows Embedded Compact 7 to run on your device. A BSP consists of four major components, one of which is a boot loader.

The boot loader is typically the first code that is executed when you power on a device that runs Windows Embedded Compact 7. The sample BSPs that are provided with Windows Embedded Compact 7 include two sample boot loaders. The original boot loader, described in The Basics of Bringing up a Hardware Platform (http://go.microsoft.com/fwlink/?LinkID=205801), is based on the BLCOMMON library. It was included with earlier versions of Windows CE and is included in Windows Embedded Compact 7 also. The newer boot loader, CE Boot, which is described in this article, was released with Windows Embedded Compact 7. It uses a framework called CE Boot.

The BLCOMMON boot loader is used by the ARM and MIPS sample BSPs that are included with Windows Embedded Compact 7. CE Boot is used by the BSPs that are based on the x86 CPU architecture, which are the CEPC, Virtual CEPC, and eBox BSPs. CE Boot is also implemented for the ARM architecture, although the ARM sample BSPs use the BLCOMMON library. This article covers the implementation of CE Boot for the x86 CPU architecture.

The CE Boot framework was created with the following goals in mind:

- **Reusability**. You can reuse the core boot loader code for different devices because the hardware-specific operations are encapsulated by the boot driver code.
- **Extensibility**. You can create different boot scenarios by assembling various combinations of drivers, because the drivers are initialized and accessed through a well-defined generic interface.
- **Dynamic memory allocation**. CE Boot can use memory on the heap.

This article contains the following sections:

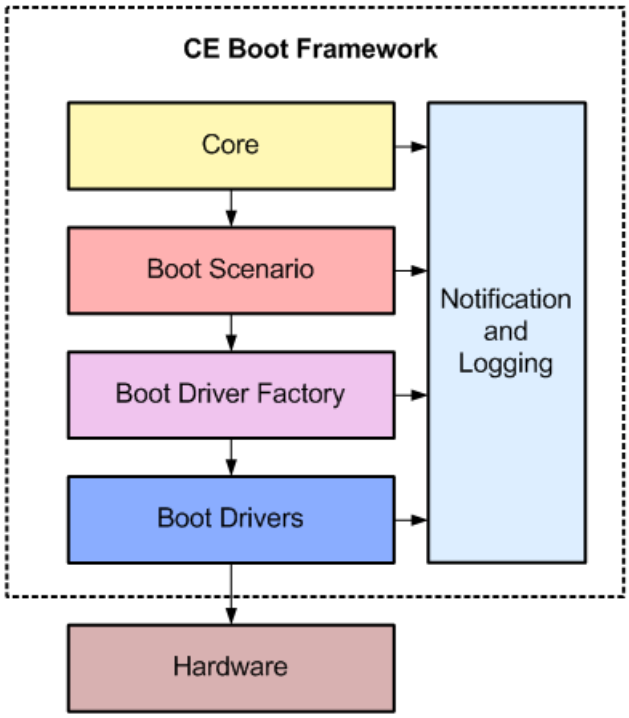
- **CE Boot Architecture**. A description of the components of CE Boot.
- **Boot Sequence States**. The states that CE Boot cycles through to transition from device power-up to starting the OS image.
- **CE Boot Code**. The main APIs in the core, notification and logging, boot driver, and boot driver factory code.
- **Boot Driver Examples**. Descriptions of the boot drivers that are included with Windows Embedded Compact 7.

CE Boot Architecture

The CE Boot framework consists of the components shown below in Figure 1. The core code calls the code in the boot scenario, which calls the boot driver factory to instantiate boot drivers. The boot drivers communicate with the hardware to do the work (such as downloading the OS over Ethernet or reading it from persistent storage).

© 2011 Microsoft. All rights reserved.

Figure 1: CE Boot Framework Architecture



The CE Boot components shown in Figure 1 are:

- **Core**. The core code controls the flow of execution, memory mapping between physical and virtual addresses, and memory allocation. For more information, see Core Code.
- **Boot Scenario**. The boot scenario consists of code called by the core code to perform individual tasks, such as loading the OS into memory. For more information, see Boot Scenario Code.
- **Boot Driver Factory**. The boot driver factory is a function that creates and initializes a boot driver and returns a handle to it. For more information on the boot driver factory, see Boot Driver Factory Code.
- **Boot Drivers**. The boot driver code standardizes access to hardware devices by using only IOCTLs and a deinitialization function to communicate with them. For more information on boot drivers, see Boot Driver Code. (Note that the core CE Boot library can be used without any boot drivers.)
- **Notification and Logging**. Notification functions interact with the user by providing information about events and by accepting feedback. Logging functions are typically used for debugging. For more information on notification and logging, see Notification and Logging Code.

© 2011 Microsoft. All rights reserved.

Boot Sequence States

Ultimately, the purpose of a boot loader is to take the device from the point at which it turns on to the point at which it starts the OS. To accomplish this task, the CE Boot core code repeatedly calls boot scenario functions depending on the existing state. The boot scenario code then changes the state of the boot loader. This process cycles the boot loader through a series of states. The successful completion of the series of states is a jump to the OS entry point.

The sequence of states that the boot process goes through depends on the OS image type and how the image is to be downloaded to the device. All possible states are listed in Table 1 below. The first state is **BOOT_STATE_POWERON**. The only states that are required are **BOOT_STATE_FAILURE** and **BOOT_STATE_RUN**.

The four most typical scenarios are described below the table.

Table 1: CE Boot States

State	Description
BOOT_STATE_POWERON	The initial state of the boot loader.
BOOT_STATE_FAILURE	A failure has occurred at some point in the boot process. If this state is reached, the device turns off.
BOOT_STATE_CONFIG	The boot loader reads the saved or default boot loader configuration from persistent storage, and the user can choose configuration options from a boot menu.
BOOT_STATE_DOWNLOAD	The boot loader downloads the image into RAM or persistent storage.
BOOT_STATE_PRELOAD	The boot loader determines if it will boot the Update Loader (ULDR), which is a tiny kernel image that can be started to update the firmware of the device.
BOOT_STATE_LOAD_ULDR	The ULDR updates the image in memory.
BOOT_STATE_LOAD_OS	The boot loader loads the image from persistent storage into memory.
BOOT_STATE_RUN	The boot loader places the OS into memory. When the OS starts, the boot process is finished. This is the final boot loader state.

© 2011 Microsoft. All rights reserved.

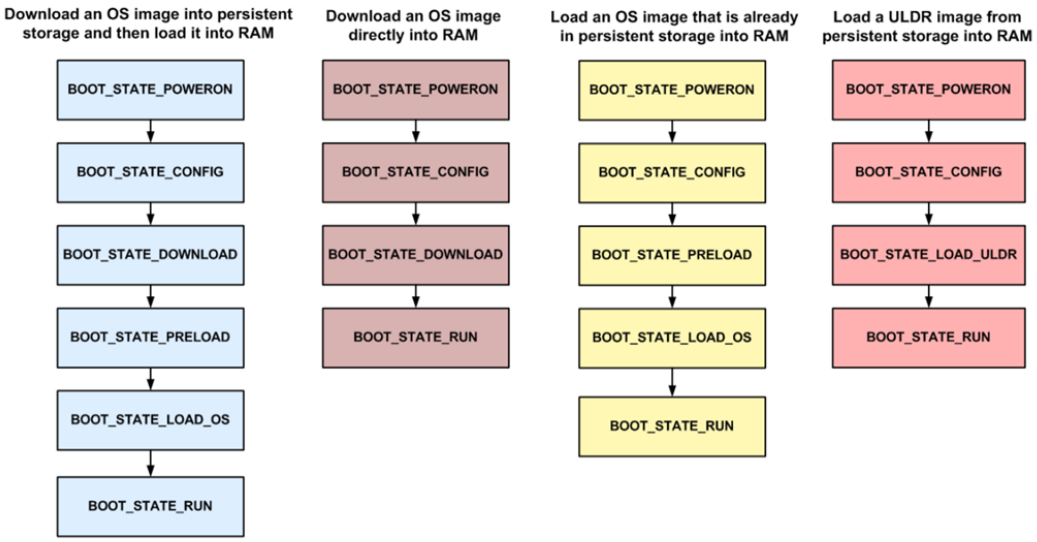
The boot loader can place a Windows Embedded Compact image into memory (RAM) and then execute it, or it can first store it on the device in persistent storage and execute it from there. The sequence of states that the boot loader enters during the boot process depends on the OS image type and how the image is to be downloaded to the device.

The four main scenarios are listed below:

- Download the OS image into persistent storage and then into RAM
- Download the OS image directly into RAM
- Load the OS image that is already in persistent storage into RAM
- Load a ULDR image from persistent storage into RAM

The sequence of states in each of these scenarios is shown in Figure 2 below. The scenarios show a successful boot process. Failure at any point results in a state of **BOOT_STATE_FAILURE**.

Figure 2: Boot State Sequences



The sequences of states shown in Figure 2 are explained below.

Download the OS image into persistent storage and then into RAM

1. When the device is started, the boot loader is in **BOOT_STATE_POWERON**.
2. In **BOOT_STATE_CONFIG**, the boot loader loads the saved or default configuration boot settings, and the user can select choices from a boot menu.
3. In **BOOT_STATE_DOWNLOAD**, the boot loader downloads the image into persistent storage, such as flash memory.

© 2011 Microsoft. All rights reserved.

4. In **BOOT_STATE_PRELOAD**, the boot loader determines if it will start the Update Loader (ULDR). In this scenario, it doesn't start it.
5. In **BOOT_STATE_LOAD_OS**, the boot loader loads the image from persistent storage into RAM.
6. Finally, **BOOT_STATE_RUN** signals the successful end to the boot process.

Download the OS image directly into RAM

1. When the device is started, the boot loader is in **BOOT_STATE_POWERON**.
2. In **BOOT_STATE_CONFIG**, the boot loader loads the saved or default configuration boot settings, and the user can select choices from a menu.
3. In **BOOT_STATE_DOWNLOAD**, the boot loader loads the image into RAM.
4. Finally, **BOOT_STATE_RUN** signals the successful end to the boot process.

Load the OS image that is already in persistent storage into RAM

1. When the device is started, the boot loader is in **BOOT_STATE_POWERON**.
2. In **BOOT_STATE_CONFIG**, the boot loader loads the saved or default configuration boot settings, and the user can select choices from a menu.
3. In **BOOT_STATE_PRELOAD**, the boot loader determines if it will start the ULDR. In this scenario, it doesn't start it.
4. In **BOOT_STATE_LOAD_OS**, the boot loader loads the image from persistent storage into RAM.
5. Finally, **BOOT_STATE_RUN** signals the successful end to the boot process.

Load a ULDR image from persistent storage into RAM

1. When the device is started, the boot loader is in **BOOT_STATE_POWERON**.
2. In **BOOT_STATE_CONFIG**, the boot loader loads the saved or default configuration boot settings, and the user can select choices from a menu.
3. In **BOOT_STATE_ULDR**, the boot loader loads the ULDR image from persistent storage into RAM.
4. Finally, **BOOT_STATE_RUN** signals the successful end to the boot process. In this case, the ULDR image is booted instead of the OS image.

© 2011 Microsoft. All rights reserved.

CE Boot Code

CE Boot consists of code that implements the core functionality in addition to code for the boot drivers that you use to create boot scenarios. Windows Embedded Compact 7 provides the CE Boot sample code in both C and C++. In this article, we refer to C++ when possible.

The following are the locations of the CE Boot code:

- **Common code**: %_WINCEROOT%\Platform\Common\Src\Common\Bldr
- **Platform-specific code**: %_WINCEROOT%\Platform\<*BSP Name*>\Src\Boot\Bldr

(Note that %_WINCEROOT% represents the root directory of your Windows Embedded Compact 7 installation.)

Although CE Boot uses many APIs, this article covers only the most important ones. These fall into the following categories:

- **Core**. This code controls boot loader flow, memory mapping, and dynamic memory allocation.
- **Boot Scenario**. This code, which is called by the core code, performs individual tasks needed by the boot process, such as loading the OS into memory.
- **Notification and Logging**. These functions handle interactions with the user and the recording of debug information.
- **Boot Driver**. These classes and methods encapsulate boot driver functionality, such as handling the IOCTLs passed to a driver.
- **Boot Driver Factory**. This function creates and initializes an instance of a boot driver.

Each of these categories is described in the sections below.

Important

The CE Boot framework functions and other code elements described in this article are only a subset of those that are available. You can explore the CE Boot source code provided with Windows Embedded Compact 7 for the full set of APIs.

Core Code

The core code is divided into the following categories, which are described in the sections that follow:

- **Boot Loader Context**. This global structure contains a set of variables that are associated with the instance of the boot loader.
- **Flow**. These functions execute the boot process.
- **Memory Allocation**. These functions handle memory allocation on the heap.
- **Memory Mapping**. These functions map physical memory addresses to virtual memory addresses, and vice versa.

© 2011 Microsoft. All rights reserved.

Boot Loader Context

The boot loader context stores the variables that are needed to perform the boot process. The **BootLoader_t** structure, which is used by the BSPs that are based on the x86 CPU architecture, is a typical example. Boot drivers can update the variables in this structure. For example, they can update the display resolution by changing the values of the display variables.

```
typedef struct BootLoader_t {
    enum_t   driveId;
    uint32_t ramTop;
    uint32_t videoRam;
    bool_t   ramestimated;
    uint8_t  eddBiosMajor;
    uint8_t  eddBiosMinor;
    uint16_t eddBiosIfcs;
    uint8_t  apmBiosMajor;
    uint8_t  apmBiosMinor;
    uint8_t  apmBiosFlags;
    uint8_t  pciBiosMajor;
    uint8_t  pciBiosMinor;
    uint8_t  pciLastBus;
    uint8_t  pciBusMode;
    uint8_t  vbeBiosMajor;
    uint8_t  vbeBiosMinor;
    size_t   devices;
    const Device_t *pDevices;
    uint32_t runAddress;
    uint32_t ipAddress;
    flags_t  kitlFlags;
    uint32_t KitlTransportMediaType;
    DeviceLocation_t bootDevice;
    DeviceLocation_t kitlDevice;
    size_t  displayWidth;
    size_t  displayHeight;
    size_t  displayBpp;
    size_t  displayLogicalWidth;
```

© 2011 Microsoft. All rights reserved.

```
    size_t  displayLogicalHeight;
    uint8_t  comPort;
    uint8_t  baudDivisor;
    flags_t  imageUpdateFlags;
    bool_t formatUserStore;
} BootLoader_t;
```

Table 2, below, describes the members of the **BootLoader_t** structure.

Table 2: BootLoader_t Structure Members

Name	Type	Description
driveId	*enum_t*	Boot driver basic input/output system (BIOS) ID
ramTop	*uint32_t*	The highest memory address of the available RAM, in bytes
videoRam	*uint32_t*	Amount of video RAM, in bytes
ramestimated	*bool_t*	If true, the RAM value was not retrieved from the BIOS
eddBiosMajor	*uint8_t*	Major version number of the BIOS Enhanced Disk Device (EDD) services supported
eddBiosMinor	*uint8_t*	Minor version number of the BIOS EDD services supported
eddBiosIfcs	*uint16_t*	Bitmap of BIOS EDD interface support
apmBiosMajor	*uint8_t*	Major version number of the BIOS Advanced Power Management (APM) supported
apmBiosMinor	*uint8_t*	Minor version number of the BIOS APM supported
apmBiosFlags	*uint8_t*	BIOS APM flags
pciBiosMajor	*uint8_t*	Major version number of the BIOS Peripheral Component

© 2011 Microsoft. All rights reserved.

Name	Type	Description
		Interconnect (PCI) extension supported
pciBiosMinor	*uint8_t*	Minor version number of the BIOS PCI extension supported
pciLastBus	*uint8_t*	Last PCI bus on the device
pciBusMode	*uint8_t*	Special cycle and configuration flags
vbeBiosMajor	*uint8_t*	Major version number of the BIOS Video Electronics Standards Association (VESA) supported
vbeBiosMinor	*uint8_t*	Minor version number of the BIOS VESA supported
devices	*size_t*	Number of supported devices
pDevices	*const Device_t **	Pointer to supported devices
runAddress	*uint32_t*	Memory address of the OS image
ipAddress	*uint32_t*	The IP address of the device
kitlFlags	*flags_t*	Flags indicating KITL settings, such as whether KITL is in poll mode or interrupt mode
KitlTransportMediaType	*uint32_t*	Media type received from Platform Builder
bootDevice	*DeviceLocation_t*	Location of boot device
kitlDevice	*DeviceLocation_t*	Location of KITL device
displayWidth	*size_t*	Width of the display
displayHeight	*size_t*	Height of the display
displayBpp	*size_t*	The bits per pixel of the display
displayLogicalWidth	*size_t*	Logical width of the display
displayLogicalHeight	*size_t*	Logical height of the display

© 2011 Microsoft. All rights reserved.

Name	Type	Description
comPort	*uint8_t*	Debug port
baudDivisor	*uint8_t*	Baud divisor for the debug port
imageUpdateFlags	*flags_t*	Update mode
formatUserStore	*bool_t*	If true, the user store needs to be formatted

Flow

The core functions listed below control the execution of the boot loader from the time the device powers up to the time control jumps to the entry point of the OS:

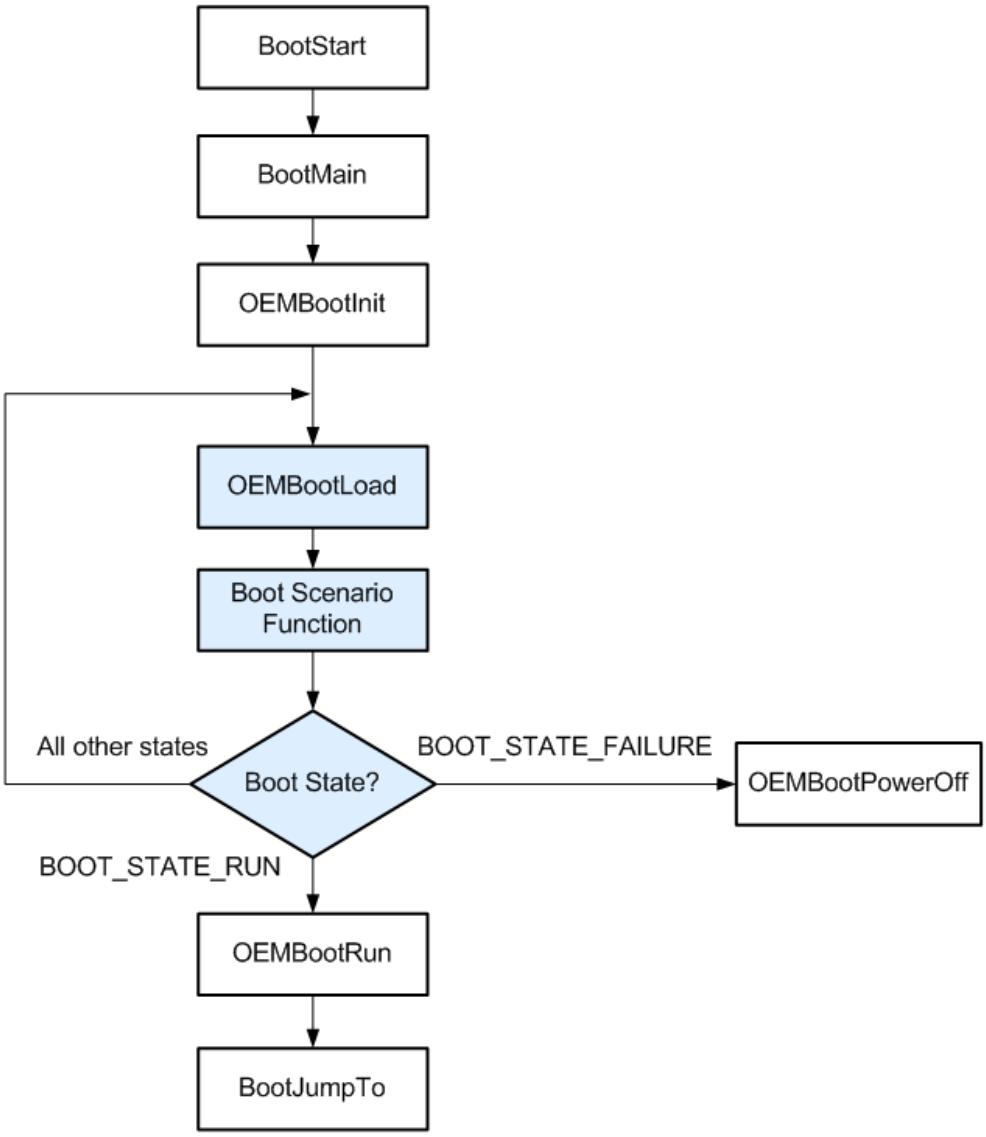
- **BootStart**
- **BootMain**
- **OEMBootInit**
- **OEMBootLoad**
- **OEMBootRun**
- **OEMBootPowerOff**
- **BootJumpTo**

Figure 3 below shows a simplified version of the order in which the core flow functions are called. (See the text below the figure for which function actually calls each subsequent function.) The code that drives the state machine is shaded blue. **OEMBootLoad** is called repeatedly until a boot scenario function returns **BOOT_STATE_RUN** or **BOOT_STATE_FAILURE**.

The figure shows only the main function calls; see the source code for all function calls. In Figure 3, the core function call sequence starts when the device powers up and ends when the boot loader passes control to the OS.

© 2011 Microsoft. All rights reserved.

Figure 3: Flow of the Boot Process



The flow of the boot process as shown in Figure 3 is described below.

1. The entry point of CE Boot is the assembly code **BootStart** function (the name may vary depending on the platform), which initializes a minimum of hardware.
2. **BootStart** calls the **BootMain** function, which calls the rest of the core functions, as described below. **BootMain** does not return because upon successful boot-up, the OS is started, and on unsuccessful boot-up, the device is powered off or reset.

© 2011 Microsoft. All rights reserved.

3. **BootMain** initializes the heap by calling the **BootHeapInit** function (not shown), and then calls the **OEMBootInit** function. **OEMBootInit** finishes initializing the hardware that is used by the boot loader and the global variables that store the parameters for this instance of the boot loader.
4. **BootMain** calls the **OEMBootLoad** function, passing in a pointer to the boot loader's context and the state of the device. **OEMBootLoad** calls the boot scenario functions, which move the boot loader through a series of states. (See Boot Scenario Code for the boot scenario functions and Boot Sequence States for a discussion of the states.) **BootMain** calls **OEMBootLoad** repeatedly until a boot scenario function returns **BOOT_STATE_RUN** or **BOOT_STATE_FAILURE**.
5. If a boot scenario function returns **BOOT_STATE_FAILURE**, the device powers off. If a boot scenario function returns **BOOT_STATE_RUN**, **BootMain** calls the **OEMBootRun** function. **OEMBootRun** prepares the device's information to be passed to the OEM Adaptation Layer (OAL) and returns the physical starting address of the image's entry point.
6. Finally, **BootMain** calls the **BootJumpTo** function, which jumps to the physical address of the entry point to the OS.

These functions are described individually in the sections below.

BootStart Function

BootStart is the entry point to the boot loader. It is platform-dependent and is written in assembly language. (The function name and implementation file name may vary.) This function initializes the cache and the memory management unit (MMU) (optional) and then jumps to **BootMain**.

Its syntax is:

```
void BootStart( );
```

This function has no input parameters and does not return a value.

BootMain Function

The main purpose of **BootMain** is to call the other core flow functions, as shown in Figure 3 above. **BootMain** drives the state machine by calling **OEMBootLoad** until **OEMBootLoad** returns with a state of **BOOT_STATE_RUN**, signaling that the OS can be started.

This function is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\Common\Main.c

Its syntax is:

```
void BootMain( );
```

This function has no input parameters and does not return a value.

OEMBootInit Function

OEMBootInit initializes the boot loader context structure, **BootLoader_t**. For information about the variables in this structure, see Boot Loader Context.

This function is implemented in:

© 2011 Microsoft. All rights reserved.

%_WINCEROOT%\Platform\<*BSP Name*>\Src\Boot\Bldr\Init.c

Its syntax is:

```
void* OEMBootInit ( VOID );
```

This function returns a pointer to the boot loader context structure. If it returns **NULL**, then initialization failed. In that case, **BootMain** calls **OEMBootPowerOff** to power off or reset the device.

OEMBootLoad Function

OEMBootLoad drives the boot loader's state machine by calling boot scenario functions to perform operations that change the state. For example, it can call **BootLoaderConfig** to configure the boot loader, which, if successful, changes the state from **BOOT_STATE_CONFIG** to **BOOT_STATE_DOWNLOAD** if the image is intended to be downloaded to the device. For examples of transitions between boot states, see Figure 2 above, and for other examples of functions that **OEMBootLoad** calls, see Figure 3 above.

BootMain calls **OEMBootLoad** repeatedly until **OEMBootLoad** returns **BOOT_STATE_RUN**, indicating that the OS is ready to be started, or **BOOT_STATE_FAIL**, in which case the device is powered down or reset.

OEMBootLoad is implemented in:

%_WINCEROOT%\Platform\<*BSP Name*>\Src\Boot\Bldr\Main.c

Its syntax is:

```
enum_t OEMBootLoad ( void *pContext, enum_t state );
```

Table 3: OEMBootLoad Function Input Parameters

Name	Type	Description
pContext	*void **	A pointer to the boot loader's **BootLoader_t** structure.
state	*enum_t*	The current boot loader state. See Boot Sequence States for a list of states.

This function returns the current boot loader state.

OEMBootRun Function

This function prepares the boot arguments that pass information to the OEM Adaptation Layer (OAL) of the Windows Embedded Compact 7 OS image and returns the physical start address of the OS image.

OEMBootRun is implemented in:

%_WINCEROOT%\Platform\<*BSP Name*>\Src\Boot\Bldr\Run.c

Its syntax is:

© 2011 Microsoft. All rights reserved.

```
uint32_t OEMBootRun ( void *pContext );
```

Table 4: OEMBootRun Function Input Parameters

Name	Type	Description
pContext	*void **	A pointer to the boot loader's **BootLoader_t** structure

This function returns the physical starting address of the OS image's entry point.

OEMBootPowerOff Function

This function powers down or resets the device when there is an unrecoverable error in the boot process.

OEMBootPowerOff is implemented in:

%_WINCEROOT%\Platform\<*BSP Name*>\ Src\Boot\Bldr\PowerOff.c

Its syntax is:

```
void OEMBootPowerOff ( void *pContext );
```

Table 5: OEMBootPowerOff Function Input Parameters

Name	Type	Description
pContext	*void **	A pointer to the boot loader's **BootLoader_t** structure

This function does not return a value.

BootJumpTo Function

This function, which is specific to the CPU architecture, jumps to the entry point of the OS.

For the x86 CPU architecture, **BootJumpTo** is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\x86_bios\JumpTo.c

Its syntax is:

```
void BootJumpTo ( uint32_t address );
```

Table 6: BootJumpTo Function Input Parameters

Name	Type	Description
address	*uint32_t*	The address of the OS in RAM

© 2011 Microsoft. All rights reserved.

This function does not return a value.

Memory Allocation

The memory allocation functions initialize, allocate, and free memory from the heap.

The memory allocation functions are:

- **BootHeapInit**
- **BootAlloc**
- **BootFree**

These functions are described individually below.

BootHeapInit Function

This function, which is called by **BootMain**, determines the amount of memory available for the heap and sets the global heap pointer, **s_bootHeap**, to the beginning of the heap. If this function exits with **s_bootHeap** equal to **NULL**, then not enough memory was available for the heap.

BootHeapInit is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\Common\Heap.c

The syntax of **BootHeapInit** is:

```
void BootHeapInit( );
```

This function has no input parameters and does not return a value.

BootAlloc Function

This function allocates memory from the heap by looking for a free block of memory large enough to accommodate the amount of memory specified by the *size* parameter. The *size* limit is 512 KB.

The memory is allocated but not cleared.

BootAlloc is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\Common\Heap.c

The syntax of **BootAlloc** is:

```
void* BootAlloc ( size_t size );
```

Table 7: BootAlloc Function Input Parameters

Name	Type	Description
size	*size_t*	The size, in bytes, of the memory block to be allocated

If successful, the return value is a pointer to the allocated memory block.

© 2011 Microsoft. All rights reserved.

If unsuccessful, the return value is **NULL**.

BootFree Function

This function frees memory that was allocated from the heap using **BootAlloc**.

BootFree is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\Common\Heap.c

The syntax of **BootFree** is:

```
void BootFree ( void *pMemory );
```

Table 8: BootFree Function Input Parameters

Name	Type	Description
pMemory	*void **	A pointer to the memory block to be freed

This function does not return a value.

Memory Mapping

The memory mapping functions map physical memory addresses to virtual memory addresses, and vice versa. These functions are:

- **BootPAtoVA**
- **BootVAtoPA**
- **BootImageVAtoPA**

These functions are described individually below.

BootPAtoVA Function

This function maps a physical memory address to a virtual memory address, so the address can be used when the MMU is active (virtual addresses are being used).

BootPAtoVA is typically implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\<*CPU Family*>\Memory.c

Its syntax is:

```
void* BootPAtoVA ( uint32_t pa, bool_t cached );
```

Table 9: BootPAtoVA Function Input Parameters

Name	Type	Description
pa	*uint32_t*	The physical memory address

© 2011 Microsoft. All rights reserved.

Name	Type	Description
cached	*bool_t*	This parameter is not used

This function returns the virtual memory address. In physical mapping mode (that is, the MMU isn't active), it simply returns the passed address.

BootVAtoPA Function

This function maps a virtual memory address to a physical memory address, so it can be used for direct memory access (DMA) transfers.

BootVAtoPA is typically implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\<*CPU Family*>\Memory.c

Its syntax is:

```
uint32_t BootVAtoPA ( void *pAddress );
```

Table 10: BootVAtoPA Function Input Parameters

Name	Type	Description
pAddress	*void **	The virtual memory address

This function returns the physical memory address. In physical mapping mode (that is, the MMU isn't active), it simply returns the passed address.

BootImageVAtoPA Function

Because the downloaded OS image uses virtual addresses, and the memory mapping can differ between the boot loader and the OS, this function maps an OS image's virtual memory address to a physical address. On platforms where the boot loader and the OS use the same memory mapping, this function is equal to **BootVAtoPA**.

BootImageVAtoPA is typically implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Core\<*CPU Family*>\Memory.c

Its syntax is:

```
uint32_t BootImageVAtoPA ( void *pAddress );
```

Table 11: BootImageVAtoPA Function Input Parameters

Name	Type	Description
pAddress	*void **	A virtual memory address used by the OS image

© 2011 Microsoft. All rights reserved.

This function returns the physical memory address.

Boot Scenario Code

The core code calls boot scenario functions to perform individual tasks needed by the boot process. The boot scenario function that is called depends on the boot loader state. The boot scenario function then returns a new state. The process of calling the boot scenario functions repeats until a boot scenario function returns **BOOT_STATE_FAILURE** or **BOOT_STATE_RUN**.

Several of the boot scenario functions create and use boot drivers to do the work needed by the boot process. For an example of the life cycle of a driver that was created by using the boot driver factory function **OEMBootCreateDriver**, see Boot Driver Code.

The boot scenario functions, and their corresponding states, are shown in Table 12 below. The boot scenario functions have only one input parameter, which is a pointer, **pLoader**, to the boot loader context. The return value is the updated boot loader state.

Table 12: Boot Scenario Functions

Boot Scenario Function	Corresponding State	Description
BootLoaderPowerOn	**BOOT_STATE_POWERON**	Simply returns the next boot state (**BOOT_STATE_CONFIG**), although you can add your own functionality instead.
BootLoaderConfig	**BOOT_STATE_CONFIG**	Reads the boot loader configuration information from persistent storage, which the user can configure by selecting options from a boot menu.
BootLoaderDownload	**BOOT_STATE_DOWNLOAD**	Calls other functions to download the image to the device if it is not already in persistent storage.
BootLoaderPreLoad	**BOOT_STATE_PRELOAD**	Determines if the device should boot the ULDR image instead of the OS image.
BootLoaderLoadUldr	**BOOT_STATE_LOAD_ULDR**	Loads the ULDR image into RAM if the ULDR is used.
BootLoaderLoadOs	**BOOT_STATE_LOAD_OS**	Loads the OS image into RAM.

© 2011 Microsoft. All rights reserved.

Notification and Logging Code

You can use the notification and logging functions to monitor and record the progress of the boot process. Inform the user of boot loader events by using the notification code. Use the logging functions only for debugging.

Notification Code

There is one notification function, **OEMBootNotify**, as described below.

OEMBootNotify Function

This function is the primary means of all communication with the user. For example, you can use it to inform the user when the OS image has finished downloading to the device.

OEMBootNotify is implemented in:

%_WINCEROOT%\Platform\<*BSP Name*>\ Src\Boot\Bldr\Notify.c

The syntax of **OEMBootNotify** is:

```
void OEMBootNotify (
   void *pContext,
   enum_t notifyCode,
   void *pNotifyInfo,
   size_t notifyInfoSize
);
```

Table 13: OEMBootNotify Function Input Parameters

Name	Type	Description
pContext	*void **	A pointer to the boot loader's **BootLoader_t** structure
notifyCode	*enum_t*	The event that has occurred
pNotifyInfo	*void **	A pointer to information that pertains to the notification
notifyInfoSize	*size_t*	The size of the information that pertains to the notification

This function does not return a value.

Logging Code

The main logging function is **BootLog**, as described below.

© 2011 Microsoft. All rights reserved.

Note

Do not use this function to communicate with the user; use **OEMBootNotify**. Even if you route logging and notification messages to the same physical channel, you should separate this functionality as a good practice. It can be helpful later, for example, when you disable logging in a release build.

BootLog Function

You use this function to produce a formatted string (though it calls other logging functions to actually perform these operations).

BootLog is implemented in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Log\Log.c

BootLog takes a variable number of parameters. Use it as you would use the standard **printf** function.

Its syntax is:

```
void BootLog ( wcstring_t format, ... );
```

Table 14: BootLog Function Input Parameters

Name	Type	Description
format	*wcstring_t*	A string. As with **printf**, use % characters to specify the formatting of the output.

This function does not return a value.

Boot Driver and Boot Driver Factory Code

CE Boot encapsulates hardware-specific boot driver implementations so that it can use drivers of different types through the same interface. All boot drivers are created by a boot driver factory function (**OEMBootCreateDriver**) and are then accessed through only two functions: an I/O control code (IOCTL) function (**BootDriverIoCtl**) and a deinitialization function (**BootDriverDeinit**).

To provide a consistent interface, these two functions are associated with their hardware-specific implementations through a look-up table. This look-up table is a structure of type **BootDriverVTable_t**, which contains a pointer to the deinitialization function (**pfnDeinit**) and a pointer to the IOCTL function (**pfnIoCtl**). Because each boot driver is associated with a **BootDriverVTable_t** structure, boot loader functions can simply call the generic IOCTL and deinitialization functions, which dereference these pointers. Each driver's header file defines which functions these pointers are associated with. For an example of this translation, see Figure 4 in the section Boot Driver Code.

© 2011 Microsoft. All rights reserved.

Boot Driver Factory Code

The boot driver factory's only responsibility is to create boot drivers that interact with the boot loader by using a consistent interface; therefore, the boot driver factory's only function is **OEMBootCreateDriver**, as described below.

OEMBootCreateDriver Function

This function creates a boot driver and returns a handle to it.

OEMBootCreateDriver is implemented in:

%_WINCEROOT%\Platform\<*BSP Name*>\ Src\Boot\Bldr\Drivers.c

Its syntax is:

```
handle_t OEMBootCreateDriver (
   void *pContext,
   enum_t classId,
   enum_t index
);
```

Table 15: OEMBootCreateDriver Function Input Parameters

Name	Type	Description
pContext	*void **	A pointer to the boot loader's **BootLoader_t** structure.
classId	*enum_t*	The driver class. In the sample CE Boot implementation in Windows Embedded Compact 7, the driver class possibilities are: **BOOT_DRIVER_CLASS_TERMINAL**, **BOOT_DRIVER_CLASS_TRANSPORT**, **BOOT_DRIVER_CLASS_DOWNLOAD**, **BOOT_DRIVER_CLASS_DISPLAY**, **BOOT_DRIVER_CLASS_BLOCK**, and **BOOT_DRIVER_CLASS_FILESYSTEM**. For more information about these driver classes, see the section Boot Driver Examples.
index	*enum_t*	The instance of this boot driver type. You can use this value for reference-counting. In the sample CE Boot implementation in Windows Embedded Compact 7, this value must be 0 (zero).

© 2011 Microsoft. All rights reserved.

The return value is a handle to the driver instance.

Boot Driver Code

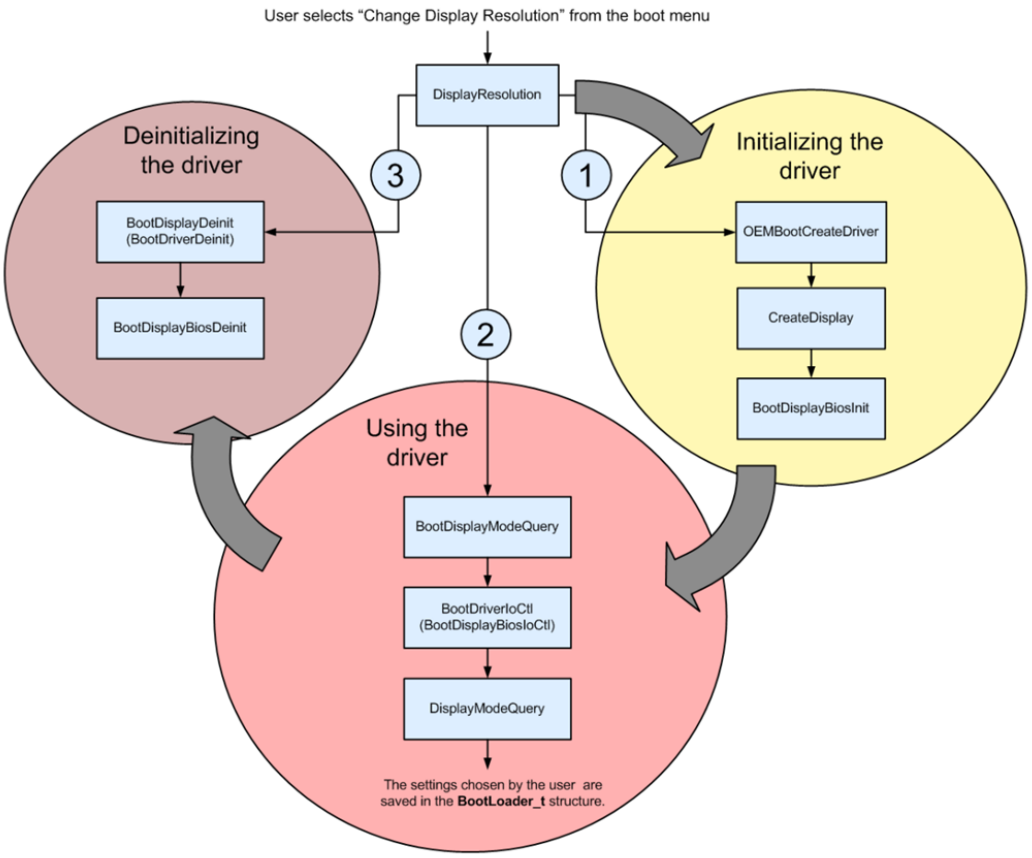
In CE Boot, boot drivers use an interface similar to that of a Windows Embedded Compact stream driver. They are similar in that after a boot driver is created, it is accessible to the boot loader only by using a set of IOCTLs for the particular capabilities of the driver's device. After the boot loader is finished with the boot driver, it closes the driver by using a deinitialization function. The common IOCTL and deinitialization functions call hardware-specific boot driver code (see Boot Driver Examples for more information).

A boot driver must be implemented as a singleton. It may be created multiple times, but each call returns a handle to the same instance of the driver. Therefore, we recommend that boot drivers support reference-counting to control their use.

Figure 4 below shows an example of the life cycle of a boot driver. In this example, the user chooses to change the resolution of the display. To do so, the boot loader uses a display driver to query the capabilities of the display device, presents the options to the user, and saves the user's choice to the boot loader context. When finished, it deinitializes the display driver.

© 2011 Microsoft. All rights reserved.

Figure 4: Example of a Boot Device Driver Life Cycle



In Figure 4, the user selects **Change Display Resolution** from the boot configuration menu. This selection triggers a call to the **DisplayResolution** function, which sets into motion the following sequence of function calls:

1. **Initializing the driver**. The boot display driver is created and initialized during the steps below.
   a. **DisplayResolution** calls **OEMBootCreateDriver**, which is the boot driver factory. The input parameters are a handle to the boot loader context, the class ID (**BOOT_DRIVER_CLASS_DISPLAY**), and an index of 0 (zero) because it is the first driver of this class to be created.
   b. Because the class ID is **BOOT_DRIVER_CLASS_DISPLAY**, **OEMBootCreateDriver** calls the **CreateDisplay** function with a handle to the boot loader context and the index (0).
   c. **CreateDisplay** calls the **BootDisplayBiosInit** function with no parameters. **BootDisplayBiosInit** initializes the boot display driver and returns a handle to it. This handle to the display driver is then returned by both **CreateDisplay** and **OEMBootCreateDriver** back to **DisplayResolution**.

© 2011 Microsoft. All rights reserved.

2. **Using the driver**. Because the user chose to change the display resolution, the following events occur.
   a. **DisplayResolution** makes repeated calls to the **BootDisplayModeQuery** function to query the settings that the display device supports. To do this, **BootDisplayModeQuery** creates a structure to hold display mode settings (such as height and width), and passes the address of that structure in addition to a handle to the display driver and the IOCTL code (**BOOT_DISPLAY_IOCTL_MODE_QUERY**) to the **BootDriverIoCtl** function.
   b. **BootDriverIoCtl** looks up the address of the display driver's IOCTL function in the **BootDriverVTable_t** element of the display driver's handle. This IOCTL function, which is **BootDisplayBiosIoCtl** for the display driver, is then called by **BootDriverIoCtl**, passing a handle to the display driver, the IOCTL, a pointer to the display mode setting structure, and the size of the display mode structure as parameters.
   c. **BootDisplayBiosIoCtl** calls the **DisplayModeQuery** function, with a handle to the display driver and pointers to a number of display settings, which it uses to pass supported display modes to the user in **DisplayResolution**.
   d. After the user chooses the resolution settings, **DisplayResolution** saves the settings to the boot loader's **BootLoader_t** structure, which contains information on different aspects of the boot loader.
3. **Deinitializing the driver**. Because **DisplayResolution** is finished using the handle to the display driver, it releases the driver by performing the following actions.
   a. **DisplayResolution** calls the **BootDisplayDeinit** function with a handle to the display driver. **BootDisplayDeinit** is defined in the boot display code to be the **BootDriverDeinit** function.
   b. **BootDriverDeinit** looks up the address of the display driver's deinitialization function in the **BootDriverVTable_t** element of the display driver's handle. The address that it finds is to the **BootDisplayBiosDeinit** function, which releases any memory that it allocated, and deletes the display driver by clearing its handle from memory.

The common boot driver code is implemented using inline functions in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootdriver.h

The boot driver base class, described below, is:

- **BootDriver_t**

The boot driver functions, described below, are:

- **BootDriverDeinit**
- **BootDriverIoCtl**

BootDriver_t Class

In C++, you implement a boot driver by deriving from the abstract base class **BootDriver_t**. The only two members of **BootDriver_t** are the deinitialization and IOCTL functions, which must be overridden.

The syntax of **BootDriver_t** is:

© 2011 Microsoft. All rights reserved.

```
class __declspec(novtable) BootDriver_t
{
   public:
   virtual bool_t __cdecl DeInit() = 0;
   virtual bool_t __cdecl IoCtl( enum_t code, void *pBuffer, size_t size ) = 0;
}
```

Table 16: BootDriver_t Class Members

Name	Type	Description
DeInit	*Function*	Pure virtual deinitialization method.
IoCtl	*Function*	Pure virtual IOCTL method. The *code* parameter is the IOCTL code. The *pBuffer* parameter is a pointer to information regarding the IOCTL. The *size* parameter is the size of the information pointed to by *pBuffer*.

BootDriverIoCtl Function

Depending on its capabilities, each boot driver has a set of IOCTL codes that it can respond to. The boot loader accesses these capabilities by passing the driver's handle and an IOCTL code to the **BootDriverIoCtl** function. **BootDriverIoCtl** then calls the driver's hardware-specific IOCTL function by looking up its address in the driver's **BootDriverVTable_t** structure.

The syntax of **BootDriverIoCtl** is:

```
bool_t BootDriverIoCtl (
      handle_t hDriver,
      enum_t code,
      void *pBuffer,
      size_t size
);
```

© 2011 Microsoft. All rights reserved.

Table 17: BootDriverIoCtl Function Input Parameters

Name	Type	Description
hDriver	*handle_t*	A handle to the driver
code	*enum_t*	The IOCTL code for the operation
pBuffer	*void **	Parameters that are associated with the IOCTL code
size	*size_t*	The size of the parameters that are associated with the IOCTL code

This function returns **TRUE** if the operation was successful; otherwise, it returns **FALSE**.

BootDriverDeinit Function

This function releases any memory that was used by the boot driver and deletes the boot driver's handle by clearing it from memory.

The syntax of **BootDriverDeinit** is:

```
bool_t BootDriverDeinit ( handle_t hDriver );
```

Table 18: BootDriverDeinit Function Input Parameters

Name	Type	Description
hDriver	*handle_t*	A handle to the driver

This function returns **TRUE** if the operation was successful; otherwise, it returns **FALSE**.

Boot Driver Examples

The CE Boot implementation in Windows Embedded Compact 7 includes sample boot drivers. You can create your own driver classes or you can extend these driver classes by adding new IOCTL codes. As discussed previously, the set of IOCTL codes that are associated with a boot driver define its functionality and enable the driver to respond to requests by the boot loader.

The boot driver classes that are provided with Windows Embedded Compact 7 are:

- Boot Block
- Terminal
- Display

© 2011 Microsoft. All rights reserved.

- Download
- File System
- Transport

(In the code, there are also class IDs defined for a keypad, battery, and OEM driver, but they are not fully implemented.)

The boot driver class list is in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootdriverclasses.h

Boot Block

The Boot Block driver is represented by an abstract class (**BootBlock_t**) that derives from **BootDriver_t**. This driver allows the boot loader to access a device's persistent storage inside a logical disk partition and exposes storage that has binary regions identified by a number (starting at 0). Reserved regions are identified by 16-character names, and partitions are identified by type and ordinal number. After the code opens a region or partition, it can use read, write, and erase commands to work with sectors within the region or partition. This operation is analogous to operations on a file that is opened by the CreateFile (http://go.microsoft.com/fwlink/?LinkId=217950) function of Windows Embedded Compact.

The driver class of the Boot Block driver is **BOOT_DRIVER_CLASS_BLOCK**.

The Boot Block driver interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\BootBlock.hpp

The IOCTL codes that a Boot Block driver can respond to are listed in Table 19 below.

Table 19: Boot Block Driver IOCTLs

IOCTL	Description
BOOT_BLOCK_IOCTL_INFO	Gets the following information from the store: Flags indicating whether binary and reserved regions are supported, the sector size, the number of sectors, and how many binary regions, reserved regions, and partitions there are
BOOT_BLOCK_IOCTL_FORMAT	Formats the store
BOOT_BLOCK_IOCTL_LOCK_MODE	Sets the store to a specified lock mode
BOOT_BLOCK_IOCTL_INFO_BINARY	Gets the index and number of sectors in a binary region
BOOT_BLOCK_IOCTL_INFO_RESERVED	Gets the index, name, and number of sectors in a reserved region

© 2011 Microsoft. All rights reserved.

IOCTL	Description
BOOT_BLOCK_IOCTL_INFO_PARTITION	Gets the index, file system type, file system index, and number of sectors in a partition
BOOT_BLOCK_IOCTL_OPEN_BINARY	Opens a binary region on the store and returns a handle to it
BOOT_BLOCK_IOCTL_OPEN_RESERVED	Opens a reserved region on the store and returns a handle to it
BOOT_BLOCK_IOCTL_OPEN_PARTITION	Opens a partition on the store and returns a handle to it
BOOT_BLOCK_IOCTL_READ	Reads data from the store
BOOT_BLOCK_IOCTL_WRITE	Writes data to the store
BOOT_BLOCK_IOCTL_ERASE	Erases data from the store

Driver classes that derive from the Boot Block driver class are listed in Table 20 below. Their interfaces are defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Block\<*Class Name*>

Table 20: BootBlock Driver Derived Function Interface Files

Driver name	Class name	Description
BIOS	**BootBlockBios_t**	Provides access to a device's BIOS
FAL	**BootBlockFal_t**	Provides access to the flash abstraction layer (FAL) of a NAND flash device
Flash	**BootBlockFlash_t**	Provides access to a flash memory device
IDE	**BootBlockIde_t**	Provides access to an Integrated Device Electronics (IDE) device

Terminal

This driver encapsulates functionality to read from and write to a terminal. Its driver class is **BOOT_DRIVER_CLASS_TERMINAL**.

© 2011 Microsoft. All rights reserved.

Its interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootterminal.h

The IOCTLs that a terminal driver can respond to are listed in Table 21 below.

Table 21: Terminal Driver IOCTLs

IOCTL	Description
BOOT_TERMINAL_IOCTL_WRITE_STRING	Writes a string to the terminal
BOOT_TERMINAL_IOCTL_READ_CHAR	Reads a character from the terminal
BOOT_TERMINAL_IOCTL_VPRINTF	Writes a formatted string to the terminal

Display

This driver encapsulates functionality to use a display. Its driver class is **BOOT_DRIVER_CLASS_DISPLAY**.

Its interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootdisplay.hpp

The IOCTL codes that a display driver can respond to are listed in Table 22 below.

Table 22: Display Driver IOCTLs

IOCTL	Description
BOOT_DISPLAY_IOCTL_FILLRECT	Fills a rectangle with a specified color
BOOT_DISPLAY_IOCTL_BLTRECT	Displays a rectangle
BOOT_DISPLAY_IOCTL_SLEEP	Not implemented
BOOT_DISPLAY_IOCTL_AWAKE	Not implemented

Download

This driver encapsulates functionality to download the OS image to the device. Its driver class is **BOOT_DRIVER_CLASS_DOWNLOAD**.

Its interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootdownload.h

The IOCTL codes a download driver can respond to are in Table 23 below.

© 2011 Microsoft. All rights reserved.

Table 23: Download Driver IOCTLs

IOCTL	Description
BOOT_DOWNLOAD_IOCTL_IMAGE_TYPE	Gets the OS image type (for example, whether the image is to be downloaded to persistent storage or to RAM)
BOOT_DOWNLOAD_IOCTL_GET_DATA	Downloads the data into storage or to RAM
BOOT_DOWNLOAD_IOCTL_RAM_INFO	Gets information about the RAM on the device, such as its starting address and size
BOOT_DOWNLOAD_IOCTL_RAM_OFFSET	Finds the RAM offset
BOOT_DOWNLOAD_IOCTL_STORE_INFO	Gets the persistent storage information, such as sector size, whether the image is to be downloaded into storage
BOOT_DOWNLOAD_IOCTL_GET_OS_INFO	Passes the request for OS information to the transport

File System

This driver encapsulates functionality to read and write from a file. Its driver class is **BOOT_DRIVER_CLASS_FILESYSTEM**.

Its interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Bootfilesystem.hpp

The IOCTL codes that a file system driver can respond to are listed in Table 24 below.

Table 24: File System Driver IOCTLs

IOCTL	Description
BOOT_FILESYSTEM_IOCTL_OPEN	Opens a file
BOOT_FILESYSTEM_IOCTL_DELETE	Deletes a file
BOOT_FILESYSTEM_IOCTL_FREE_SPACE	Determines how much free space is on the disk
BOOT_FILESYSTEM_IOCTL_READ	Reads a file
BOOT_FILESYSTEM_IOCTL_WRITE	Writes to a file
BOOT_FILESYSTEM_IOCTL_SEEK	Seeks within a file
BOOT_FILESYSTEM_IOCTL_GET_SIZE	Gets the size of a file
BOOT_FILESYSTEM_IOCTL_FLUSH	Flushes the buffer

© 2011 Microsoft. All rights reserved.

IOCTL	Description
BOOT_FILESYSTEM_IOCTL_GET_POSITION	Gets the position, within the file, at which the file is being accessed

Transport

This driver enables the boot loader to use a transport (such as a serial or Ethernet connection) to download the OS image to the device and to use KITL. Its driver class is **BOOT_DRIVER_CLASS_TRANSPORT**.

Its interface is defined in:

%_WINCEROOT%\Platform\Common\Src\Common\Bldr\Inc\Boottransport.h

The IOCTL codes that a transport driver can respond to are listed in Table 25 below.

Table 25: Transport Driver IOCTLs

IOCTL	Description
BOOT_TRANSPORT_IOCTL_READ	Reads data over the transport
BOOT_TRANSPORT_IOCTL_GET_OS_CONFIG_INFO	Gets the OS configuration information, such as the KITL media type received from Platform Builder

Conclusion

When you run a Windows Embedded Compact 7 operating system, you typically use a boot loader to load the OS image into memory and jump to the first instruction. Windows Embedded Compact 7 includes two boot loaders with its sample BSPs. CE Boot, which is discussed in this paper, is new to Windows Embedded Compact 7. It is based on the CE Boot framework and is used by the x86 sample BSPs (CEPC, Virtual CEPC, and eBox).

The CE Boot framework consists of core code, which controls the flow of execution, memory mapping, and memory allocation; boot drivers, which provide access to hardware devices; a boot driver factory, which creates boot drivers; notification and logging functions, which inform of boot events; and boot scenarios, which are a particular combination of drivers.

The architecture of CE Boot enables you to reuse code by separating the non-hardware-specific core functions from the hardware-specific boot driver code. By encapsulating boot driver implementation details and only allowing drivers to be accessed through a well-defined, common interface, the boot loader does not need to handle the hardware-specific details in order to use the drivers. CE Boot is also

© 2011 Microsoft. All rights reserved.

extensible because you can create different boot scenarios by assembling particular combinations of drivers. Another capability of CE Boot is that you can use memory on the heap, so you only allocate the memory that you need.

Although you can create your own boot drivers, the Windows Embedded Compact 7 implementation of CE Boot provides sample boot drivers, including drivers for a boot block (access to persistent storage), terminal, display, download, file system, and transport. You can use the sample drivers, or you can extend them with your own functionality. With these options, you can create various boot scenarios to meet different design goals.

Additional Resources

- The Basics of Bringing up a Hardware Platform (http://go.microsoft.com/fwlink/?LinkID=205801)
- Windows Embedded Compact website (http://go.microsoft.com/fwlink/?LinkId=210130)

© 2011 Microsoft. All rights reserved.

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.

© 2011 Microsoft. All rights reserved.