



Windows Embedded Compact 7: Improving Your Build Process

Windows Embedded Compact 7 Technical Article

Writer: Travis Hobrla

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

As you develop a Windows Embedded Compact 7 OS design, building can be complex and time-consuming. This article provides strategies designed to reduce your build times and fast-track the reflection of code changes in your image. The intended audience is intermediate and advanced users of Windows Embedded Compact 7.

By using appropriate build commands and applying the targeted build preparation algorithm to code changes, you can pinpoint build dependencies in order to build only what you must.

If you must modify code from the public tree, you can protect public shared code by cloning public code into the platform tree. Protecting public code can prevent reinstallations of Platform Builder and provide you with a fall-back solution if your modifications become temporarily unbuildable.

Custom makefile rules extend your control of build.exe beyond that offered by SOURCES macros. Actions such as copying files and running external programs can be specified to run before or after the compile phase.

When developing a board support package (BSP), properly filtering DIRS, .bib and .reg files will ensure that your BSP works, regardless of selected SYSGEN or catalog items.

Using Release Directory Modules, you can load modules directly from the Release Directory instead of from the image. For modules that are loaded and unloaded from memory during use, changes can be reflected on the device without a reboot.

Overview

When you develop a Windows Embedded Compact 7 OS design, you can customize the build process to reduce your build times and fast-track the reflection of specific code changes in your image. In this article, you will learn building strategies designed to give you more control over building your Windows Embedded Compact 7 OS design.

This article assumes you have read [Windows Embedded Compact 7 Build Process](http://go.microsoft.com/fwlink/?LinkID=208071) (<http://go.microsoft.com/fwlink/?LinkID=208071>) that explains the build process in detail. While you will still get value from this article, an understanding of the build process will help you in applying the strategies presented.

Several included examples address the modification of public code. Modifying code in the public tree is dangerous because rolling back changes to public binaries requires the reinstallation of Platform Builder. We do not recommend the modification of public code, nor do we provide support for this practice. Public code modification examples are intended for an expert audience only.

Terminology and Syntax

\$(VARIABLE) syntax will be 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.

Select the Appropriate Build Command

Choosing the most appropriate build command for your situation will help you to avoid stale binaries and prevent rebuilding unmodified code. You can determine which integrated development environment (IDE) build command to use by matching your situation against the following six build commands:

blddemo -q (Sysgen / Build OSDesign / Build Solution)

When to use: Once each time you create a new OS design in Platform Builder.

This command runs the Sysgen phase without doing any cleanup. If you have changed SYSGEN_ variables (or catalog items), you must instead use blddemo clean -q to avoid stale binaries. If you have not changed SYSGEN_ variables (or catalog items), blddemo -qbsp will be faster while still compiling all of the needed binaries.

blddemo clean -q (Clean Sysgen / Rebuild OSDesign / Rebuild Solution)

When to use: Whenever you have changed SYSGEN_ variables (or catalog items).

This command cleans the Sysgen output directory and then runs a build starting from the Sysgen phase.

blddemo (Build and Sysgen)

When to use: When you have modified public code and have not changed SYSGEN variables or catalog items, but are not doing a targeted build (as described in the [Build in the Fastest Way After Changing <X> \(Targeted Builds\)](#) section). Recall from the [Overview](#) section that

public code modification is not recommended or supported; it is considered to be for experts only.

This command starts the build from the Build OS phase but does not clean the Sysgen output directory. This is only useful when you have modified public code.

blddemo clean cleanplat -c (Rebuild and Clean Sysgen)

When to use: When you have modified public code and have also changed SYSGEN variables. Again, recall from the [Overview](#) section that public code modification is not recommended or supported; it is considered to be for experts only.

This command starts the build from the Build OS phase and does a forced recompile of all files. It also cleans the Sysgen output directory.

blddemo -qbsp (Build Current BSP and Subprojects)

When to use: When you have not changed _SYSGEN variables (or catalog items) but have changed files or code in platform\common or platform\\$_(TGTPLAT).

This command starts the build from the Platform\common phase but does not do a forced recompile.

blddemo -c -qbsp (Rebuild Current BSP and Subprojects)

When to use: When you have not changed _SYSGEN variables (or catalog items) but you have changed code in platform\common or platform\\$_(TGTPLAT) code that needs a forced recompile, such as custom makefile rules.

This command starts the build from the Platform\common phase and does a forced recompile.

Build in the Fastest Way After Changing <X> (Targeted Builds)

Targeted building is the fastest way to perform a build. The goal of a targeted build is to get a specific code change represented in the image as fast as possible. It is especially useful when iteratively developing or debugging. The main challenge in this task is tracing the downstream dependencies of what you are building. The algorithm for tracing these dependencies is represented in Figure 1:

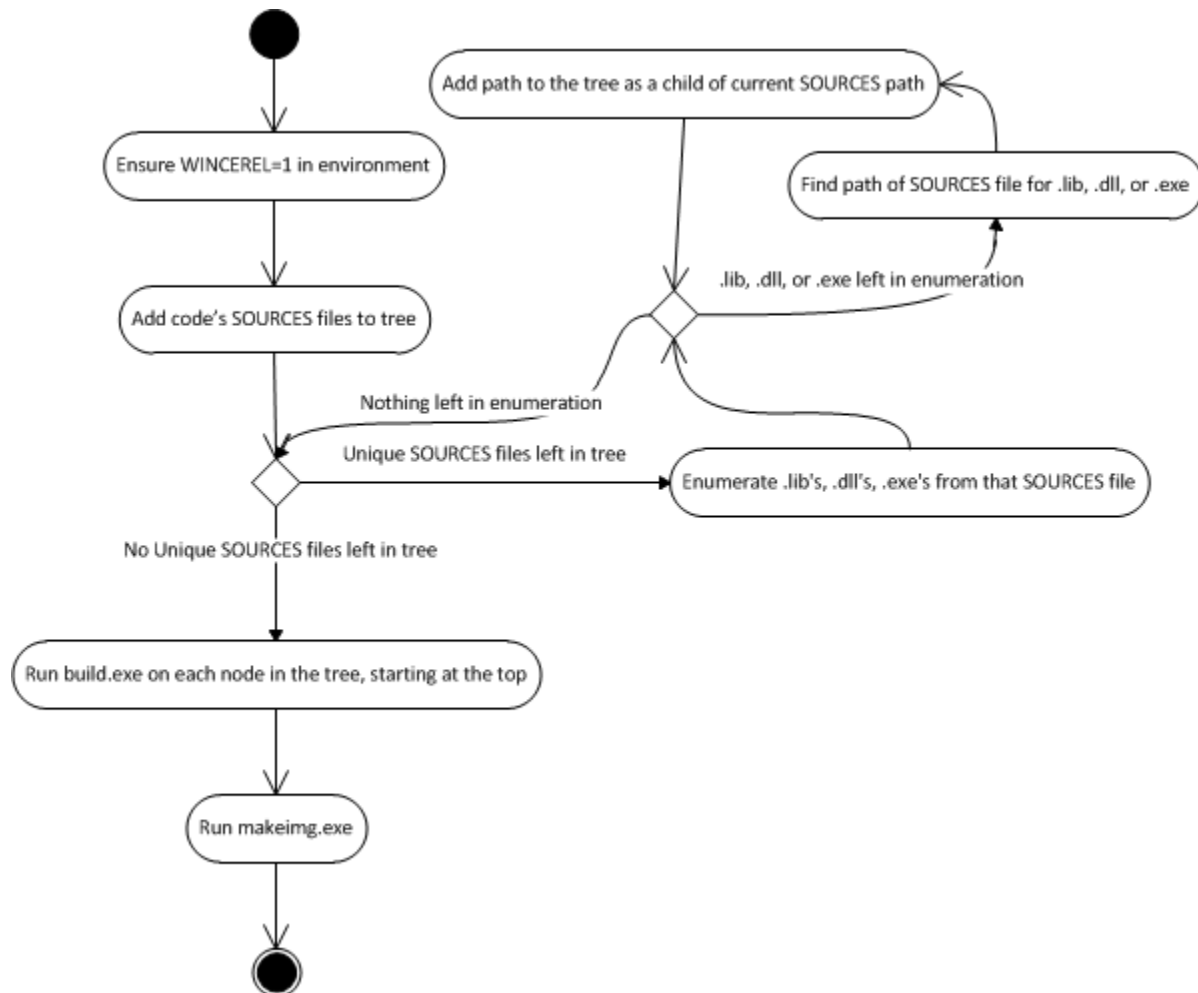


Figure 1. The targeted build preparation algorithm

Finding the path of the SOURCES file for a .lib, .dll, or .exe file can be done from the command line using "findstr /sip library_name SOURCES", or by way of a full text search using Windows Explorer.

If your dependency tree grows complex (10 or more nodes), a targeted build may actually be slower than running the appropriate blddemo command. In the following sections, you will see the targeted build preparation algorithm applied to two substantial examples.

Perform a Targeted Build in the Public Tree

Code in the public tree builds during the Build OS or Sysgen OS step. Thus, a targeted build is not recommended unless you are an expert; the implication is that you have modified code in the public\ or private\ tree instead of cloning it.

When doing a targeted build of public code, the targeted build preparation algorithm needs a modification. First, search other SOURCES files in the \public and \private trees. Then, search the cesysgen\makefiles in those trees. Finally, search SOURCES files in \platform and subprojects.

You can walk through the targeted build preparation algorithm by following the next detailed example. For this example, consider that the changed file is `public\common\oak\drivers\pm\mdd\pmsysstate.cpp`, and that the `$(_TGTPLAT)` is the `TI_EVM_3530` platform.

Looking at `public\common\oak\drivers\pm\mdd\SOURCES`, you see that it produces `pm_mdd.lib`. This becomes the root node of the tree. If you scan other `SOURCES` files in the public and private trees, you see that there are no further dependencies on this `.lib`.

The next step is to look at `public\common\cesysgen\makefile`. There you will find the following rules (some lines omitted for simplicity):

```
pm:: pm_mdd.lib pm_default_pdd.lib pmstubs.lib pm_pdd_common.lib
```

```
pm_mdd pm_default_pdd pmstubs pm_pdd_common:
    @set SOURCELIBS=%SOURCELIBS% $(SG_INPUT_LIB)\$@.lib
```

```
pm:: $(PM_COMPONENTS)
    @set TARGETTYPE=DYNLINK
    @set TARGETNAME=$@
    $(MAKECMD) /NOLOGO $(SG_OUTPUT_OAKTGT)\$@.dll
```

Looking at `public\CEBase\oak\misc\winceos.bat`, you see that `pm_mdd` is a member of `PM_COMPONENTS`. So `pm.dll` depends on the `pm_mdd` rule, which adds `pm_mdd.lib` to its `SOURCELIBS`. Therefore, `pm.dll` depends on `pm_mdd.lib`. The tree now is two nodes large—`pm_mdd.lib`, the parent and `pm.dll`, the child.

After Sysgen OS, the next build steps are `platform\common` and `platform\TI_EVM_3530`. If you scan the `SOURCES` files there, you see one reference to `pm_mdd.lib`: `platform\TI_EVM_3530\src\drivers\pm\pdd\sources`. This `SOURCES` file actually creates `pm.dll` as well, which will override the `pm.dll` that was produced during the Sysgen OS step. So, at the end of this exercise, the targeted build tree looks like this:

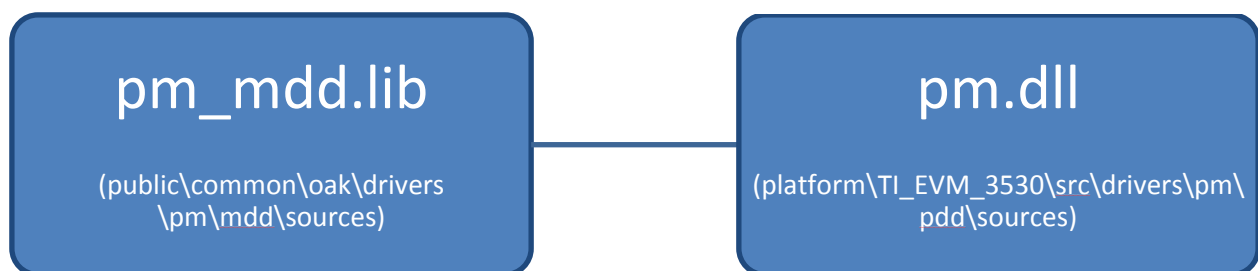


Figure 2. The targeted build tree for the modified `public\common\oak\drivers\pm\mdd\pmsysstate.cpp`

Now that the tree is complete, performing the build is straightforward. First, run `build.exe` in `public\common\oak\drivers\pm\mdd`.

Next, run the `makefile` rule in `public\common\cesysgen\makefile`. To do this, use the `sysgen.bat` utility located in `public\common\oak\misc`. The basic syntax is: `sysgen.bat -p <_DEPTREE> <makefile rule>`. `_DEPTREE` is the value from `$(_DEPTREES)` that corresponds to the directory of the `makefile` under `public\`. So, in this case, the `_DEPTREE` is

'common' and the entire command is: 'sysgen -p common pm'. This runs the 'pm' makefile rule in the 'common' _DEPTREE.

It is important to note that you need to run the cesysgen\makefile rule even though the pm.dll from the Sysgen OS build step is not used by the TI platform. This is because the TI platform does use pm_mdd_lib. When you ran build.exe in public\common\oak\drivers\pm\mdd, pm_mdd_lib.lib was output to public\common\oak\lib\\$_(TGTCPU)\\$(WINCEDEBUG). However, platforms must only link with files in the \$(SG_OUTPUT_ROOT). So you must make sure that pm_mdd_lib gets copied to the \$(SG_OUTPUT_ROOT). Because the pm makefile rule is dependent on pm_mdd_lib, 'sysgen -p common pm' will still accomplish the copy.

The next step is to run build.exe in the platform\TI_EVM_3530\src\drivers\pm\pdd directory. Then you can run makeimg, and the results of the original change to pmsysstate.cpp will be represented in the image.

Putting the theory aside, here is a review of the actual build steps taken:

1. Run build.exe in public\common\oak\drivers\pm\mdd.
2. Run 'sysgen -p common pm'.
3. Run build.exe in platform\TI_EVM_3530\src\drivers\pm\pdd.
4. Run makeimg.exe.

You can see from the complexity of the steps alone why modifying code in the public\ or private\ trees is not recommended. Next, consider the second targeted build preparation algorithm example.

Perform a Targeted Build in the Common or Platform Tree

A targeted build in platform\common or platform\\$_(TGTPLAT) is an easier and more common scenario. First, apply the targeted build preparation algorithm as shown in Figure 1 for a change to

platform\common\src\soc\Freescale\mxarm11_fsl_v1\IPU\BASE\ipu_base.cpp. The SOURCES file in this directory creates ipu_base_mxarm11_fsl_v1.lib. Scanning the SOURCES files in the platform tree, you see four other references to this library:

- IMX313DS\SRC\DRIVERS\IPU\BASE\sources
- IMX313DS\SRC\DRIVERS\IPU\CAMERA\sources
- IMX313DS\SRC\DRIVERS\IPU\PF\sources
- IMX313DS\SRC\DRIVERS\IPU\PP\sources

Checking each of these SOURCES files, you see that all four produce a .dll file. So, the targeted build tree looks like Figure 3:

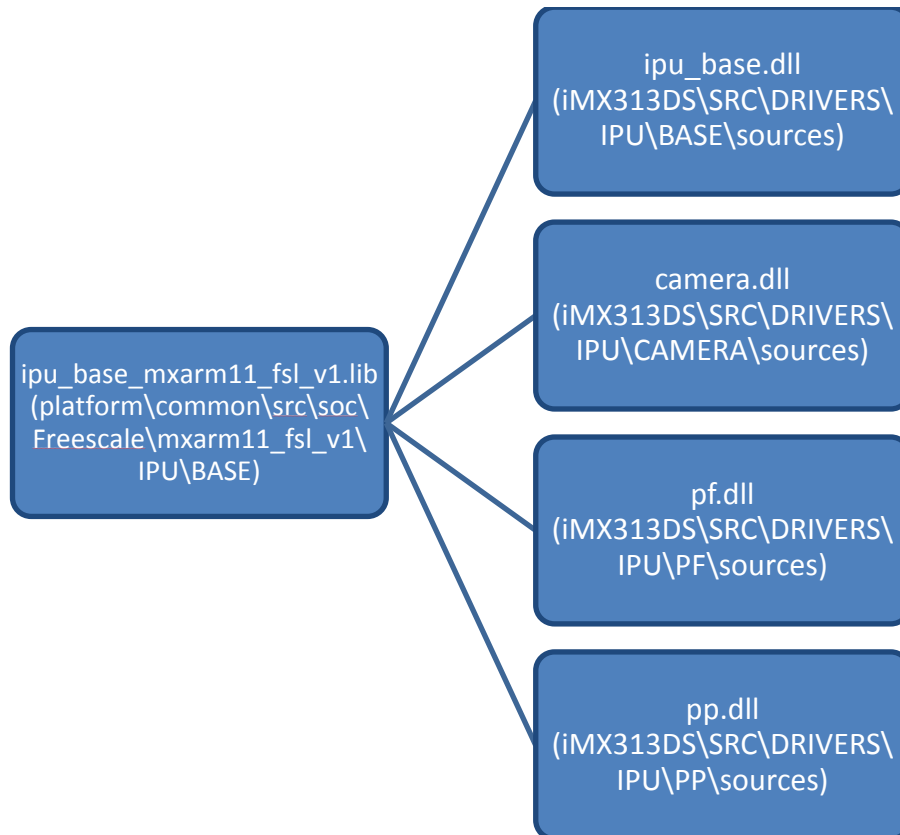


Figure 3. The targeted build tree for modified platform\common\src\soc\Freescale\mxarm11_fsl_v1\IPU\BASE\ipu_base.c pp

In this case, the build steps are:

1. Run build.exe in platform\common\src\soc\Freescale\mxarm11_fsl_v1\IPU\BASE.
2. Run build.exe in each of the four leaf directories.
3. Run makeimg.exe.

With practice, performing targeted builds will become second nature and you will drastically reduce the amount of time spent on builds.

Modify Code from the Public Tree by Cloning into platform\\$_(TGTPLAT)

To avoid modification of the public shared code, you can create a clone of the directory you want to modify in the platform tree. Building the code in the platform tree offers you a few advantages. First, it preserves the public tree, thereby avoiding a reinstallation of Platform Builder to restore the original version. Second, it makes targeted builds simpler because the code is built later in the build process and avoids going through the Sysgen OS phase. Finally, if the cloned code breaks, you can likely complete the build using the original version as a temporary solution.

Begin cloning by copying the code that you want to modify into a new directory in platform\\$_(TGTPLAT), along with its SOURCES file. Now, look at the dependencies using

the targeted build preparation algorithm from Figure 1. Each dependency before the `platform\$_TGTPLAT` step must be copied as well.

After copying the files, begin modifying the SOURCES files. Typically, references in the SOURCES files to `$_COMMONOAKROOT`, `$_COMMONSDKROOT`, and `$_COMMONDDKROOT` must be changed to `$_SYSGENOAKROOT`, `$_SYSGENSDKROOT`, and `$_SYSGENDDKROOT`, respectively. This will include and link files against the Sysgen output directory `$(SG_OUTPUT_ROOT)` instead of the Build OS phase directories. The `RELEASETYPE` must be changed from `OAK` to `PLATFORM` to reflect the new build location. You will also need to examine the `cesysgen\makefile` in the corresponding public code to make sure that any special rules are represented.

The next example demonstrates cloning the code in `public\common\oak\drivers\nleddrvr\mdd` into the `TI_3530_EVM` platform.

It is important to do some research up front to find out how this code is built and which SOURCES files and `cesysgen\makefiles` consume the output. By reviewing the SOURCES files in the directory that you want to clone, you see that `nleddrvr_mdd.lib` is linked. You need to update anything that consumes this library to instead consume the cloned version. Using the command `"findstr /sip nleddrvr sources makefile"` in the `public\` and `platform\TI_3530_EVM\`, you can create a list of what needs to be examined:

- `public\common\oak\drivers\nleddrvr\lib\SOURCES`
- `public\common\oak\drivers\nleddrvr\dll\SOURCES`
- `public\common\cesysgen\makefile`

Looking at the first occurrence, `nleddrvr_mdd.lib` is linked into `nleddrvr_lib.lib`. Because `nleddrvr_lib.lib` is a library, it can potentially be linked into other downstream binaries, so you need to use `findstr` to do the same enumeration on it:

- `public\common\cesysgen\makefile`
- `public\common\oak\drivers\nleddrvr\dll\SOURCES`
- `platform\TI_3530_EVM\src\drivers\nled\SOURCES`

Now you can examine the full listing of files that use `nleddrvr_mdd.lib` and `nleddrvr_lib.lib`. Looking at `public\common\oak\drivers\nleddrvr\dll\SOURCES`, you can see that this creates `nleddrvr.dll`, linking both libraries along with a stub `pdd` library. Because `platform\TI_3530_EVM\src\drivers\nled\SOURCES` also links `nleddrvr.dll` in a later build step, it will replace the one created from the public tree. Thus, `public\common\oak\drivers\nleddrvr\dll` is now accounted for, and you are free to focus on the other libraries in the build.

In `public\common\cesysgen\makefile`, you see several references to `nleddrvr_mdd` and `nleddrvr_lib`. First is an inclusion of makefile rules for both libraries as a dependency of the `driverlibs` rule. `Driverlibs`, as seen in the makefile, is a rule that is always run during the preprocessing phase of Sysgen. You can also see that each library has a rule to copy itself to `$(SG_OUTPUT_OAKLIB)`. You must make sure that any SOURCES files that consume `nleddrvr_lib` from `$(SG_OUTPUT_OAKLIB)` instead consume the cloned library. Fortunately, there is only one real consumer:
`platform\TI_3530_EVM\src\drivers\nled\SOURCES`.

Now that you have done the research, you can perform the actual implementation. Start by copying the code found in `public\common\oak\drivers\nleddrvr\mdd` to a new directory in the platform. Create the `platform\TI_EVM_3530\src\drivers\nledlib` directory and copy all of the files in `public\common\oak\drivers\nleddrvr\mdd` into it. Then modify

platform\TI_EVM_3530\src\drivers\dirs, adding an entry for nledlib just before the nled entry. By adding it here, you pick up the appropriate conditioning on CE_MODULES_NLEDDRVR (learn more on why this is correct in the [Filter BSP DIRS, .bib, and .reg Files Using CESYSGEN Variables](#) section). This conditioning ensures that NLED dependencies such as ceddk are included, but it also includes nleddrvr_stubpdd.lib, which you may need later.

Next, modify platform\TI_EVM3530\src\drivers\nledlib\SOURCES to match the fact that it is building in the platform directory. You pick up RELEASETYPE=PLATFORM from platform\TI_EVM3530\sources.cmn, so you don't need to add it here. What you will need to change is the DEFFILE and WINCETARGETFILE0 macros, which presently refer to a .def file in the public\ tree. If the .def file has CESYSGEN filtering in it, it is best to repoint it to the .def file in the \$(SG_OUTPUT_ROOT), so that you can pick up that filtering. Looking at platform\TI_3530_EVM\src\drivers\nled\SOURCES, it does not have filtering and has already been cloned. So you can remove the TARGETDEFNAME, DEFFILE, and WINCETARGETFILE0 macros, leaving a simple SOURCES file.

The last step is to point platform\TI_3530_EVM\src\drivers\nled\SOURCES at the new nleddrvr_mdd.lib. This SOURCES file links against nleddrvr_lib.lib, which includes nleddrvr_mdd.lib (which you have cloned) and nleddrvr_stubpdd.lib (which you have not cloned). So the new SOURCELIBS must look like this:

```
$(PLATLIB)\$(CPUINDPATH)\nleddrvr_mdd.lib \
$(SYSGENOAKROOT)\lib\$(CPUINDPATH)\nleddrvr_stubpdd.lib \
```

This picks up the cloned code from the \$(PLATLIB) directory, which is what the output directory build.exe uses for RELEASETYPE=PLATFORM and TARGETTYPE=LIBRARY. It also picks up the nleddrvr_stubpdd.lib file from the SYSGEN output directory.

Add a Custom Makefile Rule

In some cases, SOURCES macros aren't enough to cover what you want to do when build.exe runs on a directory. Maybe you want to copy some files or run an external program as part of the build. For these situations, you can create a custom makefile rule that does what you want.

One common usage of custom makefile rules is to turn a compiled bootloader into a binary image file that can be downloaded and flashed to a target device. The next example demonstrates how to do this in platform\SMP865x\src\bootloader\base (a MIPSII platform).

In the SOURCES file, the WINCETARGETFILES macro is used to call out a makefile rule named BootImage. To avoid editing the makefile directly, specify the rule in makefile.inc, which will be automatically included by makefile.def.

Looking at makefile.inc in this directory, you see that there is only one rule, which is a series of simple commands that creates the boot.bin and boot.sre files. This rule could just as easily copy files, delete files, or do anything else that you desire.

If you need a rule to run before the compile phase, use WINCETARGETFILES0 instead.

Filter BSP DIRS, .bib, and .reg Files Using CESYSGEN Variables

Properly filtering DIRS, .bib and .reg files by using CESYSGEN variables ensures that a BSP works, regardless of what SYSGEN or catalog items are selected. This is an important aspect of constructing a robust BSP.

While you will probably only need to write a few lines of filtering markup per driver or component, determining the correct lines is not trivial. The algorithm is as follows. For each driver/platform-specific component:

1. Look at the SOURCES files in the component for libraries that the component consumes.
2. Look at the cesysgen\makefiles in the public tree to determine the makefile rules that govern the libraries' inclusion in \$(SG_OUTPUT_ROOT).
3. Look at the SYSGEN rules in public\cebase\oak\misc to determine which modules cause the makefile rules to fire.

The next example applies this algorithm to platform\iMX313DS\src\drivers\sdhc. The SOURCES file here links with sdcardlib.lib, sdhclib.lib, and sdbus.lib from \$(_SYSGENOAKROOT). You must add a condition on MODULES variables that bring these libraries in. You must also investigate sdhc_mxarm11_fsl_v1.lib and make sure that it does not depend on any additional MODULES. sdhc_mxarm11_fsl_v1.lib is built in platform\common\src\soc\Freescale\mxarm11_fsl_v1\SDHC. Fortunately, it does not link with any libraries, so you add no conditioning.

Next, look at which makefile rules bring in sdcardlib.lib, sdhclib.lib, and sdbus.lib. Each of these libraries has a makefile rule to bring it in that corresponds with its name: sdcardlib, sdhclib, and sdbus. In the makefile, the sdcardlib and sdhclib rules are set by default in the driverlibs rule, so they are always available, regardless of SYSGEN settings. However, sdbus is not part of driverlibs or any other automatic inclusion rule (take care not to confuse sdbus.lib with sdbus_lib.lib, the latter of which is included); therefore, you must look for an sdbus module being set in public\cebase\oak\misc.

Looking at public\cebase\oak\misc\winceos.bat, sdbus is added to the list of CE_MODULES when SYSGEN_SDBUS==1, so you need to condition the driver on CE_MODULES_SDBUS. This conditioning must be present in three places:

1. platform\iMX313DS\src\drivers\dirs:


```
# @CESYSGEN IF CE_MODULES_SDBUS
SDHC      \
# @CESYSGEN ENDIF CE_MODULES_SDBUS
```

Condition the building of the sdhc directory. If you don't condition at this point, then the build will fail when SYSGEN_SDBUS is not set. The link phase will fail when build.exe tries to build the sdhc directory and finds that sdbus.lib is not available.

2. platform\iMX313DS\files\platform.bib:


```
; @CESYSGEN IF CE_MODULES_SDBUS
sdhc.dll      $(_FLATRELEASEDIR)\sdhc.dll          NK SHK
; @CESYSGEN ENDIF CE_MODULES_SDBUS
```

Condition the inclusion of sdhc.dll. If you don't condition at this point, the build will attempt to include sdhc.dll, even when it has not been built (per the conditioning in the DIRS file). The makeimg phase will fail because sdhc.dll cannot be found.

3. platform\iMX313DS\files\platform.reg:
; @CESYSGEN IF CE_MODULES_SDBUS
#include "\$(_TARGETPLATROOT)\SRC\DRIVERS\SDHC\sdhc_arm11.reg"
; @CESYSGEN ENDIF CE_MODULES_SDBUS

Condition the registry specific to sdhc.dll. If you don't condition at this point, you will unnecessarily bloat the registry, and will also try to load sdhc.dll when the image loads. This will not cause image boot failure, but it will consume extra cycles during boot.

Use Release Directory Modules

Recall from the section titled [Build in the Fastest Way After Changing <X> \(Targeted Builds\)](#) that the last step of performing a targeted build was typically to run makeimg.exe. Running makeimg.exe generates a new image that includes the targeted updates to .exe and .dll files. However, using a special functionality of Platform Builder, you can avoid this step and save even more time.

Using Release Directory Modules is the last step of a truly fast targeted build. In the IDE, click Target->Release Directory Modules and add modules for Platform Builder to load directly from the Release Directory, as opposed to loading from the image. For example, if you added sdhc.dll to the Release Directory Modules, you could perform a targeted build of sdhc.dll and add sdhc.dll to the Release Directory Modules list. Then, to reload the sdhc.dll driver with the updated code, only a reset is necessary.

Sdhc.dll typically stays resident in memory from the time of device boot until device shutdown. The Release Directory Modules functionality is even more useful for modules that are loaded and unloaded from memory, because the updated code is reflected on the device without even a reboot of the device.

Conclusion

By using the appropriate build command for each situation, you can avoid both unnecessary rebuilding and stale binaries. To fast-track the reflection of specific code changes in your image, you can perform targeted builds using the targeted build preparation algorithm. Cloning public shared code for modification can prevent reinstallations of Platform Builder and provide you with a backup solution if problems occur with modifications. You can use custom makefile rules to gain precise control over building actions performed by build.exe. By properly filtering DIRS, .bib and .reg files, you can create robust BSPs that work independent of selected SYSGEN or catalog items. By using Release Directory Modules, code changes in some modules can be reflected without rebooting your device.

As you develop your Windows Embedded Compact 7 OS design, using these building strategies can streamline your builds and help to shorten your development time.

Additional Resources

- [Windows Embedded Compact 7 Build Process](http://go.microsoft.com/fwlink/?LinkId=208071) (http://go.microsoft.com/fwlink/?LinkId=208071)
- [Windows Embedded website](http://go.microsoft.com/fwlink/?LinkId=183524) (http://go.microsoft.com/fwlink/?LinkId=183524)
- [Windows CE Base Team Blog](http://go.microsoft.com/fwlink/?LinkId=205449) (http://go.microsoft.com/fwlink/?LinkId=205449)

Copyright

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.