



Windows Embedded Compact 7 Build Process

Writer: Travis Hobrla

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

Describes the entire build process in detail, focusing on use of the command line.

- Take advantage of the power and flexibility of the command line
- Save time with targeted builds.
- Create a stable and robust build process.
- Debug build issues faster and more easily by understanding the process in detail

Contents

Introduction	3
Terminology and Syntax.....	3
Motivation	3
Build System Purpose and Overview	3
From the Bottom Up: Build.exe.....	4
Build Output Files.....	5
Build Input Files	6
Compiler/Linker Input	6
Synchronization	6
Additional Build Steps	7
From the Top Down: Blddemo and the Build Process	9
Inputs.....	9
Outputs.....	9
Build OS (blddemo [-c])	12
Sysgen OS (blddemo -q).....	13
Build Platform\common	14
Build Platform\\$(_TGTPLAT).....	15
Buildrel	15
Building "Clean"	16
Miscellaneous Useful Build Variables	16
Optimization.....	17
Conclusion	17

Introduction

The Windows Embedded Compact 7 build system is a complex process. This article, which is intended for advanced users of Windows Embedded Compact 7, describes the entire process in detail, focusing on the command-line build process. This article does not cover the differences between Windows Embedded Compact 7 and previous versions or building with the integrated development environment (IDE).

Terminology and Syntax

- The syntax `$(VARIABLE)` is used to denote environment variables.
- Directory listings are assumed to start with `$(_WINCEROOT)`, which is the install path of the shared source code for Windows Embedded Compact 7.

Motivation

Why do you need to understand the Windows Embedded Compact 7 build system? The most important reason is that it will save you time. The build system is a complex tool. With an in-depth understanding, you will be able to perform targeted builds, in which you build only what you need and nothing more. Also, you will improve your ability to quickly debug build issues in your own projects. Finally, you will be able to provide stable, robust build logic for your customers, saving them time also.

It is important to understand the Compact 7 build system from the command line because the command-line tools offer more flexibility and power than the IDE. The Platform Builder IDE is simply a shell that makes calls to the command line. If you have a solid understanding of the command line build, understanding build using the IDE is an easy step.

Build System Purpose and Overview

The Compact 7 build system is responsible for building an entire operating system. Unlike building a software application, building an entire operating system includes a large variety of components:

- The core operating system (kernel, file system, security features)
- Built-in applications and services (Media Player, Internet Explorer, Remote Desktop Connection)
- Built-in driver stacks (USB, networking)
- Custom drivers and hardware abstractions (display driver, audio driver, OEM adaptation layer)
- Custom applications

The first objective of the build system is to combine all of these components into a single binary image. An image is typically a single file that contains all of the above components laid out in memory so that they can be downloaded to a device and then executed by the device. The second objective of the

build system is to allow components to be easily selected and substituted so that only the necessary components are included in the image.

You can use Compact 7 on a wide variety of embedded devices. Some devices may not take advantage of a particular operating system component. For example, a device that has no display or audio hardware has no use for a media player application. If the build system included a media player application regardless, it would waste memory on the device that otherwise could be used to increase performance or reduce cost. The build system must make it easy to build only the necessary components.

From the Bottom Up: Build.exe

At the lowest level, the build system uses build.exe to compile and link code. Build.exe takes a number of arguments but is most often used in one of two ways:

- Build.exe (no arguments): compiles and links only files that have changed since the last time build.exe was called
- Build.exe -c: compiles and links regardless of file time stamps

Build.exe starts executing in the directory in which it's called. There, it will output log files that describe what occurred during its execution. These files are build.err (for errors only), build.wrn (for warnings), and build.log (for all output, including errors and warnings). It will also output the compiled and linked source code to a variety of locations, which will be covered later.

To produce the compiled and linked source code, build.exe executes recursively. Starting from the directory in which it's called, it parses DIRS files to determine which subdirectories to run on. For example (public\COMMON\oak\drivers\sdcard\DIRS):

```
DIRS= \
    SDHCLib \
    SDCard_Lib \
    SDCardLib \
    SDBus \
    SDBus_dll \
    SDHCDrivers \
    SDClientDrivers
```

When build.exe sees this file, it will scan all of the subdirectories in the order they are listed in the DIRS file. If the subdirectory contains a DIRS file it will continue scanning recursively. This process continues until all subdirectories with a DIRS entry have been enumerated. When build.exe finishes enumerating all of the DIRS, it checks for SOURCES files in each of the leaf-node directories. It will compile and link source code based on several types of input:

- Sources.cmn (a common environment file typically found at the root directory)
- The SOURCES file in the directory that it is compiling and/or linking
- Environment variables in the local environment

Below is an example of a SOURCES file (platform\common\src\common\bldr\log):

```
TARGETNAME=boot_log
TARGETTYPE=LIBRARY

INCLUDES=..\inc;$(INCLUDES);

SOURCES= \
    format.c \
    log.c \
    dump.c \
    oalLog.c \
    debugLog.c \
    kitlLog.c

WARNLEVEL=4
WARNISERROR=1
```

When build.exe is called on this SOURCES file, it compiles all of the .c files listed, including headers from the standard include path and the ..\inc directory. It links the output file boot_log.lib, and it uses the highest level of compiler warnings, throwing an error if any warnings are found during compilation.

You can find the full list of SOURCES macros in the Windows Embedded Compact 7 reference documentation. Some of the more common macros that are encountered during normal development are explained in the following sections.

Build Output Files

- **TARGETTYPE:** Determines what kind of binary file to build. Common values are DYNLINK for .dll's, PROGRAM for .exe's, LIBRARY for .lib's, or NOTARGET for special SOURCES files that don't produce an output binary.
- **TARGETNAME:** The name of the DLL, EXE, or LIB to build.
- **EXEENTRY** or **DLLENTRY:** If an .exe or .dll is to be built, this macro is the entry point (function).
- **RELEASETYPE:** Determines the output directory.

Build Input Files

- **DEFFILE:** This macro is the module definition file for .dll's.
- **SOURCES:** This macro lists the source code files to compile.
- **TARGETLIBS:** This macro lists libraries containing object code needed for symbol resolution. Typically, this macro is used when creating a .dll or .exe.
- **SOURCELIBS:** This macro lists libraries whose object code is included in its entirety. Typically, this macro is used when creating a static .lib file.

Compiler/Linker Input

- **CDEFINES / ADEFINES:** These macros are passed in as #defines to the compiler and assembler, respectively.
- **INCLUDES:** This macro specifies the include path to search for the header file that are included by the source files.

Synchronization

Synchronization macros are useful in the multithreaded build environment, which build.exe uses by default. During build.exe's execution, these macros ensure that modules that have dependencies are not built until those dependencies are satisfied. Build.exe generally handles dependencies properly, but if you use special makefile rules, these macros may be necessary.

- **SYNCHRONIZE_BLOCK:** Deprecated in Windows Embedded CE 6.0 but still supported and occasionally used. This macro delays all future SOURCES files in the DIRS tree until the current one has finished building completely.
- **SYNCHRONIZE_DRAIN:** Deprecated in Windows Embedded CE 6.0 but still supported and occasionally used. This macro forces all previous SOURCES files in the DIRS tree to finish building before the current directory is built.
- **XPASSLEVEL:** This macro provides a more granular control mechanism than **SYNCHRONIZE_BLOCK / SYNCHRONIZE_DRAIN**. It assigns a passlevel to the SOURCES file. Build.exe follows the ordering within each pass. For example, build.exe would always execute the LIB build pass for a SOURCES file with LIBPASSLEVEL=0 before a SOURCES file with LIBPASSLEVEL=1. The build passes are defined in public\common\oak\misc\buildtable.xml.
- **PASS_PRODUCES:** Similar to XPASSLEVEL, this macro provides granular control by allowing a SOURCES file to specify what it produces so that another SOURCES file can depend on it via **PASS_CONSUMES**.
- **PASS_CONSUMES:** Instructs the SOURCES file not to execute the pass until the corresponding **PASS_PRODUCES** SOURCES file has completed that pass. For example, a SOURCES file containing LINK_CONSUMES: library1 would not perform any linking until the SOURCES file containing LINK_PRODUCES: library1 finished its link pass.

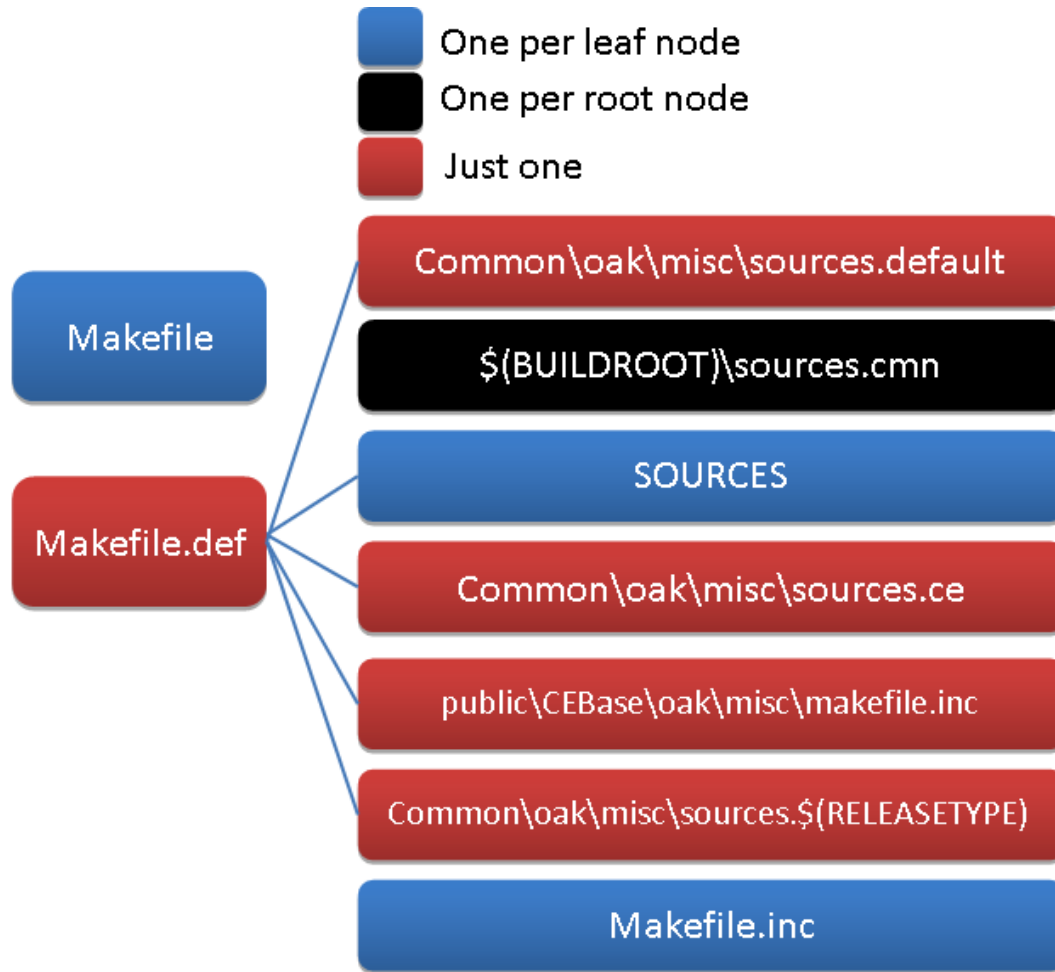
Additional Build Steps

- **WINCETARGETFILE0:** This macro specifies extra makefile rules to run once before all build passes (not once per pass).
- **WINCETARGETFILES:** This macro specifies extra makefile rules to run once after the link pass (not once per pass).

To understand exactly how these SOURCES macros work, you need to examine the build system in more depth. While it's true that build.exe is responsible for compiling and linking code, it actually calls nmake.exe to do the real compilation and linking. Nmake.exe uses makefiles to determine dependencies and what to compile and link. Thus, each leaf-node directory that contains a SOURCES file also contains a makefile. This makefile is a dummy makefile that is basically a pointer to the master makefile: public\common\oak\misc\makefile.def.

Makefile.def is a large file that is ultimately responsible for providing the right information to nmake.exe. It interprets the SOURCES macros, creating a set of flags to pass to the compiler and linker. So makefile.def is the place to go if you want to understand exactly how a particular SOURCES macro works. Makefile.def also includes several other files beyond the SOURCES file (see Figure 1).

Figure 1: Makefile include ordering



With this understanding of makefile.def in mind, review this slightly more complex SOURCES file (public\COMMON\oak\drivers\sdcard\sdbus_dll\SOURCES).

```
TARGETNAME=SDBus
TARGETTYPE=DYNLINK
RELEASETYPE=OAK
DLLENTY=_DllEntryCRTStartup
DEFFILE=..\sdbus\sdbus.def

SOURCES=

TARGETLIBS= \
```

```

$( _COREDLL) \
$( _CEDDK)

SOURCELIBS= \
    $( _PUBLICROOT) \common\oak\lib\$( _CPUINDPATH) \SDBus_LIB.lib \
    $( _PUBLICROOT) \common\oak\lib\$( _CPUINDPATH) \defbuslib.lib \
    $( _PUBLICROOT) \common\oak\lib\$( _CPUINDPATH) \sdcardlib.lib

```

By looking at TARGETNAME, TARGETTYPE, and DLLENTY, you can see that it creates SDBus.dll, and its entry point will be _DllEntryCRTStartup. It has a module definition file located in ..\sdbus, which will typically describe its function exports. In this case, the code is not compiled, but coredll.lib and ceddk.lib are linked with several libraries specific to this driver that have already been built. It's important to use SOURCELIBS for these libraries in order to include their object code so that the function exports in sdbus.def are valid. If all of these are set to TARGETLIBS, because there isn't any source code that requires their symbols for resolution, the result is a DLL with nothing in it.

Finally, the RELEASETYPE for SDBus.dll is OAK. If you look in makefile.def, you can see that this causes public\common\oak\misc\Sources.ReleaseType_OAK to be included, which will set the output directory to \$(__PROJROOT)\oak\target\\$(__CPUDIR). The next section provides more information on how to decode these environment variables.

From the Top Down: Blddemo and the Build Process

Now that you understand how the build works at a low level, you can examine the tools that orchestrate calls to build.exe. You can also examine other supporting tools that are used on the way to making your final image. At the highest level, building the operating system is a simple equation of inputs and outputs.

Inputs

- Environment variables
- Source code
- Configuration files (such as SOURCES, DIRS, .def, and .bib files)

Outputs

- An image that you can download to and execute on the device

- A "flat" release directory with all of the modules in the image
- Many intermediate libraries

The build process uses the environment to determine what to build and how to build it.

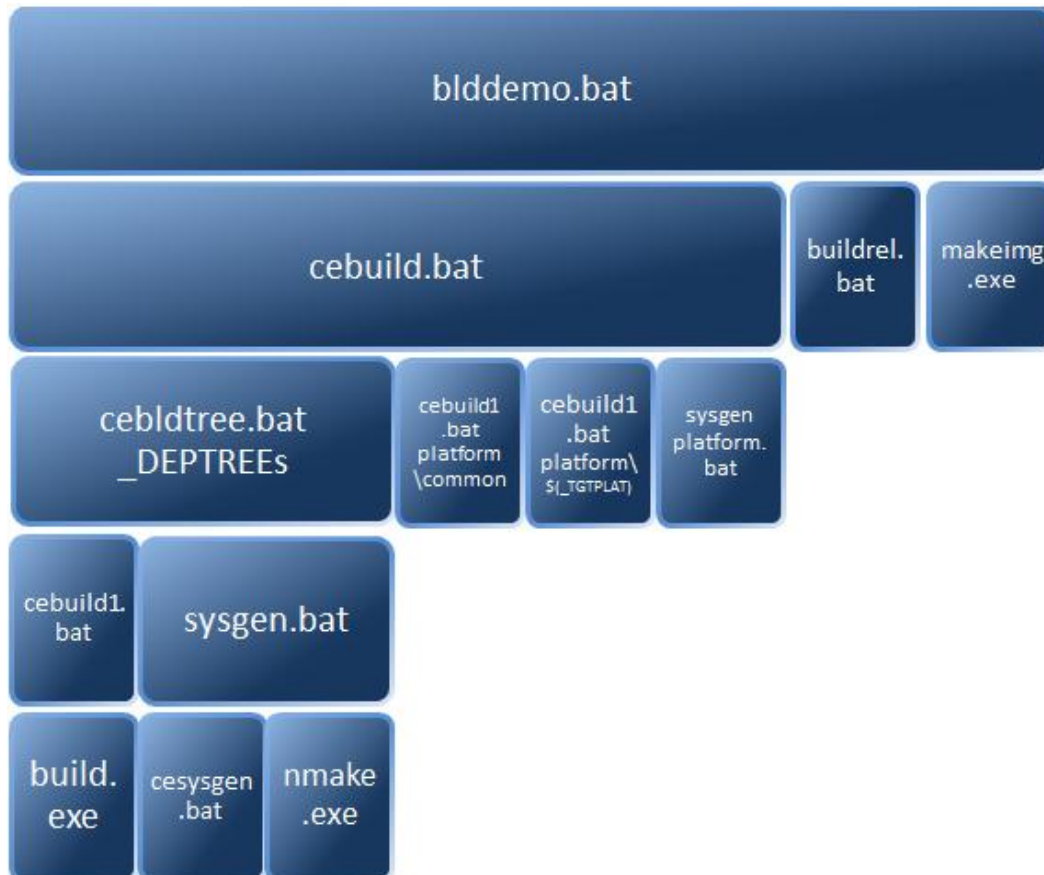
Public\common\oak\misc\wince.bat is responsible for setting up the environment with most of its default values. The caller of wince.bat is responsible for filling in a few blanks as listed below:

- \$(_TGTPLAT): The platform (board support package) to be built
- \$(_TGTPROJ): The OSDesign that is used (for Windows Embedded Compact 7 this is always CEBase)
- \$(_TGTCPU): The CPU architecture that is being used

Opening a build window is as simple as opening a Command Prompt window and calling wince.bat with the parameters listed above. (A build window can also be created in Platform Builder, but that is out of scope of this article.) After the parameters are set, wince.bat calls other batch files during its execution to set up the rest of the environment.

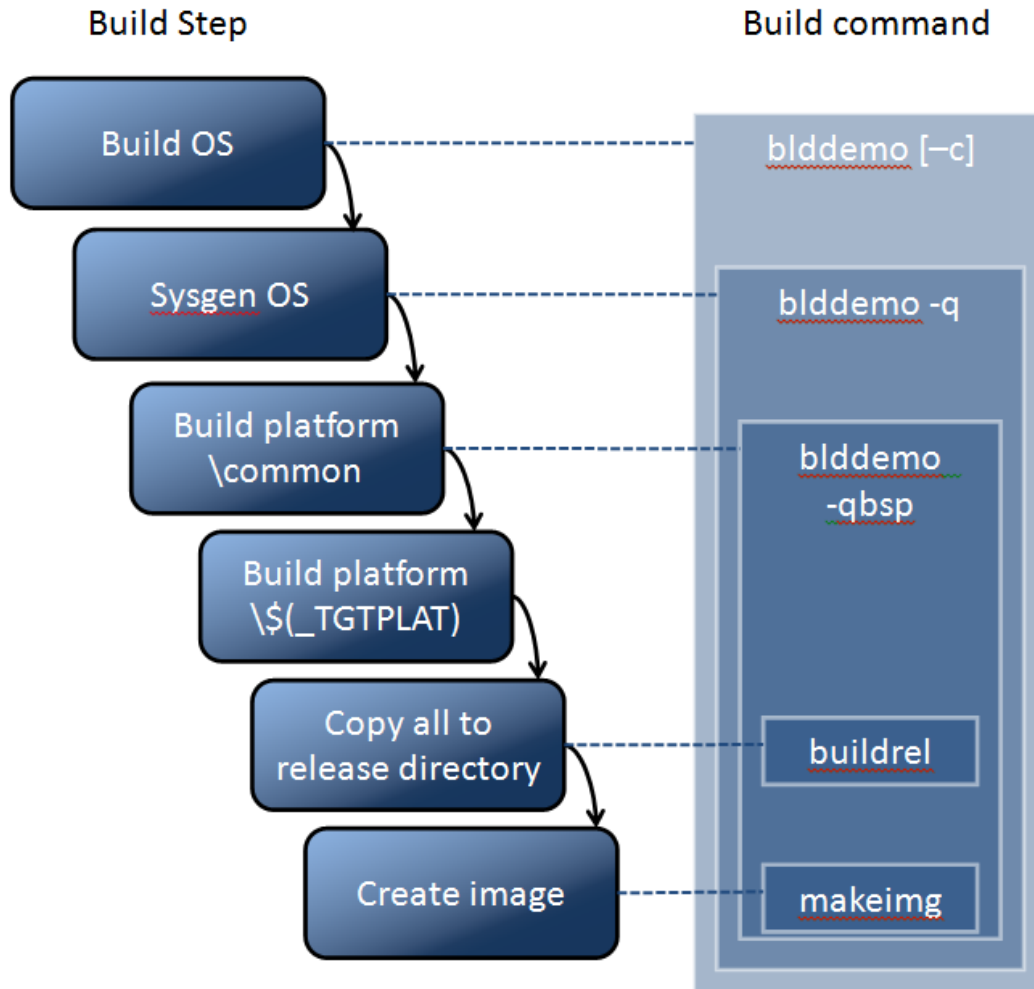
Wince.bat is not the only batch file that the build process uses. Although wince.bat sets up the initial environment the execution of the whole build process hinges on additional batch files. These batch files are located in public\common\oak\misc along with most of the other files that govern the build. To learn more about a build step or to understand the build in more detail, you can examine the batch and build files directly. By examining files in this directory, you can decode the \$(__PROJROOT) and \$(__CPUDIR) variables from the more difficult SOURCES file example. The following figure, Figure 2, shows the order of the batch file calls to blddemo, which will help you understand the high-level build steps discussed in the following section. In this figure, the calls to the batch files go depth-first, from top to bottom and then left to right, so the first six calls in the figure would be blddemo.bat, cebuild.bat, cebldtree.bat, cebuild1.bat, build.exe, and sysgen.bat.

Figure 2: Build process batch file calls



Using just the environment set up by wince.bat, all of the source code and configuration files, and the build batch files in public\common\oak\misc, the build process can start in earnest. The build process is a waterfall of steps. Each step occurs in order and feeds the next step. See Figure 3 for a full description of the steps. Note that the [-c] syntax denotes that -c is optional.

Figure 3: Build process steps



Build OS (blddemo [-c])

The Build OS step is the first and longest step of the build process. Fortunately, this step is already completed once Platform Builder is installed. The output libraries and .dll files resulting from this step are included with Platform Builder— that's what takes up the gigabytes of space for each CPU architecture that you select during installation.

The Build OS process is straightforward: For each of the `$_(DEPTREES)` set up by `wince.bat`, call `build.exe` on the private tree and then on the public tree. The result of these `build.exe` calls are (static) .lib files and .dll files in the following locations:

- `Public_DEPTREE\oak\lib\$_(TGTCPU)\$_(WINCEDEBUG)` - for .lib files that are usable by OEMs

- Public_DEPTREE\oak\target\\$_(TGTCPU)\\$_(WINCEDEBUG) - for .dll and .exe files that are usable by OEMs
- Public_DEPTREE\sdk\lib\\$_(TGTCPU)\\$_(WINCEDEBUG) - for .lib files that are usable by application developers

After the Build OS step is completed, these directories contain everything that you need for the later build steps. This is important because having everything in these directories after the Build OS step is completed means that you don't need to run the Build OS step again unless code in the public or private tree is changed.

Note that changing code in the public and private trees can cause serious build failures, which is why Microsoft takes care to compile this source code ahead of time and provide the binaries to you in Platform Builder. Thus, changing this code is advisable only for expert users. If you do change the code, then you must run blddemo so that the Build OS step will pick up the changes. Targeted builds of the public/private tree can be much faster. However, that is outside of the scope of this article.

Sysgen OS (blddemo -q)

The Sysgen OS step links together the libraries created during the Build OS step. The Sysgen OS step manages link dependencies and selects only the components that are needed for the operating system. To determine which components are desired, this step reads SYSGEN_ variables from the environment. These SYSGEN_ values represent operating system components and correspond to Catalog selections in Platform Builder.

How does the Sysgen OS step determine which components to include and how to manage the dependencies? The build batch files call another set of batch files to translate the SYSGEN_ variables from the environment into a set of MODULES and COMPONENTS variables. The batch files that are responsible for the translation are:

- Platform\\$_(TGTPLAT)\cebasecesysgen.bat
- \$_(PROJECTROOT)\oak\misc_DEPTREE.bat
- Public\CEBase\oak\misc_DEPTREE.bat
- Public_DEPTREE\cebasecesysgen.bat

After the translation is complete, a full listing of MODULES and COMPONENTS is created in \$(SG_OUTPUT_ROOT)\oak\files\ceconfig.h. This listing of MODULES and COMPONENTS variables is used to call the cesysgen\makefile for each value in \$_(DEPTREEs). A makefile rule is executed for each MODULE and COMPONENT variable in the environment. Some larger .lib files such as coredll.dll may be linked without certain functionality if the corresponding MODULE or COMPONENT is not present. For example, in coredll.dll, the **RequestBluetoothNotifications** function is not included by the linker unless CE_MODULES_BT is set, which in turn is set by the corresponding Bluetooth SYSGEN_ variable (SYSGEN_BTH). The function may also be filtered out of header files in the same way. The end result of all of this dependency checking and linking is that all of the needed files are linked and copied to \$(SG_OUTPUT_ROOT).

A new feature in Windows Embedded Compact 7 is the pre-linking in the Build OS step of many binaries that used to be linked in the Sysgen OS step. As a result, the Sysgen OS step executes more quickly. Don't be surprised if when looking at a cesysgen\makefile you see some rules that only copy instead of link.

After the Sysgen OS step is completed, \$(SG_OUTPUT_ROOT) directory contains everything that you need for the later build steps. You don't need to run it again unless the SYSGEN_ variables change.

Build Platform\common

After the Sysgen OS step, the build process gets substantially simpler. The platform\common build step runs build.exe in the platform\common directory. The resulting libraries and .dll files are output to the following two directories, respectively:

- \$(SG_OUTPUT_ROOT)\platcomm\\$_TGTPLAT\lib\\$_TGTCPU\\$_WINCEDEBUG
- \$(SG_OUTPUT_ROOT)\platcomm\\$_TGTPLAT\target\\$_TGTCPU\\$_WINCEDEBUG

Some directories in platform\common are CPU-specific, and their DIRS file will have a special directive such as DIRS_ARM. The directive indicates that the files in this directory, for example, are compiled only if an ARM CPU architecture is selected during the call to wince.bat.

Platform\common directories may depend on libraries from the Sysgen OS step and may rely on certain MODULES and COMPONENTS being present. For example, you don't want to build a Bluetooth driver in platform\common if you didn't build Bluetooth functionality into the operating system during the Sysgen OS step. To accomplish this, filtering is done on DIRS files based on ceconfig.h. Below is a more complex DIRS example (platform\common\src\soc\omap35xx_tps659xx_ti_v1\omap\tps659xx):

```
DIRS= \

# @CESYSGEN IF CE_MODULES_DEVICE

    twl \

    gpio \

# @CESYSGEN IF GWES_KBDUI

    keypad \

# @CESYSGEN ENDIF GWES_KBDUI

    wave \

# @CESYSGEN ENDIF CE_MODULES_DEVICE
```

After the Build platform\common step is completed, \$(SG_OUTPUT_ROOT) directory contains all of the platform\common files that you need for the later build steps. It does not need to be run again unless the platform\common files change.

Build Platform\\$(_TGTPLAT)

This step simply runs build.exe in platform\\$(_TGTPLAT). These directories may depend on libraries from the Sysgen OS step or the Build platform\common step. Again, DIRS filtering is used to build only the directories for which the right operating system components are present.

Sysgenplatform.bat is called once before and after build.exe to handle any special rules (such as filtering configuration files based on ceconfig.h and copying them to \$(SG_OUTPUT_ROOT)).

After the Build platform step is completed, \$(SG_OUTPUT_ROOT) directory contains all of the platform\\$(_TGTPLAT) files that you need for the later build steps. You don't need to run it again unless the platform\\$(_TGTPLAT) files change. In fact, at this point in the build process, you are finished compiling and linking and are almost ready to build your image.

Buildrel

The Buildrel step copies all of the needed files from \$(SG_OUTPUT_ROOT) directory to the \$(_FLATRELEASEDIR) directory, also known as the Flat Release Directory. This directory contains all of the final module code for the image.

Makeimg:

The makeimg step reads configuration files from the Flat Release Directory and combines modules into a single image file. It calls several different tools to accomplish this task, which is explained in more detail in the [Make Binary Image Tool \(Makeimg.exe\)](http://go.microsoft.com/fwlink/?LinkId=208442) (<http://go.microsoft.com/fwlink/?LinkId=208442>) topic. Although there are a number of configuration file types that makeimg uses, two kinds are of critical importance: .bib files and .reg files.

BIB files contain instructions for makeimg about which modules to include and where they will be placed in the image. BIB files can have CESYSGEN conditionals just like the DIRS files in platform\common and platform\\$(_TGTPLAT). These conditionals are parsed by the cefilter.exe tool and are used to eliminate unneeded modules. BIB files can also have BSP and IMG conditionals, which are parsed by makeimg. These conditionals follow simple IF logic and are useful for quickly removing or including a module or file in the image.

The FILES section of the .bib file lists all of the files that will be included in the image. The MODULES section of the .bib file lists all of the code binaries that are included in the image. Each of these listings can include flags that describe various properties of the file, such as whether it is a system file or whether it is compressible. BIB files may also instruct makeimg to name files in the image differently than their original name in the Flat Release Directory. Here is an example from (public\common\oak\files\common.bib):

```
locale.dll      $( _FLATRELEASEDIR)\nlslocale.dll      NK   SHQ
```

This line means that nlslocale.dll in the Flat Release Directory will be included in the image as locale.dll, a system file (S flag) that is hidden (H flag) and available both in user mode and kernel mode (Q flag).

Finally, .bib files may contain fix-up variables. Fix-up variables are data substitutions that occur at makeimg time and enable quick adjustment of important startup variables.

REG files define the starting registry for the image. Like .bib files, they can have CESYSGEN conditionals, which are parsed by cefilter.exe, and BSP or IMG conditionals, which are parsed by makeimg.

Each _DEPTREE and platform has its own .bib and .reg files in its \files directory. Cefilter filters the SYSGEN tags out of these files, and the resulting file is copied to intermediate directories:

- \$(SG_OUTPUT_ROOT)\oak\files — for _DEPTREEs
- \$(SG_OUTPUT_ROOT)\platform\\$_TGTPLAT\files — for platform\\$_TGTPLAT

Makeimg concatenates all of the .bib and .reg files in the Flat Release Directory into ce.bib and reginit.ini, respectively. These two files describe the final image in its entirety.

When the Makeimg step is complete, an image is produced in the Flat Release Directory that can be downloaded to the device and executed. In other words, the build process is finished!

To fully understand the build process, only a few more concepts need explanation.

Building "Clean"

By default, build.exe only compiles if targets are out of date; that is, if the time stamps for the sources files don't match the .lib files that they produced. As discussed previously, calling build.exe with the -c flag forces a recompile and relink. You can force a recompile all throughout the build process by calling blddemo -c, although this is not commonly done.

Blddemo "clean" deletes \$(SG_OUTPUT_ROOT) so that the Sysgen OS step writes to an empty directory. It is important to pass the "clean" flag when SYSGEN_ variables change so that stale binaries don't remain in the \$(SG_OUTPUT_ROOT). Similarly, blddemo "cleanplat" marks the binaries in platform\common and platform\\$_TGTPLAT for deletion. .bif files track whether platform\common and platform\\$_TGTPLAT are up to date, and "cleanplat" removes these .bif files, triggering build.exe to be called. The binaries previously marked for deletion are overwritten by the new binaries created by build.exe. The file public\common\oak\misc\CleanOS.bat is responsible for interpreting the "clean" and "cleanplat" flags.

Miscellaneous Useful Build Variables

WINCEREL: This variable acts as an amortized version of buildrel when it is set. Most files are copied to the Flat Release Directory as they are built instead of in a large batch when Buildrel executes.

WINCECOD: This variable instructs the compiler to generate .cod files during compilation for each source file compiled. These .cod files are large files that provide an assembly-level description of the compiled code. They can be useful when trying to understand compiler optimizations.

BUILD_MULTIPROCESSOR: The value of this variable specifies the number of threads that build.exe is allowed to run simultaneously. If not set, build.exe will minimize build time by using multiple threads based on the number of CPUs in the system. Setting this value to 1 can be useful for debugging intermittent build failures, which are often the result of synchronization issues in SOURCES files.

Optimization

The final aspect of the build process not yet discussed in this article is optimization of binaries. On a shipping device, the binaries produced by the build process are heavily optimized by the compiler. Such optimization is unwanted during device development when you need to debug. Therefore, the build system offers granular control over the type of binaries that are built (see Figure 4).

Figure 4: Compiler optimization and debug messaging

WINCEDEBUG value	Compiler Optimized	Contains RETAILMSGs	Contains Asserts	Contains DEBUGMSGs
Debug		X	X	X
Checked	X	X	X	X
Retail	X	X		
Ship (WINCEDEBUG=retail and WINCESHIP=1)	X			

COMPILE_DEBUG=1 overrides WINCEDEBUG and forces a debug compile.

ENABLE_OPTIMIZER=0 overrides compiler optimizations (turning them off)

Conclusion

This article describes the Windows Embedded Compact 7 command-line build process in detail. The build is a complex process with many steps. With a complete understanding of the process from start to finish, you can now understand each step fully and debug any build failures that occur.

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.