



# **Over Air Download For CC2430**

**Document Number: F8W-2008-0003**

**Version 1.0**

# TABLE OF CONTENTS

1. PURPOSE .....	4
2. FUNCTIONAL OVERVIEW .....	4
3. ASSUMPTIONS .....	4
4. DEFINITIONS, ABBREVIATIONS, ACRONYMS .....	4
5. REFERENCES .....	4
6. REVISION HISTORY .....	4
7. DESIGN CONSTRAINTS .....	5
7.1 EXTERNAL CONSTRAINTS / FEATURES .....	5
7.2 INTERNAL CONSTRAINTS / REQUIREMENTS .....	5
8. DESIGN.....	5
8.1 OAD CONTEXT .....	5
8.2 FUNCTIONAL DESCRIPTION.....	5
8.2.1 Boot Code .....	5
8.2.2 ZLOAD Application.....	6
9. OAD HIGH LEVEL CUSTOMER IMPLEMENTATION .....	6
9.1 ADD THE ZLOAD APPLICATION TO THE IAR PROJECT.....	7
9.1.1 Add the ZLOAD Library – <i>preferred method</i> .....	7
9.1.2 Add the ZLOAD Source files – <i>if not using preferred method above</i> .....	7
9.2 CONFIGURE LINKER OPTIONS FOR THE ZLOAD FUNCTIONALITY. ....	8
9.2.1 Configure the linker command file for OAD-compliant mapping. ....	8
9.2.2 Configure the linker to generate extra output. ....	9
9.2.3 Configure the linker extra output file format.....	10
9.3 CONFIGURE BUILD ACTIONS TO INVOKE THE OAD POST-PROCESSING TOOL. ....	11
9.4 CONFIGURE COMPILER OPTIONS FOR ZLOAD.....	12
9.4.1 Add the ZLOAD path. ....	12
9.4.2 Configure the compiler options to remove code overage. ....	13
9.4.3 Configure the compiler options to match the pre-built ZBOOT library. ....	14
9.5 ADD THE ZLOAD OSAL COMPONENTS. ....	15
9.6 DEFINE PROPRIETARY PREAMBLE FIELDS.....	19
9.7 BUILD AND DEBUG THE APPLICATION CODE.....	19
9.8 LOAD THE FINAL OAD-CAPABLE CODE.....	19
10. OAD HIGH LEVEL CUSTOMER INTERACTION.....	20
10.1 OAD EVENT CALLBACK - <i>OPTIONAL</i> .....	20
10.2 RESULTS OF THE BUILD.....	20
10.3 CAVEATS AND ADDITIONAL INFORMATION.....	21

## TABLE OF FIGURES

Figure 1: Architectural Placement of the OAD Z-Stack Component.....	6
Figure 2: Adding the ZLOAD Library to the IAR Project, Linker Extra Options. ....	7
Figure 3: Changing the linker command file to implement OAD-compliant mapping. ....	8
Figure 4: Configuring the linker to generate an extra output file.....	9
Figure 5: Configuring the linker extra output file format.....	10
Figure 6: Configuring the build actions to invoke the OAD post-processing tool. ....	11
Figure 7: Adding the ZLOAD path to the IAR Project Options. ....	12
Figure 8: Removing unnecessary functionality from GenericApp to free code space. ....	13
Figure 9: Adding compiler configuration to match pre-built ZBOOT code. ....	14
Figure 10: Selecting the OSAL_GenericApp.c file for editing. ....	15
Figure 11: Adding ZLOAD includes and macro to OSAL_GenericApp.c. ....	16
Figure 12: Adding the ZLOAD event to OSAL.....	17
Figure 13: Adding the ZLOAD initialization function to OSAL. ....	18
Figure 14: Adding preamble.h to the IAR Project. ....	19

## 1. Purpose

The purpose is to provide a developer's guide to quickly and easily enable OAD functionality in any customer's proprietary Z-Stack Application.

## 2. Functional Overview

OAD is an extended stack feature (patent pending) 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. This document and all code referenced herewith and provided as an extension to the Z-Stack for OAD assumes the use of the off-chip NV storage, connected by SPI to the CC2430, which is provided by reference design TBD – what is it?

## 4. Definitions, Abbreviations, Acronyms

Term	Definition
NV	Non-volatile (memory that persists through power cycles.)
OAD	Over Air Download

## 5. References

- [1] ZigBee Implementers Guide (F8W-2004-0007)
- [2] Z-Stack Applications User's Guide(F8W-2007-0022)

## 6. Revision History

Date	Writer's name	Document Version	Description of changes
03/10/08	S. Löhr	1.0	Used example Z-Stack application of GenericApp. Updated the OSAL task initialization.
04/11/08	S. Löhr	1.0	Added explicit instruction to use the SmartRF programming tool to download the <i>AppName_SRF04SoC.hex</i> file after all debugging is complete and OAD functionality is desired – Sec 9.7-9.8 & 10.2.

## 7. Design Constraints

### 7.1 External Constraints / Features

An off-chip NV device must be used to store the new OAD image - 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 separately compiled images: the Boot Code and the Z-Stack with the customer specific and the ZLOAD Applications. The post-processing tool (invoked by the IAR build of the Z-Stack with applications) “glues” these two separate builds together into one image file for downloading by either the IAR debugger or SmartRF04 programmer. These two builds have no functional API and only one data structure API. The details of this data structure are beyond the scope of this document and familiarity is not necessary to successfully implement ZOAD. The location of each of the two images in Flash and the location of the shared data structure in XDATA RAM are handled by the IAR linker script files referenced in Section 9.

### 8.2 Functional Description

#### 8.2.1 Boot Code

The OAD solution requires the use of boot code to determine the sanity of the active code image before handing off control to the active image. This is mainly to guard against an incomplete or incorrect flashing of the active image area. The OAD boot code provides the following functionality:

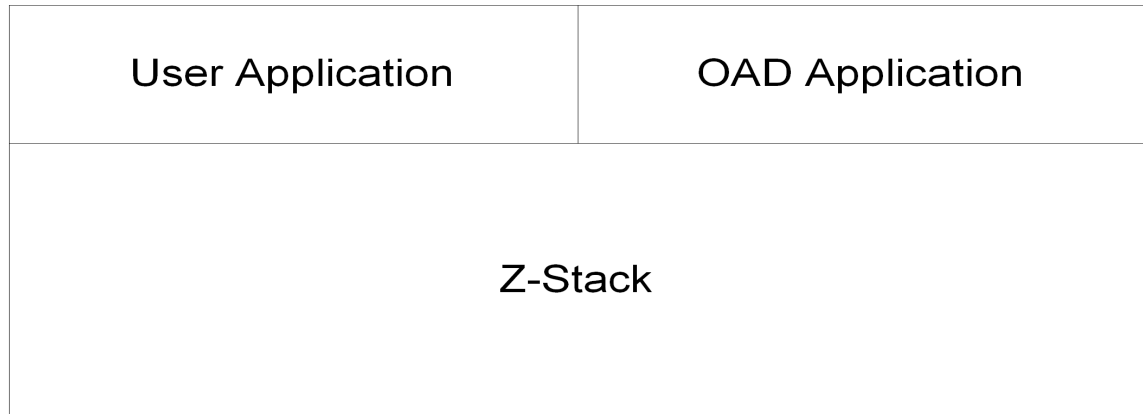
1. It will be the target of the reset vector and therefore contain startup code.
2. It will support flash read/write/erase functionality. It must be able to read the external flash part containing the downloaded image and it must be able to write the on-chip flash to install a new active image.
3. It will provide sanity checks of the downloaded application code. This is needed to guard against programming the active flash sectors with downloaded code whose integrity may be compromised.
4. It will guard against interrupted re-flashing by checking the validity of the active application code image when appropriate. If the image is not valid the boot/startup code will again re-flash the downloaded image. This situation can occur if there was a power interruption during flashing of the new image over the old.

The Boot Code requires 6 KB of code space (3 flash pages) and must occupy the first flash page so that it can intercept the startup vector and the last two flash pages so that an OAD does not interfere with the IEEE address.

The Boot Code must be included in a Z-Stack build that is physically downloaded to the target, and after that, only an OAD-enabled build that does not include Boot Code can be downloaded to the board by OAD.

### 8.2.2 ZLOAD 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 ZLOAD Application must be included in the build of the Customer's Application in order to enable OAD functionality (the Client and Server logical components only – the Commissioner component is implemented in the ZPortApp sample application which is included with the Z-Stack distribution.)

The step-by-step details as to how to integrate ZLOAD functionality into an existing Z-Stack application are described in Section 9.

## 9. OAD High Level Customer Implementation

The Z-Stack sample application known as GenericApp is used below for demonstration purposes only - the Customer would apply the following steps in his/her own, proprietary Z-Stack application and make the corresponding changes to all of the paths below that are specific to GenericApp.

## 9.1 Add the ZLOAD Application to the IAR Project.

### 9.1.1 Add the ZLOAD Library – ***preferred method***.

Add the following line to the IAR Project → Options → Linker Extra Options (as shown in hi-light below.)

```
-C $PROJ_DIR$..\..\..\Libraries\TI2430DB\bin\ZLOAD.lib
```

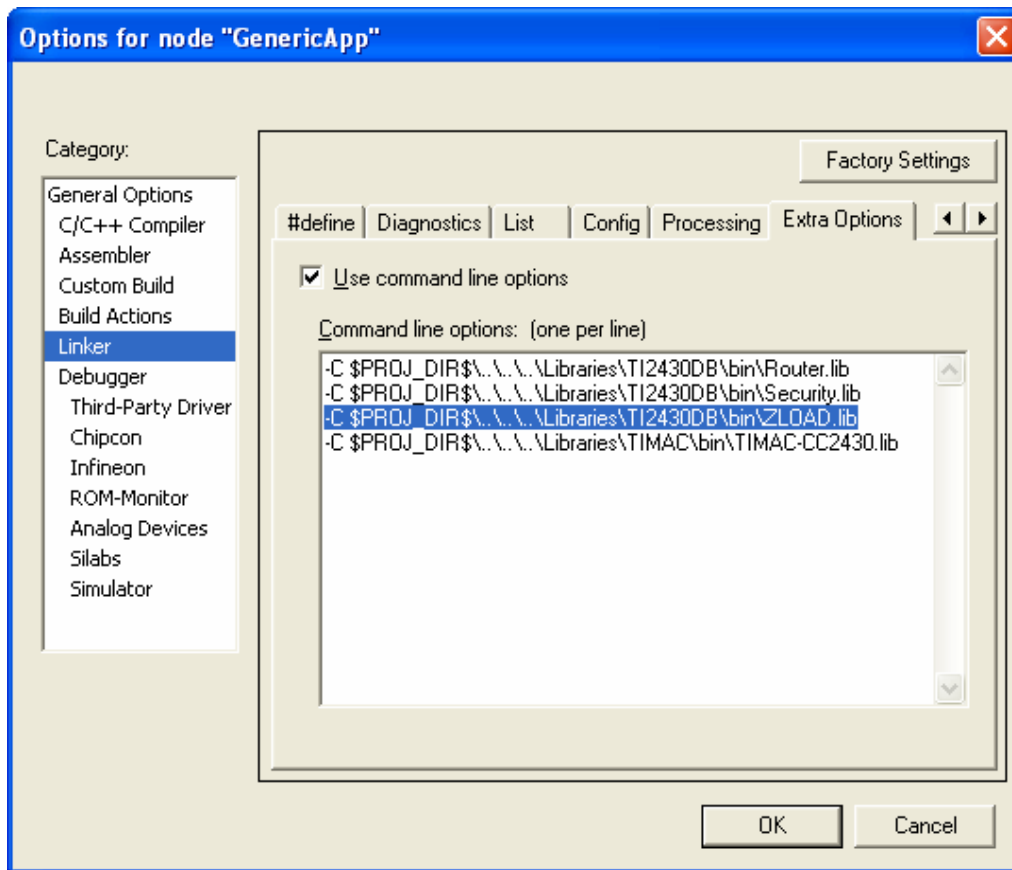


Figure 2: Adding the ZLOAD Library to the IAR Project, Linker Extra Options.

### 9.1.2 Add the ZLOAD Source files – ***if not using preferred method above***.

Adding the source build for ZLOAD to an existing application is beyond the scope of this document. Doing so would be modeled after the source build project included with the Z-Stack distribution which is named *ZPortApp*.

## 9.2 Configure linker options for the ZLOAD functionality.

### 9.2.1 Configure the linker command file for OAD-compliant mapping.

Use the following line for the IAR Project → Options → Linker → Config as shown in highlight below.

```
$PROJ_DIR$\\...\\Tools\\CC2430DB\\f8w2430oad.xcl
```

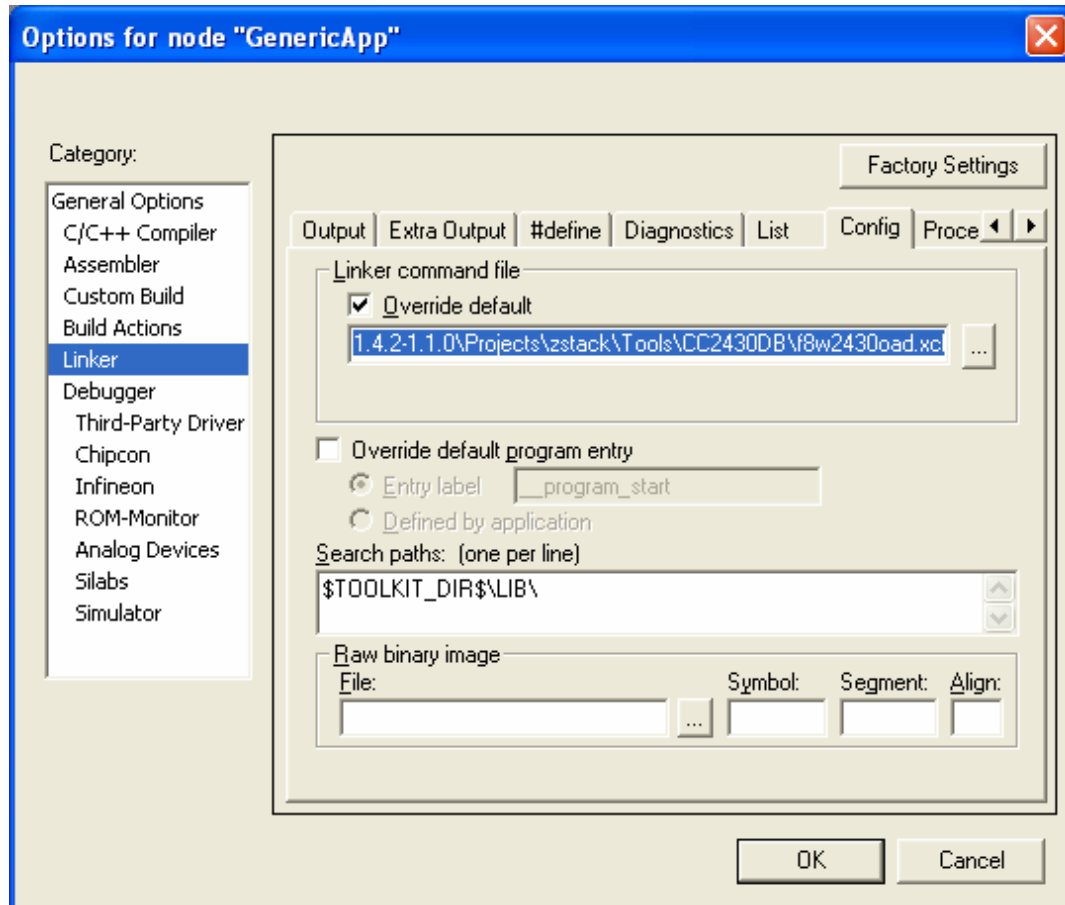
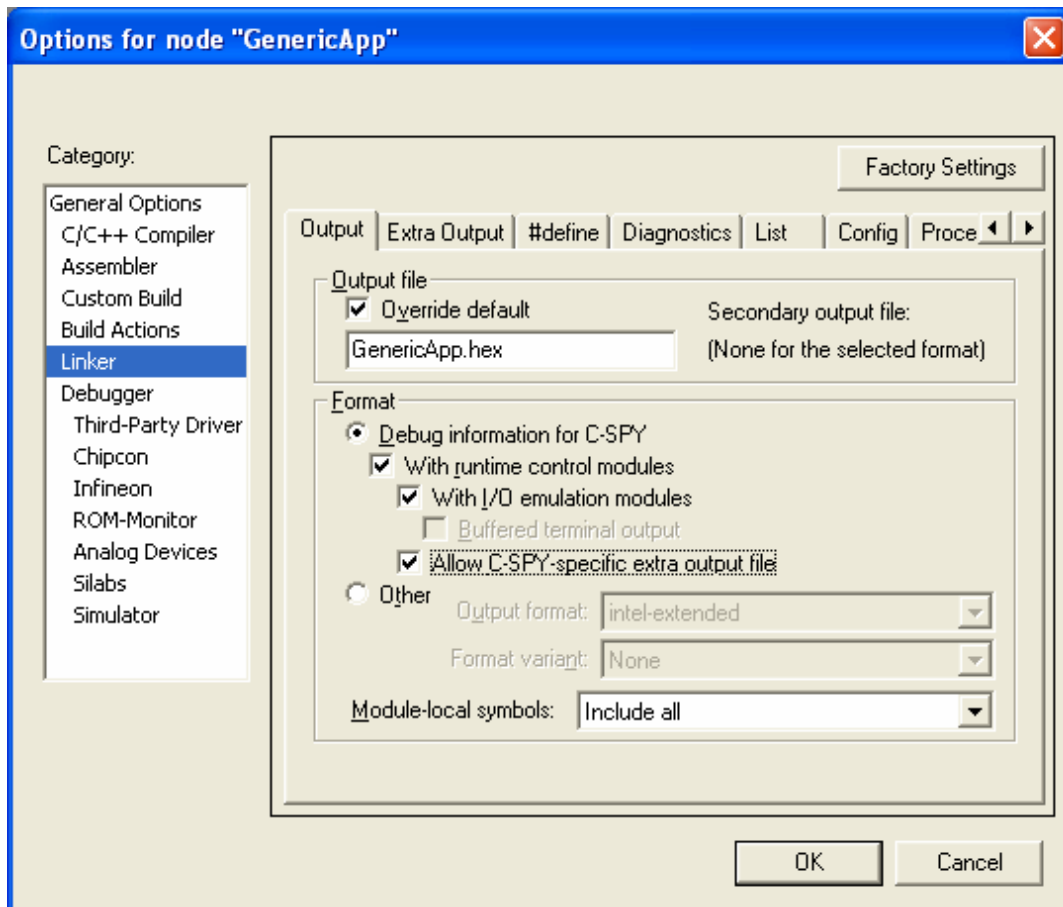


Figure 3: Changing the linker command file to implement OAD-compliant mapping.



### 9.2.2 Configure the linker to generate extra output.

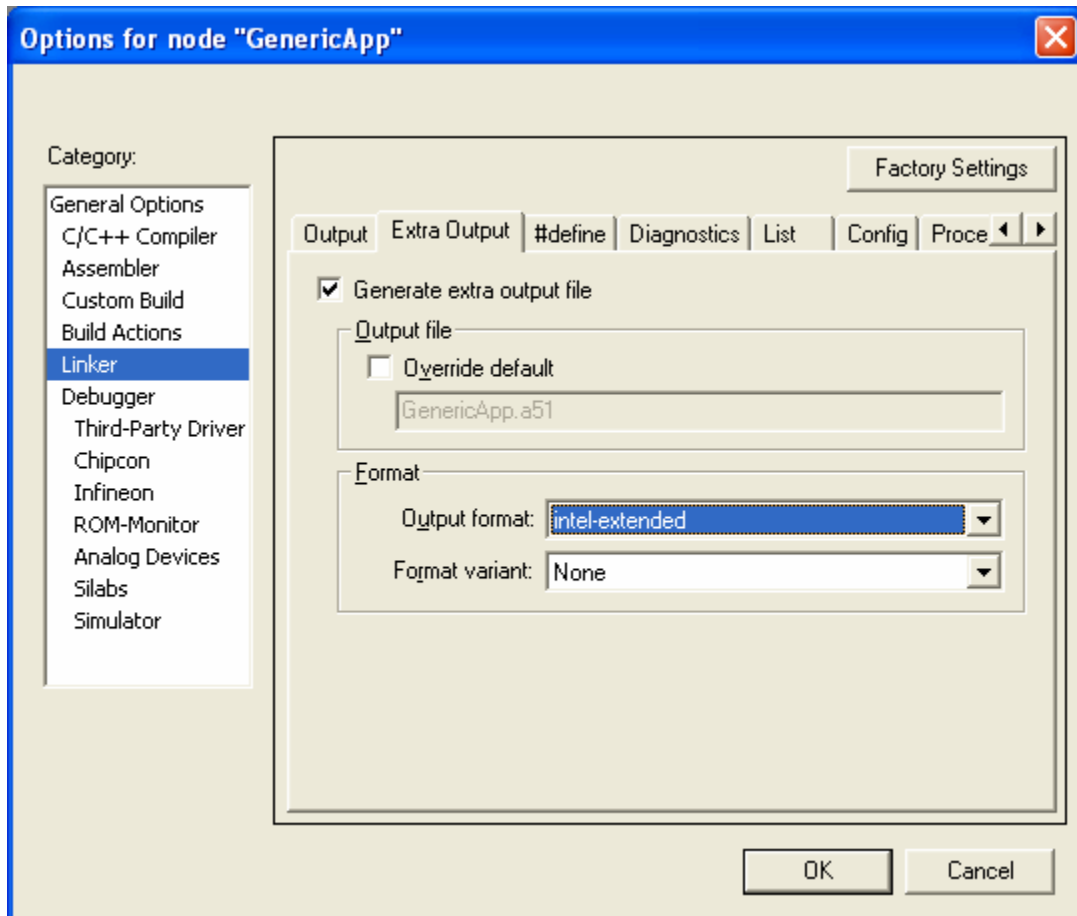
Check the checkbox to “Allow C-SPY-specific extra output file” as shown below.



**Figure 4: Configuring the linker to generate an extra output file.**

### 9.2.3 Configure the linker extra output file format.

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



**Figure 5: Configuring the linker extra output file format.**

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

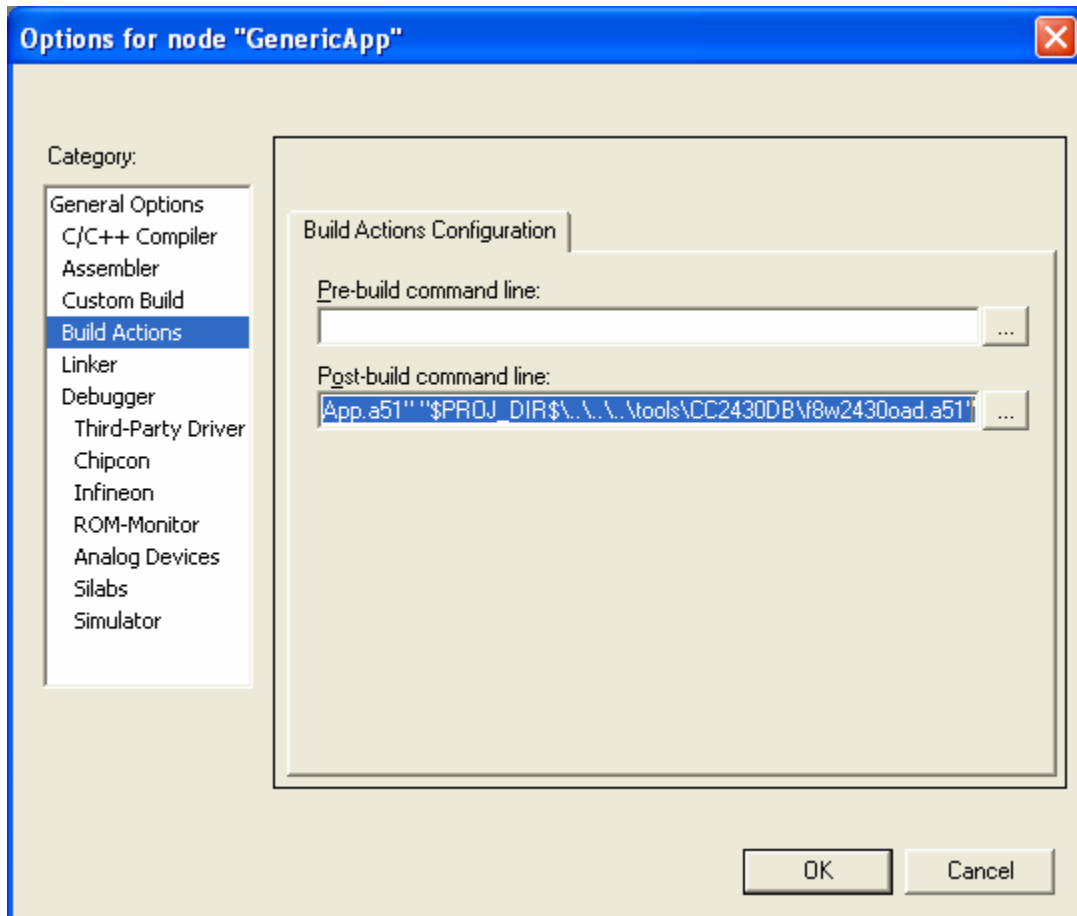
Use the following line for the IAR Project → Options → Build Actions as shown in hi-light below.

```
"$PROJ_DIR$\..\..\tools\CC2430DB\8w2430oad.exe"
```

```
"$PROJ_DIR$\RouterEB\Exe\GenericApp.a51"
```

```
"$PROJ_DIR$\..\..\tools\CC2430DB\8w2430oad.a51"
```

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



**Figure 6: Configuring the build actions to invoke the OAD post-processing tool.**

## 9.4 Configure compiler options for ZLOAD.

### 9.4.1 Add the ZLOAD path.

Add the following line to the IAR Project → Options → Additional include directories (as shown in hi-light below.)

```
$PROJ_DIR$\\..\..\..\Utilities\ZLOAD\Source
```

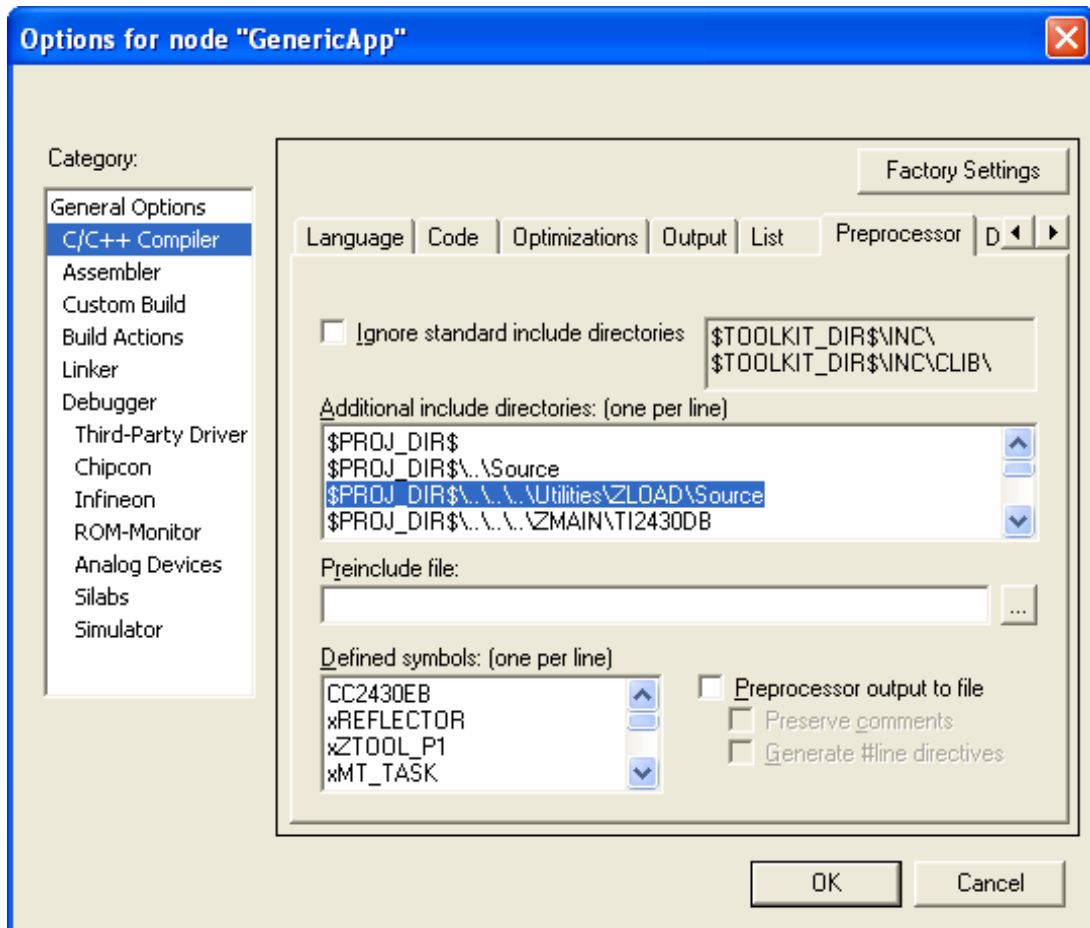


Figure 7: Adding the ZLOAD path to the IAR Project Options.

#### 9.4.2 Configure the compiler options to remove code coverage.

The 6-KB reserved for ZBOOT and the code space required by ZLOAD causes a code coverage with the default setting in GenericApp, so remove unnecessary ZTOOL support by adding a lower-case 'x' before the corresponding compiler options as shown in hi-light below.

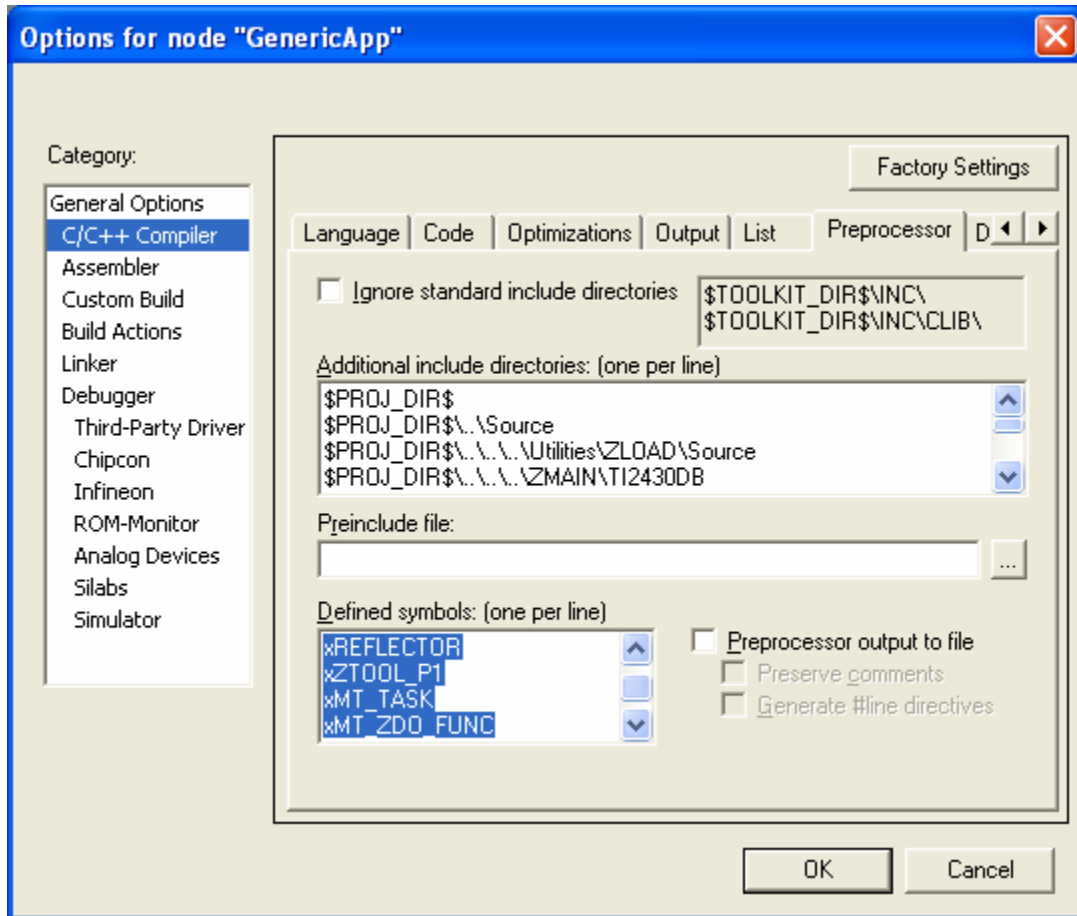


Figure 8: Removing unnecessary functionality from GenericApp to free code space.

### 9.4.3 Configure the compiler options to match the pre-built ZBOOT library.

Add the following three lines as shown below in hi-light.

```
TOP_BOOT_PAGES=2  
MINIMIZE_ROOT  
NV_IS_SPI
```

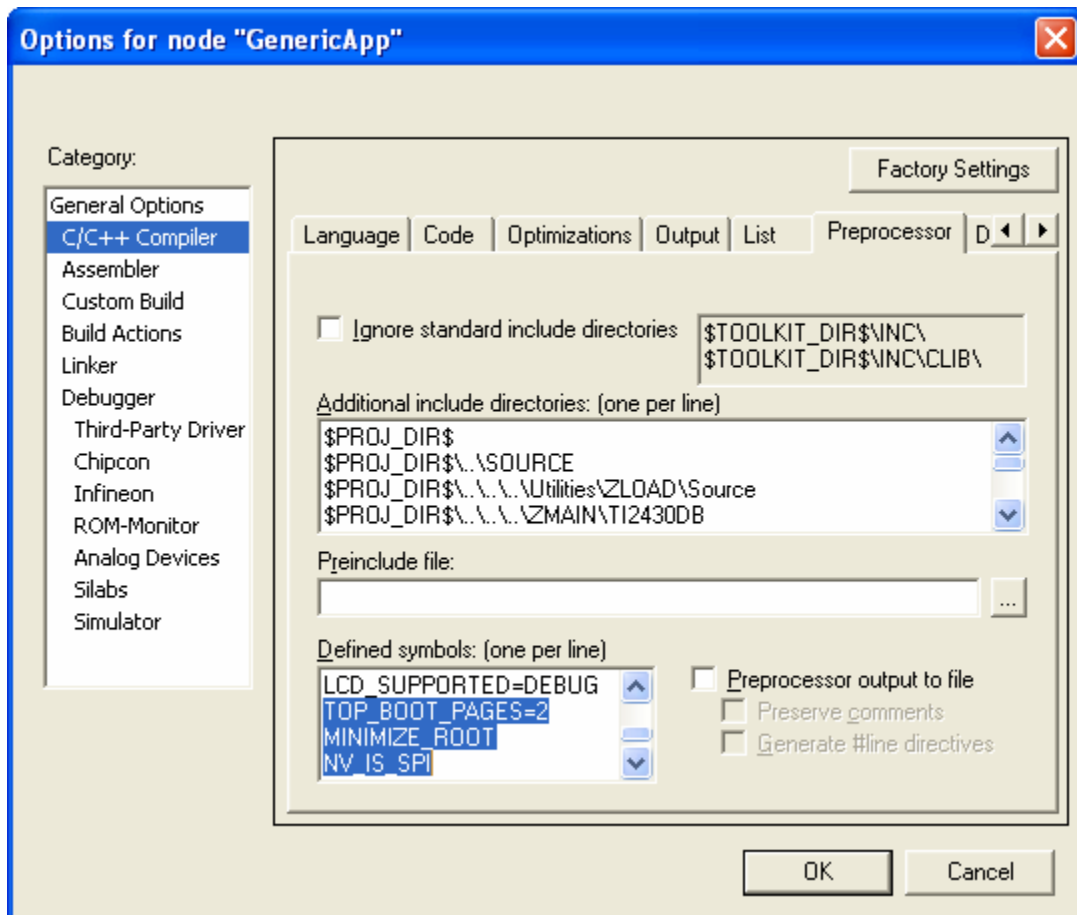


Figure 9: Adding compiler configuration to match pre-built ZBOOT code.

## 9.5 Add the ZLOAD OSAL components.

Edit the *OSAL\_GenericApp.c* file as follows:

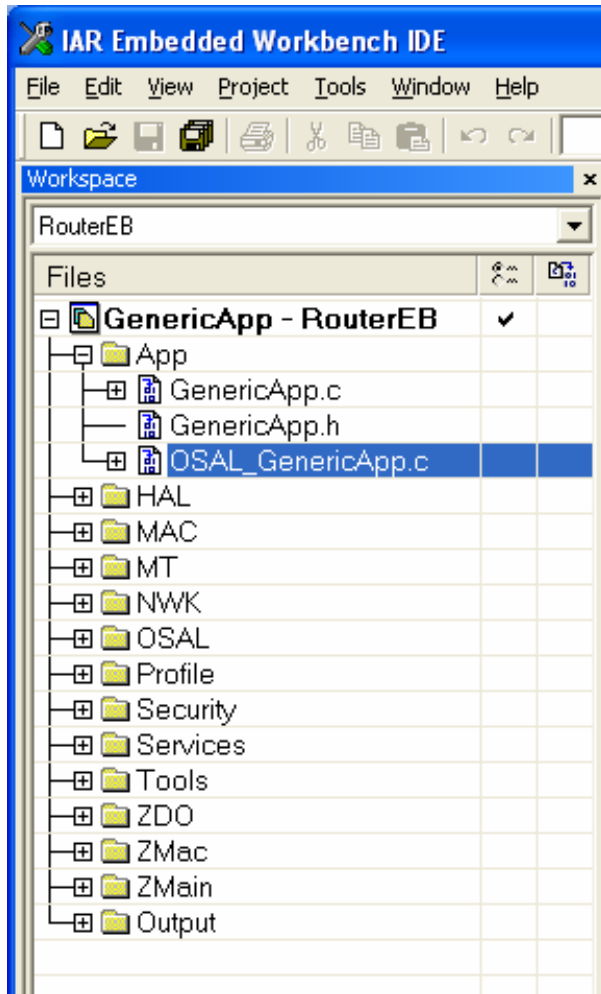


Figure 10: Selecting the *OSAL\_GenericApp.c* file for editing.

1. Add the following three lines to the top of the file with the other #include directives (as shown in hi-light below.)

```
#include "ZLOAD_IF.h"
#include "preamble.h"
OAD_MAKE_IMAGE_ID();
```

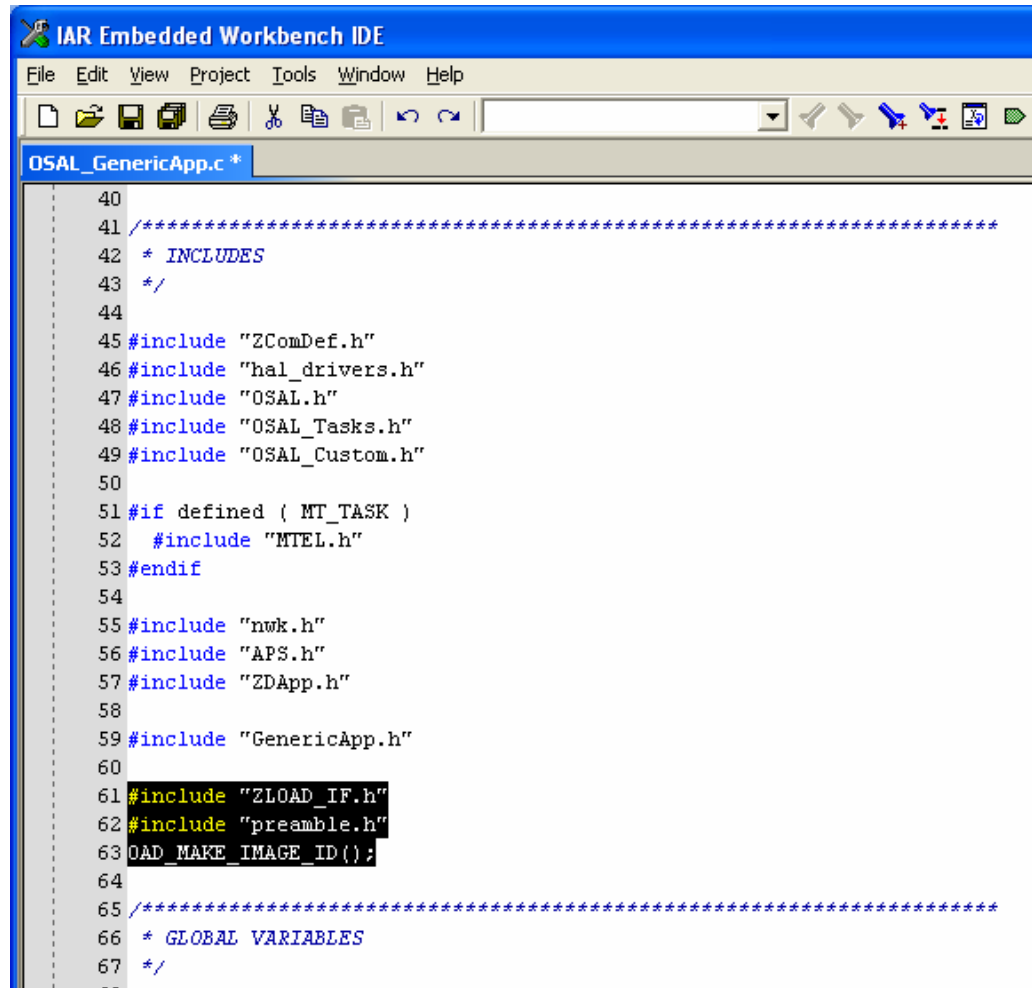


Figure 11: Adding ZLOAD includes and macro to OSAL\_GenericApp.c.



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

ZLOADApp\_ProcessEvent,

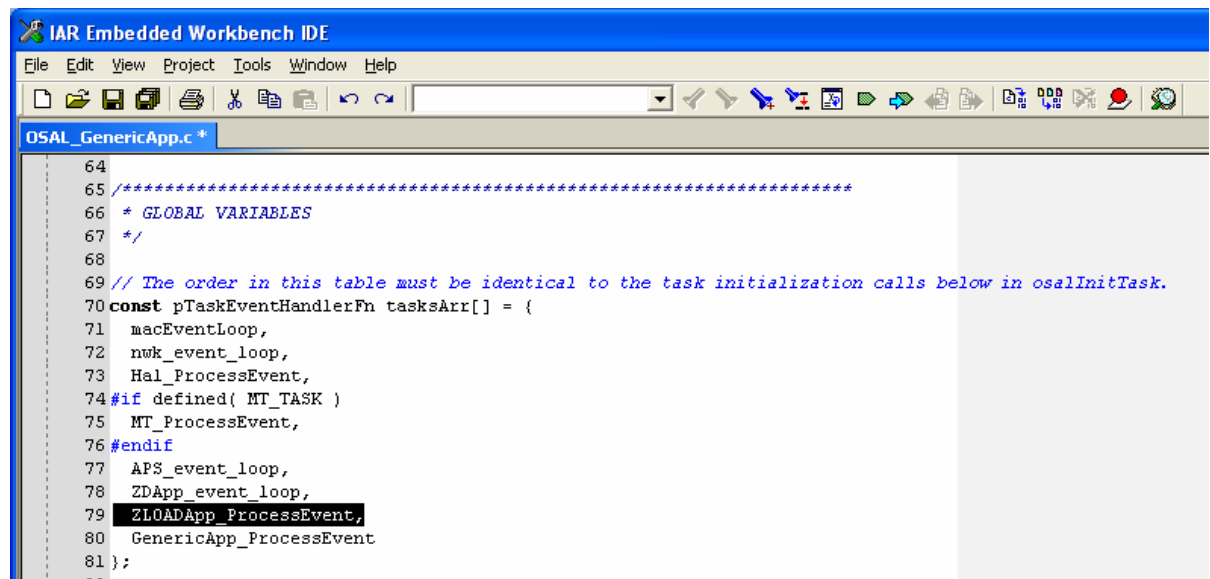
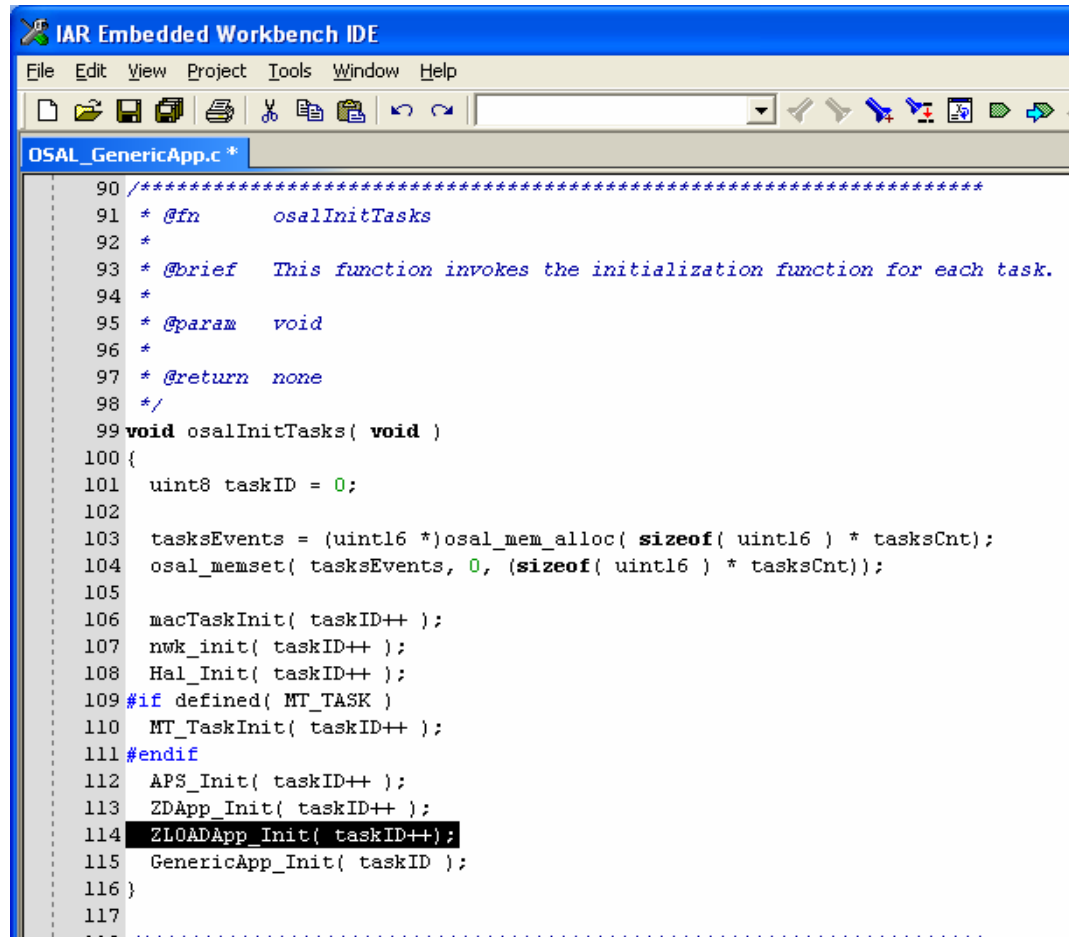


Figure 12: Adding the ZLOAD event to OSAL.

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

```
ZLOADApp_Init( taskID++ );
```



```
IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
[Icons]
OSAL_GenericApp.c *
90 /*****
91  * @fn      osalInitTasks
92  *
93  * @brief   This function invokes the initialization function for each task.
94  *
95  * @param   void
96  *
97  * @return  none
98  */
99 void osalInitTasks( void )
100 {
101     uint8 taskID = 0;
102
103     tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
104     osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
105
106     macTaskInit( taskID++ );
107     nwk_init( taskID++ );
108     Hal_Init( taskID++ );
109     #if defined( MT_TASK )
110     MT_TaskInit( taskID++ );
111     #endif
112     APS_Init( taskID++ );
113     ZDApp_Init( taskID++ );
114     ZLOADApp_Init( taskID++ );
115     GenericApp_Init( taskID );
116 }
117
```

Figure 13: Adding the ZLOAD initialization function to OSAL.

## 9.6 Define proprietary preamble fields.

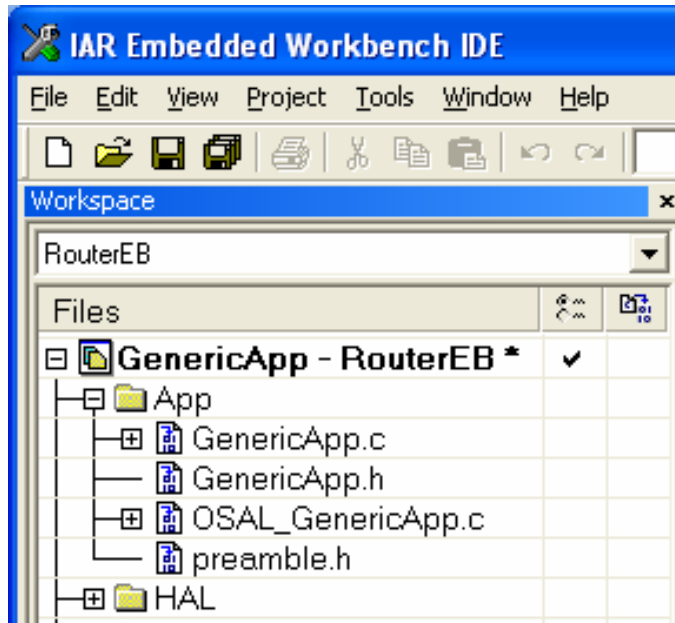
Move (don't copy) the header file *preamble.h* from this location:

```
C:\TexasInstruments\ZStack-1.4.3-1.1.0\Projects\zstack\Utilities\ZLOAD\Source
```

To the application source directory:

```
C:\Texas Instruments\ZStack-1.4.3-1.1.0\Projects\zstack\Samples\GenericApp\Source
```

Add *preamble.h* to the App folder of the IAR Project:



**Figure 14: Adding preamble.h to the IAR Project.**

By adding the *preamble.h* header file to the project, your proprietary changes to the preamble fields will trigger a re-build as appropriate.

There are six bytes available for unique identification of an image. 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. But if a device is to activate an OAD image, the identifying information must be different in at least one byte from that of the image currently active.

## 9.7 Build and Debug the Application code.

Simply build and debug the code from the IAR IDE (running with breakpoints, etc.) as you normally would. The code size is properly constrained to the space available to support OAD even though the crucial Boot Code is not installed when running, downloading, and/or debugging from within the IAR IDE. Thus, OAD is not possible while debugging.

## 9.8 Load the final OAD-capable code.

After making the final build of your application from the IAR IDE, you must use the TI SmartRF programming tool to load the output file named *GenericApp\_SRF04SoC.hex* (this has the added benefit of allowing you to set/preserve the unique IEEE address of your device.)

## 10. OAD High Level Customer Interaction

### 10.1 OAD Event Callback - *optional*.

To maintain platform stability coexisting ZigBee applications may need to know when an OAD session is in progress. This is particularly true of platforms acting in the client role (i.e. a platform receiving a new image.) 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.

To help maintain stability the capability is provided for an application to be alerted when and OAD session is in progress. This is done through a callback function that has registered to be invoked when any or all of the tracked OAD events occur.

The registration prototype and OAD event bitmap defines can be found in the *ZLOAD\_IF.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 *ZLOAD\_IF.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 occur 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.

### 10.2 Results of the build.

After a project is built with the ZLOAD Application as described in Section 9, the target Exe directory will contain three files:

1. A raw binary file with the extension **.bin**. This is the file that is used as the image to be uploaded OTA in subsequent OAD procedures. This is a binary image of the Z-Stack and Application(s) only (no boot code) and is specific to the project built. The image will be OAD-capable so that it too can be updated via OAD.
2. An Intel extended-hex file with the extension **\_SRF04SoC.hex**. This is the file used to populate a device initially to make it OAD-capable. It is a file that contains both the application image and the boot code image. Both will be programmed onto the device when this file is downloaded using the TI SmartRF programming tool. Once this initial population of a device is done using this combination application-boot code hex file, the hex file is never again used. Even though one will result from each subsequent build it can be ignored. (It is easier to just keep the IAR IDE configuration the same and ignore the file than to modify the IDE settings.)
3. A file with the extension **.a51 - *Deprecated***. This file has the same content as the .hex file above, but this one is to be downloaded from within the IAR IDE **after changing Project Options**, which is beyond the scope of this document.

### 10.3 Caveats and additional information.

1. The OAD implementation depends on boot code in 6K bytes of flash set aside at Page 0 and the top two pages (62 and 63) of the address space. If you currently run boot code you will have to merge the functionality of the TI ZigBee stack boot code and your own.
2. The segments **PREAMBLE** and **OAD\_IMAGE\_ID** *must* remain at the location designated in the linker script file. It is strongly recommended that these segments not be moved because the post build tool will then not work properly.
3. The **ZLOAD** application uses the watchdog timer to reset the MCU when the downloaded code is to be enabled. If you are using the watchdog these two uses will have to be reconciled. If your application is resetting the watchdog at a rate faster than the prescaler setting that the **ZLOAD** application uses, then the MCU will never reset. You can register for and use the callback mechanism provided by **ZLOAD** to disable your watchdog reset function. This will guarantee a clean watchdog reset initiated by the **ZLOAD** application.
4. The image identification fields (defined in the file 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.
5. If you are building an End Device and you change the polling period to a value larger than the default (1 second) it could interfere with OAD. During an OAD transfer session there is a timeout constant for retries on the client side. This value is set to 1000 milliseconds. On a timeout there are 5 retries before the client gives up. Therefore if the End Device polling rate is longer than 5 seconds it is likely that the client side will quit during an image transfer. If you are using the library version of the OAD support and do not have source code you will have to ensure the End Device polling rate is less than 5 seconds to accomplish OAD. If source code is available you can change the constant **SDR\_WAIT\_TO** in the source file **ZLOAD\_App.c** to the proper number of milliseconds to accommodate the End Device polling rate.
6. 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.
7. **Debugging note:** Once the combination ZLOAD Z-Stack/ZBOOT image file is downloaded onto a device, code should *not* be directly loaded onto the board again unless it is the combination image file at a later time. This restriction includes the IAR debugger when the 'Suppress Download' check box is unchecked.

The reason for this is that when code is downloaded through any of these tools the chip is first erased. This means that things will not work correctly with respect to OAD.

8. If you decide to ignore the above for debugging just be sure that the combination image file is reloaded before the device is placed into service. Otherwise it will not support OAD and will likely crash if any OAD functionality is invoked.