



The Basics of Bringing up a Hardware Platform

Writers: Wendy Giberson and Jina Chan

Technical Reviewer: Travis Hobrla

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

Helps you implement a board support package (BSP) for your platform.

- Prepare BSP components such as the boot loader, the OEM adaptation layer (OAL), the kernel independent transport layer (KITL), and configuration files to create a run-time image for your hardware platform
- Clone and adapt an existing BSP to create a new BSP
- Configure binary image builder (.bib) files to describe your OS image and memory layout
- Customize OAL startup functions for the kernel startup sequence
- Initialize timers, configure power management, handle interrupts, and manage caches
- Configure serial debugging and enable KITL
- Download your OS image to your hardware platform for testing and debugging

This white paper assumes familiarity with Platform Builder 7.

Contents

| | |
|--|----|
| Introduction | 4 |
| PART 1: BSP Component Overview | 5 |
| BSP Components | 5 |
| OAL..... | 5 |
| PQOAL..... | 6 |
| Memory Layout..... | 6 |
| Power Management | 6 |
| Timers..... | 7 |
| Interrupts..... | 7 |
| Boot Loader | 7 |
| Boot Loader Binary Images | 8 |
| Boot Arguments..... | 8 |
| Configuration Files..... | 9 |
| .Bib Files | 9 |
| .Reg Files..... | 9 |
| Device Drivers | 9 |
| Kernel Independent Transport Layer | 10 |
| Run-Time Image | 10 |
| PQOAL Code Organization | 11 |
| BSP Directory Organization | 11 |
| PART 2: BSP Code Overview..... | 12 |
| Boot Loader | 13 |
| Startup..... | 13 |
| Memory Layout..... | 15 |
| Serial Debugging (Boot Loader) | 16 |
| Download Transport..... | 16 |
| Flash Memory..... | 17 |
| OAL | 19 |
| Startup..... | 20 |
| Memory Layout..... | 22 |
| Real-Time Clock | 22 |
| Cache..... | 23 |
| Timer..... | 24 |
| Power Management..... | 25 |
| Serial Debugging (OAL)..... | 28 |
| Interrupts | 28 |

| | |
|---|----|
| Module Inclusion..... | 29 |
| KITL..... | 29 |
| Interrupt and Polling KITL | 31 |
| Active and Passive KITL | 32 |
| PART 3: BSP Development Overview | 32 |
| Requirements | 33 |
| Step 1 Selecting an Existing BSP | 33 |
| Using a BSP Provided by Microsoft..... | 34 |
| Importing a BSP from a Third Party..... | 34 |
| Using a BSP Migrated from an Earlier Version of Windows Embedded Compact..... | 35 |
| Step 2 Cloning a BSP | 35 |
| Step 3 Adapting a BSP..... | 36 |
| Adapting the Boot Loader | 36 |
| Startup | 37 |
| Memory Layout..... | 37 |
| Serial Debugging (Boot Loader)..... | 37 |
| Download Transport | 38 |
| Flash Memory..... | 38 |
| Adapting the OAL | 38 |
| Startup | 39 |
| Memory Layout..... | 39 |
| Real-Time Clock..... | 39 |
| Cache..... | 40 |
| Timer..... | 40 |
| Power Management | 41 |
| Serial Debugging (OAL)..... | 41 |
| Interrupts..... | 41 |
| Module Inclusion | 42 |
| Enabling KITL..... | 43 |
| Adapting Device Drivers..... | 43 |
| Step 4 Testing a BSP..... | 43 |
| Step 5 Designing and Building an OS..... | 44 |
| Step 6 Starting an OS | 44 |
| Connecting the Development Computer to the Device..... | 44 |
| Downloading the Image onto the Device | 45 |
| Conclusion | 46 |
| Additional Resources | 46 |

Introduction

For Windows Embedded Compact to run on a specific hardware platform, it needs a board support package (BSP) that is customized to the target device. Whereas the OS kernel is generic to a CPU architecture (x86, ARM, or MIPS), the BSP is closely tied to the specific hardware. In other words, the BSP encompasses all of the code that is hardware-specific. This hardware-specific code is linked to the CPU-generic kernel code to become part of the run-time image.

You can develop a BSP on your own or start from an existing BSP, such as one included with Platform Builder or provided by a third party. Starting from an existing BSP, whether from Platform Builder or a third party, can significantly reduce the amount of development work required because you can take advantage of existing code that is common to a CPU architecture, a specific CPU, or a system-on-a-chip (SOC). SOC architectures consolidate the major components of a computing device onto a single package of silicon. This consolidation enables smaller, thinner devices while reducing the amount of power required for the device.

Copying an existing BSP is called cloning. After you clone the BSP, you then customize the clone to your device, and use it as a basis for your OS design. Because customizing a cloned BSP is the easiest path to developing a BSP, we recommend that approach in this article. Even if you extensively modify the clone, starting from an existing BSP is still the quickest way to develop one.

A description of the BSP components, how the BSP as a whole fits into the Windows Embedded Compact architecture, and the Windows Embedded Compact startup process provide context for the tasks that you must complete to bring up a hardware platform. In addition, the overview of the source code directory organization will help you find hardware-dependent code that you might need to modify. If you are familiar with this background information, you can skip ahead to the implementation steps.

Because of the wide range of possible hardware configurations and the number of potential differences between your hardware platform and the platform of the existing BSP platform, we can't tell you exactly which BSP code to modify to get the cloned BSP code to work with your device. Instead, we guide you to the functions and configuration files that you might want to start with.

The information in this article is based on the assumption that:

- You have installed and are familiar with using Platform Builder.
- Your hardware platform uses a CPU that is based on the ARM, MIPS, or x86 architecture, the CPU architectures that Windows Embedded Compact 7 supports.

Microsoft provides several types of documentation for Windows Embedded Compact 7. To find additional documentation, see this [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=210130) (<http://go.microsoft.com/fwlink/?LinkId=210130>).

PART 1: BSP Component Overview

Familiarity with the following components will assist you when bringing up your hardware platform.

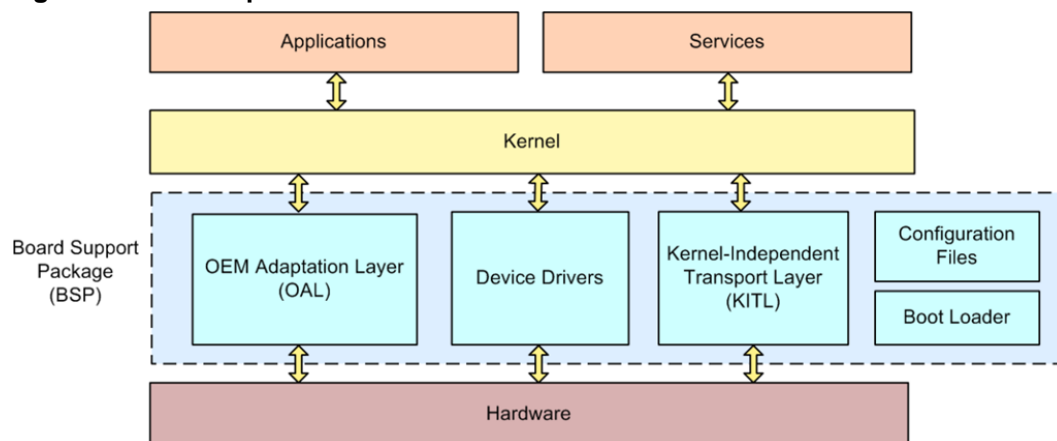
- BSP components
- Contents of the run-time image
- Production-quality OAL (PQOAL) code organization

The following sections contain high-level overviews of these components. For detailed information about these components, see this [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=205390) (<http://go.microsoft.com/fwlink/?LinkId=205390>).

BSP Components

The BSP includes the boot loader, OEM Adaptation Layer (OAL), device drivers, configuration files, and the kernel independent transport layer (KITL). The OS features that you can use with your hardware platform are directly proportional to the enabling code that you provide in your BSP. Figure 1 shows the relationship between the BSP components and the kernel.

Figure 1: BSP components



The top layer of the system architecture consists of the applications and services. Both of these components interact with the next layer, the kernel. The layer below that is the BSP, containing the OAL, device drivers, KITL, configuration files, and boot loader. Three BSP components, the OAL, the device drivers, and KITL, interact with the kernel and also with the bottom layer, which is the hardware.

OAL

The OAL is the layer of code that adapts the kernel, which is generic to a type of CPU architecture (such as ARM or x86), to the specific CPU, or to a system-on-a-chip (SOC) that is used in your hardware platform. The OAL includes code to support timers, specify memory, clean and flush caches,

get and set the time, and other hardware-dependent operations. Creating an OAL is a complex task in bringing up a hardware platform. In general, the best method is to copy the OAL implementation from a working OS design and modify it to suit your hardware platform.

PQOAL

Production-quality OAL (PQOAL) is a way of organizing the OAL source code into reusable libraries by hardware platform. The OAL libraries are a collection of functional, static libraries that you can assemble, in a modular approach, to create an OAL or boot loader. The libraries use a set of functions and structures that are consistent across all CPU architectures. They are organized in a directory structure, which is explained in the section [PART 2: BSP Code Overview](#), according to the level of hardware specificity required. OAL code may be common to all hardware platforms, specific to a type of architecture (such as ARM), specific to a system on a chip (SOC), or specific to a hardware platform (such as Freescale i.MX27). This organization minimizes confusion when you adapt an existing OAL to your hardware platform, and makes it easier to reuse the code if you are bringing up multiple hardware platforms or revisions. If you are creating a BSP that supports different hardware configurations, create a separate directory for each configuration.

PQOAL implements the core functions for the OAL that interact at the chip level. Board-level operations, such as interrupt request (IRQ) routing or glue-logic interactions, remain in the directory for the hardware platform, but they are simplified and abstracted across all architectures whenever possible.

Memory Layout

The binary image builder (.bib) files configure the OS image and the boot loaders. You use the MEMORY section of the config.bib and eboot.bib files to select the address regions for your kernel and other OS image components, the regions for applications and RAM-based file systems, and reserved regions. For more information, see [Understanding Memory Sections in config.bib, boot.bib, and OEMAddressTable in Windows CE 5.0 and 6.0](#) (<http://go.microsoft.com/fwlink/?LinkId=210131>).

Power Management

A Windows Embedded Compact device can be in one of five power states, which by default are named On, UserIdle, SystemIdle, Suspend, and Off/Reset. A change in the system power state can occur because a timer has expired, an application has requested a power state change, or a hardware event has occurred (such as the user pressing a reset button). The OAL participates in power management by responding to CPU idle and suspend requests, and system power-down and restart requests. The OAL does not change the power states of peripheral devices; changing the states is done by the Power Manager, which interacts with the device drivers to change their power consumption. The Power Manager can identify what power states a device and its peripheral devices are capable of, determine their current power states, request a new state, and validate that the driver can support the transition.

Timers

The OAL can manage time by interacting with the following three timers:

- The real-time clock (RTC) tracks the time of day and the date
- The system timer tracks the number of milliseconds since system startup
- The (optional) high-performance timer is needed for profiling

Interrupts

The OAL implements interrupt handlers and interrupt service routines (ISRs). Windows Embedded Compact 7 can handle interprocessor interrupts (IPIs) to support symmetric multiprocessing (SMP). For more information, see [Windows Embedded Compact 7 Symmetric Multiprocessing Guide](http://go.microsoft.com/fwlink/?LinkID=205448) (<http://go.microsoft.com/fwlink/?LinkID=205448>).

Boot Loader

The boot loader is responsible for initializing a minimum amount of hardware, placing the OS image into memory, and jumping to the OS start routine. It resides in nonvolatile memory on the device and can obtain the run-time OS image in a number of different ways, including loading it over a connection such as Ethernet, a USB, or a serial port. Without a boot loader, transferring a run-time image to your device is a slow manual process. The boot loader is used primarily during BSP development.

After the boot loader stores the OS image on the device, you can remove the boot loader from the final product. At this point, the system reset process handles any initialization and jumps to the OS start routine. Hardware platforms that need to perform tasks such as run-time image updates before startup might include the boot loader in the final product.

A boot loader performs the following mandatory tasks:

- Loads the run-time OS image from a storage device to RAM.
- Jumps to the OS start routine.

A boot loader may also perform the following optional tasks:

- Handles the reset vector.
- Determines and handles the reset reason.
- Initializes the CPU, including I/O pins.
- Initializes the memory management unit (MMU) for use within the boot loader; the MMU must then be disabled before jumping to the OS.
- Initializes other hardware as needed.
- Performs power on self test (POST) to identify the devices present.
- Displays a splash screen.
- Supports a debug menu.
- Reads from a removable drive.
- Calls another boot loader that loads the OS image via Ethernet or another transport.

- Downloads a new boot loader version and safely replaces itself in nonvolatile memory with the new version.

We recommend that you design your boot loader to load both run-time images and boot-loader images, to both RAM and nonvolatile storage, to enable product testing.

The sample BSPs that are provided with Windows Embedded Compact 7 include two sample boot loaders. The original boot loader, based on the BLCOMMON library, is described in this article. It was included with earlier versions of Windows CE and is included in Windows Embedded Compact 7 also. The second boot loader, based on the CE Boot framework, is described in the article [CE Boot Framework](http://go.microsoft.com/fwlink/?LinkId=221315) (<http://go.microsoft.com/fwlink/?LinkId=221315>). The BLCOMMON boot loader is used by the ARM and MIPS sample BSPs. 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, which is described in this article.

Boot Loader Binary Images

After you have developed your boot loader, you will have two binary images: a .bin file and an .nb0 file. The .nb0 file is the boot loader image as it appears in memory on your device, and its size is the maximum amount of memory reserved for the boot loader. You use this file to place the initial boot loader image on the device, using either a built-in monitor program provided by the board manufacturer or with a Joint Test Action Group (JTAG) programmer. After the .nb0 image is stored on the device, it can download and update itself using the smaller .bin file if your boot loader supports that functionality. The .bin file is the most common format for Windows Embedded Compact binary images. This format contains header information that the .nb0 file does not, and it removes the need to pad between records. The .bin file is smaller than the .nb0 file because it does not need to fill the entire memory reserved for the boot loader.

Boot Arguments

The boot loader can pass information to the OS by using the Boot Arguments (sometimes called bootargs) area, which is shared between the boot loader and OAL. Drivers and applications can query the OAL for this information through I/O controls (IOCTLs). You determine the format and contents of the Boot Arguments area. Information typically stored in the Boot Arguments includes:

- Hardware revision number
- Universally unique identifier (UUID) for the device
- Settings for debugging and remote tools
- Network device used for downloading the OS image: base address, interrupt, IP address, subnet mask, and MAC address
- Boot arguments version number, which the OAL checks

The information in Boot Arguments is stored by the boot loader and read by the OAL in the BSP_ARGS structure. To create and use a Boot Arguments area, you need:

- A definition of the structure's contents in an include file shared by the boot loader and the OAL
- A definition of the memory location in boot.bib (see the section [.Bib Files](#) in this article)
- A definition of the memory location in config.bib (see the section [.Bib Files](#) in this article)
- A definition of the memory location in an include file shared by the boot loader and the OAL (typically the args.h file in the %_WINCEROOT%\Platform\<Hardware Platform Name>\Src\Inc directory)
- Source code in the boot loader that fills in the structure's contents and stores it in the shared memory location
- Source code in the OAL that reads the structure's contents from the shared memory location

Configuration Files

Windows Embedded Compact uses a number of configuration and batch files to create a run-time image. Some configuration files contain settings that are specific to your hardware platform; other configuration files determine which of the platform-independent OS components to include in the run-time image. These configuration files are part of the build process, which is described in the article [Windows Embedded Compact 7 Build Process](http://go.microsoft.com/fwlink/?LinkId=209954) (<http://go.microsoft.com/fwlink/?LinkId=209954>).

.Bib Files

The .bib files contain configuration information for images. The platform.bib file determines device drivers and other hardware-specific modules included in the run-time image. The config.bib file defines the memory layout and configuration options for the run-time image, and the boot.bib or eboot.bib file defines the memory layout and configuration options for the boot-loader image. For examples of these files, see the directories %_WINCEROOT%\Platform\BSPTemplate\Files and %_WINCEROOT%\Platform\BSPTemplate\Src\Bootloader\Boot. The Boot Arguments area of memory is defined in the config.bib file, for example:

```
ARGS      800FF000      00001000      RESERVED      ; BOOT ARGS
```

.Reg Files

The .reg files contain registry entries for the run-time image. The platform.reg file contains hardware registry settings for the Windows Embedded device. Each device driver may also have a .reg file. The combination of all .reg files in the build process determines which device drivers load when the run-time image boots. For more information, see the article [Building and Testing Your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210199) (<http://go.microsoft.com/fwlink/?LinkId=210199>).

Device Drivers

A device driver presents hardware functionality to higher-level user processes in a generic way. The device drivers that are required for a specific board depend on the peripherals, the features available on the board, and its intended use. Device drivers are described in detail in the articles [Planning Your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210236) (<http://go.microsoft.com/fwlink/?LinkId=210236>), [Implementing Your Device Driver](#)

(<http://go.microsoft.com/fwlink/?LinkId=210237>), and [Building and Testing Your Device Driver](#) (<http://go.microsoft.com/fwlink/?LinkId=210199>).

Kernel Independent Transport Layer

The kernel independent transport layer (KITL) is a communication link between the development computer and the Windows Embedded Compact device. KITL, which is a transport mechanism used for debugging, is called by the OAL. KITL clients are services that receive messages from KITL. There are three default clients: debug messages (DBGMSG), the kernel debugger (KDBG), and parallel port shell (PPSH). You can also create and register your own clients by calling the function **KITLRegisterClient**. Registering a client assigns it a KITL identifier, which enables it to receive KITL messages.

Run-Time Image

To run Windows Embedded Compact on a device, you ultimately need to produce the run-time OS image, which is a binary file typically named nk.bin or nk.nb0. This run-time image is burned onto a ROM chip or is loaded into RAM by a boot loader. The run-time image contains the components listed in Table 1. For more information about how these files become the run-time OS image, see the article [Windows Embedded Compact 7 Build Process](#) (<http://go.microsoft.com/fwlink/?LinkId=209954>).

Table 1: Contents of the Run-Time Image

| File/Component | Description |
|----------------|--|
| Oal.exe | Created from the BSP source code; becomes nk.exe when the build system adds the OAL program to the kernel image |
| Kernel.dll | CPU architecture-specific file provided with Windows Embedded Compact |
| Device Drivers | Dynamic-link libraries (DLLs) that provide support for peripherals |
| Kitl.dll | (Optional but highly recommended) Programming elements required to support KITL, the transport mechanism used for debugging |
| OS Files | OS components such as core services, applications, and configuration files |

PQOAL Code Organization

The BSP code in Windows Embedded Compact 7 is organized to take advantage of the similarities between different hardware platforms. Whereas OAL is a general term for the abstraction layer between the hardware and the kernel, production-quality OAL (PQOAL) describes the modularization of the OAL code into reusable libraries. For example, there are separate libraries to support cache, interrupts, Ethernet, KITL, and so on. For more information about PQOAL libraries, see [Production-Quality OAL Components](http://go.microsoft.com/fwlink/?LinkId=205391) (<http://go.microsoft.com/fwlink/?LinkId=205391>) on MSDN.

The PQOAL model also separates non-hardware-specific code from code that is unique to CPU architectures and specific hardware platforms. This organization enables code reuse between different hardware platforms, making it easier to maintain.

BSP Directory Organization

The default root directory of Windows Embedded Compact 7, which is typically `c:\WINCE700`, is referred to as `%_WINCEROOT%` in files used by the build process. The `%_WINCEROOT%\Platform` directory contains the BSP code. In compliance with the PQOAL model, the directories are organized into the following groups:

- Non-hardware-specific
- CPU architecture
- Specific CPU
- SOC
- Hardware platform

The Platform directory also contains a directory called `BSPTemplate`, which contains commented BSP stub code and tutorials that can be helpful when you are learning about BSPs.

In this article, we refer you to the source code located in the directories in the table below. The directories are listed from the most common to the most specific. The last column of the table contains an example for a BSP called `3DS_iMX27`, which is a BSP that Windows Embedded Compact 7 provides. The `3DS_iMX27` BSP is for a hardware platform that uses an SOC with an ARM926 CPU.

Table 2: Location of BSP Code

| Code type | Location within the <code>%_WINCEROOT%\Platform</code> directory | Example locations for <code>3DS_iMX27</code> BSP |
|------------------------------------|--|--|
| Code that is not hardware-specific | <code>Common\Src\Common</code> | <code>Common\Src\Common</code> |
| Code common to a CPU architecture | <code>Common\Src<CPU Architecture Name>\Common</code> | <code>Common\Src\ARM\Common</code> |
| Code common to a | <code>Common\Src<CPU Architecture</code> | <code>Common\Src\ARM\ARM926</code> |

| Code type | Location within the %_WINCEROOT%\Platform directory | Example locations for 3DS_iMX27 BSP |
|--|---|-------------------------------------|
| specific CPU | Name>\<CPU Name> | |
| Code common to a specific SOC | Common\Src\SOC\<SOC Name> | Common\Src\SOC\iMX27_MS_v2 |
| Code that is unique to a specific hardware platform. A <Hardware Platform Name> directory exists for each installed BSP. | <Hardware Platform Name>\Src | 3DS_iMX27\Src |

For more information about all of the subdirectories in %_WINCEROOT%\Platform, see [PLATFORM Directory](http://go.microsoft.com/fwlink/?LinkId=205393) (http://go.microsoft.com/fwlink/?LinkId=205393) on MSDN.

PART 2: BSP Code Overview

To customize a BSP to your device, you will need to modify some of the hardware-specific code. The sections below describe the main functions of the BSP. The descriptions include:

- The typical location of the source code. (The exact location of the source code depends on the hardware platform that you are using.) The most typical directories are listed.
- The main functions for each component (such as flash memory, cache, and so on).

Important

Not all functions, configurations, or other supporting files that Windows Embedded Compact requires are listed below. If a function is listed, you may still have to customize the code to your particular implementation, and you will often need to explore lower-level methods that are called by the functions. For a detailed description of each function and configuration file, see the Windows Embedded Compact [reference documentation](http://go.microsoft.com/fwlink/?LinkId=209956) (http://go.microsoft.com/fwlink/?LinkId=209956) on MSDN.

- In some cases, a figure and a description of the calling sequence for a typical scenario. The figures are color-coded to indicate which functions are part of the kernel and which are part of the BSP. This distinction is important because it indicates which functions you need to implement. For example, you do not need to implement any kernel functions because Windows Embedded Compact provides the kernel.

Of the BSP functions, some functions are required and some are optional. (See the [reference documentation](http://go.microsoft.com/fwlink/?LinkId=209956) (<http://go.microsoft.com/fwlink/?LinkId=209956>) on MSDN or the comments in the BSP template code in %_WINCEROOT%\Platform\BSPTemplate for more information.) Many of these functions, both required and optional, are already implemented in the common PQOAL library. You can use this common code, or you can use a mix of common code and your own code by replacing the common library implementations as long as the interface matches that specified in the header files.

After you clone an existing BSP for the hardware platform that most closely matches your device, you may then want to pull in code for specific components from other sample BSPs. To find out if code exists that you can use with a particular component (such as the cache) on your hardware platform, first look at the directory names under %_WINCEROOT%\Platform\Common. The directory names will indicate whether code is common to all BSPs, CPUs within a CPU architecture, CPUs within a CPU family, or a particular SOC.

For example, there is cache code in a common directory for the ARM architecture and also in directories for different CPUs (such as ARM920T and ARM1136), and different SOC's (such as OMAP35xx_TPS659xx). Often, the comments in the code will provide useful information on which hardware the code supports. You can also search the Platform directory for a specific function and see which directory it is implemented in. The directory names imply the level of code commonality.

Boot Loader

For ease of discussion, the boot loader code is divided into the following categories:

- Startup
- Memory layout
- Serial debugging
- Download transport
- Flash memory

Note that "Platform" refers to the %_WINCEROOT%\Platform directory.

Startup

Some of the startup tasks performed by the boot loader include initializing the CPU, clearing the instruction and data caches, clearing the translation look-aside buffers (TLBs), configuring and enabling the RAM controller, clearing the interrupts, and initializing the real-time clock (RTC). For details, see the [Boot Loader](#) section.

All boot loaders can use the **BLCOMMON** library that is provided with Windows Embedded Compact. The library provides much of the boot loader functionality. The **BLCOMMON** code is located in Platform\Common\Src\Boot\Blcommon. The interface is defined in blcommon.h.

Although much boot loader functionality is provided for you, you must implement some hardware-specific functions that interact with the **BLCOMMON** library. The hardware-specific code is typically located in one of the following locations:

- Platform\<BSP Name>\Src\Boot
- Platform\<BSP Name>\Src\Bootloader

To enable basic functionality, you typically modify the functions shown in the table below. In addition to these functions, you modify the initialization functions for these components (**OEMInitDebugSerial**, **OEMPreDownload**, and others) that are listed in the tables in the sections [Serial Debugging \(Boot Loader\)](#) and [Download Transport](#) later in this article.

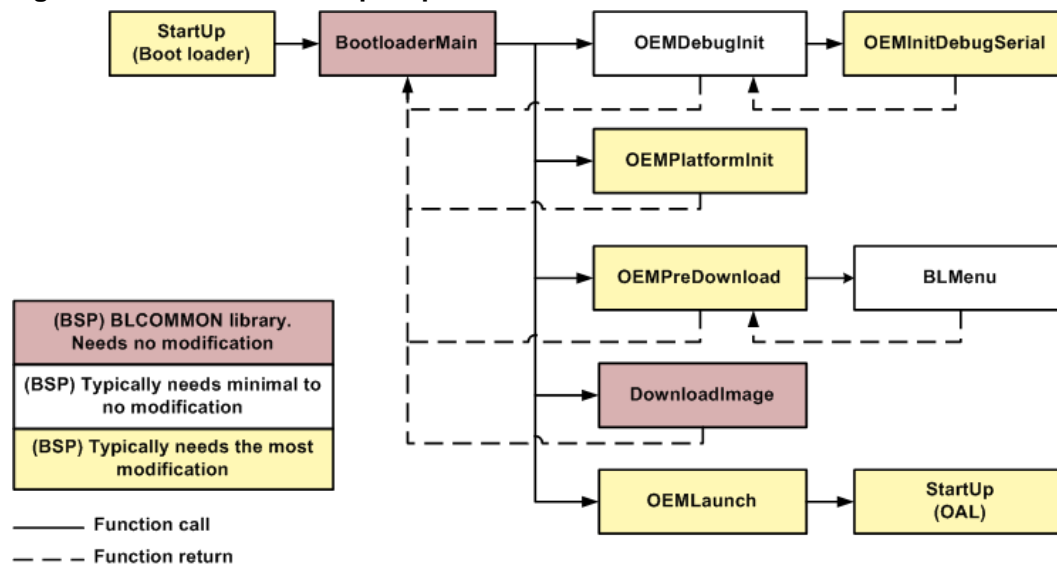
Table 3: Boot Loader Startup Functions

| Purpose | Function name | Typical file name |
|--|--|-------------------|
| Enters the boot loader and initializes the CPU | StartUp (http://go.microsoft.com/fwlink/?LinkId=210098) | startup.s |
| Initializes hardware components | OEMPlatformInit (http://go.microsoft.com/fwlink/?LinkId=210097) | main.c |
| Launches the run-time image | OEMLaunch (http://go.microsoft.com/fwlink/?LinkId=210096) | main.c |

See [Boot Loader Reference](http://go.microsoft.com/fwlink/?LinkId=209957) (http://go.microsoft.com/fwlink/?LinkId=209957) on MSDN for a complete list of boot loader functions.

Figure 2 below shows the function call sequence for the boot loader until it passes control to the OAL.

Figure 2: Boot loader startup sequence



Below is a description of the boot loader startup sequence shown in Figure 2.

1. The **StartUp** function is the entry point to the boot loader executable; it is the first function the boot loader executes. The **StartUp** function, which is typically written in assembly language, performs low-level hardware initialization.
2. The **StartUp** function calls the **BootloaderMain** function, which is implemented in the **BLCOMMON** library, so you do not need to modify it. The **BootloaderMain** function performs further initialization of the hardware and calls several BSP functions, as described below.
 - a. The **BootloaderMain** function calls the **OEMDebugInit** function, which initializes the debug serial port by calling the **OEMInitDebugSerial** function. After the debug serial port is initialized, you can communicate with your device through a serial port connection by using a terminal emulator to display the boot loader's output and accept your input.
 - b. The **BootloaderMain** function then calls the **OEMPlatformInit** function to further initialize hardware such as an Ethernet controller.
 - c. The **BootloaderMain** function calls the **OEMPreDownload** function, which can be customized to prompt for user input, such as to modify the network settings of an Ethernet controller. The **OEMPreDownload** function uses the **BLMenu** function to provide the user interface for the configuration.
 - d. The **BootloaderMain** function then calls the **DownloadImage** function, which is implemented in the **BLCOMMON** library, to download the run-time image into RAM on the device. See the [Flash Memory](#) section for an example of the functions that are called when the boot loader downloads the run-time image into flash memory.
 - e. Finally, the **BootloaderMain** function calls the **OEMLaunch** function to find and jump to the address of the first instruction of the run-time image. At this point, the boot process ends and the boot loader jumps to the OAL's **Startup** function, which is the entry point of the run-time image. That process is described in the OAL [Startup](#) section.

Memory Layout

Windows Embedded Compact supports a 4-GB virtual address space. This address space is divided into areas for the boot loader, the OS, and a place to pass information between the boot loader and the OS.

To define this address space, you will need to modify the boot.bib configuration file (which may also be called eboot.bib, for Ethernet boot loaders, or sbboot.bib, for serial boot loaders). This boot loader BIB file is typically located in:

- Platform\<BSP Name>\Src\Bootloader\Eboot
- Platform\<BSP Name>\Src\Bootloader\Sboot

For a description of the contents of BIB files, see [Binary Image Builder \(.bib\) File](#) (<http://go.microsoft.com/fwlink/?LinkId=209958>).

Serial Debugging (Boot Loader)

The serial debug functions initialize and communicate with a debug message output device. Typically, this device is a Universal Asynchronous Receiver/Transmitter (UART) connected over a null modem cable to a terminal emulator on the host computer. You may be able to use a single implementation of the serial debug functions for the boot loader and the OAL.

The header file that defines the serial debug interface is `Public\Common\OAK\Inc\knintr.h`. The boot loader may use serial debug functions that are implemented in a library shared with the OAL, or the boot loader may use functions that are implemented in its own source code file.

The following are typical serial debug code locations:

- `Platform\Common\Src\Common\Other`
- `Platform\<BSP Name>\Src\Bootloader\Eboot`
- `Platform\<BSP Name>\Src\Boot\Serial`
- `Platform\<BSP Name>\Src\OAL\Oallib`

Table 4 below lists the serial debug functions. For more information, see [Boot Loader Debug Functions](http://go.microsoft.com/fwlink/?LinkId=209959) (<http://go.microsoft.com/fwlink/?LinkId=209959>) on MSDN.

Table 4: Serial Debug Functions

| Purpose | Function name | Typical file name |
|-----------------------------------|---|-------------------|
| Initializes the debug serial port | OEMDebugInit (http://go.microsoft.com/fwlink/?LinkId=209960), OEMInitDebugSerial (http://go.microsoft.com/fwlink/?LinkId=209965) | init.c, debug.c |
| Outputs debug messages | OEMWriteDebugString (http://go.microsoft.com/fwlink/?LinkId=209966), OEMWriteDebugByte (http://go.microsoft.com/fwlink/?LinkId=209967) | debug.c |
| Accepts input to the boot loader | OEMReadDebugByte (http://go.microsoft.com/fwlink/?LinkId=209968) | debug.c |

Download Transport

The download transport copies the run-time image onto the device by using Ethernet, USB, or serial transfer. Most often, an Ethernet connection is used to download the run-time image to the device. You can also use the Ethernet connection to communicate with the development computer for debugging using KITL.

The header file that defines the Ethernet interface is `Public\Common\OAK\Inc\halether.h`. The Ethernet functions are typically linked into `eboot.lib`.

The following are typical Ethernet code locations:

- Platform\<BSP Name>\Src\Bootloader
- Platform\<BSP Name>\Src\Bootloader\Eboot
- Platform\<BSP Name>\Src\Boot\Eboot

There are also common Ethernet libraries that support specific Ethernet cards in Platform\Common\Src\Common\ethdrv.

Table 5 below lists the main download transport and Ethernet functions. For more information, see [Boot Loader Download Functions](http://go.microsoft.com/fwlink/?LinkId=209969) (<http://go.microsoft.com/fwlink/?LinkId=209969>) on MSDN.

Table 5: Download Transport and Ethernet Functions

| Purpose | Function name | Typical file name |
|--|--|------------------------|
| Initializes the download transport (along with other pre-download initializations) | OEMPreDownload (http://go.microsoft.com/fwlink/?LinkId=209971) | main.c |
| Reads data from the download transport | OEMReadData (http://go.microsoft.com/fwlink/?LinkId=209972) | main.c |
| Sends and receives data over an Ethernet connection | OEMEthGetFrame (http://go.microsoft.com/fwlink/?LinkId=209973), OEMEthSendFrame (http://go.microsoft.com/fwlink/?LinkId=209974) | ether.c, eth.c, main.c |

See the following section, [Flash Memory](#), for an example of the function calls that are used when downloading a run-time image into flash memory on a device.

Flash Memory

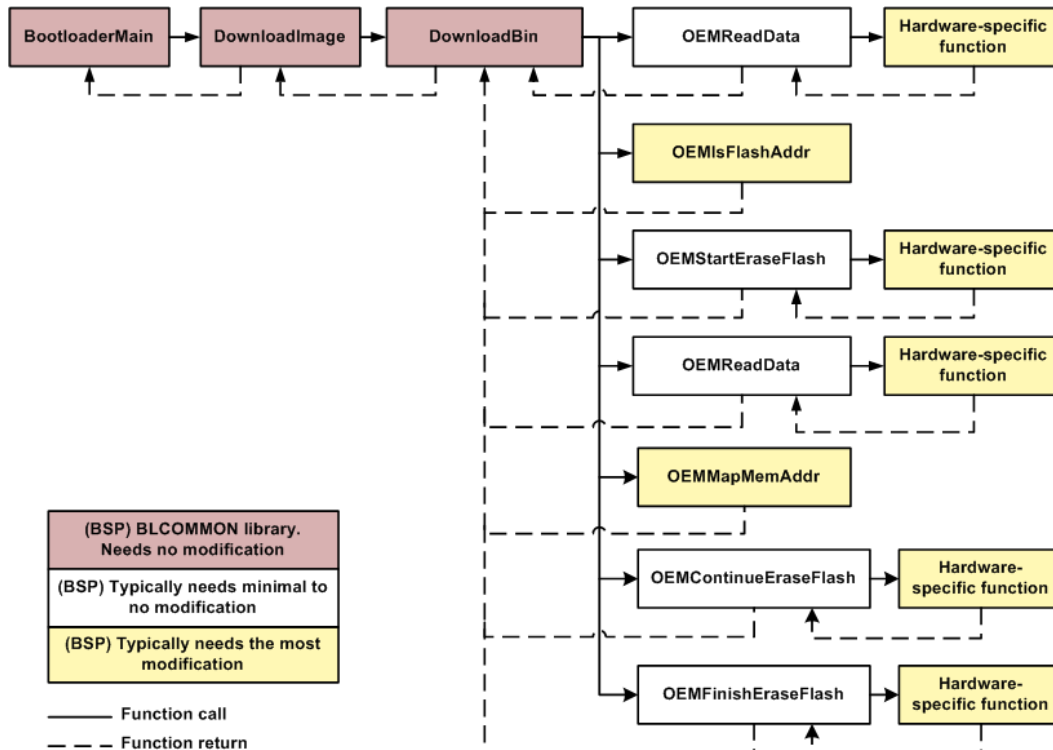
The **BLCOMMON** library provides an infrastructure for downloading the run-time image into flash memory, although you may need to modify some of the functions that the **BLCOMMON** functions call to work with your hardware. The header file that defines the flash memory interface is Public\Common\OAK\Inc\blcommon.h. The flash memory functions are typically located in Platform\<BSP Name>\Src\Bootloader\Eboot and linked into oal_flash.lib or directly into the boot loader executable (such as eboot.exe).

Table 6 below lists the flash memory functions. For more information, see [Boot Loader Flash Functions](http://go.microsoft.com/fwlink/?LinkId=209975) (<http://go.microsoft.com/fwlink/?LinkId=209975>) on MSDN.

Table 6: Flash Memory Functions

| Purpose | Function name | Typical file name |
|--|---|-------------------|
| Determines if an address lies in flash memory | OEMIsFlashAddr (http://go.microsoft.com/fwlink/?LinkId=209976) | flash.c |
| Remaps a flash memory address to a RAM address | OEMMapMemAddr (http://go.microsoft.com/fwlink/?LinkId=209980) | flash.c |
| Erases flash memory | OEMStartEraseFlash (http://go.microsoft.com/fwlink/?LinkId=209981), OEMContinueEraseFlash (http://go.microsoft.com/fwlink/?LinkId=209982), OEMFinishEraseFlash (http://go.microsoft.com/fwlink/?LinkId=209983) | flash.c |
| Writes to flash memory | OEMWriteFlash (http://go.microsoft.com/fwlink/?LinkId=209984) | flash.c |

Figure 3 below shows an example of the function call sequence when you download a run-time image into flash memory.

Figure 3: Downloading a run-time image into flash memory on a device

Below is a description of the process of downloading the OS image into flash memory, as shown in Figure 3.

1. The **BootloaderMain** function calls the **DownloadImage** function, which in turn calls the **DownloadBin** function when the run-time image is in BIN format (or a similar function if the image is in NB0 format). All three functions are implemented by the **BLCOMMON** library and therefore you do not need to modify them. The **DownloadBin** function calls several BSP functions, as described below.
 - a. The **DownloadBin** function calls the **OEMReadData** function to read the run-time image's starting address and length. The **OEMReadData** function typically calls a hardware-specific function to perform the read operation.
 - b. The **DownloadBin** function then calls the **OEMIsFlashAddr** function to check if the run-time image will be downloaded into flash memory.
 - c. If the image is destined for flash memory, the **DownloadBin** function then calls the **OEMStartErase** function so that the flash memory will be erased as the image is downloaded. The **OEMStartErase** function typically calls a hardware-specific function to start erasing the flash memory.
 - d. The **DownloadBin** function then calls **OEMReadData** function to copy the records of the BIN image to flash memory.
 - e. As it reads the data of the BIN image, the **DownloadBin** function calls the **OEMMapMemAddr** function to map the flash memory addresses to a temporary location in RAM, which allows the run-time image to be downloaded while the slower flash memory erase operation takes place.
 - f. The **DownloadBin** function calls **OEMContinueEraseFlash** to continue erasing the flash memory as it copies the data records of the BIN image into it.
 - g. After all of the data records have been copied into flash memory, the **DownloadBin** function calls the **OEMFinishEraseFlash** function, which typically calls a hardware-specific function to finish the flash memory erasure.

OAL

For ease of discussion, the OAL code is divided into the following categories:

- Startup
- Memory layout
- Real-time clock
- Cache
- Timer
- Power management
- Serial debugging
- Interrupts

- Module inclusion



Note

The OAL library is typically named oal.lib. However, you may name it anything you like.

Startup

Some of the startup tasks performed by the OAL include initializing the CPU, jumping to the entry point of kernel.dll, and then initializing other components of the system at the kernel's request. For details about the tasks that the OAL performs, see [OAL](#) in the BSP Component Overview section earlier in this article.

The OAL startup code is located in Platform\<BSP Name>\Src\OAL\Oallib. It is linked into oal.lib.

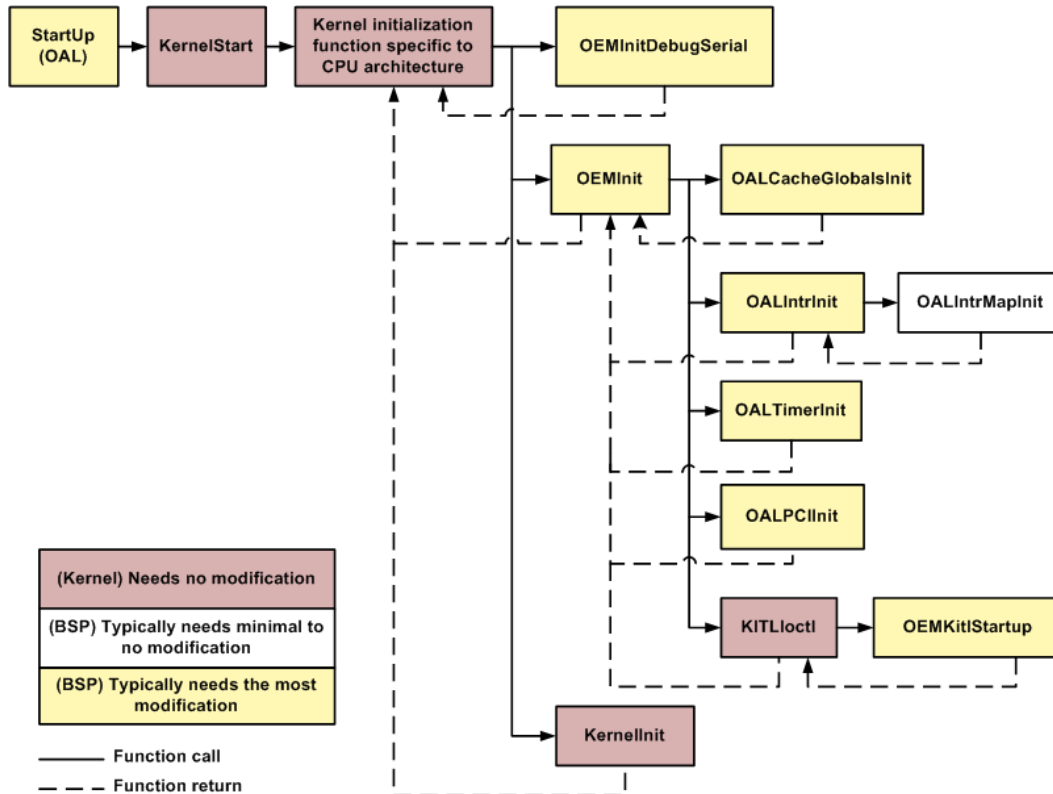
Table 7 below lists the main OAL startup functions; you will need to customize them to your hardware platform. In addition to these functions, initialization functions such as **OALCacheGlobalsInit**, **OALTimerInit**, **OALIntrInit**, and **OEMKitIStartup** are listed in the tables in the following sections: [Cache](#), [Timer](#), [Interrupts](#) and [KITL](#).

Table 7: OAL Startup Functions

| Purpose | Function name | Typical file name |
|---------------------------------|--|-------------------|
| Puts the CPU in a known state | StartUp | startup.s |
| Initializes hardware interfaces | OEMInit (http://go.microsoft.com/fwlink/?LinkId=209987) | init.c |

Figure 4 below shows an example of the system startup sequence beginning where the boot loader left off. The OAL starts the kernel, and then the kernel calls OAL functions to initialize various components.

Figure 4: OAL/kernel startup sequence



Below is a description of the OAL and kernel startup sequence shown in Figure 4.

1. At the end of the boot process, the boot loader jumps to the OAL's **StartUp** function, which is the entry point of the run-time image. (For details on what led up to this point, see the [Boot Loader Startup](#) section.) The **StartUp** function flushes the cache and calls **KernelStart**, which is a CPU architecture-specific function implemented by the kernel.
2. The **KernelStart** function performs minimal initialization, turns on the memory management unit (MMU), initializes the kernel globals, and then jumps to the entry point of kernel.dll. The entry point of kernel.dll is a CPU architecture-specific kernel function that performs the rest of the initialization. The **KernelStart** function never returns.
3. In the kernel initialization function, the kernel exchanges global pointers with the OAL. From this point on, the kernel has access to all functions and variables that are defined by the OAL in the **OEMGLOBAL** structure, and the OAL has access to all functions and variables that are defined by the kernel in the **NKGLOBAL** structure. The kernel initialization function also performs additional CPU setup, such as setting up interrupt vectors and enabling the cache.
4. The kernel initialization function then calls the **OEMInitDebugSerial** function to initialize the debug serial port on the device. The serial port initialization process is similar to the one performed by the boot loader.

5. The kernel initialization function then calls the **OEMInit** function, at which time optional functions in the **OEMGLOBAL** structure are initialized or overridden if applicable. The **OEMInit** function also performs any other hardware-specific initialization, such as initializing additional global variables, initializing a watchdog timer, and initializing coprocessors. To do this initializing, the **OEMInit** function calls several BSP functions, such as those described below.
 - a. The **OEMInit** function calls the **OALCacheGlobalsInit** function to initialize the global variables that hold the cache parameters.
 - b. The **OEMInit** function initializes the interrupt hardware by calling the **OALIntrInit** function, which in turn calls the **OALIntrMapInit** function to initialize the mappings between interrupt requests (IRQs) and SYSINTR logical interrupts.
 - c. The **OEMInit** function calls the **OALTimerInit** function, which initializes the timer state structure.
 - d. The **OEMInit** function calls the **OALPCInit** function, which initializes the PCI bus, if there is one.
 - e. The **OEMInit** function calls the kernel's **KITLloctl** function with the **IOCTL_KITL_STARTUP_IOCTL**. The **KITLloctl** function then calls the **OEMKitlStartup** function. See the [KITL](#) section for the continuation of the KITL startup procedure.
6. After the **OEMInit** function returns, the kernel initialization function calls the **KernelInit** function, which completes the kernel initialization by setting up the heap, initializing threads, and so on. The **KernelInit** function then schedules the first thread, and the OS startup sequence is complete.

Memory Layout

The placement of the run-time image in memory is hardware-dependent and is determined by configuration files and address tables.

The config.bib file determines the memory layout for the run-time image. There is one config.bib file for each BSP. It is located in Platform\<BSP Name>\Files. For a description of the layout of BIB files, see [Binary Image Builder \(.bib\) File](http://go.microsoft.com/fwlink/?LinkId=209958) (<http://go.microsoft.com/fwlink/?LinkId=209958>) on MSDN. The most important entries in the config.bib file are the RAM and ROMIMAGE definitions.

For the ARM and x86 architectures, you will also need to define the physical-to-virtual address mappings in [OEMAddressTable](http://go.microsoft.com/fwlink/?LinkId=209988) (<http://go.microsoft.com/fwlink/?LinkId=209988>). To specify physical RAM beyond 512 MB, you will need to define the [OEMDeviceTable](http://go.microsoft.com/fwlink/?LinkId=209989) (<http://go.microsoft.com/fwlink/?LinkId=209989>) and [OEMRamTable](http://go.microsoft.com/fwlink/?LinkId=209990) (<http://go.microsoft.com/fwlink/?LinkId=209990>).

Real-Time Clock

The real-time clock (RTC) manages time-of-day information. The header file that defines the RTC module interface is Platform\Common\Src\Inc\oal_rtc.h. The RTC functions are typically part of a library with a name similar to oal_rtc_timer.lib, but they are ultimately linked into oal.lib.

Table 8 below lists the main RTC functions. The sample BSP will contain implementations of these functions, but you will probably need to modify them if your RTC isn't the same as the one used in the sample BSP.

For more information about these functions, see [OAL Functions](http://go.microsoft.com/fwlink/?LinkId=209991) (<http://go.microsoft.com/fwlink/?LinkId=209991>).

Table 8: Real-Time-Clock Functions

| Purpose | Function name | Typical file name |
|---------------|--|-------------------|
| Sets the time | OEMSetRealTime (http://go.microsoft.com/fwlink/?LinkId=209992), OEMSetAlarmTime (http://go.microsoft.com/fwlink/?LinkId=209993) | rtc.c |
| Gets the time | OEMGetRealTime (http://go.microsoft.com/fwlink/?LinkId=209994) | rtc.c |

Cache

Windows Embedded Compact stores data in a cache to serve data requests by the kernel and applications more quickly. The header file that defines the OAL cache module interface is Platform\Common\Src\Inc\oal_cache.h. The cache functions are linked into oal.lib.

The cache code is often divided into multiple files, some of which are in assembly language. The following are typical of cache code locations:

- Platform\Common\Src\Common\Cache
- Platform\Common\Src\SOC\<SOC Name>\OAL\Cache
- Platform\Common\Src\<CPU Family>\Common\Cache
- Platform\Common\Src\<CPU Family>\<CPU Name>\Cache

Table 9 below lists the main cache functions; for more information, see the [Cache Reference](http://go.microsoft.com/fwlink/?LinkId=209995) (<http://go.microsoft.com/fwlink/?LinkId=209995>) on MSDN. The common cache code that comes with Windows Embedded Compact contains CPU-family-specific implementations that you may be able to use with little modification if your hardware platform uses a CPU from the same family. For example, there is cache-flushing code in Platform\Common\Src\ARM\ARM926\Cache\flush.c that is suitable for ARM CPUs with separate data and instruction caches.

Table 9: Cache Functions

| Purpose | Function name | Typical file name |
|-----------------------|--|-----------------------------|
| Initializes the cache | OALCacheGlobalsInit (http://go.microsoft.com/fwlink/?LinkId=209997) | init.s, initcache.s, init.c |

| Purpose | Function name | Typical file name |
|---|---|---|
| Gets cache information | OALIoCtlHalGetCacheInfo (http://go.microsoft.com/fwlink/?LinkId=209998) (OEMIoControl (http://go.microsoft.com/fwlink/?LinkId=210039)) | ioctl.c |
| Cleans the cache | OEMCacheRangeFlush (http://go.microsoft.com/fwlink/?LinkId=209999), OALCleanDCache (http://go.microsoft.com/fwlink/?LinkId=210000), OALCleanDCacheLines (http://go.microsoft.com/fwlink/?LinkId=210001) | flush.c, cleandc.s, cleandclines.s |
| Clears the translation look-aside buffers (TLB) | OALClearDTLB (http://go.microsoft.com/fwlink/?LinkId=210004), OALClearITLB (http://go.microsoft.com/fwlink/?LinkId=210005) | cleardtlb.s, clearitlb.s |
| Flushes the cache | OALFlushDCache (http://go.microsoft.com/fwlink/?LinkId=210002), OALFlushDCacheLines (http://go.microsoft.com/fwlink/?LinkId=210003), OALFlushICache (http://go.microsoft.com/fwlink/?LinkId=210006), OALFlushICacheLines (http://go.microsoft.com/fwlink/?LinkId=210007) | flushdc.s, flushdclines.s, flushic.s, flushiclides.s |

Timer

The Windows Embedded Compact scheduler relies on a system timer to provide system ticks, power-saving functionality, and other actions and information related to time. The header file that defines the OAL timer module interface is Platform\Common\Src\Inc\oal_timer.h. The timer functions are linked into oal.lib.

The following are examples of timer code locations:

- Platform\Common\Src\Common\Timer
- Platform\Common\Src\SOC\<SOC Name>\OAL\Timer
- Platform\Common\Src\<CPU Family>\Common\Timer

Table 10 below lists the main timer functions. In addition to these functions, the [Timer Reference](http://go.microsoft.com/fwlink/?LinkId=210008) (<http://go.microsoft.com/fwlink/?LinkId=210008>) on MSDN describes other functions related to timer activity. The common timer code that comes with Windows Embedded Compact contains CPU-family-specific implementations that you may be able to use with little modification. For example,

Platform\Common\Src\MIPS\Common\Timer\Vartick\timer.c provides timer function implementations that are suitable for variable-tick timers in MIPS-based CPUs and SOCs.

Table 10: Timer Functions

| Purpose | Function name | Typical file name |
|--------------------------|--|-------------------------------|
| Initializes timer | OALTimerInit (http://go.microsoft.com/fwlink/?LinkId=210043), OALTimerInitCount (http://go.microsoft.com/fwlink/?LinkId=210044) | timer.c, init.c |
| Handles timer interrupts | OALTimerIntrHandler (http://go.microsoft.com/fwlink/?LinkId=210045) | timer.c |
| Gets timer information | OALTimerGetCount (http://go.microsoft.com/fwlink/?LinkId=210046), OALTimerGetCompare (http://go.microsoft.com/fwlink/?LinkId=210047), OALGetTickCount (http://go.microsoft.com/fwlink/?LinkId=210049) | timer.c, count.s |
| Sets timer | OALTimerSetCompare (http://go.microsoft.com/fwlink/?LinkId=210050), OALTimerUpdate (http://go.microsoft.com/fwlink/?LinkId=210051), OALTimerRecharge (http://go.microsoft.com/fwlink/?LinkId=210052) | timer.c, count.s, cntcmp.c |

Power Management

When hardware or software events generate system calls to change the device's power state, they use the power management functions. The header file that defines the OAL power management module interface is Platform\Common\Src\Inc\oal_power.h. The power management functions are linked into oal.lib.

The following are examples of power management code locations:

- Platform\Common\Src\Common\Power
- Platform\Common\Src\Common\Timer
- Platform\Common\Src\<CPU Family>\Common\Power
- Platform\Common\Src\<CPU Family>\Common\Timer
- Platform\Common\Src\SOC\<SOC Name>\OAL\Power
- Platform\Common\Src\SOC\<SOC Name>\OAL\Timer

- Platform\<BSP Name>\Src\OAL\Oallib

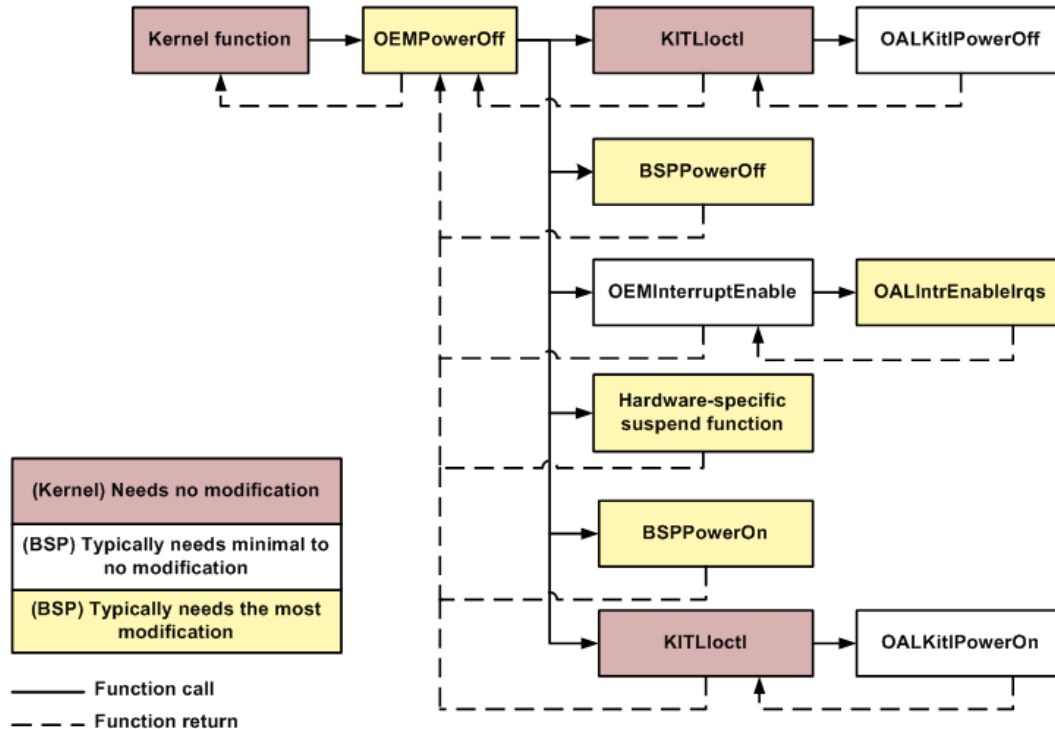
Table 11 below lists the main power management functions. The MSDN topics [OAL Functions](http://go.microsoft.com/fwlink/?LinkId=209991) (<http://go.microsoft.com/fwlink/?LinkId=209991>), [Timer Reference](http://go.microsoft.com/fwlink/?LinkId=210008) (<http://go.microsoft.com/fwlink/?LinkId=210008>), [Power Manager IOCTLs](http://go.microsoft.com/fwlink/?LinkId=210104) (<http://go.microsoft.com/fwlink/?LinkId=210104>), and [SMP Functions](http://go.microsoft.com/fwlink/?LinkId=210103) (<http://go.microsoft.com/fwlink/?LinkId=210103>) list other functions related to power management. You may need to adapt the power management code to your hardware platform.

Table 11: Power Management Functions

| Purpose | Function name | Typical file name |
|-------------------------------|---|----------------------------|
| Implements the CPU idle state | OEMIdle (http://go.microsoft.com/fwlink/?LinkId=210055), OEMIdleEx (http://go.microsoft.com/fwlink/?LinkId=210057), OALCPUIdle (http://go.microsoft.com/fwlink/?LinkId=210058) | power.c, idle.c, timer.c |
| Powers off the CPU | OEMPowerOff (http://go.microsoft.com/fwlink/?LinkId=210063) | power.c, off.c, poweroff.c |
| Enables and disables wake | OALIoCtlHalEnableWake (http://go.microsoft.com/fwlink/?LinkId=210064), OALIoCtlHalDisableWake (http://go.microsoft.com/fwlink/?LinkId=210065) OEMIoControl (http://go.microsoft.com/fwlink/?LinkId=210039) | off.c, ioctl.c |

In Figure 5, the kernel makes a call to the OAL to transition the CPU to the suspend state. An interrupt then triggers the CPU to wake.

Figure 5: Powering off the CPU and then powering it back on



Below is a description of the process of powering the CPU off and then back on, as shown in Figure 5.

1. To power off the device, the kernel calls the **OEMPowerOff** function of the OAL. The **OEMPowerOff** function is responsible for transitioning the CPU to a suspend state and then for powering it back on when signaled. The **OEMPowerOff** function typically calls the functions described below.
 - a. The **OEMPowerOff** function calls the **KITLloctl** function, which is implemented by the kernel in `kitlcore.lib`, with the `IOCTL_KITL_POWER_CALL` IOCTL. The **KITLloctl** function calls the **OALKitlPowerOff** function, which is implemented in the BSP common library `oal_kitl.lib`, to disable KITL.
 - b. The **OEMPowerOff** function then calls **BSPPowerOff**, which prepares for power-off mode on the hardware level.
 - c. Next, the **OEMPowerOff** function calls the **OEMInterruptEnable** function to enable interrupts so that a power-up signal can be received. The **OEMInterruptEnable** function calls the **OALIntrEnableIrqs** function, which is hardware-specific.
 - d. The **OEMPowerOff** function calls a hardware-specific function to put the device in a suspend state. The device stays within the suspend state until an interrupt occurs.
 - e. When the device is notified to emerge from the suspend state, the **OEMPowerOff** function calls the **BSPPowerOn** function to perform hardware-specific power-up operations.

- f. The **OEMPowerOff** function calls the **KITLloctl** function, which in turn calls the **OALKitlPowerOn** function to re-enable KITL.

Serial Debugging (OAL)

To set up the serial debugger, you can often use the boot loader's debug functions with minor modifications. For more information about serial debugging functions, see [Serial Debugging \(Boot Loader\)](#) earlier in this article.

Interrupts

Windows Embedded Compact uses interrupts to respond to events. These events may be internal, such as a timer tick, or external, such as requests by I/O peripherals. The header file that defines the OAL interrupt module interface is Platform\Common\Src\Inc\oal_intr.h. The interrupt functions are linked into oal.lib.

The following are examples of interrupt code locations:

- Platform\Common\Src\Common\Intr
- Platform\Common\Src\SOC\<SOC Name>\OAL\Intr
- Platform\Common\Src\<CPU Family>\Common\Intr
- Platform\<BSP Name>\Src\OAL\Oallib

Table 12 below shows the main interrupt functions. For more information, see the MSDN topics [Interrupt Reference](#) (<http://go.microsoft.com/fwlink/?LinkId=210105>) and [OAL Functions](#) (<http://go.microsoft.com/fwlink/?LinkId=209991>). You may need to adapt the interrupt code to your hardware platform.

Note that in some cases, the OAL and BSP functions have similar names (for example, [OALIntrDisableIrqs](#) (<http://go.microsoft.com/fwlink/?LinkId=210066>) and [BSPIntrDisableIrq](#) (<http://go.microsoft.com/fwlink/?LinkId=210067>)). The name similarity comes about because the BSPIntr* functions are called by the OALIntr* functions if your implementation uses hardware platform callbacks.

Table 12: Interrupt Functions

| Purpose | Function name | Typical file name |
|---------------------|--|-------------------|
| Enables interrupts | OALIntrEnableIrqs (http://go.microsoft.com/fwlink/?LinkId=210068), BSPIntrEnableIrq (http://go.microsoft.com/fwlink/?LinkId=210069) | intr.c |
| Disables interrupts | OALIntrDisableIrqs (http://go.microsoft.com/fwlink/?LinkId=210066), BSPIntrDisableIrq (http://go.microsoft.com/fwlink/?LinkId=210067) | intr.c |

| Purpose | Function name | Typical file name |
|---|---|-------------------|
| Initializes interrupts | OALIntrInit (http://go.microsoft.com/fwlink/?LinkId=210070), BSPIntrInit (http://go.microsoft.com/fwlink/?LinkId=210071), OALIntrMapInit (http://go.microsoft.com/fwlink/?LinkId=210072) | intr.c |
| Handles interrupt requests | OALIntrRequestIrqs (http://go.microsoft.com/fwlink/?LinkId=210073), BSPIntrRequestIrqs (http://go.microsoft.com/fwlink/?LinkId=210074) | intr.c |
| Releases interrupts | OALIntrReleaseSysIntr (http://go.microsoft.com/fwlink/?LinkId=210075) | map.c |
| Handles interrupt translation and mapping | OALIntrStaticTranslate (http://go.microsoft.com/fwlink/?LinkId=210076), OALIntrTranslateIrq (http://go.microsoft.com/fwlink/?LinkId=210077) | map.c |

Module Inclusion

The build process uses configuration files and macros to determine which binaries and files to include in the run-time image. The following files specify the hardware-dependent modules and files (such as driver files) for the device:

- The platform.bib configuration file. It is located in Platform\<BSP Name>\Files. There is one platform.bib file for each BSP. For a description of the layout of BIB files, see [Binary Image Builder \(.bib\) File](#) (<http://go.microsoft.com/fwlink/?LinkId=210106>) on MSDN.
- The SOURCES files, which determine which files to compile during the build process. For more information about the build process, see [Windows Embedded Compact 7 Build Process](#) (<http://go.microsoft.com/fwlink/?LinkId=209954>). Note that the SOURCES files must compile the modules specified by platform.bib. If the modules are not compiled, the files will be missing when the run-time image is created at the end of the build, which will cause the run-time image creation to fail.

KITL

Because the kernel independent transport layer (KITL) is the transport mechanism that you use to debug the kernel, we strongly recommend that you support it for BSP development. For security reasons, you should disable KITL in any device that you release.

Some KITL support functions are implemented in the kernel's KITL library (kitlcore.lib). You do not need to modify these KITL support functions; you only need to modify the [OEMKitlStartup](http://go.microsoft.com/fwlink/?LinkId=210078) (<http://go.microsoft.com/fwlink/?LinkId=210078>) function, which is in the OAL. The following are typical BSP KITL code locations:

- Platform\Common\Src\Common\KITL
- Platform\<BSP Name>\Src\KITL
- Platform\Common\Src\<CPU Family>\Common\KITL

Table 13 lists the main KITL functions. Other related functions are described in [Kernel Independent Transport Layer](http://go.microsoft.com/fwlink/?LinkId=210107) (<http://go.microsoft.com/fwlink/?LinkId=210107>) on MSDN.

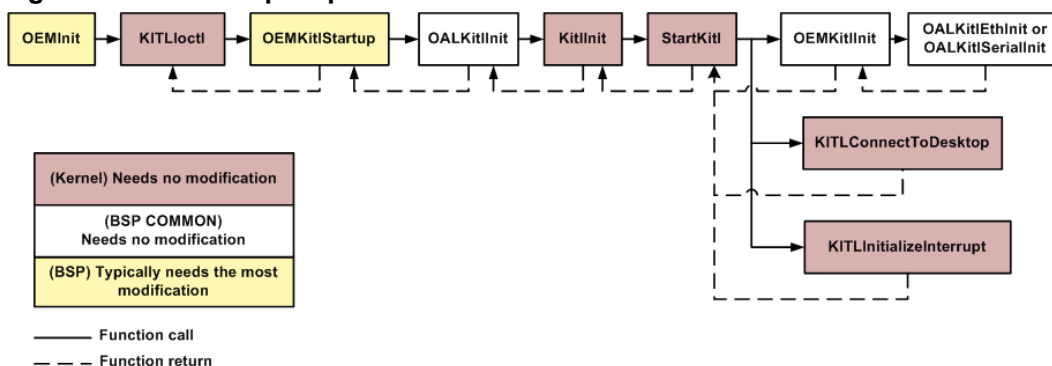
Table 13: KITL Functions

| Purpose | Function name | Typical file name |
|------------------|--|-------------------|
| Initializes KITL | OALKitlInit (http://go.microsoft.com/fwlink/?LinkId=210079), OEMKitlInit (http://go.microsoft.com/fwlink/?LinkId=210080) | kitl.c |
| Starts KITL | OEMKitlStartup (http://go.microsoft.com/fwlink/?LinkId=210078), StartKitl (http://go.microsoft.com/fwlink/?LinkId=210081) | kitl.c |

You can use the common implementation of **OALKitlInit**, **OEMKitlInit**, and **StartKitl**, but you will need to customize **OEMKitlStartup** for your hardware platform. **OEMKitlStartup** is the entry point to kitl.dll. It reads any boot arguments, chooses and initializes the KITL hardware transport, and calls **OALKitlInit**. **OALKitlInit** sets KITL to use active or passive mode, interrupt or polled transport (described below), initializes the KITL variables, such as function pointers for each required KITL function, and then starts KITL.

Figure 6 below shows the KITL startup sequence.

Figure 6: KITL startup sequence



Below is a description of the KITL startup sequence shown in Figure 6.

1. During the system startup process, the **OEMInit** function calls **KITLIoctl** with **IOCTL_KITL_STARTUP** as the IOCTL. (For the sequence of events leading to this point, see the [Startup](#) section.)
2. The **KITLIoctl** function, which is implemented in the kernel (`kitlcore.lib`), calls the **OEMKitlStartup** function, which is implemented in the OAL. The **OEMKitlStartup** function reads the boot arguments, chooses and initializes the KITL hardware transport (such as an Ethernet controller), and sets up data structures to contain the KITL parameters and the KITL device information. The **OEMKitlStartup** function is the primary function that you customize to use KITL with your hardware platform.
3. The **OEMKitlStartup** function calls the **OALKitlInit** function, passing in a parameter that determines whether KITL will be used in active or passive mode (these modes are described below). The **OALKitlInit** function is implemented in the BSP common KITL library (`oal_kitl.lib`).
4. The **OALKitlInit** function calls the **KitlInit** function, which is implemented in the kernel (`kitlcore.lib`). The **KitlInit** function's main task is to call the **StartKitl** function, which is also implemented in the kernel (`kitlcore.lib`).
5. The **StartKitl** function detects and initializes the Ethernet hardware and calls the BSP functions described below.
 - a. The **StartKitl** function calls the **OEMKitlInit** function, which is implemented in the common KITL library (`oal_kitl.lib`). If the KITL connection is over Ethernet, the **OEMKitlInit** function calls the **OALKitlEthInit** function. If the KITL connection is serial, the **OEMKitlInit** function calls the **OALKitlSerialInit** function.
 - b. The **StartKitl** function then calls the **KITLConnectToDesktop** function, which is implemented in the kernel (`kitlcore.lib`), to exchange transport information with the development computer.
 - c. Finally, if interrupt-mode KITL is used (which is optional), the **StartKitl** function calls the **KITLInitializeInterrupt** function, which is also implemented in the kernel (`kitlcore.lib`), to start the interrupt service routine (ISR) thread and unmask the hardware interrupts.

Interrupt and Polling KITL

Interrupt transport is faster than polled transport. Interrupt transport is useful for a completed KITL driver. Polled transport is less complicated than interrupt transport, but slower. It is useful for developing the KITL driver. When an interrupt model is in place, the kernel calls **PFN_ENABLEINT** to enable or disable the KITL transport interrupt. The interrupt is enabled when KITL is started, on a system resume, and when the kernel is started. Interrupt transport is the default.

To set up polling, the **OEMKitlStartup** function must call **OALKitlInit** with **OAL_KITL_FLAGS_POLL** in the **flags** member of the **pArgs** parameter. This value may be passed to the KITL initialization code by the boot loader, or it may be set in **OEMKitlStartup** as an override. If you are using the common KITL library (`oal_kitl.lib`), you do not need to do anything else; the rest is handled in the **OALKitlEthInit**

or **OALKitlSerialInit** function. When a polling model is in place, the kernel will not call **TransportEnableInt** to enable or disable the KITL transport interrupt. Rather, KITL actively calls **PFN_RECVFRAME** in a polling manner to retrieve data from the development computer.

Active and Passive KITL

KITL can operate in active or passive mode. In active mode, all KITL clients are initialized and registered when the device boots the OAL. In passive mode, KITL is initialized but not connected until an exception occurs on the device.

You must determine which mode the device enters on boot, by setting the flag **KTS_PASSIVE_MODE**. This flag is passed to the **OALKitlInit** function, which starts KITL in active or passive mode. You can set **KTS_PASSIVE_MODE** in Platform Builder.

▶ To set **KTS_PASSIVE_MODE** in Platform Builder

1. In Platform Builder, on the **Target** menu, click **Connectivity Options**.
2. On the left pane, click **Core Service Settings**.
3. On the right pane, do one of the following:
 - To enable passive KITL (set **KTS_PASSIVE_MODE**), clear the **Enable KITL on device boot** check box.
 - To enable active KITL (unset **KTS_PASSIVE_MODE**), select the **Enable KITL on device boot** check box.
4. Click **Apply**.

If you are using the **BLCOMMON** library, the **KTS_PASSIVE_MODE** flag is among the OS settings passed from Platform Builder to the boot loader during download. The boot loader then places it in the Boot Arguments area, where the BSP's KITL initialization code in **Kitl.dll** can detect the flag and add **OAL_KITL_FLAGS_PASSIVE** to the flags sent to **OALKitlInit**.

Active KITL is best suited for the development process so the debugger can maintain a constant connection. Passive KITL is better suited to a scenario in which the debugger is not constantly needed, for example, the later stages of testing a device, but before it is released as a retail device. The benefit of the passive mode is that you can create a device that does not have to be tethered to the development computer tools. The device can even be mobile, and if it enters a state where the development computer tools are needed, it initiates a KITL connection between the device and development computer.

PART 3: BSP Development Overview

The majority of the work in enabling Windows Embedded Compact to run on a specific hardware platform is customizing the BSP for that device. The development process to bring up a hardware platform from beginning to end includes the steps listed below:

1. Selecting an existing BSP
2. Cloning a BSP
3. Adapting a BSP
4. Testing a BSP
5. Designing and building an OS
6. Starting the OS

This article covers steps 1-3 and step 6. It does not cover steps 4 and 5 and instead directs you to relevant documentation.

Assumptions:

- You are using Platform Builder.
- You select an existing BSP that conforms with PQOAL (the Microsoft-supplied BSPs do).
- Your hardware platform is based on one of the CPU architectures supported by Windows Embedded Compact: ARM, MIPS, or x86.
- You chose an existing BSP with a CPU in the same architecture as the one on your hardware platform.

Requirements

Ensure that your device meets the minimum requirements to support Windows Embedded Compact as listed below.

CPU

- A CPU that supports one of the following instruction set architectures: x86, ARMv5, ARMv6, ARMv7, MIPSII, or MIPSII_FP
- Memory management unit capable of 4 GB 32-bit virtual memory

Synchronous dynamic RAM (SDRAM)

- 1 MB of contiguous physical SDRAM
- SDRAM must be accessible on **byte**, **word**, or **dword** boundaries

Other

- Must have an internal timer for scheduling



Note

There are no required peripherals, power considerations, or I/O mechanisms.

Step 1 Selecting an Existing BSP

Cloning an existing BSP is the easiest way to develop a BSP for a new piece of hardware even if you make extensive modifications to the code.

► To select an existing BSP

- Choose a BSP that uses a CPU from the same CPU architecture as the one on your device and has a hardware platform that best approximates the characteristics of the device you are working with.



Important

Choosing which BSP to start with is an important step, so be sure to carefully consider how well the features (such as the system timer, RTC, cache, and so on) of your device match those of the reference BSP's device.

You can:

- Select a BSP provided by Microsoft.
- Import a BSP from a third party.
- Migrate a BSP from a previous version of Windows Embedded Compact.

For more information about these options, see the sections below.

Using a BSP Provided by Microsoft

Windows Embedded Compact 7 provides several reference BSPs that support the ARM, MIPS, and x86 CPU architectures. When you install Platform Builder, you can choose which CPU architectures to install. You can then choose which BSP to use for your OS design when you step through Platform Builder's OS Design Wizard. To help you decide which reference BSP to clone, see the reference documentation on [ARM BSPs](http://go.microsoft.com/fwlink/?LinkId=210083) (<http://go.microsoft.com/fwlink/?LinkId=210083>), [MIPS BSPs](http://go.microsoft.com/fwlink/?LinkId=210084) (<http://go.microsoft.com/fwlink/?LinkId=210084>), and [x86 BSPs](http://go.microsoft.com/fwlink/?LinkId=210085) (<http://go.microsoft.com/fwlink/?LinkId=210085>) for a list of the hardware platform characteristics for each of the reference BSPs.

Platform Builder also provides a BSP template that you can use with different CPU architectures. The template can be very helpful when you are learning about the mandatory and optional APIs for a specific CPU architecture. However, because the APIs are not implemented, starting from a BSP template is usually more time-consuming than adapting a cloned implemented BSP.

Implementing a BSP by starting only with a template is outside the scope of this article, but for learning purposes you can select a BSP template in the OS Design Wizard by choosing **BSPTemplate:<CPU Architecture>**, where **<CPU Architecture>** is ARMv5, ARMv6, ARMv7, MIPSII, MIPSII_FP, or x86. The BSP template directories also contain tutorials for learning about the different parts of the BSP.

Importing a BSP from a Third Party

Several independent hardware vendors have developed Windows Embedded Compact BSPs. You can find a list of BSPs at [Find Board Support Packages](http://go.microsoft.com/fwlink/?LinkId=205434) (<http://go.microsoft.com/fwlink/?LinkId=205434>). The BSP provider will typically instruct you on how to install the BSP. Often, installation is performed using a Microsoft Installer Package (MSI).

Using a BSP Migrated from an Earlier Version of Windows Embedded Compact

If you want to start with a BSP from an earlier version of Windows Embedded Compact, we recommend that you first migrate it to Windows Embedded Compact 7 before cloning it. For instructions on porting a Windows CE 5.0 or Windows Embedded CE 6.0 BSP to Windows Embedded Compact 7, see [BSP Porting Guide for Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkId=205436) (<http://go.microsoft.com/fwlink/?LinkId=205436>).

Step 2 Cloning a BSP

When you clone a BSP, you copy the entire directory of the existing BSP into a new directory, so that you can modify your cloned BSP without affecting the code of the existing BSP. The easiest way to clone a BSP is by using Platform Builder as described in the procedure below.

For this procedure, we assume that you chose to install some BSPs when installing Platform Builder. If you did not, reinstall Platform Builder and include support for the BSPs that you think most closely match your device.

To clone a BSP

1. On the Platform Builder **Tools** menu, point to **Platform Builder**, and then click **Clone BSP**. The **Clone Board Support Package** dialog box appears.
2. In the **Clone Board Support Package** dialog box, browse for and choose **OS build tree (WINCEROOT)**, which is the Windows Embedded Compact 7 installation directory (typically c:\WINCE700).
3. In the **Source BSP** list, review the BSPs that you have installed on your computer, and select the BSP you want to clone.
BSP names are in the form **<BSP Name>:<CPU Architecture>**.
4. Under **New BSP information**, enter your new BSP name, description, Platform directory name, vendor, and version for the BSP.

Note

The Platform directory is the location that the cloned BSP is copied to. Your new directory will appear under the %_WINCEROOT%\Platform directory. The name for your new directory cannot contain spaces.

5. Click **Clone** to create the BSP.

A folder for the cloned BSP appears in the %_WINCEROOT%\Platform directory.

Later, when you use the OS Design Wizard in Platform Builder to create your OS design, the cloned BSP will be available on the **Board Support Packages** page. The cloned BSP will also be available in the **Source BSP** list (under its new name) the next time you clone a BSP.

For information about using Platform Builder, see [Using Platform Builder in Windows Embedded Compact 7](http://go.microsoft.com/fwlink/?LinkId=205680) (<http://go.microsoft.com/fwlink/?LinkId=205680>).

Step 3 Adapting a BSP

The guidelines in this section point out a small subset of the many functions and configuration files required by Windows Embedded Compact. For a list of required functions, see [Board Support Package \(BSP\)](http://go.microsoft.com/fwlink/?LinkId=205390) (<http://go.microsoft.com/fwlink/?LinkId=205390>). You can also refer to the comments in the BSP template source files to find information on which functions you must implement. The amount of BSP code that you must modify depends on how well your hardware platform matches the device of the BSP that you cloned. You can often reuse the existing code that is common to your device's CPU or SOC.

The general steps to adapt your cloned BSP for your device are below.

To adapt your BSP for your device

1. Adapt the boot loader to perform hardware-specific operations, such as:
 - Startup
 - Memory layout
 - Serial debugging (boot loader)
 - Download transport
 - Flash memory
2. Adapt the OAL of the BSP to your hardware configuration to handle:
 - Startup
 - Memory layout
 - Real-time clock (RTC)
 - Cache
 - Timer
 - Power management
 - Serial debugging (OAL)
 - Interrupts
 - Module inclusion
3. Enable KITL
4. Adapt device drivers

Adapting the Boot Loader

To adapt the boot loader, follow the steps below.

Startup

▶ To configure the boot loader startup process

- To perform hardware-specific initialization, you may need to edit the following functions:
 - Boot loader's **StartUp** (note that the OAL has a **StartUp** function also)
 - **OEMPlatformInit**
 - **OEMLaunch**

Memory Layout

▶ To define the memory layout on the device

- To specify where the boot loader resides in the device's memory, or to specify a shared memory area for the boot loader arguments that pass information to the OS, you may need to modify:
 - The boot.bib configuration file (it may also be called eboot.bib or sboot.bib).

Serial Debugging (Boot Loader)

▶ To support serial debugging

- To initialize the debug device, you may need to modify:
 - **OEMDebugInit**
 - **OEMInitDebugSerial**
- To output debug messages, you may need to modify:
 - **OEMWriteDebugString**
 - **OEMWriteDebugByte**
- To accept input to the boot loader, you may need to modify:
 - **OEMReadDebugByte**

Download Transport

▶ To configure an Ethernet download transport

- To initialize the download transport, you may need to modify:
 - **OEMPreDownload**
- To read data from the download transport, you may need to modify:
 - **OEMReadData**
- To send and receive data over an Ethernet connection, you may need to modify:
 - **OEMEthGetFrame**
 - **OEMEthSendFrame**

Flash Memory

▶ To support flash memory operations

- To support flash address functions, you may need to modify:
 - **OEMIsFlashAddr**
 - **OEMMapMemAddr**
- To support erasing of flash memory, you may need to modify:
 - **OEMStartEraseFlash**
 - **OEMContinueEraseFlash**
 - **OEMFinishEraseFlash**
- To support writing to flash memory, you may need to modify:
 - **OEMWriteFlash**

Adapting the OAL

To adapt the OAL, follow the steps below.

Startup

▶ To configure the OAL startup

- To further initialize the hardware and call the kernel initialization function, you may need to modify:
 - OAL's **StartUp** (note that the boot loader has a **StartUp** function also)
 - **OEMInit**

Memory Layout

▶ To define the memory layout for the run-time image

- To configure the memory layout, you may need to modify:
 - The config.bib configuration file
- To define the physical-to-virtual address mappings (for ARM and x86 only), you may need to modify:
 - **OEMAddressTable**
 - **OEMDeviceTable** (for ARM and x86 architectures using more than 512 MB of RAM)
 - **OEMRAMTable** (for ARM and x86 architectures using more than 512 MB of RAM)

Real-Time Clock

▶ To configure the RTC

- To set the time, you may need to modify:
 - **OEMSetRealTime**
 - **OEMSetAlarmTime**
- To get the time, you may need to modify:
 - **OEMGetRealTime**

Cache

▶ To handle the cache

- To initialize the cache, you may need to modify:
 - **OALCacheGlobalsInit**
- To get cache information, you may need to modify:
 - **OALIoCtlHalGetCacheInfo**
- To clean the cache, you may need to modify:
 - **OEMCacheRangeFlush**
 - **OALCleanDCache**
 - **OALCleanDCacheLines**
- To clear the translation look-aside buffers (TLBs), you may need to modify:
 - **OALClearDTLB**
 - **OALClearITLB**
- To flush the cache, you may need to modify:
 - **OALFlushDCache**
 - **OALFlushDCacheLines**
 - **OALFlushICache**
 - **OALFlushICacheLines**

Timer

▶ To configure the timer

- To initialize the timer, you may need to modify:
 - **OALTimerInit**
 - **OALTimerInitCount**
- To handle timer interrupts, you may need to modify:
 - **OALTimerIntrHandler**

- To get and set the timer, you may need to modify:
 - **OALTimerGetCount**
 - **OALTimerGetCompare**
 - **OALGetTickCount**
 - **OALTimerSetCompare**
 - **OALTimerUpdate**
 - **OALTimerRecharge**

Power Management

▶ To configure power management

- To place the CPU in an Idle state, you may need to modify:
 - **OEMIdle, OEMIdleEx** (preferred)
 - **OALCPUIIdle**
- To power down or place the CPU in a Suspend state, you may need to modify:
 - **OEMPowerOff**
- To enable or disable an interrupt from waking the system, you may need to modify:
 - **OALIoCtlHalEnableWake**
 - **OALIoCtlHalDisableWake**

Serial Debugging (OAL)

To set up the serial debugger, you can often use the boot loader's debug functions with minor modifications. For more information, see [Serial Debugging \(Boot Loader\)](#) above.

Interrupts

▶ To handle interrupts

- To enable interrupts, you may need to modify:

- **OALIntrEnableIrqs**
- **BSPIntrEnableIrq**

- To disable interrupts, you may need to modify:
 - **OALIntrDisableIrqs**
 - **OALIntrDisableIrq**

- To initialize interrupts, you may need to modify:
 - **OALIntrInit**
 - **BSPIntrInit**
 - **OALIntrMapInit**

- To handle interrupt requests, you may need to modify:
 - **OALIntrRequestIrqs**
 - **BSPIntrRequestIrqs**

- To release interrupts, you may need to modify:
 - **OALIntrReleaseSysIntr**

- To handle interrupt translation and mapping, you may need to modify:
 - **OALIntrStaticTranslate**
 - **OALIntrTranslateIrq**

Module Inclusion

To specify which hardware modules to include in the build

- To specify the hardware-dependent modules and files (such as driver files) for the device, you may need to modify:
 - The platform.bib configuration file.
 - The SOURCES files, which determine which files to compile during the build process. For more information about the build process, see [Windows Embedded Compact 7: Integrated Development Environment Build Process](http://go.microsoft.com/fwlink/?LinkId=210089) (http://go.microsoft.com/fwlink/?LinkId=210089) and [Windows Embedded Compact 7 Build Process](http://go.microsoft.com/fwlink/?LinkId=209954) (http://go.microsoft.com/fwlink/?LinkId=209954).

- The platform.reg file, which contains registry settings. To modify platform.reg, see [Building and Testing your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210199) (http://go.microsoft.com/fwlink/?LinkId=210199).

Enabling KITL

▶ To set up KITL

- To enable KITL, you may need to modify:
 - **OEMKitlInit**
- To configure the KITL startup process, you may need to modify:
 - **OEMKitlStartup**

Adapting Device Drivers

Device drivers are another part of the BSP that enable the kernel and applications to communicate with devices. Windows Embedded Compact comes with device driver sample code. For details about device drivers, see [Planning Your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210236) (http://go.microsoft.com/fwlink/?LinkId=210236), [Implementing Your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210237) (http://go.microsoft.com/fwlink/?LinkId=210237), and [Building and Testing Your Device Driver](http://go.microsoft.com/fwlink/?LinkId=210199) (http://go.microsoft.com/fwlink/?LinkId=210199).

Step 4 Testing a BSP

As you configure the boot loader, adapt the OAL, and bring up device drivers, you must test each component.

▶ To test each component

- Select and run the appropriate tests available from the wide variety in the Windows Embedded Compact Test Kit (CTK).

For more information about the CTK, see [Compact Test Kit \(CTK\)](http://go.microsoft.com/fwlink/?LinkId=205780) (http://go.microsoft.com/fwlink/?LinkId=205780) on MSDN.

Step 5 Designing and Building an OS

After you have created your BSP, you can design an OS based on that BSP by selecting your BSP from the **Board Support Packages** page in Platform Builder's OS Design Wizard. For information about how to design an OS, see [Developing an Operating System Design](http://go.microsoft.com/fwlink/?LinkId=210187) (<http://go.microsoft.com/fwlink/?LinkId=210187>).

After you have designed your OS, you can build the run-time image by using Platform Builder or the command line. For more information, see [Windows Embedded Compact 7: Integrated Development Environment Build Process](http://go.microsoft.com/fwlink/?LinkId=210089) (<http://go.microsoft.com/fwlink/?LinkId=210089>) and [Windows Embedded Compact 7 Build Process](http://go.microsoft.com/fwlink/?LinkId=209954) (<http://go.microsoft.com/fwlink/?LinkId=209954>).

Step 6 Starting an OS

After you have completed your OS image, Platform Builder provides a mechanism for you to download the run-time image to your device by using Ethernet, a universal serial bus (USB), or a serial connection. First, you must set up a connection between the development computer and the device. After that connection is configured, you can download the run-time image.

Connecting the Development Computer to the Device

To download a run-time image to a device, you must first configure a connection from a development computer to the device. The steps below use the default values as often as possible to get you up and running quickly. You may need to tailor the settings to your device.

To connect the development computer to the device

1. Physically connect the development computer to the target device by using the connection method (Ethernet, USB, or serial) present on your device and supported by the boot loader.
2. In Platform Builder, select **Target**, and then select **Connectivity Options**.
The **Target Device Connectivity Options** dialog box appears.
3. In the left pane, under **Device Configuration**, click **Add Device**.
4. Then do the following:
 - Enter a name for your device.
 - Click the **Associated OS Design/SDK** menu and then click **Windows CE**.
 - Click **Add**.
5. While still in the **Target Device Connectivity Options** dialog box, in the left pane under **Service Configuration**, click **Kernel Service Map**.
6. Click the **Kernel Download** menu, and then select the method you want to use to download the run-time image onto the device.
7. Click the **Settings** button that is next to the **Kernel Download** menu to configure the download settings.

8. Click the **Kernel Transport** menu, and then select the transport that you want to use to establish a kernel-level connection between your development computer and your device for debugging. We recommend that you match the setting you chose in step 6.
9. Click the **Settings** button that is next to the **Kernel Transport** menu to configure the transport settings.
10. Click the **Kernel Debugger** menu, select **KdStub** if your BSP supports KITL and you enabled kernel debugging during the OS build process.
11. Click the **Settings** button that is next to the **Kernel Debugger** menu to configure the kernel debugger settings.
12. Click **Apply**.
13. To configure the KITL settings, in the left pane, under **Service Configuration**, click **Core Service Settings**.
14. On the right pane, do one of the following:
 - To enable passive KITL, clear the **Enable KITL on device boot** check box.
 - To enable active KITL, select the **Enable KITL on device boot** check box.
15. Click **Apply**.
16. Click **Close**.

Downloading the Image onto the Device

After you have configured a connection between your development computer and your device, you can download the run-time image onto the device by using Platform Builder.

To download the image to the device

1. Load your boot loader into nonvolatile storage on the device.
The method you use to do so is hardware-dependent. Developers typically use a joint test action group (JTAG) programmer.
2. Put your boot loader in a state in which it is waiting to receive the OS image.
How you perform this task depends on your specific device.
3. In Platform Builder, click **Device**, and select your target device from the list.
4. To initiate the download, in Platform Builder, click **Target**, and then select **Attach Device**.
A dialog box displays a progress indicator for the download.

When the download is complete, your device will boot Windows Embedded Compact 7.

For an example of how to load the OS onto a virtual PC-based device (CEPC), which emulates an x86-based hardware platform, see [Getting Started with Virtual CEPC](http://go.microsoft.com/fwlink/?LinkId=205781) (<http://go.microsoft.com/fwlink/?LinkId=205781>).

Conclusion

To run Windows Embedded Compact 7, you need a board support package (BSP) to enable the OS to communicate with the hardware of your device. The BSP consists of a boot loader, an OAL, device drivers, KITL, and configuration files. The boot loader loads the OS into memory and jumps to the first instruction. The OAL then translates run-time requests from the OS to the hardware and notifies the OS of hardware activity. Device drivers enable the OS to communicate with peripherals such as a USB drive or an accelerometer, and configuration files specify the settings that are used to build the run-time image.

To maximize code reusability, the BSP code is organized using the PQOAL model. The PQOAL model separates code common to all hardware platforms from code specific to CPU architectures and specific hardware platforms. For example, boot loaders can use the **BLCOMMON** library, which is common code that provides much of the boot loader functionality. The **BLCOMMON** library, in turn, interacts with hardware-specific boot loader code.

The development of a BSP is easiest when you start from an existing BSP that supports Windows Embedded Compact 7, such as one that comes with Windows Embedded Compact, one developed by a third party, or one that you have migrated from a previous version of Windows Embedded Compact. After selecting a BSP, you can then customize it to your hardware configuration by modifying the code and configuration files that enable the hardware and the OS kernel to communicate.

The benefit of cloning a BSP, rather than creating a BSP from scratch, is that it speeds the development process because the code infrastructure is already in place, much of the code is already written, and some can be used “as is” depending on your needs. In addition, the code that you reuse has already been tested and may contain helpful comments. After you modify the BSP so that it works on your hardware platform, you can use it as a basis for developing an OS design so that you can take advantage of the features of your device. With a finished OS, you merely need to download it to your device and run it.

Additional Resources

- [Windows Embedded Compact website](http://go.microsoft.com/fwlink/?LinkID=210130) (http://go.microsoft.com/fwlink/?LinkID=210130)

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.