Over Air Download
For CC2530 SoC

Document Number: SWRA257

Version 1.3

TABLE OF CONTENTS

TABLE OF FIGURES

1. Purpose

The purpose of this document is to provide a developer's guide to enable the proprietary TI OAD functionality in any sample or proprietary Z-Stack Application using the CC2530.

2. Functional Overview

OAD is an extended stack feature (US Patent 7814478) provided as a value-enhancing solution for updating code in fielded devices without the cost of physically accessing them. OAD is effected as a managed client-server mechanism which requires three logical components:

1. A **Commissioner** component to initiate and control the deployment of images to devices and to activate them when appropriate.
2. A **Client** component on any device that is to receive and activate a new image.
3. A **Server** component to supply client devices with a new image.

3. Assumptions

1. Off-chip NV assumes the use of the Numonyx NV storage device, connected by SPI to the CC2530 SoC, and provided by reference design of the SmartRF05 board.

4. Acronyms

Term	Definition
NV	**N**on-**V**olatile (memory that persists through power cycles.)
OAD	Proprietary Texas Instruments **O**ver **A**ir **D**ownload
OTA	ZigBee Alliance **O**ver **T**he **A**ir download

5. References

[1] Z-Stack Developer's Guide (SWRA176)

[2] Z-Stack User's Guide For SmartRF05EB and CC2530 (SWRU189)

6. Revision History

Date	Revision	Description of changes
11/07/08	0.1	New document – used "OAD for MSP430" as a template.
02/18/09	0.2	Internal only with minimal update to enable testing of a 2.2 Beta release.
03/10/09	0.3	Finish updates for OAD in 2.2 release code.
04/03/09	1.0	Finalize for release.
02/17/10	1.1	Add requirement to define MAKE_CRC_SHDW.
07/28/10	1.2	Emphasized that this is the proprietary TI OAD and removed obsolete references.
11/11/10	1.3	Updated "patent pending" statement, removed use of OTA term for OAD functionality.

7. Design Constraints

7.1 External Constraints / Features

An off-chip NV device must be used to store the new OAD image when code size exceeds the available internal flash[1] - the means by which this occurs is beyond the scope of this document.

7.2 Internal Constraints / Requirements

The OAD image must be a complete and integral Z-Stack Application (i.e. only sending the stub Application layer is not supported.)

8. Design

8.1 OAD Context

The OAD system is comprised of two images: the 'Boot Code' and the Z-Stack with the OAD Application (as well as any other application(s)) – the 'Application Code'. The placement of each of the two images into the internal flash is handled by the unique IAR linker command file used by each.

8.2 Functional Description

8.2.1 Boot Code

The OAD solution requires the use of boot code to check the integrity of the active image before jumping to it. This check guards against an incomplete or incorrect programming of the active image. The OAD boot code provides the following functionality:

1. Boot Code will be the target of the reset vector and therefore contain startup code.
2. When the active image indicates, Boot Code will program the download image into the active image area and will thusly complete the final step of an OAD process: code instantiation.
3. Boot Code will guard against interrupted programming of the active image area by checking the validity of the active image. If the image is not valid the boot code will again program the download image into the active image area.
4. The Boot Code requires the first flash page so that it can intercept the startup vector.
5. The Boot Code can only be physically downloaded.

[1] The available internal flash is half of the subtotal of the total flash pages less the page used by the OAD boot code and less the pages reserved for NV and less the last page used for flash lock bits.

8.2.2 OAD Application

The OAD Z-Stack component is implemented as a standard ZigBee Application:

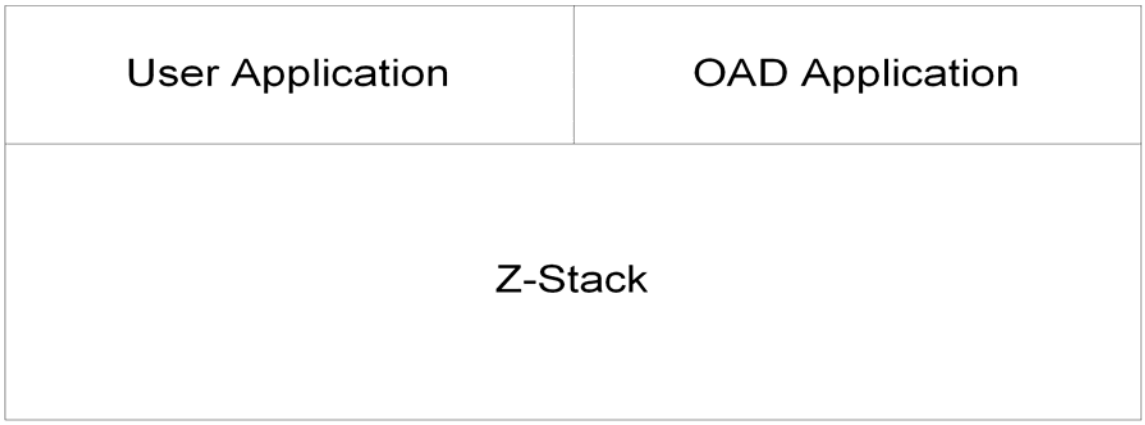

Figure 1: Architectural Placement of the OAD Z-Stack Component.

Thus, the OAD Application must be included in the build of the sample or proprietary application in order to enable OAD functionality (the Client and Server logical components only – the Commissioner component is implemented in a ZigBee tool such as Daintree Professional as well as in the Dongle sample application which is included with the Z-Stack distribution.)

9. Producing OAD Boot Code to be programmed.

9.1 Separate Build & Debug of Boot Code

The Boot Code is separately built and debugged or programmed via the IAR IDE by opening the OAD Boot Project here:

$INSTALL_DIR$\Projects\zstack\Utilities\OAD\CC2530DB\Boot.eww

The default configuration is with the download option to erase flash in order to start a CC2530 SoC with clean flash (and thus clean NV). Before debugging or physically programming the OAD Application code produced in the next section, this OAD Boot code must first be programmed into the flash (but only this once, since, as the following section mentions, the default option for application code is to preserve this OAD Boot code on successive debugging or programming.)

10. Producing OAD Application Code to debug or send OAD.

The “RouterEB” build of the Z-Stack sample application known as GenericApp is used below for demonstration purposes only - the Customer would apply the following steps in her own, proprietary Z-Stack application and make the corresponding changes to all of the paths below that are specific to GenericApp. The RouterEB is also used below for demonstration only – these same steps apply to any of the CC2530 targets supported by the Z-Stack release. It is only requisite that the paths specific to the RouterEB target be changed accordingly.

10.1 Configure linker options for the OAD functionality.

10.1.1 Configure the linker to generate extra output.

Check the checkbox to “Allow C-SPY-specific extra output file” as shown below.

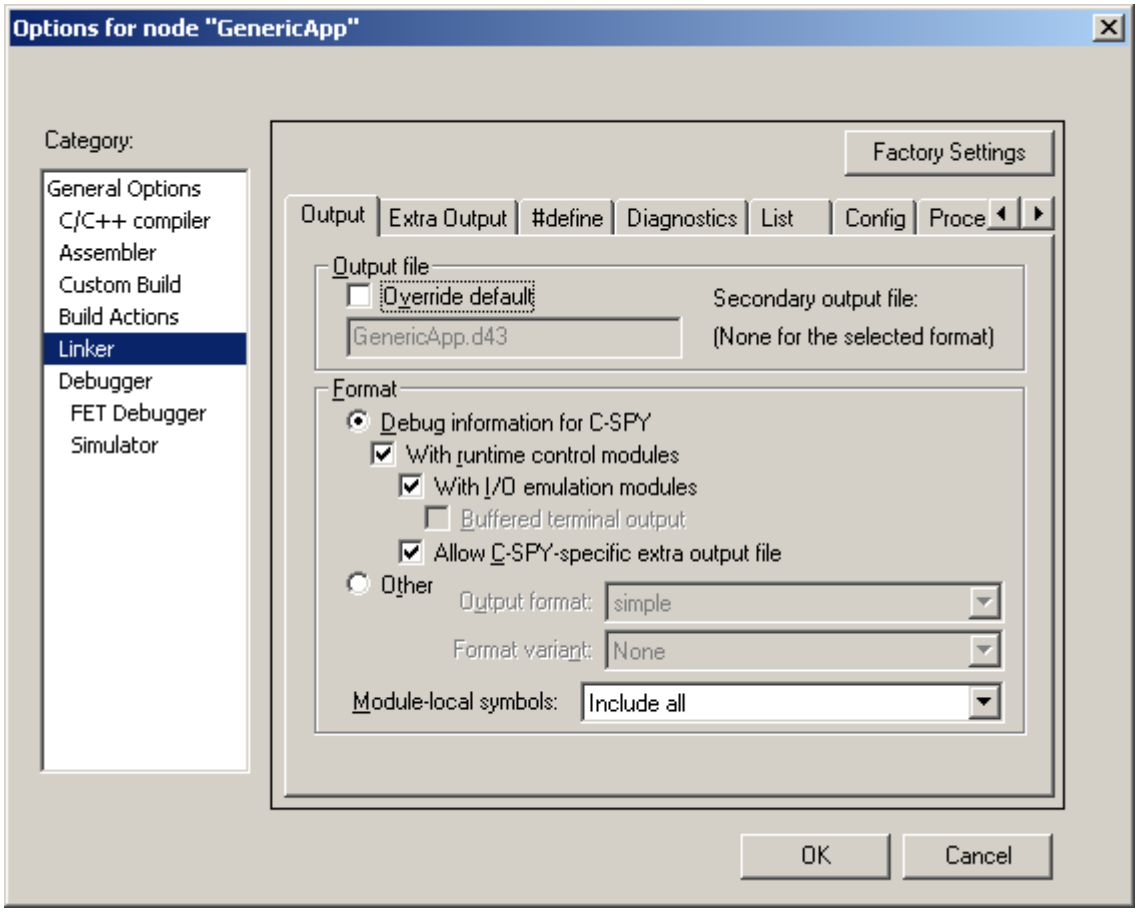

Figure 2: Configuring the linker to generate an extra output file.

10.1.2 Configure the linker extra output file format.

Check the checkbox to “Generate extra out file” and choose the “Output format:” as *simple-code* as shown below.

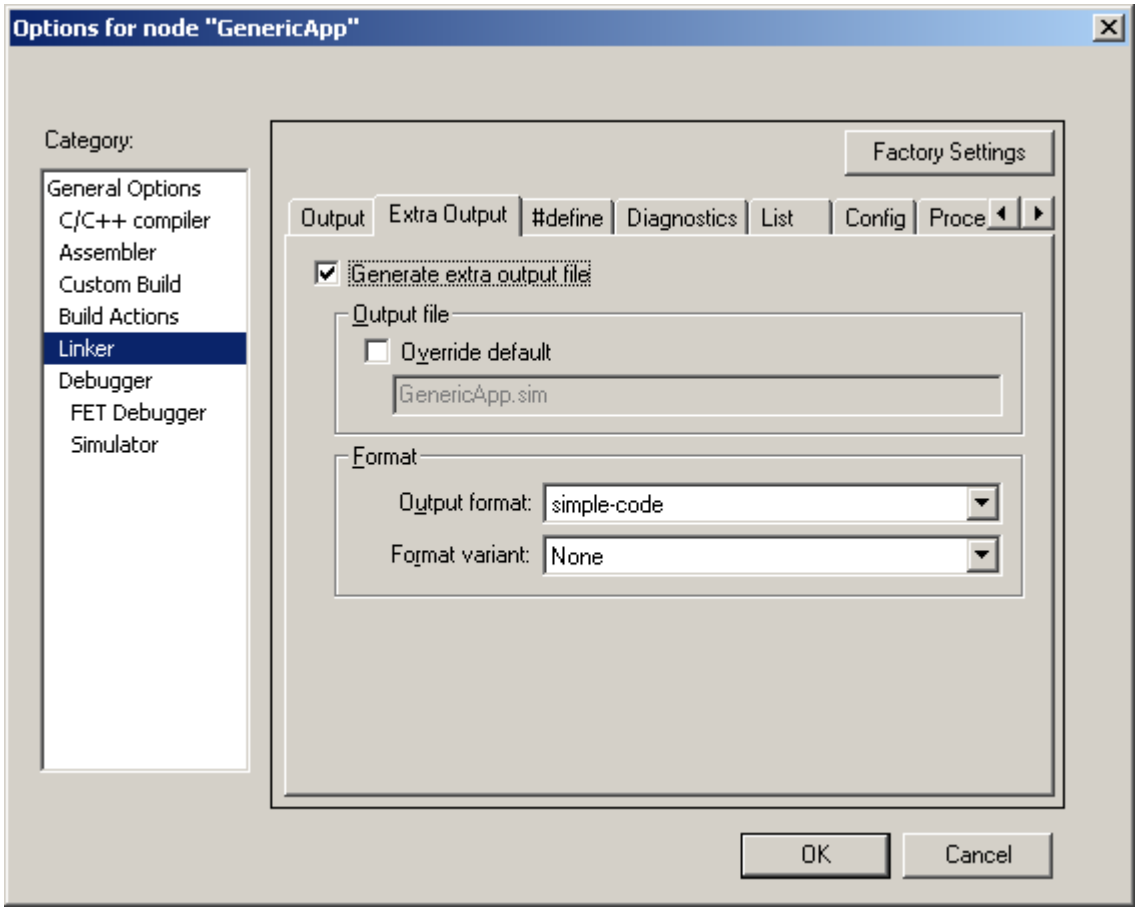

Figure 3: Configuring the linker extra output file format.

10.1.3 Configure the linker command file for OAD-compliant mapping.

Use the following line for the “Override default” command string:

```
$PROJ_DIR$\..\..\..\Tools\CC2530DB\oad.xcl
```

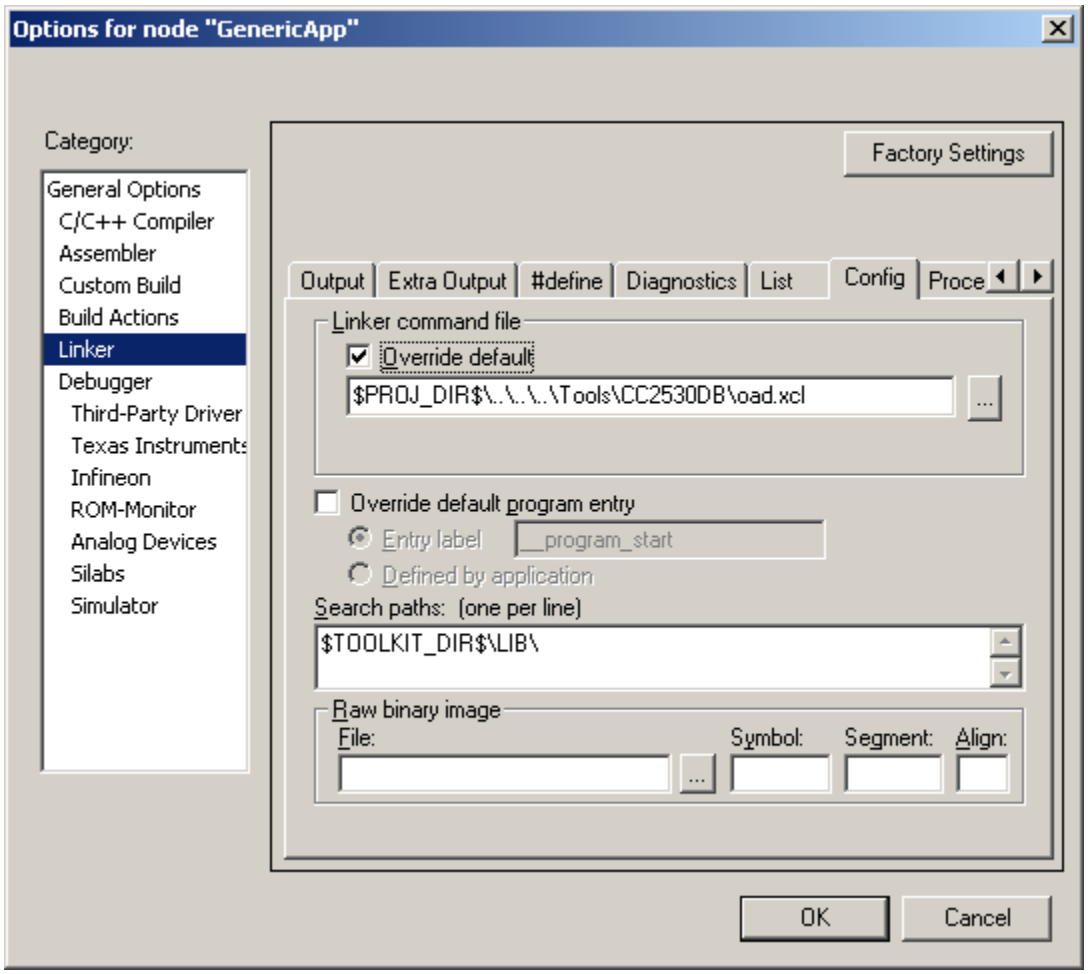

Figure 4: Changing the linker command file to implement OAD-compliant mapping.

10.2 Configure build actions to invoke the OAD post-processing tool.

Use the following line for the “Post-build command line:”

```
"$PROJ_DIR$\..\..\..\Tools\CC2530DB\oad.exe"
"$PROJ_DIR$\RouterEB\Exe\GenericApp.sim"
"$PROJ_DIR$\RouterEB\Exe\GenericApp.bin"
```

The above lines must be pasted as a single line into the dialog box with one space separating each block in parenthesis.

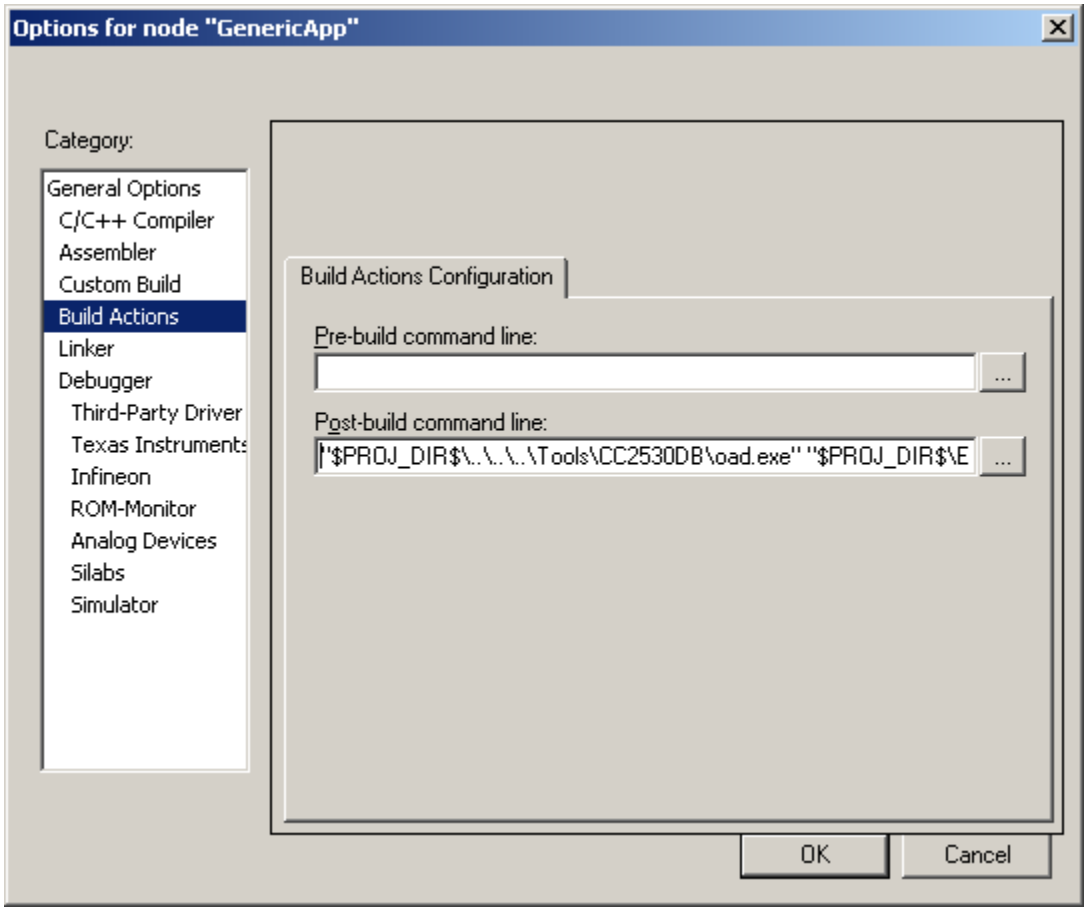

Figure 5: Configuring the build actions to invoke the OAD post-processing tool.

10.3 Configure C/C++ Compiler Defines for OAD.

Add the following to the C/C++ Compiler Preprocessor defined symbols:

`MAKE_CRC_SHDW`

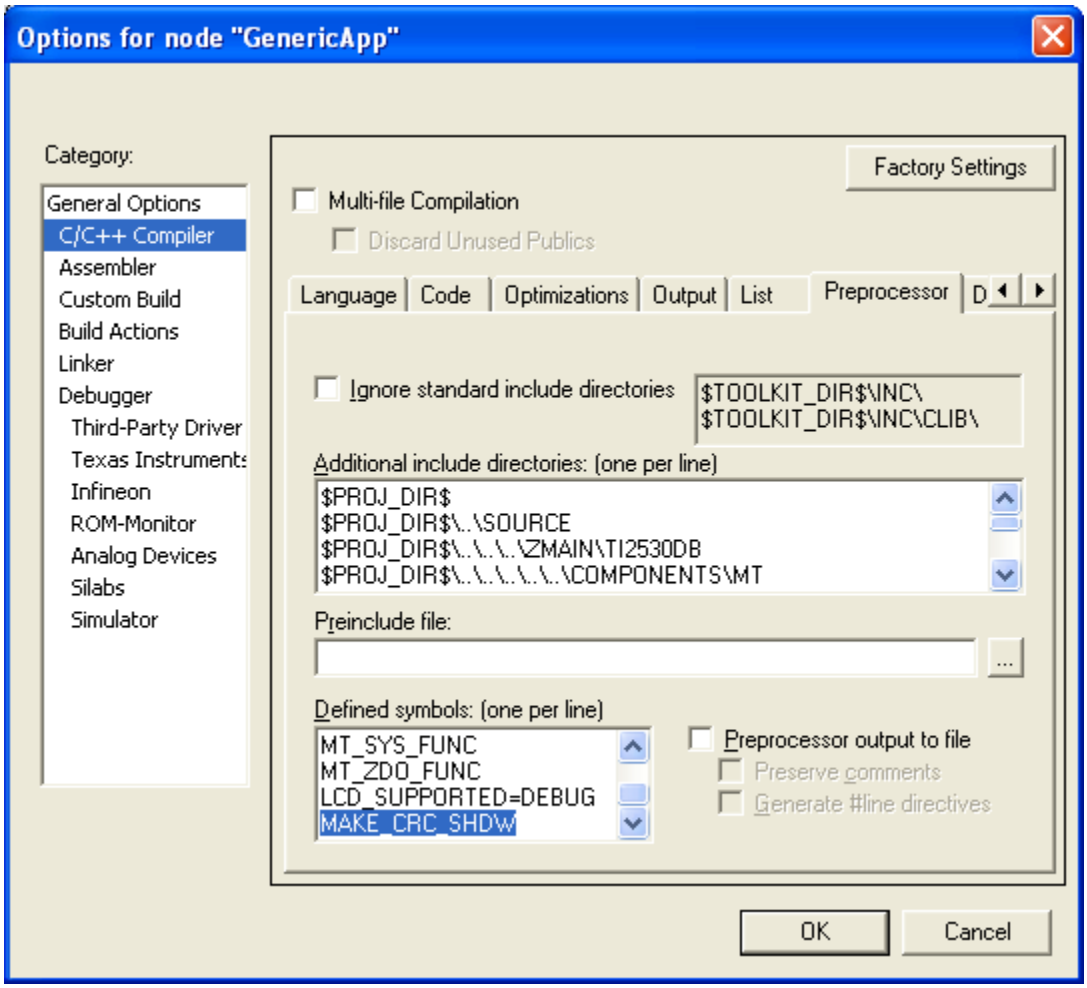

Figure 6: Configuring the C/C++ Compiler options.

10.4 Add the OAD Application to the IAR Project.

10.4.1 Add three OAD Application Source files that are found here:

```
$INSTALL_DIR$\Projects\zstack\Utilities\OAD\Source
```

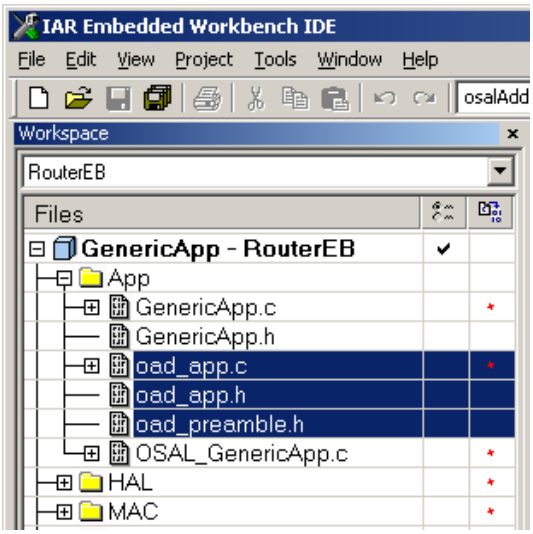

Figure 7: Adding the OAD Application source files to the IAR Project.

10.4.2 Define Proprietary Preamble Fields.

There are six bytes available for unique identification of an image in the file *oad_preamble.h*. Although the macros are given names in the header file, the fields carry no semantics other than the object type (uint16). The fields are identifying information that may be used in any way desired for identification purposes.

10.4.3 Add the OAD OSAL components.

Edit the *OSAL_GenericApp.c* file as follows:

1. Add the following two external function declarations as shown in hi-light:

```
extern void oadAppInit(uint8 id);
extern uint16 oadAppEvt(uint8 id, uint16 evts);
```

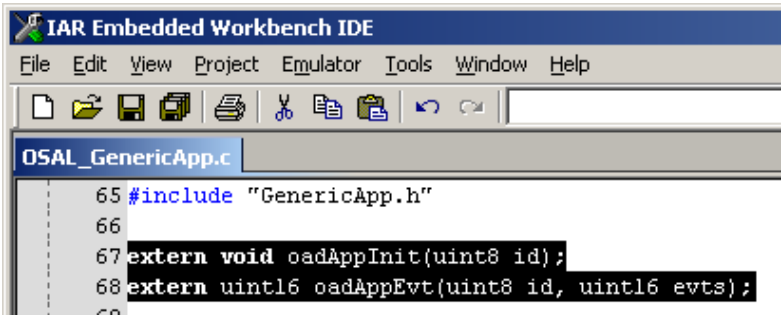

Figure 8: Adding external function declarations to OSAL_GenericApp.c.

2. Add the following line to the data structure *tasksArr[]* exactly as shown in hi-light.

```
oadAppEvt,
```

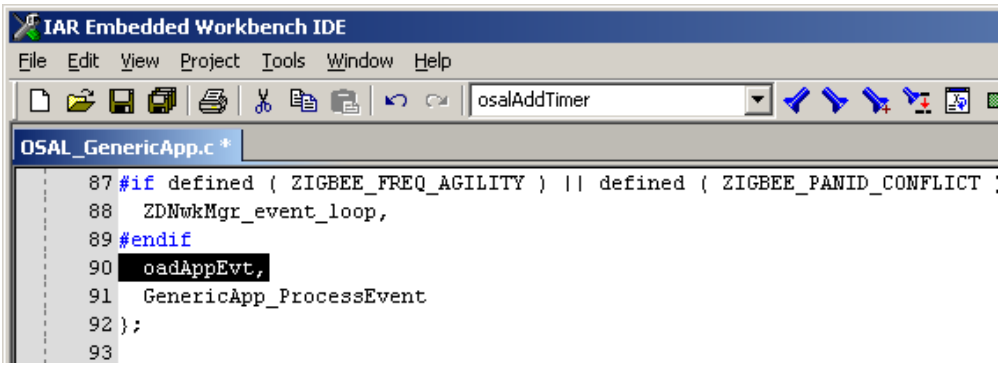

Figure 9: Adding the OAD event to OSAL.

3. Add the following line to the function *osalInitTasks()* exactly as shown in hi-light.

```
oadAppInit(taskID++);
```

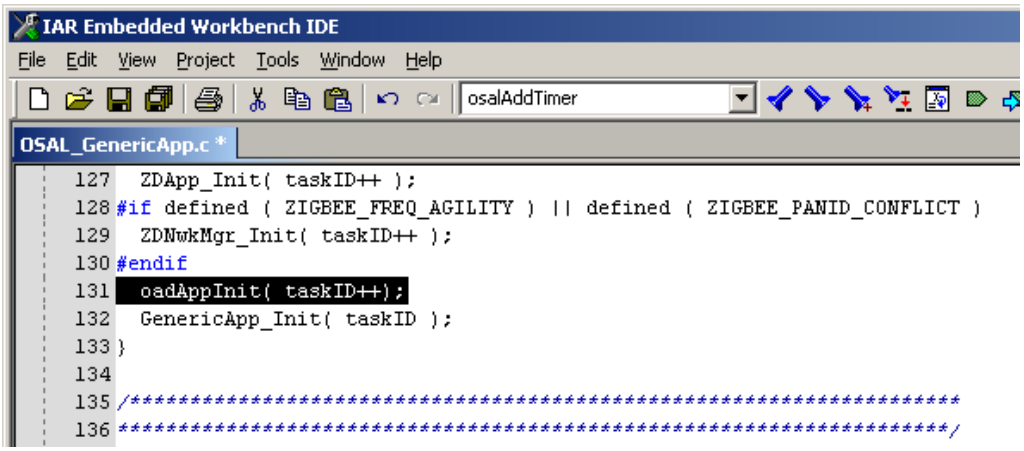

Figure 10: Adding the OAD initialization function to OSAL.

10.4.4 Add the OAD HAL component.

Add one OAD HAL file that is found here:

`$PROJ_DIR$\..\..\..\..\..\Components\hal\target\CC2530EB\hal_oad.c`

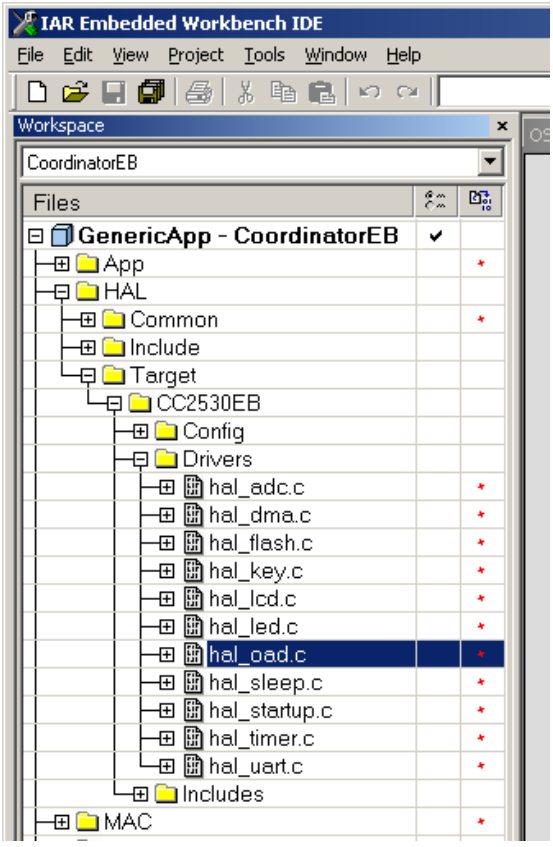

Figure 11: Adding the OAD HAL file to the IAR Project.

10.5 Building the Application Code.

Simply build from the IAR IDE as you normally would.

The binary file produced, which is to be sent over-the-air, is found here:

$PROJ_DIR$\RouterEB\Exe\GenericApp.bin

10.6 Debugging the Application Code.

In order to run or debug the Application Code, a Boot Code image must have already been downloaded to the CC2530 SoC (see the previous section.) So as not to destroy the Boot Code image, preserve the space by checking the “Retain unchanged memory” option as follows:

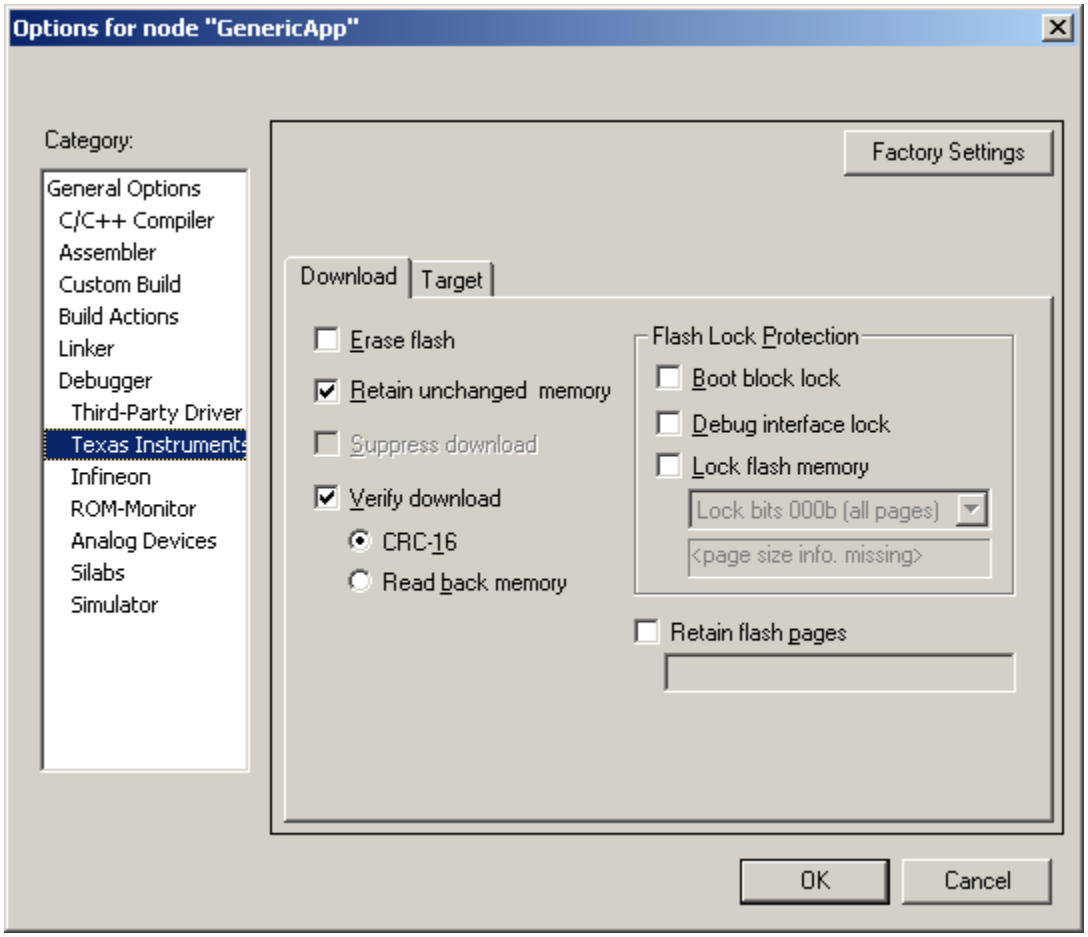

Figure 12: Preserving boot code while debugging.

Now simply build and debug the code from the IAR IDE (running with breakpoints, etc.) as you normally would.

11. Producing OAD Application Code with Boot Code to be programmed.

For mass-production programming, it will be important to have a single image containing both the OAD Boot and Application code so that the part must only be programmed once. The following example assumes that the SmartRF Programming Tool will be used for programming an Intel-hex formatted file into the CC2530 SoC.

11.1 Build the Application Code hex image.

11.1.1 Configure the linker to generate Intel-hex output.

Check the checkbox to “Override default” and make the suffix “.hex” Also check the radio button for “Other” Output file Format and choose the Output format drop-down selection for ‘intel-extended’ as shown below.

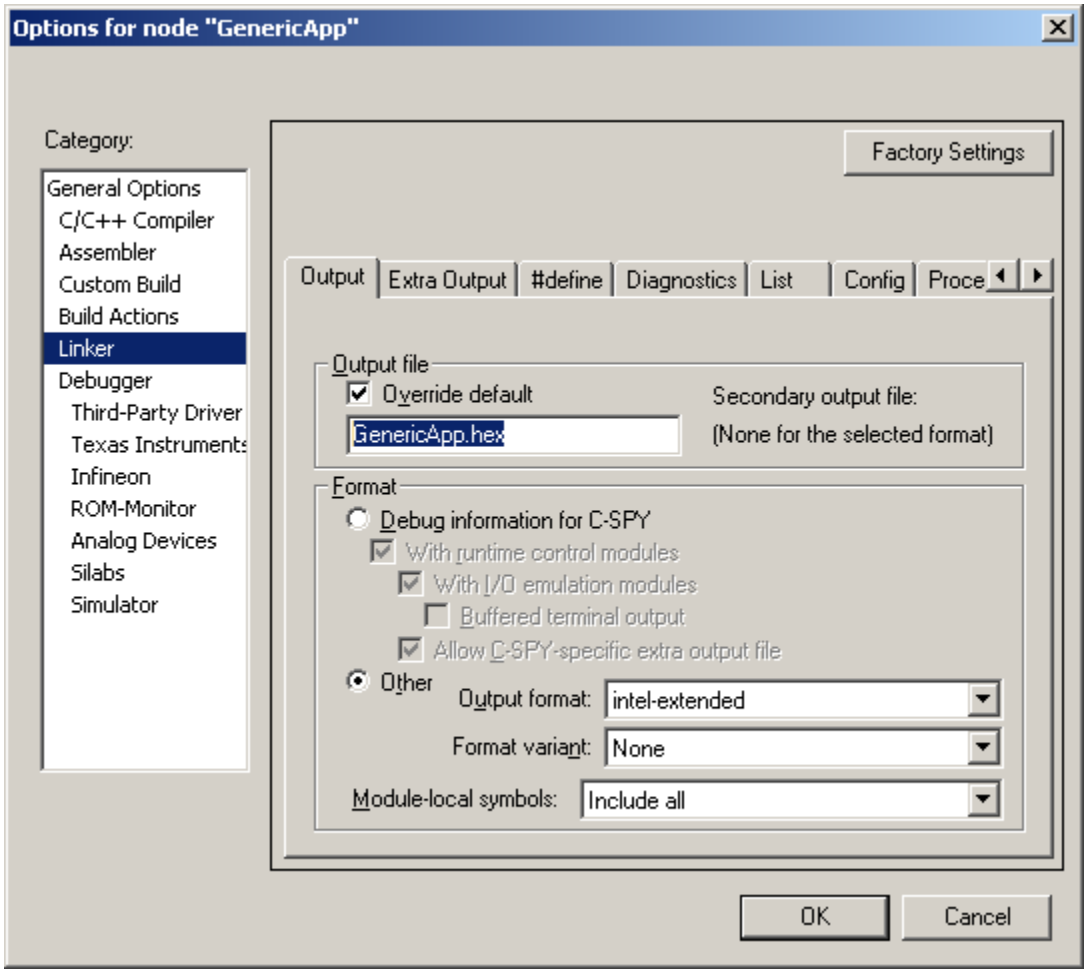

Figure 13: Configuring the linker to generate Intel-hex output.

11.1.2 Configure the linker control file to generate output compatible for the SmartRF Programmer tool.

Remove the comments from the –M option in oad.xcl as shown in hi-light.

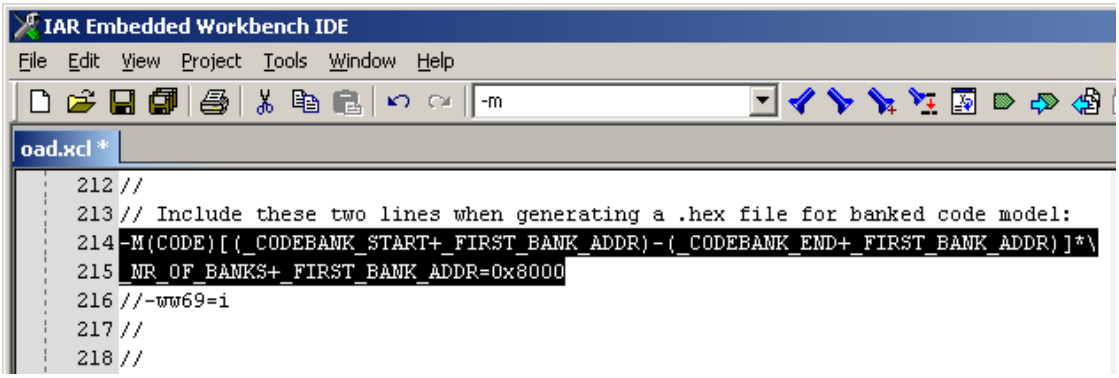

Figure 14: Enabling –M option for SmartRF Programmer tool.

11.1.3 Re-build the Application Code to generate the .hex file.

Having already been built, just pressing the 'F7' key and linking will be sufficient

11.2 Pre-pend the Boot Code hex image to the Application Code hex image.

1) Use any text editor to open the Application Code file produced here:

   $PROJ_DIR$\RouterEB\Exe\GenericApp.hex

2) Delete this first line from the file:

   :020000040000FA

3) Use any text editor to open the Boot Code file produced here:

   $PROJ_DIR$\OAD-Boot\Exe\Boot.hex

4) Delete these last two lines from the file:

   :040000050000079E52

   :00000001FF

5) Copy the edited contents of the Boot Code file to the top of the Application Code file and save it.

6) Use the SmartRF Programmer to install the edited Application Code hex image into the CC2530 SoC.

12. Producing OAD Dongle Code.

12.1 Separate Build & Debug of OAD Dongle Code

The OAD Dongle is just a Z-Stack Sample Application that is built and debugged or programmed like any other; the project is found here:

$INSTALL_DIR$\Projects\zstack\Utilities\OAD\CC2530DB\Dongle.eww

A PC Tool is required to drive the OAD Dongle via the serial port and effect the 'Commissioner' component of the OAD trilogy; it is found here:

$INSTALL_DIR$\Tools\ZOAD\ZOAD.exe

13. OAD High Level Customer Interaction

13.1 OAD Event Callback - *optional*.

Significant time is required to store the new image as it is downloaded. Typical programming times for flash parts are on the order of 10-15 milliseconds per page. So to help maintain stability, the capability is provided for an application to be alerted when an OAD session is in progress. This is done through a callback function that has registered to be invoked when any of the tracked OAD events occur.

The registration prototype and OAD event bitmap defines can be found in *hal_oad.h*. The registration prototype is presented below as well. The call registers a void function that takes a uint16 argument. The named function will be invoked when the events marked in the bit map indicated in the **eventMask** occur. The argument to the callback function will indicate the event that occurred:

```
void ZLOADApp_RegisterOADEventCallback(void(*pCBFunction)(uint16),
                                       uint16 eventMask)
```

The events supported are defined in the *hal_oad.h* interface header. There will be no race conditions inherent in the sequence because the callback is done in the same thread as the event itself. The **begin** event callbacks occurs before file transfer begins and the **end** event callbacks occur after completion. The reset event callback occurs just before the platform is reset. In each case the process will not proceed until the callback returns.

Only a single callback can be registered. Only the most recent callback function and event set registration is respected. Deregistration can be accomplished by providing either a null function pointer or a null event mask.

13.2 Results of the build.

After a project is built with the OAD Application as described above, the target Exe directory will contain three files:

1. A monolithic binary file with the extension **.bin**. This is the file that is used as the image to be downloaded over-the-air in subsequent OAD procedures. The image will be OAD-capable so that it too can be updated via OAD.
2. An IAR debug file with the extension **.d43**. This file is used by the IAR IDE for debugging.
3. A file with the extension **.sim**. This file is input to the IAR post-processing tool which produces the monolithic binary file above.

13.3 Caveats and additional information.

1. If anything changes that will affect the boot code, then the boot code must be re-compiled and then must be physically downloaded – OAD is impossible with parameters that are not identical to the fielded boot code. The applicable parameters are HAL_NV_PAGE_CNT when HAL_OAD_XNV_IS_INT and everything within the hal_board_cfg.h section labeled “Xtra-NV (used by OAD) implemented by internal flash or SPI eeprom.”

2. The image identification fields (defined in the file oad_preamble.h) are not used to validate a downloaded image. That is, they are not used as part of any sanity check for the legality of an image. They have no semantics associated with them.

3. During an OAD transfer session there is a timeout constant of 1 second for retries on the client side. There are 10 retries before the client gives up. Therefore, if the End Device polling rate is 10 seconds or longer, it is likely that the client side will quit during an image transfer. So ensure compatible network settings between the polling rate and OAD retry rate and retry count to ensure OAD success.

4. The OAD event callback can be used to synchronize the OAD session with other platform functionality. This can be important when the platform is in the client role. The image download process can generate significant interruptions in processing when the downloaded image is being written. The callback offers an opportunity to add discipline before and after a download session. These run-time adjustments should probably be done in the callback thread rather than using the task messaging interface. This will guarantee that the adjustments occur before the OAD session continues, as the session is blocked until the callback returns. If it is not important to make the runtime adjustments precisely before or after the session then the more disciplined OSAL messaging method can be used.

5. To save on sending unused flash pages over-the-air, those pages can be removed from the image to send by OAD by shortening the end of available flash to only what is required for the particular application. After building the code image, inspect the .map file produced to find the number of filler pages at the end of the image. These are not necessary, so remove them by subtracting the size of the filler at the end of the image from these 3 places:

    i. #define HAL_OAD_DL_SIZE

    ii. -P(CODE)BANKED_CODE=0x0800-0x7FFF,0x18000-0x1FFFF,… (in oad.xcl).

    iii. -J2,crc16,= 800-887,88C-7C7FF (in oad.xcl).

    When using only internal flash for OAD (i.e. external NV is not available), there is a very strong and perhaps undesired side-effect to reducing the operational code image to less than half of the available internal flash. The side-effect is that it is never possible to instantiate a code image larger than this reduced size (even though the device can participate in OAD with larger image sizes, it cannot instantiate them.) A minor change in the boot code could ameliorate this side-effect, but not eliminate it completely.