



Serial Boot Loader For CC2530 SoC

Document Number: SWRA-TBD

Version 1.1

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 SBL CONTEXT.....	5
8.2 FUNCTIONAL DESCRIPTION.....	5
8.2.1 Boot Code	5
8.2.2 SBL-compatible Z-Stack	6
9. PRODUCING SBL BOOT CODE TO BE PROGRAMMED.....	6
9.1 SEPARATE BUILD & DEBUG OF BOOT CODE.....	6
10. PRODUCING SBL-COMPATIBLE APPLICATION CODE TO DEBUG OR LOAD BY SBL.	7
10.1 CONFIGURE LINKER OPTIONS FOR THE SBL FUNCTIONALITY.	7
10.1.1 Configure the linker to generate extra output.	7
10.1.2 Configure the linker extra output file format.....	8
10.1.3 Configure the linker command file for SBL-compatible mapping.	9
10.2 CONFIGURE BUILD ACTIONS TO INVOKE THE POST-PROCESSING TOOL.....	10
10.3 ADD THE CRC SHADOW TO THE BUILD.....	10
10.4 BUILDING THE APPLICATION CODE FOR SBL.....	11
10.5 DEBUGGING THE APPLICATION CODE WITH SBL.	11
10.5.1 Preserve the SBL.	11
10.5.2 Force the CRC Shadow.	12
11. FORCING BOOT-MODE OR EARLY JUMP TO APPLICATION CODE.	13
12. PRODUCING SBL APPLICATION CODE WITH BOOT CODE TO BE PROGRAMMED.	14
12.1 BUILD THE APPLICATION CODE HEX IMAGE.....	14
12.1.1 Configure the linker to generate Intel-hex output.	14
12.1.2 Generate output compatible for the SmartRF Programmer tool.....	15
12.1.3 Re-build the Application Code to generate the .hex file.....	15
12.2 PRE-PEND THE BOOT CODE HEX IMAGE TO THE APPLICATION CODE HEX IMAGE.....	15

TABLE OF FIGURES

Figure 1: Architectural Placement of the SBL & SBL-compatible Z-Stack image.....	6
Figure 2: Configuring the linker to generate an extra output file.....	7
Figure 3: Configuring the linker extra output file format.....	8
Figure 4: Changing the linker command file to implement SBL-compatible mapping.....	9
Figure 5: Configuring the build actions to invoke the post-processing tool.	10
Figure 6: Preserving boot code while debugging.....	11
Figure 7: Configuring the linker to generate Intel-hex output.	14
Figure 8: Enabling -M option for SmartRF Programmer tool.....	15

1. Purpose

The purpose is to provide a developer's guide to implement SBL compatibility in any sample or proprietary Z-Stack Application for the CC2530 SoC.

2. Functional Overview

SBL is provided as a value-enhancing sample solution that enables the updating of code in devices without the cost of maintaining any download-related code in the user application other than ensuring a compatible flash memory mapping of the final output. SBL is effected as a managed client-server mechanism which requires a serial master to drive the process (i.e. a PC GUI application with access to the serial connection to the CC2530.)

3. Assumptions

1. SBL is a generic feature that should deviate as little as possible from one implementation to another by only supporting the idiosyncrasies of the specific medium (e.g. USB) crucial to the level of service necessary to complete a code image download in a reasonable amount of time. For the sake of example only, the USB SBL specific references will be made in this document, although merely changing the medium-specific verbiage and paths should allow this document to sufficiently describe any Z-Stack SBL.

4. Definitions, Abbreviations, Acronyms

Term	Definition
PC	Personal Computer
NV	Non-volatile (e.g. memory that persists through power cycles.)
SBL	Serial Boot Load(er)

5. References

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

6. Revision History

Date	Writer's name	Document Version	Description of changes
05/20/09	S Löhr	1.0	New document – used "OAD for CC2430" as a template.
03/18/2010	S Löhr	1.1	Update according to latest SBL behavior from Bug 3204.

7. Design Constraints

7.1 External Constraints / Features

1. A serial bus master must drive the download across the serial bus to the CC2530 - the means by which to design or implement such an application is beyond the scope of this document.
2. The Boot Code requires at least the first flash page so that it can intercept the startup vector.

7.2 Internal Constraints / Requirements

1. The image to be loaded via SBL must conform to a flash memory mapping compatible with the serial boot loader. Compatibility includes flash memory usage and ISR vector relocation (see Figure 1.)
2. The SBL must allow the bus master to force boot-load mode or an immediate jump to valid Application code after a powerup.
3. When not in boot-load mode, the SBL must immediately forward any ISR vectors that it consumes to the known relocation in the Application code (note that the Application code may not even enable such an ISR vector (e.g. USB ISR) in which case, it wouldn't need to define an ISR for it either.)

8. Design

8.1 SBL Context

The SBL system is comprised of two images: the 'boot loader code' and the Z-Stack with its Application(s) built with a compatible flash mapping – 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 SBL solution requires the use of boot code to check the integrity of the active code image before jumping to it. This check guards against an incomplete or incorrect programming of the active image area. The SBL boot code provides the following functionality:

1. Boot Code will be the target of the reset vector (as well as any vector necessary for communicating on the chosen serial bus), and therefore contains startup and ISR code.
2. When the serial bus connection is detected and the master application commands it, Boot Code will program the SBL image into the active image area and will thusly complete the final step of the SBL process: code instantiation.
3. (Optional) Boot Code will guard against interrupted, incomplete or incorrect programming of the active image area by checking the validity of the active application code image via CRC. If the image is not valid then the boot code will not allow it to run.

8.2.2 SBL-compatible Z-Stack

An SBL-compatible Z-Stack is implemented as a standard ZigBee Application build with the exception of the linker command file and some ancillary settings.

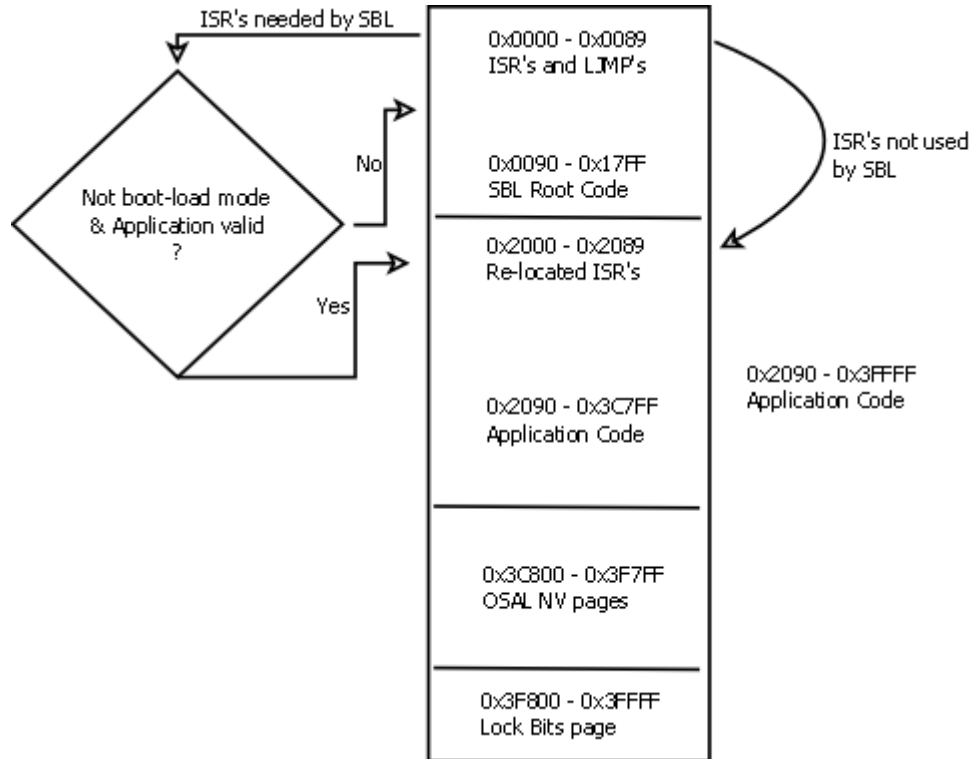


Figure 1: Architectural Placement of the SBL & SBL-compatible Z-Stack image.

9. Producing SBL 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 SBL Boot Project here:

```
$INSTALL_DIR$\Projects\zstack\Utilities\BootLoad\CC2530USB\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 SBL-compatible Application code produced in the next section, this SBL 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 SBL Boot code on successive debugging or programming.)

10. Producing SBL-compatible Application Code to debug or load by SBL.

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 CC2530USB is also used below for demonstration only – these same steps apply to any of the CC2530 targets supported by the Z-Stack release and a conforming SBL. It is only requisite that the paths specific to the CC2530USB target be changed accordingly.

10.1 Configure linker options for the SBL 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.

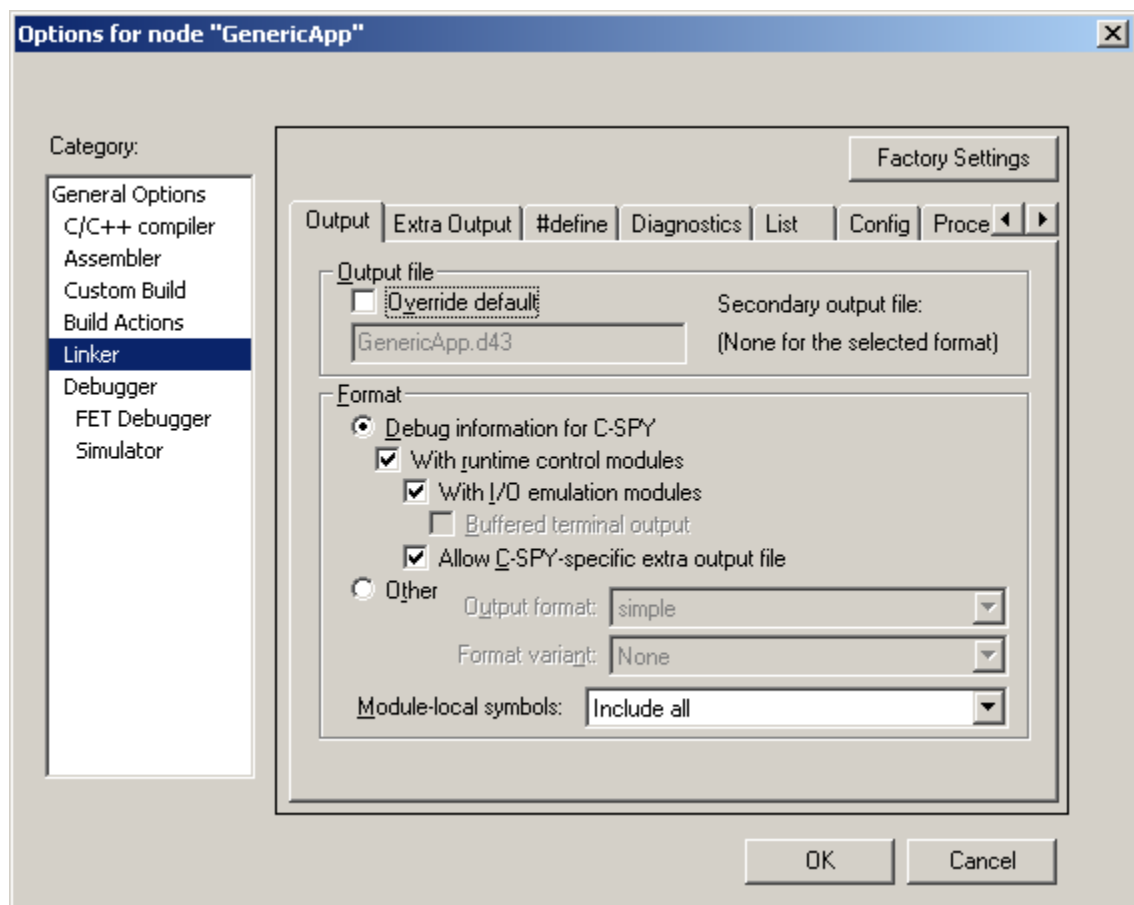


Figure 2: Configuring the linker to generate an extra output file.

10.1.2 Configure the linker extra output file format.

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

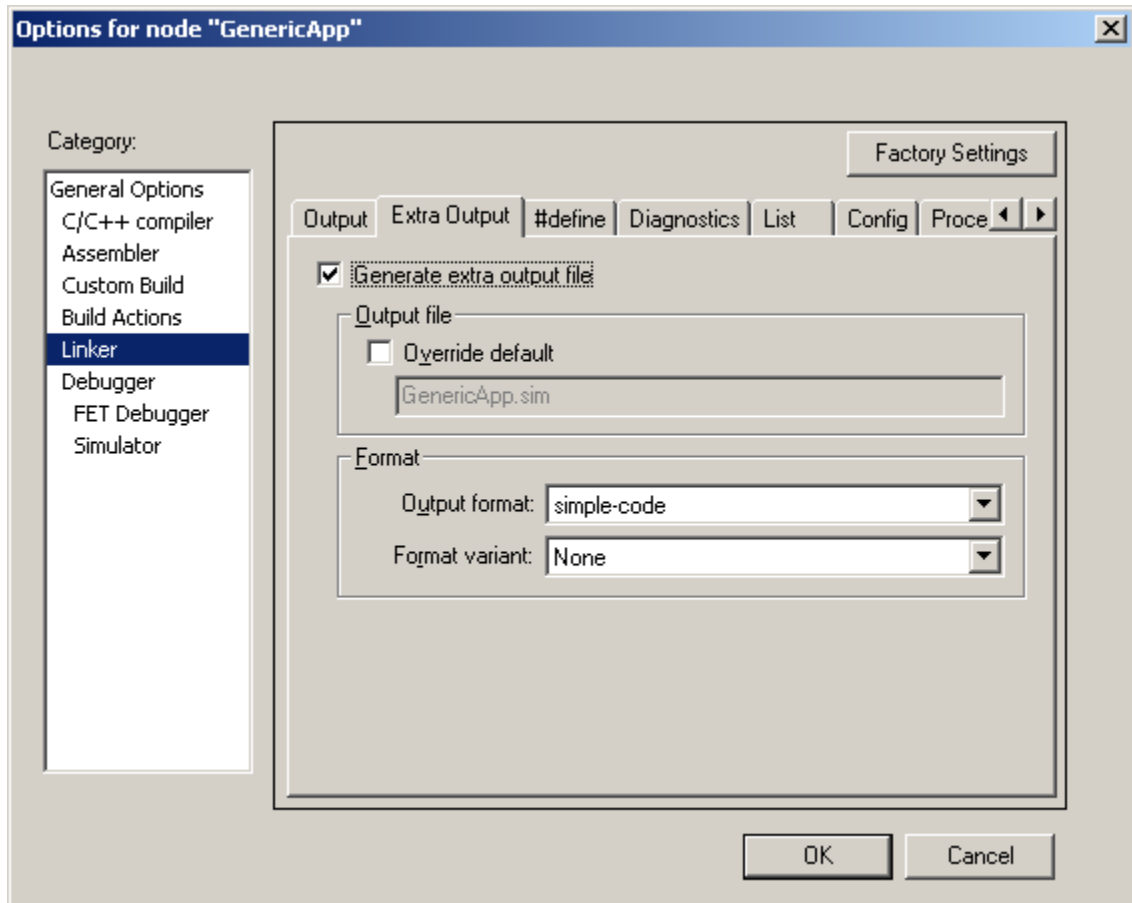


Figure 3: Configuring the linker extra output file format.

10.1.3 Configure the linker command file for SBL-compatible mapping.

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

```
$PROJ_DIR$\\.\.\.\Tools\CC2530DB\cc2530-sb.xcl
```

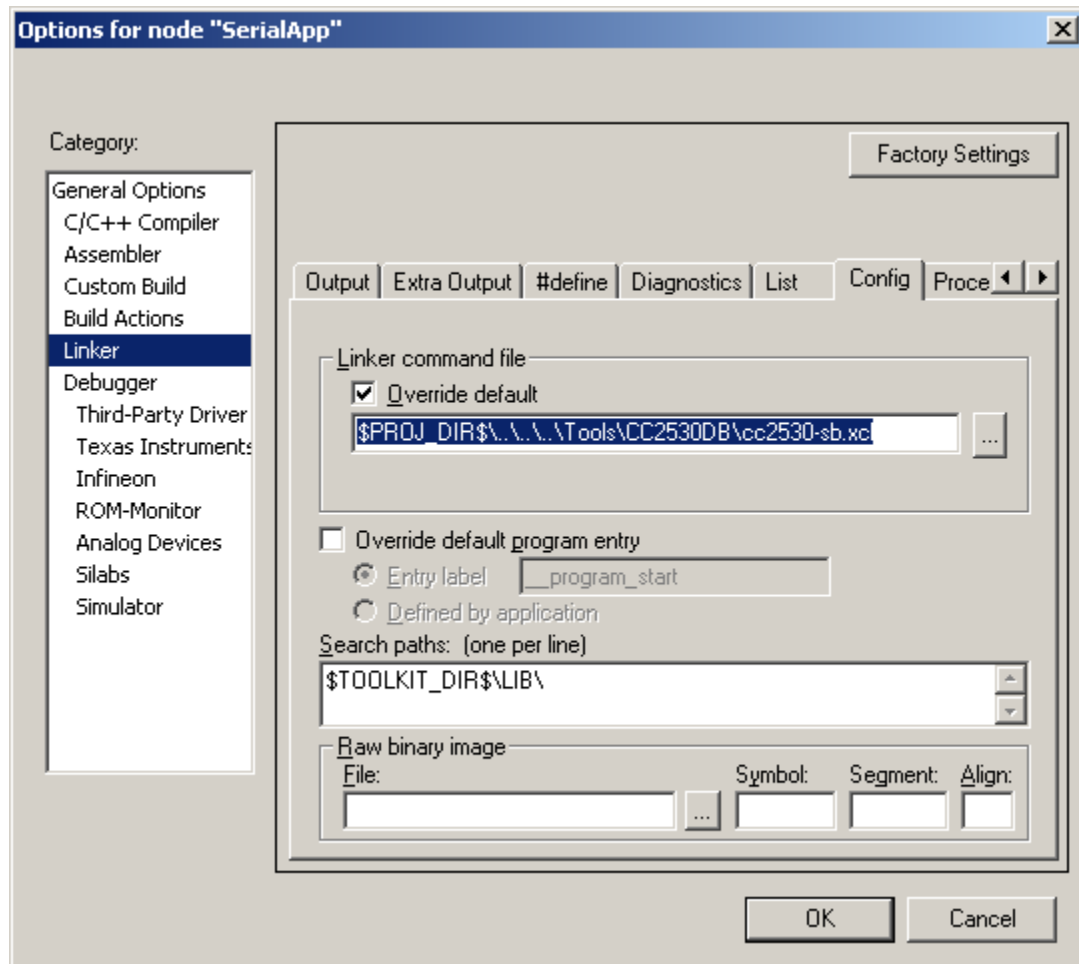


Figure 4: Changing the linker command file to implement SBL-compatible mapping.

10.2 Configure build actions to invoke the 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.

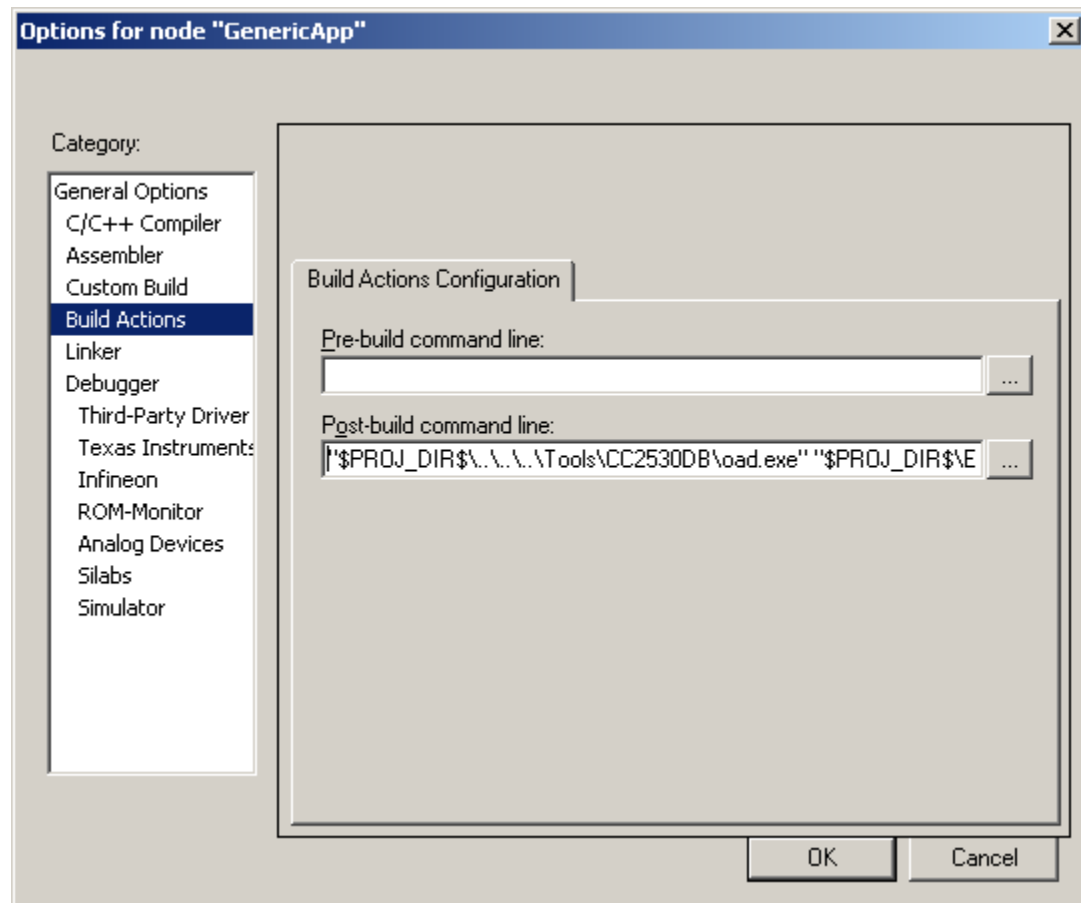


Figure 5: Configuring the build actions to invoke the post-processing tool.

10.3 Add the CRC Shadow to the build.

The CRC-shadow in OnBoard.c must be enabled by defining MAKE_CRC_SHDW somewhere (e.g. in OnBoard.c, hal_board_cfg.h, f8wConfig.cfg, or IAR project options to name a few possible locations.)

10.4 Building the Application Code for SBL.

Simply build from the IAR IDE as you normally would. The binary file produced, which is to be loaded by SBL, is found here:

```
$PROJ_DIR$\RouterEB\Exe\GenericApp.bin
```

10.5 Debugging the Application Code with SBL.

10.5.1 Preserve the SBL.

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:

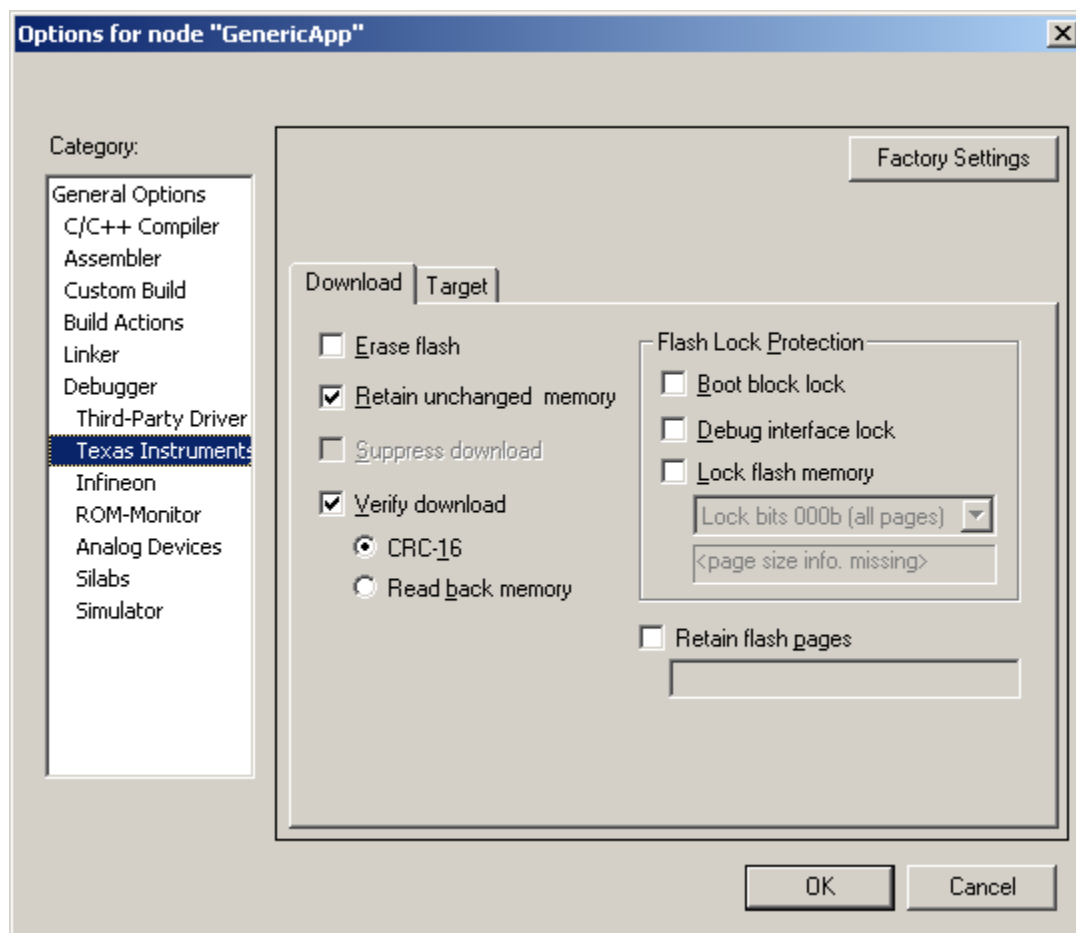
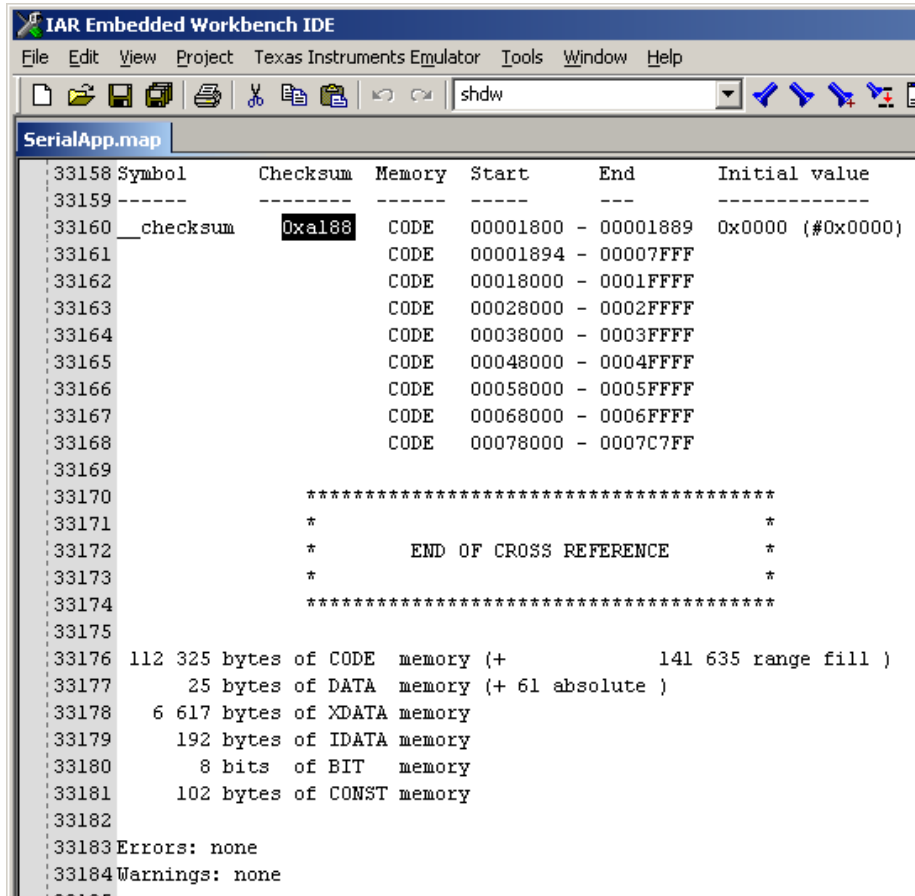


Figure 6: Preserving boot code while debugging.

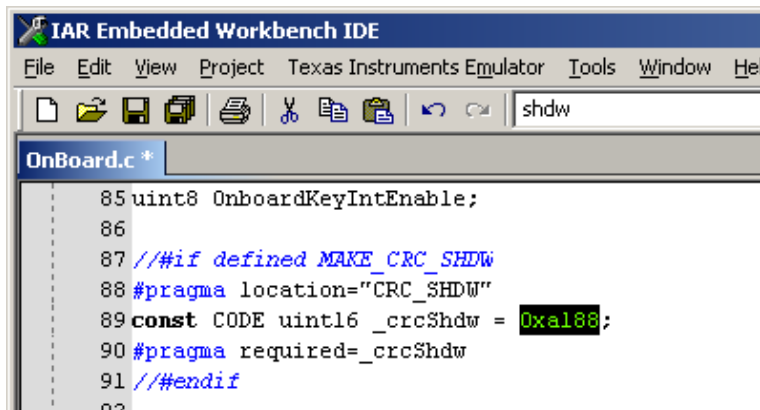
10.5.2 Force the CRC Shadow.

In order for the SBL to jump to the Application code after intercepting the reset vector, it must be able to verify that the Application code is valid. This validation takes over a minute and can be short-circuited by forcing the CRC-shadow to match the calculated CRC. **NOTE** that it is crucial that the CRC shadow be restored to its default value of 0xFFFF after debugging is complete. After each and every compile it is necessary to inspect the .map file to learn the new CRC:



```
IAR Embedded Workbench IDE
File Edit View Project Texas Instruments Emulator Tools Window Help
SerialApp.map
33158 Symbol      Checksum  Memory  Start      End      Initial value
33159 -----
33160 _checksum      0xa188   CODE    00001800 - 00001889  0x0000 (#0x0000)
33161                                     CODE    00001894 - 00007FFF
33162                                     CODE    00018000 - 0001FFFF
33163                                     CODE    00028000 - 0002FFFF
33164                                     CODE    00038000 - 0003FFFF
33165                                     CODE    00048000 - 0004FFFF
33166                                     CODE    00058000 - 0005FFFF
33167                                     CODE    00068000 - 0006FFFF
33168                                     CODE    00078000 - 0007C7FF
33169
33170                                     *****
33171                                     *
33172                                     *      END OF CROSS REFERENCE      *
33173                                     *
33174                                     *****
33175
33176 112 325 bytes of CODE memory (+          141 635 range fill )
33177    25 bytes of DATA memory (+ 61 absolute )
33178    6 617 bytes of XDATA memory
33179    192 bytes of IDATA memory
33180      8 bits of BIT  memory
33181    102 bytes of CONST memory
33182
33183 Errors: none
33184 Warnings: none
33185
```

Then copy the CRC into the CRC Shadow and only now can you run the IAR debugger with this build:



```
IAR Embedded Workbench IDE
File Edit View Project Texas Instruments Emulator Tools Window Hel
OnBoard.c *
85 uint8 OnboardKeyIntEnable;
86
87 // #if defined MAKE_CRC_SHDW
88 #pragma location="CRC_SHDW"
89 const CODE uint16 _crcShdw = 0xa188;
90 #pragma required=_crcShdw
91 // #endif
92
```

11. Forcing boot-mode or early jump to Application code.

The SBL receives control from the reset vector and verifies whether valid Application code is present. If so, then the SBL gives the bus master a window in which to force boot mode or an immediate jump to Application code.

1. If the CRC is not 0x0000 or 0xFFFF and the CRC-shadow is identical, then the Application code is valid.
2. If the CRC is not 0x0000 or 0xFFFF and the CRC-shadow is 0xFFFF, then the CRC is calculated over the Application code image area (this will take over a minute.)
 - a. If the calculated CRC matches the read CRC, program the CRC-shadow to this identical value to speed-up future power-ups.
3. If the Application code is valid, wait for the bus master to send a 0xF8 to force boot-mode or an 0x07 to force an immediate jump to the Application code.
 - a. The default wait for UART and USB transport is 1 minute.
 - b. The default wait for SPI is 50 milliseconds.
4. If the Application code is valid and the wait expires, jump to the Application code.
5. If the Application code is not valid, immediately jump to the boot-code without waiting as described above.

12. Producing SBL Application Code with Boot Code to be programmed.

For mass-production programming, it will be important to have a single image containing both the SBL Boot and Application code so that the part must only be programmed once. The following example assumes that the SmartRF04/05 Programming Tools will be used for programming an Intel-hex formatted file into the CC2530 SoC.

12.1 Build the Application Code hex image.

12.1.1 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.

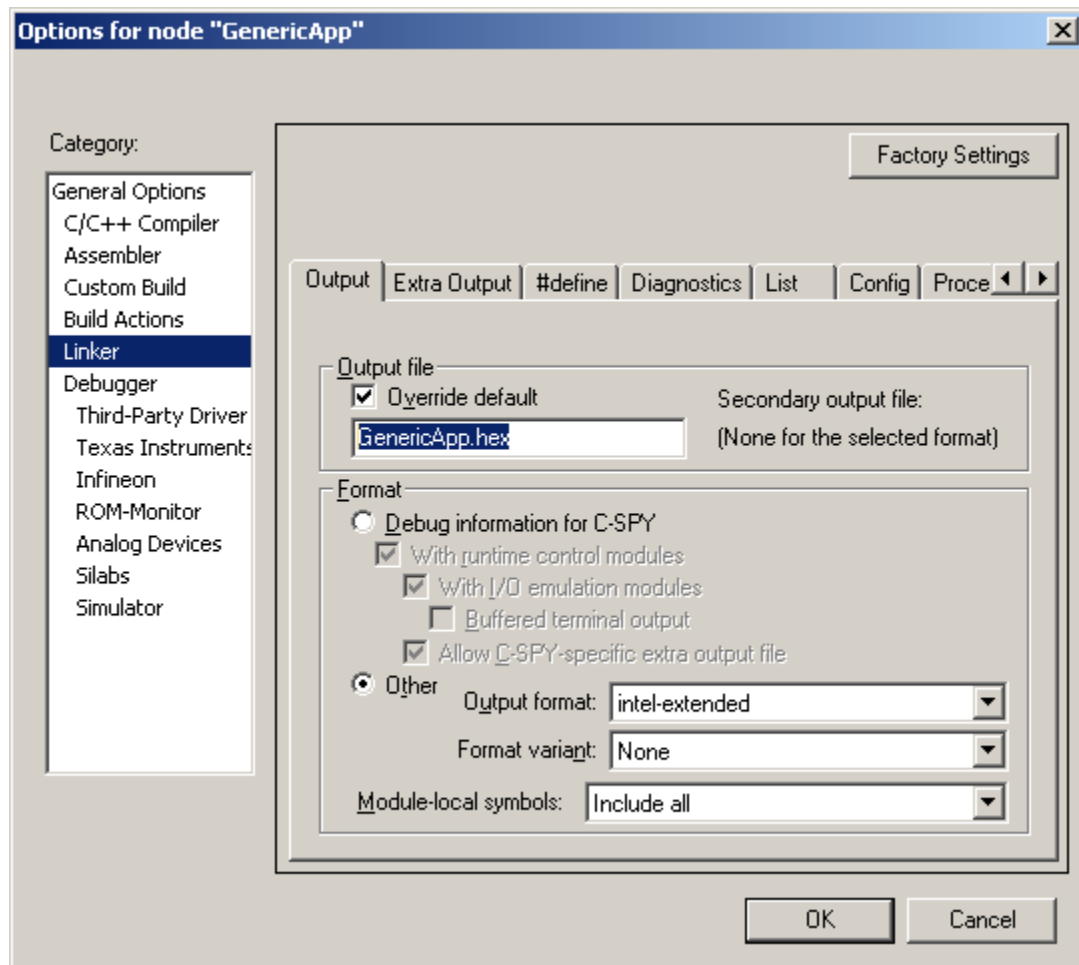


Figure 7: Configuring the linker to generate Intel-hex output.

12.1.2 Generate output compatible for the SmartRF Programmer tool.

Remove the comments from the `-M` option in `cc2530-sb.xcl` as shown in hi-light.

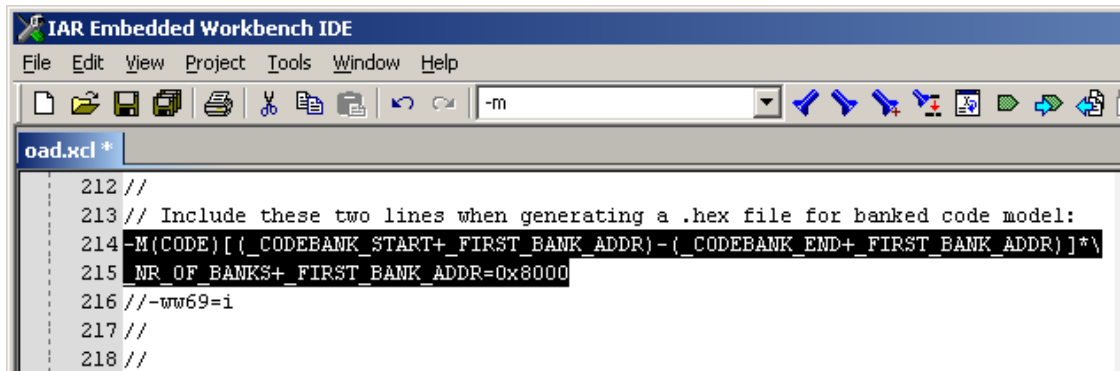


Figure 8: Enabling `-M` option for SmartRF Programmer tool.

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

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

12.2 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 SBL Boot Code file.
- 4) Delete these last two lines from the file:
:040000050000079E52
:00000001FF
- 5) Copy the edited contents of the SBL 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.