Texas Instruments

Application Report

SPRAAS7B –April 2009

Migrating a DSP/BIOS 5 Application to DSP/BIOS 6

Steven Connell *DSP/BIOS Kernel Team*

ABSTRACT

This application note introduces DSP/BIOS 5.x users to DSP/BIOS 6. It provides details and an example about using the legacy support. It further provides advice about migrating applications to take advantage of DSP/BIOS 6 features. This application note also discusses issues encountered when upgrading both from DSP/BIOS 5 to 6 *and* from Code Composer Studio 3.x to 4.

Contents

1 Introduction to DSP/BIOS 6.x Legacy Support

This application note introduces DSP/BIOS 5.x users to DSP/BIOS 6. It provides details about the legacy support that is provided and how to make use of it. It further provides advice about migrating applications to take advantage of DSP/BIOS 6 features.

The intention of this application note is to help you upgrade to DSP/BIOS 6 quickly and effectively, with minimal overhead, through examples and reference information about legacy module and API support. The information in this application note will hopefully act as a bridge to connect the old world of DSP/BIOS 5 to the new world of DSP/BIOS 6.

This application note also discusses issues encountered when upgrading both from DSP/BIOS 5 to 6 *and* from Code Composer Studio 3.x to 4. While you may use DSP/BIOS 6 with either CCStudio 3 or 4, many users may be performing both migrations at the same time. This document provides notes at the beginning of sections that apply only to CCStudio 4.

Along with DSP/BIOS 6.x, you must use XDCtools 3.x, which provides tools and APIs that implement the RTSC standard. For documentation on these additional tools, see Section 12.

1.1 What DSP/BIOS 6 Legacy Support is Provided?

DSP/BIOS 6 provides legacy support for most APIs to make it easy for you to migrate DSP/BIOS 5 applications to DSP/BIOS 6. The following bullet points highlight DSP/BIOS 6 legacy support:

- The vast majority of DSP/BIOS C APIs are fully supported and callable within DSP/BIOS 6, requiring no changes to existing C code.
- Tconf configuration (*.tcf) code is supported within DSP/BIOS 6 with some adjustments.
- A DSP/BIOS 5 program that uses supported legacy C APIs and has an updated configuration will be 100% compatible after a rebuild.

Expanding on these highlights, most DSP/BIOS 5 legacy C APIs still exist in DSP/BIOS 6, and may be called without making any C code changes. Allowing you to move C code forward, unchanged, minimizes migration overhead. A small number of modules and their APIs are no longer supported, either because they are no longer part of DSP/BIOS or because support for them did not make sense. The number of these deprecated modules and APIs was minimized.

Legacy Tconf configuration support is also provided, but in this case you must make some changes to that code in order to build. The changes needed for simple configurations are minimal. For more advanced configurations, a conversion utility is provided to translate Tconf configurations to RTSC configurations.

With these qualifications, existing DSP/BIOS 5 programs you upgrade to build in the DSP/BIOS 6 build environment will be 100% supported after updating the configuration and rebuilding, provided that their legacy program does not use any unsupported legacy APIs or modules.

Note: This document uses the term "Tconf configuration" to describe DSP/BIOS 5 configuration scripts. Note that both “Tconf configuration” and "DSP/BIOS configuration" refer to the same thing: the *.tcf file used to configure your DSP/BIOS 5 application.

1.2 What Does this Application Note Cover?

The sections in this application note are divided into the following categories:

- **Summary Information.** Section 1 provides this introduction. Section 11 and 12 provide a conclusion and a list of related documents.
- **Procedures.** Sections 2 through 6 give you step-by-step instructions for the following tasks:
  - Section 2, “Converting Your Configuration”
  - Section 3, “Converting and Updating Your CCStudio Project”
  - Section 4, “Converting Your C Code”
  - Section 5, “Migrating Memory Configurations”
  - Section 6, “Migrating Library Builds”
- **Mailbox Example.** Sections 7 through 10 demonstrate the migration process using the DSP/BIOS 5 mailbox example:
  - Section 7, “Porting the Mailbox Example to DSP/BIOS 6 Using Legacy Support”
  - Section 8, “Porting the Mailbox Example to DSP/BIOS 6 with New Modules and APIs”
  - Section 9, “Adding DSP/BIOS Tasks and Communication to a Legacy Application”
  - Section 10, “Building the Mailbox Example Using a Custom Platform File”

- **Reference.** Appendixes A through C
  - Appendix A, “Unsupported DSP/BIOS 5 APIs”
  - Appendix B, “Unsupported DSP/BIOS 5 Tconf Properties”
  - Appendix C, “Performance Benchmarks”

1.3 What is the General Migration Procedure?

In general, the migration process involves the following steps:

1. Convert a DSP/BIOS 5 Tconf configuration to a RTSC configuration. See Sections 2 and 5.
2. Migrate the build environment if necessary. See Section 3 (and Section 6 if you are building a library).
3. Update the C source file(s) if necessary. See Section 4.
4. Build and run the migrated application.

1.4 Software Requirements

Please ensure that the following components have been installed and set up properly before attempting to migrate the DSP/BIOS 5 mailbox example to DSP/BIOS 6:

- **XDCtools**
  - For setup details and instructions, please see the *XDCtools Getting Started Guide* that is included with the XDCtools installation.
  - The instructions in this application note require you to have set up a repository directory. The steps for doing this are found in the *XDCtools Getting Started Guide.*
- **DSP/BIOS 6**
  - For setup details and instructions, please see the *DSP/BIOS 6 Getting Started Guide* that is included with the DSP/BIOS installation.
  - The instructions in this application note require you to have set up the config.bld script. The steps for doing this are found in the *DSP/BIOS 6 Getting Started Guide.*
- **DSP/BIOS 5**
  - The steps in this application note are based on the “mailbox” example, which is packaged with DSP/BIOS 5.
- **CCStudio 4** or **CCStudio 3**
  - CCStudio 4 is required in order to configure DSP/BIOS using the XGCONF tool as described in Section 7.7.
  - CCStudio 3 is sufficient if you will configure DSP/BIOS with a text editor.
- **Drivers**
  - You must have already installed the appropriate drivers for your hardware configuration, if necessary.

2 Converting Your Configuration

In DSP/BIOS 6, programs no longer use a Tconf configuration script. Instead, they use a RTSC configuration script (*.cfg file), which is similar but somewhat different. Legacy users need to convert existing Tconf scripts to RTSC configuration scripts in order to build with DSP/BIOS 6 legacy support.

It's important to note that this *does not* mean that your Tconf configuration code must be thrown out! Most Tconf configuration settings and properties are still supported as they were in DSP/BIOS 5, with no changes to those configuration properties required. The degree of difficulty in making this conversion varies from script to script.

If the Tconf configuration script contains settings for modules and/or properties that are no longer supported, you will be notified of this by warning messages that are displayed either at program build time or when the conversion tool (explained in Section 2.2) is run.

To make it easier to convert your Tconf configuration to a RTSC configuration, this section categorizes Tconf configuration scripts as either "simple" or "complex":

- A "simple" Tconf configuration is one that *does not* make any changes to the default memory map.
- A "complex" Tconf configuration is one that *does* make changes to the memory map. Specifically, Tconf configurations that create a new memory segment or change the base location or length of a memory segment are considered "complex".

Changes to memory segments in DSP/BIOS 6 require non-trivial changes to the configuration, and that is why any Tconf configuration that *does* make memory map changes is *not* considered "simple." For details on how to configure memory segments, please see Section 5, "Migrating Memory Configurations".

Based on this categorization, one of two different methods of converting the Tconf configuration script to a RTSC configuration script is recommended:

- **Manual Conversion.** Recommended for "simple" configurations. See Section 2.1.
- **Using the Conversion Tool.** Recommended for "complex" configurations. See Section 2.2.

2.1 Converting a Simple Configuration Manually

Converting a DSP/BIOS 5 Tconf configuration to a RTSC configuration is straightforward *for simple Tconf configurations*. Again, a simple Tconf configuration is one that does not make any changes to the application's memory map. More specifically, a simple Tconf configuration would not create new MEM segments, or redefine any pre-existing MEM segments.

The following steps summarize how to convert a simple DSP/BIOS 5 Tconf configuration into a RTSC configuration:

1. Rename the Tconf configuration file to change its file extension from *.tcf to *.cfg.
2. Open the *.cfg file in your favorite text editor.

3. Near the beginning of the file, you should see a line that loads the platform file. For example, if your hardware platform was the “evmDM6437”, then you would see the following line of code:

   ```
   utils.loadPlatform("ti.platforms.evmDM6437");
   ```

   Delete this line of code from the file, since in CCStudio 4 the platform is defined by the project options.

4. In place of the call to `utils.loadPlatform()`, add the following line of code to the *.cfg file, in order to load the DSP/BIOS 6 legacy support:

   ```
   xdc.loadPackage('ti.bios.tconf');
   ```

5. Near the end of the file, you should see the following line of code. Delete this line of code from the file; it is no longer necessary.

   ```
   prog.gen();
   ```

6. The methods `prog.decl()` and `prog.extern()` are no longer supported. If your code uses either one, a build-time error will occur. If a *.tcf or *.tci file uses either of these methods for a function assignment, remove `prog.decl()` or `prog.extern()` and leave only the string name of the function being assigned. For example, the following code:

   ```
   tsk0.fxn = prog.extern("tsk0Fxn");
   tsk1.fxn = prog.decl("tsk1Fxn");
   ```

   needs to be changed to the following:

   ```
   tsk0.fxn = "tsk0Fxn";
   tsk1.fxn = "tsk1Fxn";
   ```

   If `prog.decl()` or `prog.extern()` are used as arguments, replace these method calls with calls to the new method `$externPtr()`. For example, the following code:

   ```
   tsk0.arg0 = prog.decl("foo");
   tsk1.arg0 = prog.extern("bar");
   ```

   needs to be changed to the following:

   ```
   tsk0.arg0 = $externPtr("foo");
   tsk1.arg0 = $externPtr("bar");
   ```

7. Save and close the file. It may now be used as a RTSC configuration file to build with DSP/BIOS 6.

Note: If your configuration file imports any additional configuration files (*.tci files) by means of calls to `utils.importFile()`, then you must update your CCStudio 4 project with the location of these files so that XDCtools can find them at configuration time. See Section 3.4, "Updating a CCStudio 4 Project’s Search Path to Find Included Configuration Files” for how to do this.

2.2 Converting a Configuration Using the Conversion Tool

DSP/BIOS 6 includes a conversion tool that is provided by the “ti.bios.conversion” package. You run this tool using the `xs` command provided by XDCtools. This conversion tool converts legacy Tconf configuration scripts to RTSC configuration scripts, and is the recommended means of translating a “complex” Tconf configuration to a RTSC configuration.

Once more, a Tconf configuration is considered to be “complex” if it makes changes to the application’s memory map. More specifically, a complex Tconf configuration is one that creates new MEM segments or redefines any pre-existing MEM segments. Such changes require non-trivial changes to the configuration. For details on configuring memory segments, please see Section 5, “Migrating Memory Configurations”.

Before using the conversion tool, please gather the following information that is needed in order to create a RTSC configuration script:

- The full paths of any Tconf include files (*.tci) that are imported by the application Tconf script (*.tcf).
- The name of the platform the application runs on.

2.2.1 *Conversion Tool Syntax*

The full syntax for the conversion tool command line is as follows:

```
xs [xs options] ti.bios.conversion [-v|--help] [-i] [-c outfile.cfg]
   [--pf platform.xs] [--pn plat] infile.tcf
```

Options:

- –c name of the output RTSC configuration file
- --pf name of the output platform file
- --pn name of the platform instance
- –v show details during build
- --xp set a package path
- --help show command line options
- -i content of imported files is copied into the output file

The `xs --xp` option sets a package path. In order to use DSP/BIOS 6, you must add the location of its packages to the path. For example:

```
xs --xp 'C:/Program Files/Texas Instruments/bios_6/packages' ti.bios.conversion
   [-v|--help] [-c outfile.cfg] [--pf platform.xs] [--pn myPlatform] infile.tcf
```

If your Tconf script imports Tconf include files (*.tci) from different directories, you must add paths to these directories to the package path. The paths can be absolute or relative to the current working directory. For example, if "infile.tcf" contains the following statement, and "helper.tci" is in a directory "sub" relative to the current working directory, then the `--xp` option should be set to ‘<*bios_install_dir*>/packages;sub’.

```
utils.importFile('helper.tci');
```

Without the `-i` option, any lines that import files are left unchanged in the generated RTSC configuration script, and the imported files are not converted. If the `-i` option is added to the command line, all imported files are converted and their contents are copied inline into the generated RTSC configuration script. Each imported file is clearly separated by comments in case you want to split the generated script into separate files again.

Note: If your *.tci files contain any calls to the methods `prog.extern()` or `prog.decl()`, we recommend that you use the `-i` option when running the conversion tool. This option causes the conversion tool to parse each *.tci file and to change all references to these methods appropriately.

Alternatively, if you do not want to use the `-i` option, you may convert all `prog.extern()` or `prog.decl()` calls manually, as described in Section 2.1. For more information on these unsupported methods, see Section 2.1.

By default, output from the conversion tool is a RTSC configuration script with the same name as the Tconf script, but with the extension .tcf replaced with .cfg. You can specify a different name for the output file with the `–c` command-line option.

The conversion tool makes the following changes to the Tconf script, and saves them in the RTSC configuration script:

Old Syntax	New Syntax
utils.loadPlatform	xdc.loadPackage(‘ti.bios.tconf’)
prog.gen()	<blank>

2.2.2 Specifying Environment Variables to the Conversion Tool

If your original Tconf script accesses any environment variables through the `environment` array, you need to define such variables on the command line for both the conversion tool. Define environment variables using the `-D` option. For example, if your script contains a statement like this:

```
utils.loadPlatform(environment[’config.platform’]);
```

Define `’config.platform’` on the command lines for the conversion tool as follows:

```
xs ti.bios.conversion –Dconfig.platform=ti.platforms.evmDM6437 ...
```

2.2.3 How Memory Map Settings Are Translated

In addition to the *.cfg file, the conversion tool generates a file called platform.xs by default. This file contains memory map configuration and platform-specific configuration (such as clockRate and deviceName). The conversion tool extracts memory map information from the Tconf script and saves equivalent commands in the platform.xs file. The platform.xs file creates a platform instance that corresponds to the memory map created by the Tconf script. The platform instance parameters are determined from bios.MEM and bios.GBL parameters in the Tconf script.

For example, if your original TCF script contains the following statement:

```
bios.GBL.C64PLUSL2CFG = "32k";
```

Then, your new platform file will contain the following line, which ensures that there are 32 KB of cache in your L2 memory.

```
l2Mode: "32k";
```

Memory-related commands in the new RTSC configuration script are ignored because, at the time that configuration script is parsed, no changes to the memory map are allowed. The memory map is already defined in platform.xs, which is processed before the RTSC configuration script. However, commands that create and determine the size of heaps are still valid and are processed at configuration time.

Also, you may override the default name for the generated platform file "platform.xs" by using the –pf command-line option. You should include the platform configuration file in the config.bld file you specify in your CCStudio 4 project.

For more on custom platform files, see Section 10. For more on memory configurations, see Section 5.

2.2.4 Steps to Convert a Tconf Script to a RTSC Configuration Script

1. At a MS-DOS command prompt, change (cd) to the directory that contains the Tconf script you wish to convert.

2. Run the following command to convert a Tconf script to a RTSC configuration script. (Notice the space after the --xp= option.)

```
>xs -–xp= "%XDCPATH%" ti.bios.conversion –c myConfig.cfg –-pf myPlatform.xs
<filename>.tcf
```

Running this command for a Tconf script that loads the “evmDM6437” platform yields output similar to the following:

```
Platform: ti.platforms.evmDM6437
     params.mem:[object Object],[object Object]
     params.catalogName:ti.catalog.c6000
     params.deviceName:DM6437
     params.clockRate:594
     params.regs:[object Object]
Target: ti.targets.C64P
Clock Rate: 594
```

This command also creates the following files:

- myConfig.cfg
- myConfig.h
- myPlatform.xs

The “ti.bios.conversion” utility converts the *.tcf file to a RTSC configuration file called myConfig.cfg. This RTSC configuration script may now be used to configure your application and build with DSP/BIOS 6.

The contents of the myConfig.cfg file will be very similar to those of the original *.tcf file. The tool essentially strips out calls to `utils.loadPlatform()`, `prog.decl()`, `prog.extern()`, and `prog.gen()`, leaving all other code intact. The tool also generates the file myConfig.h, which may need to be included by your application's C source file.

Note: If any of the generated files (myConfig.cfg, myConfig.h, or myPlatform.xs) already exist in the current directory, the conversion tool overwrites them with new generated files. To prevent any data loss, please make sure to back up any files that have the same name as those specified as command line arguments to the conversion tool.

3 Converting and Updating Your CCStudio Project

The following subsections describe changes you may need to make within your Code Composer Studio project.

3.1 Migrating Code Composer Studio Projects

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

If you are migrating from Code Composer Studio 3 to Code Composer Studio 4, you will need to migrate the build environment, which essentially means migrating the CCStudio 3 project file to a CCStudio 4 project.

Since CCStudio 4 projects are completely different from CCStudio 3 project files, you will need to recreate the project manually. As of this document's publication, there is no tool for converting CCStudio 3 *DSP/BIOS 5.x* projects to CCStudio 4 *DSP/BIOS 6.x* projects.

Use the following steps to migrate your legacy DSP/BIOS 5 application to build with DSP/BIOS 6. These steps are covered in greater detail for the mailbox example in Section 7.3, "Creating a CCStudio 4 Project for the Mailbox Program".

1. Use the CCStudio 4 project creation wizard to create a new CCStudio 4 project, selecting the correct hardware platform and enabling DSP/BIOS.
2. Configure the project options to match those from your CCStudio 3 project. (Don't worry; it's easier to do than it sounds!) While configuring the new CCStudio 4 project, it is important that you transfer the compiler options, assembler options, linker options, and include paths to the new project, as well as any special #defines configured in the CCStudio 3 project. In addition, if you set any Tconf build options, use those as well for the XDCtools build options.
3. Add your source files to the project.
4. Add the migrated RTSC configuration to the project. Details for converting a Tconf configuration to a RTSC configuration are provided in Section 2, "Converting Your Configuration". Also, please see Section 7.2, "Configuring the Mailbox Example".
5. Rebuild the project, load the program, and then run.

See the RTSC+Eclipse QuickStart topic in the RTSC-pedia wiki for details about creating a CCStudio 4 project that uses RTSC and DSP/BIOS 6 content. See the CCStudio 4 online help for more about converting CCStudio 4 projects.

3.2 Updating a CCStudio 4 Project to Build with a Generated Custom Platform File

As mentioned in Section 2.2.3, the conversion tool creates the file “myPlatform.xs”, which is a custom platform file. Custom platform files are used to create user-defined Memory segments and to change the pre-defined Memory segments that ship for a default platform. The one generated by the conversion tool contains settings for any memory changes in the Tconf configuration script passed to it.

In order to use the custom platform file in your program’s build, you must specify and update a config.bld file in the project in order to load it. Use the following steps to build your project using this generated custom platform file:

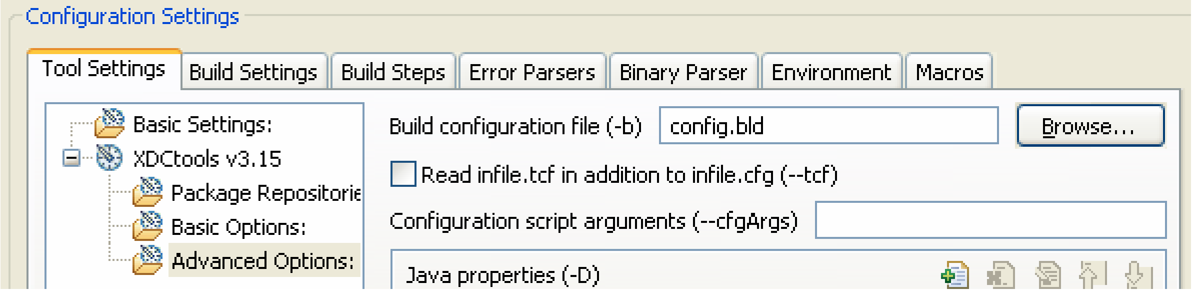
1. In your project directory, create a new file called “config.bld” and open it for editing using your favorite text editor.
2. Add the following line to the config.bld script to load the custom myPlatform.xs platform file:

```
xdc.loadCapsule("myPlatform.xs");
```

3. Save and close the config.bld file.

Note: The following steps apply to you only if you are migrating to Code Composer Studio 4.

1. In CCStudio 4, if it is not open already, open the RTSC configuration project that you want to build using a custom platform file.
2. Make sure that you are in the **C/C++** perspective by clicking the button with that label.
3. In the project pane, right-click on your RTSC configuration project, and select **Build Properties**.
4. In the **C/C++ Build** category, choose the **Tool Settings** tab, and then click the **+** sign next to XDCtools v3.15 to expand the list of XDCtools options. Choose **Advanced Options**.
5. For **Build configuration file (-b)**, type “config.bld”.

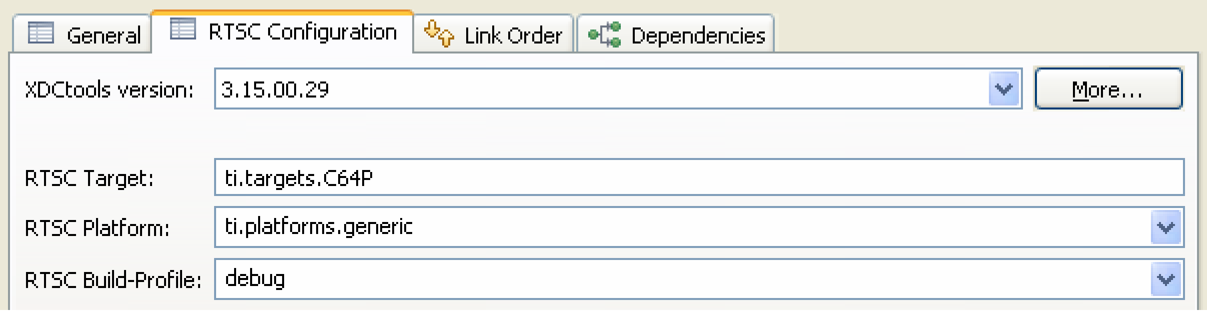


Build configuration file (-b) config.bld

□ Read infile.tcf in addition to infile.cfg (--tcf)

Configuration script arguments (--cfgArgs)

6. In the **CCS Build Settings** category, choose the **RTSC Configuration** tab.
7. For **RTSC Platform (–p)**, select “ti.platforms.generic:plat”.



XDCtools version: 3.15.00.29

RTSC Target: ti.targets.C64P

RTSC Platform: ti.platforms.generic

RTSC Build-Profile: debug

8. Click **OK** to apply these changes to the project.

For more information on custom platform files and how to use them in CCStudio 4, see Section 10.2, “Building the Mailbox Example Using the Custom evmDM6437.xs Platform File”.

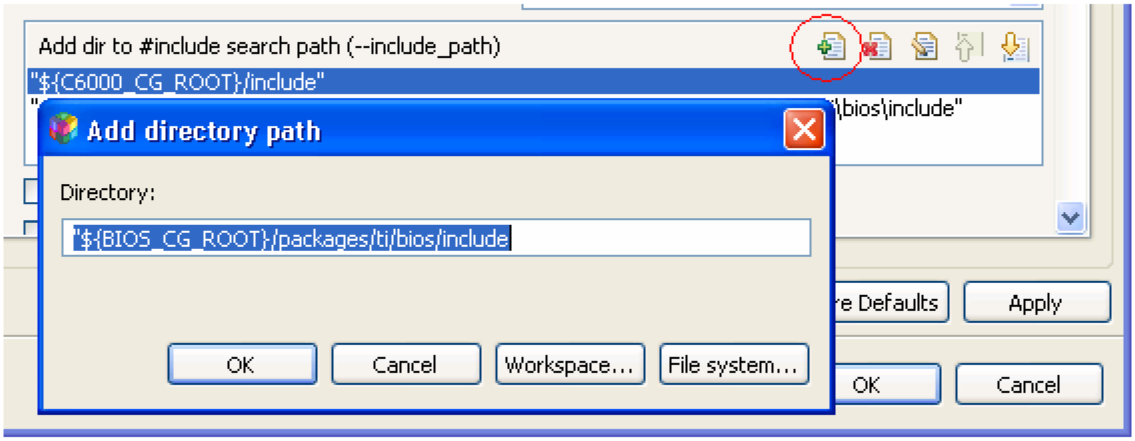
3.3 Updating the Project’s Compiler Search Path

Since your program’s C file probably contains #include directives for legacy DSP/BIOS header files, such as std.h, your project may need to be updated to pass the path to these legacy header files to the compiler. This section shows how to update your project to add the following include path:

```
<DSP/BIOS 6 Install Dir>/packages/ti/bios/include
```

Follow these steps to add this include file directory to the compiler search path. You will need to perform these steps for *both* your main project and the referenced RTSC configuration project.

1. In the CCStudio 4 project pane, right-click on your main project and select **Properties**.
2. In the left pane of the project’s properties window, select **C/C++ Build**. You will see the build properties in the right pane.
3. Under Configuration Settings, choose the **Tool Settings** tab, then click the + sign next to **C6000 Compiler v6.1** to show the list of compiler option categories:
4. Click **Include Options**, then click the + icon next to **Add dir to #include search path (--include path)**:



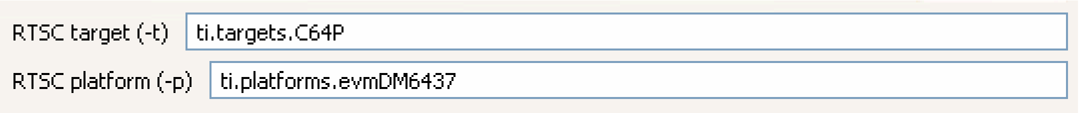
Add dir to #include search path (--include_path)

"${C6000_CG_ROOT}/include"

\bios\include"

Directory: "${BIOS_CG_ROOT}/packages/ti/bios/include

5. Type the path to the DSP/BIOS legacy include directory in the Directory field and click **OK**.
6. For the RTSC configuration project only, in the **Tool Settings** tab, click the + sign for **XDCtools v3.15** to expand the XDCtools options list. Click **Basic Options**. Update the RTSC target (-t) and RTSC platform (-p) fields with the appropriate values for your hardware. For example, if you are using the evmDM6437, then type the following values for these fields:



RTSC target (-t) ti.targets.C64P

RTSC platform (-p) ti.platforms.evmDM6437

7. Click **OK** in the Properties dialog to apply the settings you added.

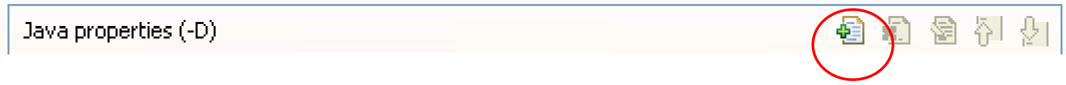
3.4 Updating a CCStudio 4 Project's Search Path to Find Included Configuration Files

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

If your configuration imports any Tconf include files (*.tci), then you must update the build options of your project to add the paths to where those files are located. This is necessary so that RTSC can locate them at configuration time. Note that if you have converted your Tconf script using the conversion tool with the `–i` option, these steps won't be necessary, as all of the content of the Tconf include files will have been migrated into the generated RTSC configuration script.

Follow these steps to update your project settings with the search path to these include files:

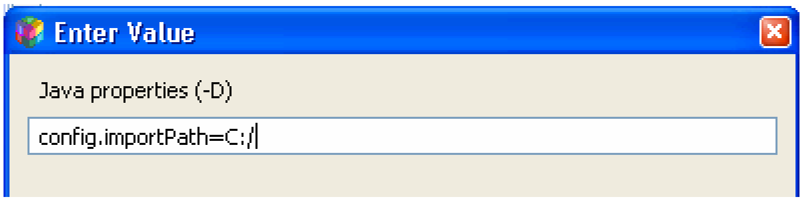
1. In CCStudio 4, make sure that you are in the "C/C++ Perspective" by clicking the button with that label.
2. If your RTSC configuration project is not open in CCStudio 4, open it by selecting the **Project → Open Project** menu option. Make sure the RTSC configuration project is the active project.
3. In the project pane, right click on your project, and select **Build Properties**. Choose the **C/C++ Build** category.
4. Under "Configuration Settings", click the tab labeled **Tool Settings** and then click the "**+**" sign next to **XDCtools v3.15** to expand the list of XDCtools options:
5. Click **Advanced Options**, then click the **"+"** icon for the box that says "Java properties (-D)".



6. In the Enter Value dialog that opens, add:

```
config.importPath=<full path to Tconf include files>
```

For example, if your Tconf include files are all stored in the root of drive C:\, you should add the following to the text box:



Java properties (-D)

config.importPath=C:/

7. Click **OK** to accept the new option.

8. Click **OK** in the "Properties for <*your project*>" dialog to apply the settings just added. All of your program's Tconf include files will now be found at configuration time the next time you build the project.

3.5 Building and Running the Project

At this point, if your legacy C code does not contain any deprecated DSP/BIOS APIs, you will be able to build your project.

1. Create a target configuration by choosing **File → New → Target Configuration**. Use the **Basic** tab to create a target configuration. Press Ctrl+S to save the configuration. Use **View → Target Configurations** to see a list of your saved configurations. Right-click on the one you want to use and choose **Set As Default** from the pop-up menu.

2. Select **Project → Build Active Project** to build the mailbox application using DSP/BIOS 6.

3. Select **Target → Debug Active Project** to launch the TI debugger and load the program.

4. Run the application by selecting **Target → Run** or by pressing the F8 key.

If you have build problems in your C code at this point, it may be because your program is using modules and or APIs that are no longer supported. Please refer to the table of legacy modules in Section 4.1 and the list of deprecated APIs in Appendix A for information on unsupported APIs and modules.

4 Converting Your C Code

Most DSP/BIOS 5 legacy C APIs still exist in DSP/BIOS 6, and may be called without making any C code changes. However, a small number of modules and their APIs are no longer supported, either because they are no longer part of DSP/BIOS or because support for them did not make sense. The number of these deprecated modules and APIs was kept to a minimum.

4.1 Legacy Module Mappings and API Guide

The following table maps legacy DSP/BIOS 5 modules to their DSP/BIOS 6 module counterparts. It lists both the legacy support module for identical APIs and the “new version” of that module introduced in DSP/BIOS 6. While most legacy modules are supported, we recommend that you use the new DSP/BIOS 6 modules if you are writing new code.

Table 1. Module Mappings from DSP/BIOS 5 to DSP/BIOS 6

DSP/BIOS 5 Legacy Module	DSP/BIOS 6 Legacy Module	Recommended DSP/BIOS 6 Module	Legacy APIs Supported?	Support Plan
ATM	None	None	Yes	Legacy support only
BCACHE	None	ti.sysbios.hal.Cache	Yes	--
BUF	ti.bios.BUF	ti.sysbios.heaps.HeapBuf	Yes	--
C62	None	ti.sysbios.hal.Hwi	Yes	--
C64	None	ti.sysbios.hal.Hwi	Yes	--
CLK	ti.bios.CLK	ti.sysbios.knl.Clock and xdc.runtime.Timestamp	Yes	--
DEV	ti.bios.DEV	ti.sdo.io.IDriver	Yes	--
DIO	ti.bios.DIO	None	Yes	Legacy support only
DGN	ti.bios.DGN	ti.sdo.io.drivers.Generator	Yes	--
DGS	None	None	No	Will not be supported
DHL	None	None	No	Will not be supported
DNL	None	None	Yes	Legacy support only
DOV	None	None	Yes	Legacy support only
DPI	ti.bios.DPI	None	Yes	Legacy support only
DST	None	None	No	Will not be supported
DTR	None	ti.sdo.io.converters.Transformer	Yes	--
ECM	ti.bios.ECM	ti.sysbios.family.c64p. EventCombiner	Yes	--
GBL	ti.bios.GBL	ti.sysbios.BIOS and ti.sysbios.hal.Cache	Yes*	--

DSP/BIOS 5 Legacy Module	DSP/BIOS 6 Legacy Module	Recommended DSP/BIOS 6 Module	Legacy APIs Supported?	Support Plan
GIO	ti.bios.GIO	ti.sdo.io.Stream	Yes	--
HOOK	ti.bios.HOOK	ti.sysbios.knl.Task	Yes	--
HST	None	None	No	Will not be supported
HWI	ti.bios.HWI	ti.sysbios.hal.Hwi	Yes*	--
IDL	ti.bios.IDL	ti.sysbios.knl.Idle	Yes	--
LCK	ti.bios.support.Lck	ti.sysbios.gates.GateMutex	Yes	--
LOG	ti.bios.LOG	xdc.runtime.LoggerBuf	Yes	Supported in XDCtools
MBX	ti.bios.MBX	ti.sysbios.ipc.Mailbox	Yes	--
MEM	ti.bios.MEM	ti.sysbios.heaps.HeapMem and xdc.runtime.Memory	Yes	--
MPC	None	None	No	To be supported in a future release
MSGQ	ti.bios.MSGQ	None	Yes	To be supported in a future release
PIP	None	None	No	Will not be supported
POOL	ti.bios.POOL	None	Yes	To be supported in a future release
PRD	ti.bios.PRD	ti.sysbios.knl.Clock	Yes*	--
QUE	ti.bios.QUE	ti.sysbios.misc.Queue	Yes	--
RTDX	ti.bios.RTDX	ti.rtdx.RtdxModule and ti.rtdx.driver.RtdxDvr	No	Supported in a separate product
SEM	ti.bios.SEM	ti.sysbios.ipc.Semaphore	Yes	--
SIO	ti.bios.SIO	ti.sdo.io.Stream	Yes	--
STS	ti.bios.STS	ti.sysbios.misc.Stats	Yes	--
SWI	ti.bios.SWI	ti.sybios.knl.Swi	Yes	--
SYS	ti.bios.SYS and ti.bios.support.Sys	xdc.runtime.System and xdc.runtime.Error	Yes	Supported in XDCtools
TRC	None	xdc.runtime.Diags	Yes	Supported in XDCtools
TSK	ti.bios.TSK	ti.sysbios.knl.Task	Yes*	--

***** Most legacy APIs are supported for this module. See Appendix A for the list of APIs that are no longer supported for this module.

4.2 Extending Your Legacy Program with DSP/BIOS 6 APIs

Once you have updated a legacy program to build and run using DSP/BIOS 6, you can expand that program with code that leverages the new DSP/BIOS 6 APIs and features.

Using a combination of legacy code and new code in the same DSP/BIOS program is supported. However, if you want to combine the old with the new like this, there are certain precautions you must take to ensure that things will work smoothly. This section provides guidelines and examples for extending a DSP/BIOS 5 legacy program using the DSP/BIOS 6 modules in the C code and RTSC configuration.

See Sections 9 and 9.2 for examples that add DSP/BIOS 6 API calls to the mailbox example.

4.2.1 Place New Configuration Code After Legacy Configuration Code

When you extend a legacy application's RTSC configuration file with configuration statements for DSP/BIOS 6 modules, you must place the new configuration code *after* the existing legacy configuration code.

If DSP/BIOS 6 modules are configured before legacy modules, the result may be a build-time error that states that certain legacy modules must be used before certain new modules. You can easily avoid this problem by adding all configuration of DSP/BIOS 6 and XDCtools modules to the end of your RTSC configuration script. This includes any `useModule()` calls, as well as any code to configure those modules.

If a configuration contains settings for equivalent properties using both the DSP/BIOS 6 and DSP/BIOS 5 names, then the DSP/BIOS 6 setting takes precedence. For example, if a configuration contained the following statements, the net result of these equivalent settings would be to configure the CPU speed to be 400MHz (the DSP/BIOS 6 setting).

```
bios.GBL.CLKOUT = 500;                          /* DSP/BIOS 5 */

xdc.global.BIOS = xdc.useModule('ti.sysbios.BIOS");   /* DSP/BIOS 6 */
BIOS.cpuFreq = 400;
```

4.2.2 Place Header File #include Directives for DSP/BIOS 6 Before Legacy Directives

When you update a legacy C file to include DSP/BIOS 6 header files, it is important that you add the #include directives for those header files before the existing #include directives for legacy DSP/BIOS header files.

Currently, if you add any DSP/BIOS 6 header file #include directives after any legacy #includes, then the DSP/BIOS 6 API calls in that C file will be listed as undefined symbols at link time. So, when updating a legacy file for use with DSP/BIOS 6, just make sure to include the DSP/BIOS 6 headers first.

There is an alternative workaround for this issue, in case you do not want to worry about the order of #include directives. If any DSP/BIOS 6 APIs are listed as unresolved symbols due to the order of header file inclusion, you may instead call those APIs using their "long names," which prepend the package name for a particular module's API.

For example, suppose the following call results in an undefined symbol error due to #include ordering:

```
Task_sleep(10);
```

You can change this DSP/BIOS 6 API call to the following to use the “long name” alternative to resolve the undefined symbol:

```
ti_sysbios_knl_Task_sleep(10);
```

4.2.3 Communicating Between Old Code and New Code

Your program can communicate between legacy code and new code that uses DSP/BIOS 6 APIs. For example, if your program contains a legacy TSK C function, and a new Task function has been added, communication between the two is allowed with some simple guidelines.

In general, it is highly recommended that your new DSP/BIOS 6 code use legacy APIs and objects to communicate with legacy code, and vice versa.

An example of this is a legacy TSK that pends on a legacy SEM object via the following call:

```
SEM_pend(legacy_sem, 10);
```

If you add a DSP/BIOS 6 Task function and want this Task function to post the SEM object legacy_sem, then the new Task function should use the *legacy* DSP/BIOS API to do so, even though this Task itself is *not* legacy code, as follows:

```
SEM_post(legacy_sem);
```

The same applies for the opposite case; if a new Task function pends on a DSP/BIOS 6 Semaphore object via the following call:

```
Semaphore_pend(new_sem, 10);
```

Then a legacy TSK function should also post the Semaphore using the new code:

```
Semaphore_post(new_sem);
```

All interactions between new and old code should match new APIs/objects and legacy APIs/objects in this manner.

To be clear, the following example shows an incorrect use of mixing and matching of old code with new code. The following **incorrect** code demonstrates how *not* to mix new APIs with legacy objects:

```
Semaphore_post(legacy_sem); // WRONG! – can’t use new API with legacy object!
```

5 Migrating Memory Configurations

DSP/BIOS 5 provided MEM, the memory manager, for managing new memory segments, heaps and sections, and for loading specific sections into certain memory segments. The MEM module was also used for dynamic memory allocation.

All this functionality is available in DSP/BIOS 6 in the Memory (xdc.runtime.Memory), HeapMem (ti.sysbios.heaps.HeapMem), and Program modules. The MEM module no longer exists.

The subsections that follow provide an overview of Memory segments and sections, Heaps, and dynamic memory allocation in DSP/BIOS 6.

5.1 Memory Segments and Sections

In DSP/BIOS 5, the MEM manager module was used to create new MEM segments for user applications. In DSP/BIOS 6, MEM segments have been replaced by XDCtools Memory segments, which are specified in the platform file.

The most significant difference between MEM segments and Memory segments is how they are created. While MEM segments were created or modified in the configuration script, Memory sections are created or modified using a platform file.

The DSP/BIOS 6 installation includes several platform files that define Memory segments. However, to change these Memory segments requires defining a custom platform file and using that to build the application. Section 10.3 shows how to define new Memory sections using a custom platform file.

The DSP/BIOS 5 MEM manager used section names that could be specified to load to a certain MEM segment. This also exists in DSP/BIOS 6, but the means to load Memory segments differs.

Memory sections in DSP/BIOS 6 are specified to load into Memory segments using the Program module’s `sectMap[ ]` array. This array maps section name strings to the Memory segments into which that section is to be loaded. This pseudo-configuration code shows a mapping:

```
Program.sectMap[“<section name>”] = <Memory segment>.name;
```

You should replace <*section name*> above with the string name of the section to be mapped. Also, replace <*Memory segment*> with the Memory segment name to load that section into.

Using the mailbox example for the evmDM6437 platform, to load the .text section of the application into the IRAM Memory segment instead of to the DDR2 Memory segment, redefine the mapping in the “mailbox.cfg” RTSC configuration script using the Program module as follows:

```
var IRAM = Program.cpu.memoryMap.IRAM;
Program.sectMap[“.text”] = IRAM.name;
```

You have access to all of the existing Memory segments via the Program module’s ”cpu.memoryMap”. Use “Program.cpu.memoryMap” to get a reference to any Memory segment defined for that program’s platform.

Some modules allow you to specify a section name for that module’s data to be loaded into. For instance, the Task module allows you to specify (or create) a section name for a given Task instance’s stack using the Task module’s Params structure member “stackSection”. Once a section name is specified using “stackSection”, that section name can be mapped to load into or run from a particular Memory segment by way of the `sectMap[ ]` array.

For example, the following configuration code creates a static Task instance, then creates a new section name "myTaskStackSection" for its stack, and maps this section to load into the IRAM Memory segment:

```
xdc.global.Task = xdc.useModule('ti.sysbios.knl.Task');
var taskParams = new Task.Params();
taskParams.stackSection = "myTaskStackSection";
Task.create('&reader', taskParams);
Program.sectMap[".myTaskStackSection"] = "IRAM";
```

5.2 Heaps

DSP/BIOS 6 provides the HeapMem module to create heaps and manage a program’s memory. Both DSP/BIOS 6 and XDCtools provide other Heap modules (each of which performs memory management duties), as well as the IHeap interface.

All DSP/BIOS and XDCtools Heaps implement the IHeap interface. In fact, you may create your own Heap module by inheriting the IHeap interface. This allows you to define your own memory management algorithms for your custom Heap.

It’s also important to note that the definition of “Heap” as applied to these modules is very generic. Heap modules are heaps in the sense that they provide memory management; this definition does not imply variable size, non-determinism, or fragmentation, only managed memory.

This section focuses on the HeapMem module for explaining Heaps. The full list of Heap modules provided by DSP/BIOS 6 and XDCtools are listed in the table at the end of this section.

A Heap is created and configured in the application’s RTSC configuration script. Heap configuration parameters include size, alignment, and section name. The following configuration code creates a Heap of size 4096 using the HeapMem module:

```
xdc.global.HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
var heapMemParams = new HeapMem.Params();
heapMemParams.size = 4096;
heapMemParams.sectionName = "myHeapSection";
Program.global.myHeap = HeapMem.create(heapMemParams);
```

This code uses the HeapMem Params structure to set properties for the new Heap. Notice that a new Memory section name is created and used to assign the Heap to a particular Memory segment.

The code to assign the Heap “myHeap” to the Memory segment IRAM using the Program module would look like this:

```
Program.sectMap["myHeapSection"] = “IRAM”;
```

A limitation of the DSP/BIOS 5 MEM manager was that for any MEM segment, you could only create a single heap in that segment.

One benefit of DSP/BIOS 6 is that you can create more than one heap for a given Memory segment. The following code creates a second IRAM Heap, HeapMem, of size 8192:

```
heapMemParams.size = 8192;
heapMemParams.sectionName = "myOtherHeapSection";
Program.global.myOtherHeap = HeapMem.create(heapMemParams);
Program.sectMap["myOtherHeapSection"] = “IRAM”;
```

DSP/BIOS 6 and XDCtools provide several Heap modules, as shown in the following table.

Table 2. DSP/BIOS 6 and XDCtools Heap Modules

Heap Module	Description
xdc.runtime.IHeap	Heap interface. All Heap modules inherit from and implement this interface. You may inherit from the IHeap interface to implement your own Heap module and perform any memory management style.
xdc.runtime.HeapMin	A simple, minimum footprint Heap.
xdc.runtime.HeapStd	This Heap is based on the C RTS functions malloc() and free().
ti.sysbios.heaps.HeapBuf	Single, fixed size buffer, split into blocks of equal size. This Heap corresponds to the DSP/BIOS 5 BUF module.
ti.sysbios.heaps.HeapMem	HeapMem corresponds to the DSP/BIOS 5 MEM module.
ti.sysbios.heaps.HeapMultiBuf	Allows you to create many different HeapBuf Heaps of different sizes.

You can find more information on Heap modules in the online CDOC reference documentation.

5.3 Dynamic Memory Allocation

The DSP/BIOS 5 MEM module supported the `MEM_alloc()` and `MEM_free()` APIs to dynamically allocate and free memory. These functions took a MEM segment ID as an argument to specify which MEM segment to allocate from.

In DSP/BIOS 6, the Memory module and the Heap modules provided by DSP/BIOS 6 and XDCtools replace the DSP/BIOS 5 MEM module’s dynamic memory allocation. The APIs were replaced by the Memory module APIs `Memory_alloc()` and `Memory_free()`, and a Heap is specified as the first argument.

DSP/BIOS 6 provides a default Heap for dynamically allocating and freeing memory without explicitly configuring and creating a new heap. The default heap is specified by passing NULL as the IHeap handle to these functions. The following C code dynamically allocates and frees memory from the default Heap:

```
Ptr buf = Memory_alloc(NULL, 512, 0, NULL);
Memory_free(NULL, buf, 512);
```

Alternately, you may create your own Heap and use it to allocate memory from in your program. The `Memory_alloc()` and `Memory_free()` functions take a Heap handle of type IHeap_Handle as the first function argument (IHeap_Handle is a generic Heap handle). To use a HeapMem Heap in the call to `Memory_alloc()`, the HeapMem handle must be cast to type IHeap_Handle. However, casting a HeapMem handle to type IHeap_Handle could result in data loss, so a special function is provided to cast the handle: `HeapMem_Handle_upCast()`.

The following code can be used to cast the HeapMem handle myHeap to type IHeap_Handle, and allocate and free memory from this heap:

```
Ptr buf;
// “Cast” myHeap to type IHeap_Handle
IHeap_Handle iHeapHandle = HeapMem_Handle_upCast( myHeap );
buf = Memory_alloc(iHeapHandle, 512, 0, NULL);
Memory_free(iHeapHandle, buf, 512);
```

DSP/BIOS 6 also allows a user-created heap to be assigned as the default system heap, overriding the default heap that is provided. This way, dynamically-created DSP/BIOS objects are allocated from the user-created heap that was set as the default heap. The following configuration code sets the default heap of the program to be myOtherHeap:

```
Memory.defaultHeapInstance = Program.global.myOtherHeap;
```

6 Migrating Library Builds

DSP/BIOS 5 and DSP/BIOS 6 applications use a configuration file. As a result of this, the include paths needed by DSP/BIOS to find header files are automatically added as options to the compiler. In DSP/BIOS 6 applications, these options are automatically placed into the generated file “compiler.opt”.

However, since library builds do not use a DSP/BIOS configuration, there are additional steps you must take to rebuild a library. You’ll need to update the library project options by adding the extra include search paths that would normally be added by the generated “compiler.opt” file.

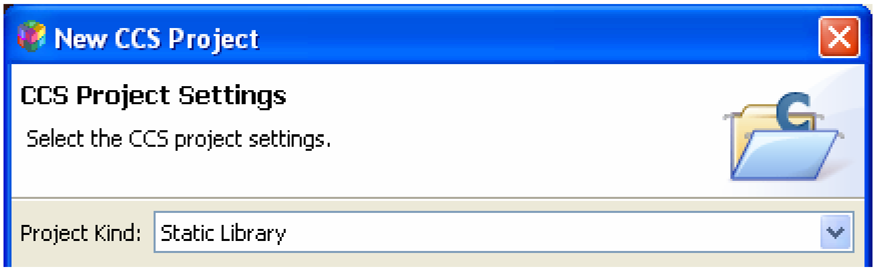
You may rebuild your existing DSP/BIOS 5 libraries using the legacy support included with DSP/BIOS 6. To do so, you’ll need to create a new project within CCStudio 4. This is required because there is currently no way to automatically convert your CCStudio 3.x project to a CCStudio 4 project.

Creating a project for a library build differs from creating a project for an executable, which has been described in the previous examples of this document. Since library builds do not use a RTSC configuration, a few vital compiler search paths will be missing. When you use a RTSC configuration, these search paths are added automatically. However, because there will be no RTSC configuration, you’ll need to add these compiler search paths to the project manually.

6.1 Creating a New Library Project for CCStudio 4

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

Follow the steps to create a new CCStudio 4 project that uses DSP/BIOS 6 as described in the RTSC+Eclipse QuickStart topic of the RTSC-pedia wiki or in the Bios_Getting_Started_Guide.pdf document. However, when you reach the **CCS Project Settings** page in the project-creation wizard, select **Static Library** from the **Project Kind** drop-down box.



New CCS Project

CCS Project Settings

Select the CCS project settings.

Project Kind: Static Library

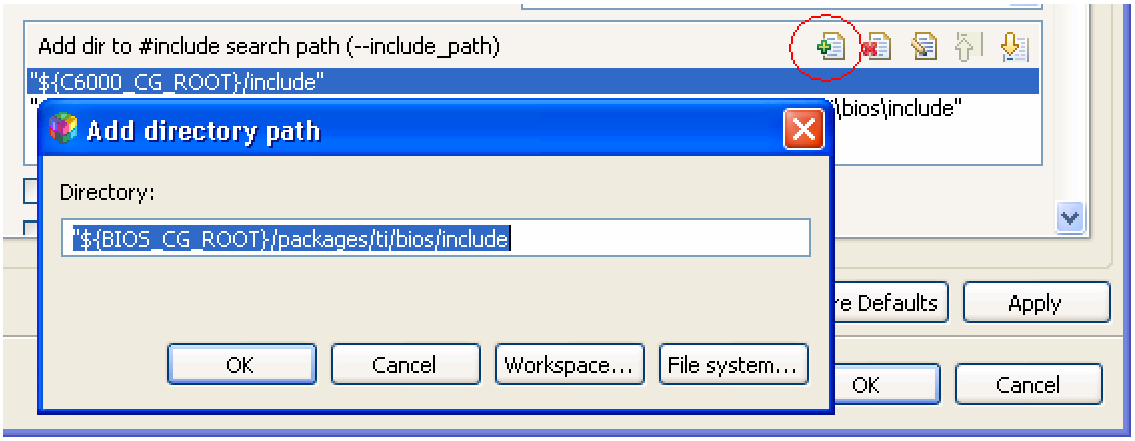
If you select the location of your existing library source files when you begin creating the project, then when you finish creating the project, all of the library source files that exist in the project directory are added to the project automatically.

6.2 Updating the Compiler Search Paths for CCStudio 4 Library Projects

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

Now that the project has been created, it's necessary to update the compiler search paths, as mentioned previously.

1. In the CCStudio 4 project pane, right-click the library project and select **Properties**.
2. In the left pane of the library project's properties window, select **C/C++ Build**. You will see the build properties in the right pane.
3. Under Configuration Settings, click the **Tool Settings** tab, then click the + sign next to **C6000 Compiler v6.1** to show the list of compiler option categories.
4. Click **Include Options**, then click the + icon next to **Add dir to #include search path (--include path)**:



Add dir to #include search path (--include_path)

"${C6000_CG_ROOT}/include"

"...\bios\include"

Add directory path

Directory: "${BIOS_CG_ROOT}/packages/ti/bios/include

Note: For legacy libraries, source files must include the DSP/BIOS file "std.h" as follows:

```
#include <std.h>
```

Including this file has always been a requirement for DSP/BIOS 5 programs, so your library sources likely already include this file correctly. If std.h is not properly included, the library build will fail due to undefined identifiers in the file "xdc/std.h".

5. Type the path to the DSP/BIOS legacy include directory in the Directory field and click **OK**.
6. For both types of library (legacy and DSP/BIOS 6 API), repeat the two previous steps. This time, add the following additional paths to the compiler search path, where you replace the <*DSP/BIOS 6 Install Dir*> and <*XDCtools 6 Install Dir*> with the actual locations:
   - ${BIOS_CG_ROOT}/packages
   - ${XDC_CG_ROOT}/packages
7. Click **OK** in the Properties dialog to apply the settings you added.
8. Select **Project → Build Active Project** to build the library using DSP/BIOS 6.

7 Porting the Mailbox Example to DSP/BIOS 6 Using Legacy Support

DSP/BIOS 6 supports most legacy DSP/BIOS 5 APIs and Tconf configuration properties. The subsections that follow show how to build and run an existing DSP/BIOS 5 application using DSP/BIOS 6 legacy support. That is, the DSP/BIOS 5 APIs are still used, but the project is built with DSP/BIOS 6.

The subsections that follow show how to convert the DSP/BIOS 5 Tconf configuration to a RTSC configuration script, modify the project and C source file, and build and run the application. Section 8 further modifies the application to take advantage of DSP/BIOS 6 modules and APIs.

7.1 Copying the Mailbox Example Files

To preserve the existing application's configuration and source files, use the following steps to copy the application source and configuration files to two new directories before making any changes. These copies will be used for all work in this application note.

1. Create a new directory named “mailbox”. This directory will be used to store the mailbox application's C source files. You may create this directory anywhere on your computer, and it will be referred to as the working directory throughout this document.
2. Copy the mailbox C source file into the "mailbox" directory from <*DSP/BIOS 5 Install Dir*>/packages/ti/bios/examples/basic/mailbox/mailbox.c.
3. Create a new directory named "mailbox_configuration". This directory will be used to store the mailbox application's DSP/BIOS configuration files. You may create this directory anywhere on your computer, and it will be referred to as the configuration directory throughout this document.
4. Copy the following mailbox Tconf configuration files into the "mailbox_configuration" directory:
   - <*DSP/BIOS 5 Install Dir*>/packages/ti/bios/examples/basic/mailbox/mailbox.tcf
   - <*DSP/BIOS 5 Install Dir*>packages/ti/bios/examples /basic/mailbox/mailbox_evmDM6437_custom.tci
   - <*DSP/BIOS 5 Install Dir*>/packages/ti/bios/examples/basic/mailbox/mailbox.tci
   - <*DSP/BIOS 5 Install Dir*>/packages/ti/bios/examples/common/evmDM6437_common.tci

Note: You should copy the Tconf include files (*.tci) to the configuration directory for convenience. If you do not want to move an application's *.tci files from their existing locations, you must update the CCStudio 4 mailbox_configuration project (which will be created in the next section) with the paths to these *.tci files. Otherwise, these files will not be found at build time. For instructions on how to update your CCStudio 4 project with the path locations of *.tci files, see Section 3.4.

Similarly, if you use the conversion tool to convert your Tconf script to a RTSC configuration script, and the script contains *.tci files that exist in different directories, you'll need to pass relative paths to these files on the command line during the conversion step. Do this using the `--xp` XDCPATH command line option to the xs command, so the conversion tool and XDCtools can find these *.tci files. For further details on this, see Section 2.2.1, “Conversion Tool Syntax”.

7.2 Configuring the Mailbox Example

While there is more than one way to migrate a DSP/BIOS 5 Tconf configuration to a RTSC configuration, you must choose which is best, based on your existing Tconf script. Section 2, ”Converting Your Configuration,” discusses configuration scripts and how to classify your Tconf configuration as “simple” or “complex.” Based on that classification, you should choose one of two means of converting a Tconf configuration to a RTSC configuration.

Looking at the mailbox.tcf Tconf configuration, you can see that it should be classified as “simple,” as it does not make changes to memory that are not supported at configuration time.

Based on this classification, the recommended route to take in moving the DSP/BIOS 5 mailbox configuration to DSP/BIOS 6 is a manual conversion. However, we’ll show both methods in the following subsections so that you can see how to perform each method.

7.2.1 Coverting the Mailbox Configuration Manually

Since the mailbox.tcf script is relatively simple and contains no changes to the evmDM6437 platform’s memory map, the easiest way to convert it into a RTSC configuration script is by hand. Because there are no memory map changes, if the conversion tool were used, it would just parse the Tconf configuration files and remove the following lines of configuration code:

- `utils.loadPlatform("ti.platforms.evmDM6437");`
- `prog.gen()`

It will also add one line of code to bring in DSP/BIOS legacy support. These changes are so simple that it is easiest to make them by hand and rename the mailbox.tcf file to mailbox.cfg.

Follow these steps to translate the mailbox.tcf script to a RTSC configuration script mailbox.cfg that will allow this program to build with DSP/BIOS 6 legacy support:

1. In the configuration directory, rename the “mailbox.tcf” file to “mailbox.cfg”.

2. Open mailbox.cfg in your favorite text editor.

3. Near the beginning of the file, delete the following line of code from the file:

```
utils.loadPlatform("ti.platforms.evmDM6437");
```

4. In place of the call to `utils.loadPlatform()`, add the following line of code to mailbox.cfg to bring in the necessary DSP/BIOS 6 legacy support:

```
xdc.loadPackage('ti.bios.tconf');
```

5. Near the end of the file, delete the following lines of code:

```
if (config.hasReportedError == false) {
    prog.gen();
}
```

6. To see the full output of the trace log statements in CCSv4, you need to enable the mailbox program to use DSP/BIOS Real Time Analysis (RTA). In order to do this, you will need to set the flush property of the trace log to “true” in the configuration file, as well as configure the RTA and RTDX modules. Note that this step is *not* necessary for the program to work with DSP/BIOS 6; the program still runs without these changes. But, it is nice to see all of the example’s output, if merely for aesthetic reasons. Also, this extra step would not be done by the conversion tool, if it were used to convert the mailbox.tcf file.

Add the following lines of code to the end of the mailbox.cfg file in order to enable flushing for the trace log and configure RTA and RTDX:

```
trace.exitFlush = true;

/* Enable RTA */

/* Bring in and configure the RTA Agent */
xdc.global.Agent = xdc.useModule('ti.sysbios.rta.Agent');

/* This example generates a lot of RTA traffic so give plenty of buffer
 * room to the Agent. */
Agent.numSystemRecords = 8192;

/* Bring in and configure the necessary RTDX modules */
xdc.global.RtdxModule = xdc.useModule('ti.rtdx.RtdxModule');
xdc.global.RtdxDvr = xdc.useModule('ti.rtdx.driver.RtdxDvr');

var rtdx = RtdxDvr.create();

xdc.global.DriverTable = xdc.useModule('ti.sdo.io.DriverTable');
DriverTable.addMeta("/rtdx", rtdx);

RtdxDvr.numInputChannels = 2;
RtdxModule.bufferSizeInWords = 4096;
```

7. Save and close the mailbox.cfg file.

7.2.2 Converting the Mailbox Configuration Using the Conversion Tool

DSP/BIOS 6 includes a conversion utility called “ti.bios.conversion”, which is run using the XDCtools command `xs`.

This section demonstrates how to convert the mailbox.tcf Tconf script to the mailbox.cfg RTSC configuration script using the conversion tool. This step is optional, and should not be done if you have already converted the mailbox.tcf script to mailbox.cfg script manually, as described in the previous section. This step is only provided in order to give you an example of how to use the conversion tool in case you are converting complex configurations.

For more information on the conversion tool, please refer to Section 2.2, “Converting a Configuration Using the Conversion Tool”.

1. At a MS-DOS command prompt, change (cd) to the configuration directory.
2. Run the following command to convert the mailbox.tcf Tconf script in the configuration directory to a RTSC configuration script. (Notice the space after the --xp= option.)

```
>xs --xp= "%XDCPATH%;.." ti.bios.conversion –c mailbox.cfg --pf evmDM6437.xs
mailbox.tcf
```

Running this command produces the following output:

```
Platform: ti.platforms.evmDM6437
     params.mem:[object Object],[object Object]
     params.catalogName:ti.catalog.c6000
     params.deviceName:DM6437
     params.clockRate:594
     params.regs:[object Object]
Target: ti.targets.C64P
Clock Rate: 594
```

This command also creates the following files:

- mailbox.cfg
- mailboxcfg.h
- evmDM6437.xs

The “ti.bios.conversion” utility converts the application’s “mailbox.tcf” file to a RTSC configuration file called “mailbox.cfg.” The contents of the “mailbox.cfg” file will be very similar to the “mailbox.tcf” file. The tool also re-generates the “mailboxcfg.h” file in its DSP/BIOS 6 form. This file is included by the C source file “mailbox.c.”

Lastly, the tool creates the file “evmDM6437.xs”, which is a custom platform file. Custom platform files can be used to create user-defined Memory segments. For more information on custom platform files and how to use them, please see Section 10, “Building the Mailbox Example Using a Custom Platform File”.

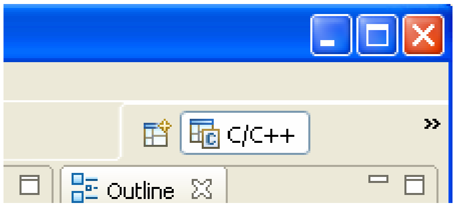
7.3 Creating a CCStudio 4 Project for the Mailbox Program

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

The **Project → Import Legacy CCSv3.3 Project** command in CCSv4 does not migrate a DSP/BIOS 5 project to the DSP/BIOS 6 project organization, which uses separate projects for the source code and the RTSC configuration. So, instead of migrating the existing project, it is recommended that you create a new DSP/BIOS 6 project.

This section walks you through the CCStudio 4 project creation wizard to create a new project for the legacy mailbox example. In CCSv4, DSP/BIOS 6 projects typically consist of 2 separate projects, one for sources, and another dependent project which handles the RTSC configuration. For more about the project creation wizard, see the RTSC+Eclipse QuickStart topic of the RTSC-pedia wiki. By following these steps, you will learn the correct settings to use when creating a DSP/BIOS 6 project in CCStudio 4, how to modify the project to choose a different hardware platform, how to update compiler and XDCtools options, and how to use XGCONF to view and change the settings in the mailbox.cfg file.

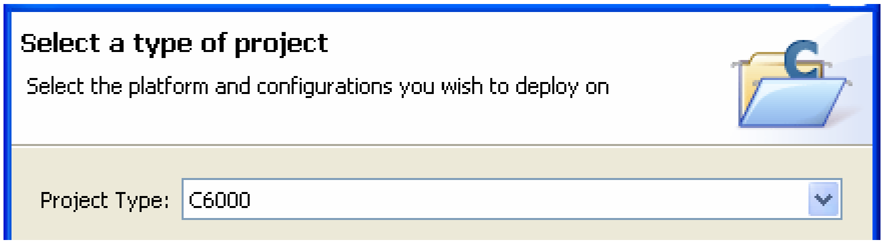
1. In CCStudio 4, enter the **C/C++** perspective by clicking the corresponding button:



2. From the File menu, select **File → New → CCS Project**. This opens the new project wizard.

3. In the Project Name field, type **mailbox**, then un-check the **Use default location** checkbox. Once unchecked, click the **Browse** button, and then navigate to the working directory, choosing it for the project location.

4. Click **Next** to bring the wizard to the **Select a type of project** page. Choose **C6000** for the project type:



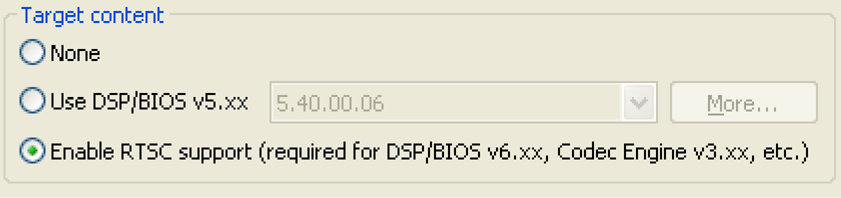
Select a type of project

Select the platform and configurations you wish to deploy on

Project Type: C6000

5. Click **Next** to bring the wizard to the **Additional Project Settings** page. No changes are necessary for this step of the wizard.

6. Click **Next** to bring the wizard to the **CCS Project Settings** page. In this screen, select **Executable** from the **Project Kind** drop-down box. Make sure the device settings are correct. Also, check the **Enable RTSC support** box. Notice that enabling RTSC support allows a project to use DSP/BIOS. All other default settings should be OK.



Target content

○ None

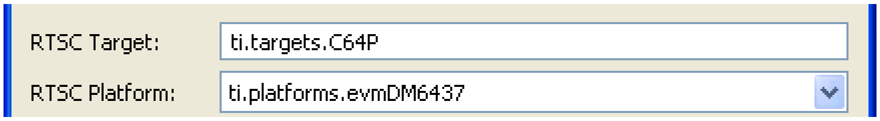
○ Use DSP/BIOS v5.xx 5.40.00.06 More...

● Enable RTSC support (required for DSP/BIOS v6.xx, Codec Engine v3.xx, etc.)

7. Click **Next** to bring the wizard to the **Referenced RTSC Configuration** page. Choose to create a new RTSC configuration project.

8. Click **Next** to bring the wizard to the **New RTSC Configuration Project** page. Use the default name "mailbox_configuration", then un-check the **Use default location** checkbox. Once unchecked, click the **Browse** button, and then navigate to the configuration directory "mailbox_configuration", choosing it for the project location.

9. Click **Next** to bring the wizard to the **RTSC Configuration Settings** page. Change the **RTSC Target** to ti.targets.C64P and the **RTSC Platform** to ti.platforms.evmDM6437.



RTSC Target: ti.targets.C64P

RTSC Platform: ti.platforms.evmDM6437

10. Click **Next** to bring the wizard to the **RTSC Configuration Templates** page. Do not check the box to create a project using a template.
11. Click **Finish** to create the new mailbox project.

Notice that all of the mailbox source files that exist in the working directory are automatically added to the mailbox project, and all of the mailbox configuration files that exist in the configuration directory are automatically added to the mailbox_configuration project.

7.4 Updating the Compiler Search Path

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

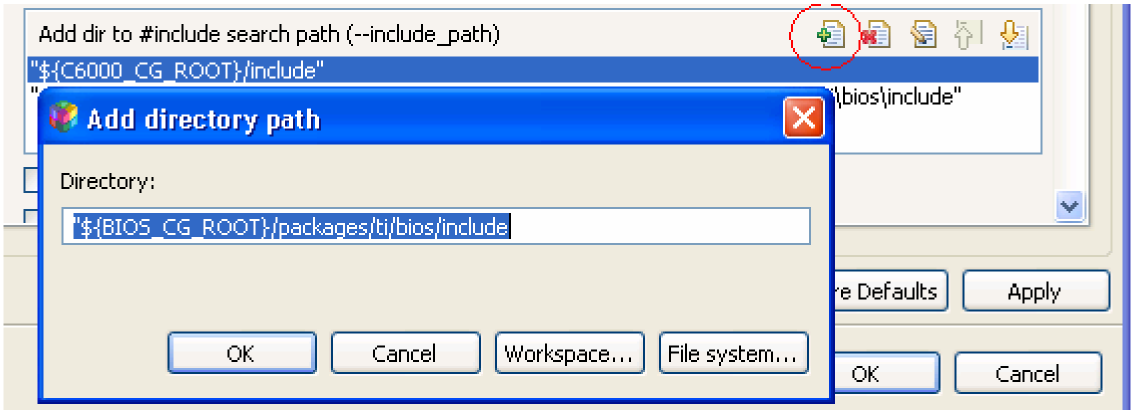
The mailbox.c file contains `#include` directives for legacy DSP/BIOS header files, such as std.h. Since the path to these legacy header files is not currently being passed to the compiler, these legacy header files won't be found at compile time. So, the mailbox project must be updated to add the following include path:

```
<DSP/BIOS 6 Install Dir>/packages/ti/bios/include
```

Additionally, the default hardware platform for the **DM643x** device that you chose during the project creation points to a different RTSC configuration platform than what we need. The steps in this section also show you how to correct the RTSC configuration platform file.

Follow these steps to add this include file directory to the compiler search path for the mailbox program:

1. In the CCStudio 4 project pane, right-click the mailbox project and select **Properties**.
2. In the left pane of the mailbox project's properties window, select **C/C++ Build**. You will see the build properties in the right pane.
3. Under Configuration Settings, click the **Tool Settings** tab, then click the + sign next to **C6000 Compiler v6.1** to show the list of compiler options:
4. Click **Include Options**, then click the + icon next to **Add dir to #include search path (--include path)**:



Add dir to #include search path (--include_path)

"${C6000_CG_ROOT}/include"

"\bios\include"

Add directory path

Directory: "${BIOS_CG_ROOT}/packages/ti/bios/include

5. Type the path to the DSP/BIOS legacy include directory in the Directory field and click **OK**.
6. Click **OK** in the Properties dialog to apply the settings you added.

7.5 Updating the C Source File

The legacy C file mailbox.c contains a reference to the header file mailboxcfg.h, which was generated by the DSP/BIOS 5 configuration. Now that DSP/BIOS 6 is being used to configure the mailbox program, it is necessary to include the new generated file. This is accomplished by including the xdc/cfg/global.h file, which indirectly includes the newly generated mailbox configuration header file.

1. Open “mailbox.c” in a text editor.
2. Replace the following line in the mailbox.c file:

```
#include “mailboxcfg.h”
```

   with this line:

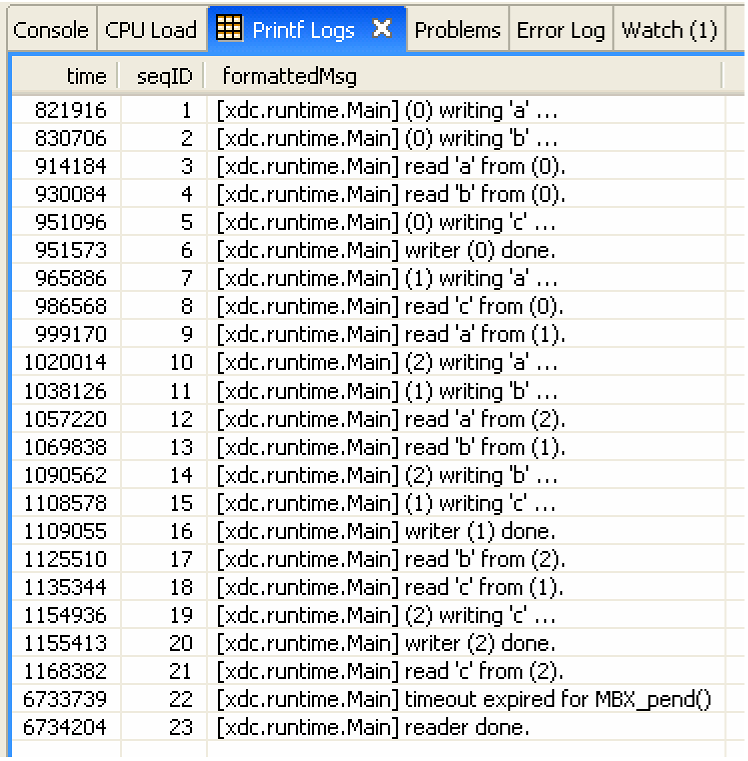
```
#include <xdc/cfg/global.h>
```

3. Save and close the file.

7.6 Building and Running the Application

1. Make sure that the active project in CCSv4 is the “mailbox” project (rather than the “mailbox_configuration” project.
2. Create and activate a target configuration by choosing **File → New → Target Configuration**. Select the default location, then click **Finish** to bring up the general setup for a target configuration.
3. Use the **Basic** tab under general setup to create a target configuration. For **Connection**, select "Spectrum Digital DSK-EVM onboard USB Emulator”. For **Device**, select "TMS320DM6437". Press Ctrl+S to save the target configuration.
4. Use **View → Target Configurations** to see a list of your saved target configurations. Right-click on the one you want to use and choose **Set As Default** from the pop-up menu.
5. Switch to the Debug perspective.
6. Open the RTA log view by selecting **Tools → RTA → Printf Logs**. This opens the RTA Printf Logs pane.
7. Choose **Project → Build Active Project** to build the “mailbox” project using DSP/BIOS 6.
8. Choose **Target → Debug Active Target** to load the mailbox.out executable.
9. Connect CCStudio 4 to the DM6437 EVM board by selecting **Target → Connect Target**.

10. Run the application by selecting **Target → Run** or pressing the **F8** key. Output is displayed in the RTA Printf Logs pane:



time	seqID	formattedMsg
821916	1	[xdc.runtime.Main] (0) writing 'a' ...
830706	2	[xdc.runtime.Main] (0) writing 'b' ...
914184	3	[xdc.runtime.Main] read 'a' from (0).
930084	4	[xdc.runtime.Main] read 'b' from (0).
951096	5	[xdc.runtime.Main] (0) writing 'c' ...
951573	6	[xdc.runtime.Main] writer (0) done.
965886	7	[xdc.runtime.Main] (1) writing 'a' ...
986568	8	[xdc.runtime.Main] read 'c' from (0).
999170	9	[xdc.runtime.Main] read 'a' from (1).
1020014	10	[xdc.runtime.Main] (2) writing 'a' ...
1038126	11	[xdc.runtime.Main] (1) writing 'b' ...
1057220	12	[xdc.runtime.Main] read 'a' from (2).
1069838	13	[xdc.runtime.Main] read 'b' from (1).
1090562	14	[xdc.runtime.Main] (2) writing 'b' ...
1108578	15	[xdc.runtime.Main] (1) writing 'c' ...
1109055	16	[xdc.runtime.Main] writer (1) done.
1125510	17	[xdc.runtime.Main] read 'b' from (2).
1135344	18	[xdc.runtime.Main] read 'c' from (1).
1154936	19	[xdc.runtime.Main] (2) writing 'c' ...
1155413	20	[xdc.runtime.Main] writer (2) done.
1168382	21	[xdc.runtime.Main] read 'c' from (2).
6733739	22	[xdc.runtime.Main] timeout expired for MBX_pend()
6734204	23	[xdc.runtime.Main] reader done.

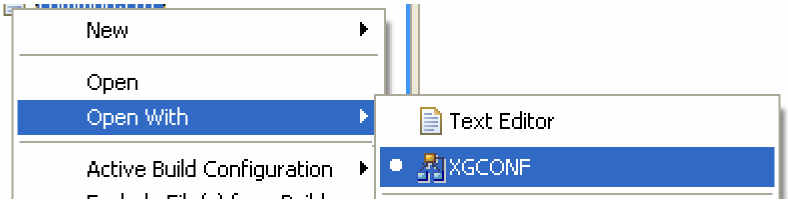
11. Halt the program by selecting **Target → Halt**.

7.7 Using XGCONF to View the DSP/BIOS Configuration

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

You can use the XGCONF tool to view a legacy DSP/BIOS 6 program's RTSC configuration graphically. (For legacy configurations, editing the DSP/BIOS configuration is not supported.) In this section, you will learn how to open the XGCONF tool and use it to view the current configuration settings of the mailbox program. For more about using XGCONF, see the XGCONF User's Guide topic of the RTSC-pedia wiki.

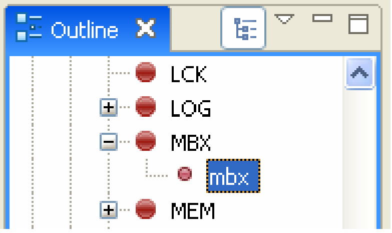
1. To open XGCONF, you must be in the **C/C++** perspective of CCStudio 4. Click the **C/C++** icon to switch back to that perspective.
2. Make sure the project that contains the RTSC configuration file, "mailbox_configuration", is set to be the "Active" project. You can do this by right-clicking on the mailbox_configuration project in the "C/C++ Projects" view and selecting **Set as Active Project**. *This step is important since XGCONF works with the properties of the current active project in the workspace.*
3. Right-click on the "mailbox.cfg" file in the project list and select **Open With → XGCONF**: It takes a few seconds for XGCONF to open. During this time, the CCStudio status bar shows that the configuration is being processed and validated.



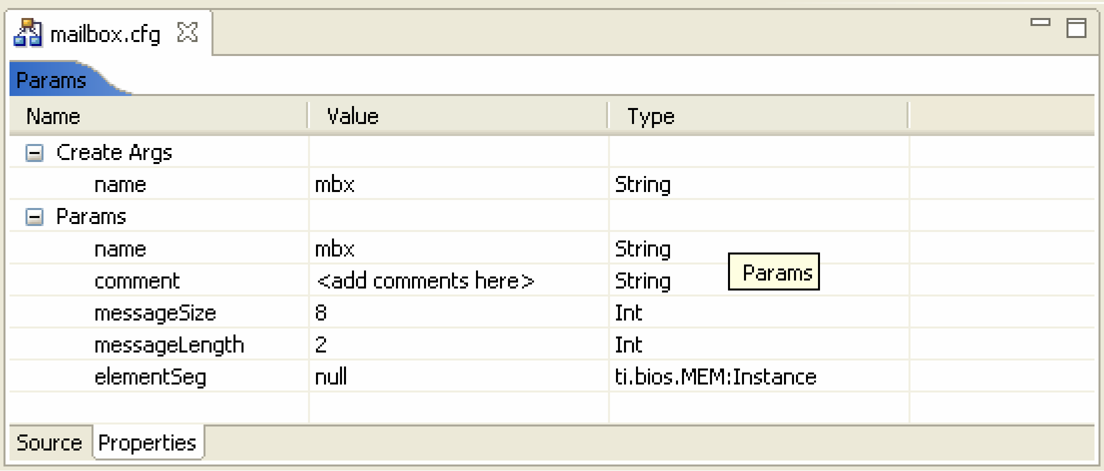
4. Let's take a look at one of the legacy DSP/BIOS configuration objects that exist in the mailbox program. Recall the following configuration code from the file "mailbox_evmDM6437_custom.tci" that creates an MBX object called "mbx":

```
var mbx;
mbx              = bios.MBX.create("mbx");
mbx.messageSize = 8;
mbx.length       = 2;
```

5. You can see the result of this configuration code in XGCONF. In the **Outline** pane, toggle the full tree view of the modules on by clicking the icon. Click the **+** sign next to the **ti.bios** package in the Outline pane to expand the view of this package, which contains legacy modules.



6. Next, click the **+** sign next to the module “MBX”. Notice that there is an instance found under “MBX” called “mbx”. Click on “mbx” to select it. You see the properties for this MBX instance in the properties tab. Notice that its properties match those set in the configuration code:



Name	Value	Type
⊟ Create Args		
name	mbx	String
⊟ Params		
name	mbx	String
comment	<add comments here>	String
messageSize	8	Int
messageLength	2	Int
elementSeg	null	ti.bios.MEM:Instance

7. You can use XGCONF in this way to view other properties of a DSP/BIOS program’s configuration. This is a good time to play around with the tool by viewing the TSK instances and seeing how the properties there match exactly what was configured.

8 Porting the Mailbox Example to DSP/BIOS 6 with New Modules and APIs

In this section, you’ll manually convert the existing, legacy RTSC configuration file and legacy C code of the mailbox example to use the *new* DSP/BIOS 6 modules and C APIs.

In the subsections that follow, you will update the configuration files (Section 8.1), C source files (Section 8.2), and project. Then you will build and run the application (Section 8.3). After following these steps, the application configuration and C file will no longer contain any DSP/BIOS legacy configuration code or C code.

As an alternative to removing all legacy code, Sections 9 and 9.2 provide examples that integrate new DSP/BIOS 6 code with existing legacy DSP/BIOS 5 code.

8.1 Updating the Configuration Files

Where DSP/BIOS 6 configuration code is very similar to DSP/BIOS 5 configuration code, conversion is straightforward. However, there are some differences that make conversion less simple. For example, heaps in DSP/BIOS 6 are configured very differently. The following steps explain the configuration changes needed to port the mailbox application to DSP/BIOS 6.

These steps must be done manually; that is, the conversion tool may not be used to do this. The conversion tool only works for converting a legacy Tconf configuration script to a legacy RTSC configuration script. The conversion tool does not support converting a RTSC configuration that uses legacy DSP/BIOS 6 modules to one that uses non-legacy DSP/BIOS 6 modules.

8.1.1 Porting the mailbox.cfg File

In DSP/BIOS 6, the `useModule()` method enables the application to use a particular module, such as a DSP/BIOS, XDCtools, or a user-defined module. The RTSC configuration script calls `useModule()` for all modules the program configures, and it automatically links all the module's libraries to the application. Calling `useModule()` returns a reference to the module, which is used to configure that module's settings.

For the mailbox example, only DSP/BIOS and XDCtools modules are necessary. The program needs to configure settings for the following modules:

- BIOS
- HeapMem
- Defaults
- Memory
- Cache
- Task
- Mailbox
- Clock
- LoggerBuf
- Diags
- Main

So, the `useModule()` method must also be called for each of these modules in order to link in their libraries and/or configure them.

Follow these steps to use all of the modules for the mailbox program and to remove unneeded legacy code:

1. Open "mailbox.cfg" in a text editor.
2. At the top of the file, add the following code to use all necessary modules:

```
/* use the BIOS module to link in the sysbios library */
xdc.useModule('ti.sysbios.BIOS');

/* use modules for memory heaps and cache configuration */
xdc.global.HeapMem   = xdc.useModule('ti.sysbios.heaps.HeapMem');
xdc.global.Defaults  = xdc.useModule('xdc.runtime.Defaults');
xdc.global.Memory    = xdc.useModule('xdc.runtime.Memory');
xdc.global.Cache     = xdc.useModule('ti.sysbios.family.c64p.Cache');

/* use modules for Task, Mailbox, and Clock functionality */
xdc.global.Task      = xdc.useModule('ti.sysbios.knl.Task');
xdc.global.Mailbox   = xdc.useModule('ti.sysbios.ipc.Mailbox');
xdc.global.Clock     = xdc.useModule('ti.sysbios.knl.Clock');

/* use modules needed for logging */
xdc.global.LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
xdc.global.Diags     = xdc.useModule('xdc.runtime.Diags');
xdc.global.Main      = xdc.useModule('xdc.runtime.Main');
```

3. Remove legacy configuration code from the file. The mailbox.cfg file contains a legacy reference that was necessary in Section 7, but is no longer needed. Delete the following line of code from the file:

```
xdc.loadPackage('ti.bios.tconf');
```

4. Save and close the file.

8.1.2 Porting the evmDM6437_common.tci File

The “evmDM6437_common.tci” file contains code to enable Real Time Analysis, memory heaps, Real Time Data Exchange, and the Task Manager. It also contains configuration code to create a heap in the memory section IRAM, to map the heap in IRAM as the segment for DSP/BIOS objects and mallocs, and to configure the cache and the clock.

In the following steps, you will convert this configuration code to DSP/BIOS 6 and remove unneeded legacy code:

1. Open “evmDM6437_common.tci” in an editor.
2. Delete the following lines of code from the file:

```
/*
 * Enable common BIOS features used by all examples
 */
bios.enableRealTimeAnalysis(prog);
bios.enableMemoryHeaps(prog);
bios.enableRtdx(prog);
bios.enableTskManager(prog);
```

It is no longer necessary to use these legacy methods to enable RTA, memory heaps, RTDX or the Task Manager. The Task Manager is enabled by default in DSP/BIOS 6 and the configuration code to enable RTA and RTDX was already added in Section 7.2.1.

8.1.2.1 Converting the IRAM Heap Creation Code

Next the evmDM6437_common.tci file creates a heap in IRAM and maps the segment as the default heap for DSP/BIOS objects and mallocs.

In DSP/BIOS 6, heaps and segments are configured using the Program, Memory, and HeapMem modules. These modules function in DSP/BIOS 6 as the equivalent of the MEM manager in DSP/BIOS 5.

The following steps create and set up a heap using the Program, HeapMem and Memory modules. The configuration that follows sets the heap’s size, maps it to the IRAM segment, and sets the heap as the default. For details on memory, see Section 5, “Migrating Memory Configurations”.

1. Delete the following code from the file “evmDM6437_common.tci”:

```
/* Enable heaps in IRAM and define label SEG0 for heap usage. */
bios.IRAM.createHeap      = true;
bios.IRAM.enableHeapLabel = true;
bios.IRAM["heapLabel"]    = prog.extern("SEG0");
bios.IRAM.heapSize        = 0x2000;
bios.MEM.BIOSOBJSEG = prog.get("IRAM");
bios.MEM.MALLOCSEG = prog.get("IRAM");
```

2. Using the HeapMem reference created by the `useModule()` method in mailbox.cfg, create a HeapMem heap instance of size 0x2000 by adding the following code:

```
var params = new HeapMem.Params;
params.size = 0x2000;
params.sectionName = "myHeap";
var heap = HeapMem.create(params);
```

3. Using the Memory reference created by the `useModule()` method in mailbox.cfg, assign this heap to be the default Heap for the application. This also assigns the heap to be the default heap for `malloc()` and `free()` calls. Add the following code:

```
Memory.defaultHeapInstance = heap;
```

4. Using the Memory reference created by the `useModule()` method in mailbox.cfg, assign this heap to be the default Heap for DSP/BIOS objects. Add the following code:

```
Defaults.common$.instanceHeap = heap;
```

5. Using the Program module, which exists during RTSC configuration, map the section name of our Heap “myHeap”, which was created in Step 2, to load data into the IRAM segment:

```
var IRAM = Program.cpu.memoryMap.IRAM;
Program.sectMap[“myHeap”] = {loadSegment: IRAM.name};
```

8.1.2.2 Converting the Cache Configuration Code

1. Delete the following lines of code from the file “evmDM6437_common.tci”:

```
bios.GBL.C64PLUSCONFIGURE = 1;
bios.GBL.C64PLUSMAR128to159 = 0x0000ffff;
```

   In DSP/BIOS 5, configuring the cache required that this intention be explicitly set in the Tconf include file (the first line of this code). This is no longer necessary in DSP/BIOS 6, as the enabling of the cache is now handled in the platform file.

2. Add a line of code to set the MAR 128 – 159 register bit mask:

```
Cache.MAR128_159 = 0x0000ffff;
```

8.1.2.3 Converting the Clock Configuration Code

The legacy module CLK corresponds to the DSP/BIOS 6 Clock module. The CLK module allowed you to take a specific timer out of reset mode. This property is not supported by the DSP/BIOS 6 Clock module, so the legacy CLK configuration code in the mailbox example should be deleted.

1. Delete the following lines of code from the file “evmDM6437_common.tci”:

```
bios.CLK.TIMERSELECT = "Timer 0";    /* Select Timer 0 to drive BIOS CLK */
bios.CLK.RESETTIMER = true;          /* Take the selected timer our of reset */
```

2. Add configuration code to set Timer 0 as the timer instance created by the Clock module:

```
Clock.timerId = 0;                   /* Select Timer 0 to drive BIOS CLK */
```

3. Save and close the file.

8.1.3 Porting the mailbox.tci File

The “mailbox.tci” file contains configuration settings for DSP/BIOS Logs and creates DSP/BIOS Tasks for the mailbox program. While the code to configure logging in DSP/BIOS 6 is different from earlier versions, DSP/BIOS 6 Task configuration is very similar to legacy TSK configuration.

In the subsections that follow, you will also convert the logging configuration code to DSP/BIOS 6 format and remove unneeded legacy code.

8.1.3.1 Converting the Log Configuration Code

In DSP/BIOS 5 applications, you printed to a DSP/BIOS LOG by defining a LOG buffer and calling the `LOG_printf()` function to write to the buffer. In DSP/BIOS 6, you will create a default logger to write log output to. One method is to use the XDCtools LoggerBuf module to create a LoggerBuf instance and set the instance as the default logger for the system so that output will be written to it.

Additionally, XDCtools provides greater control over the logging in an application by providing the diagnostics module, “xdc.runtime.Diags,” to enable or disable the logging for a particular module. For more information on Diags, see Section 8.2.7.

The following steps enable USER2 diagnostics for the mailbox program’s logging. You will remove the legacy LOG configuration and replace it by setting up an XDCtools LoggerBuf instance as the default logger in the mailbox application.

1. Open mailbox.tci for editing.
2. Delete the following lines of code:

```
var trace;
trace         = bios.LOG.create("trace");
trace.bufLen  = 256;
trace.logType = "circular";
```

3. Add the following code:

```
/* create trace logger */
var trace = LoggerBuf.create();

/* Set trace to be the default logger for the application. */
Main.common$.logger = trace;

/*
 * Enable diagnostics for USER2 events for our program. Logging in the C
 * Code will be done using USER2 events. Disabling this would disable all of
 * the logging done in mailbox.c
 */
Main.common$.diags_USER2 = Diags.RUNTIME_ON;
```

4. Delete the lines of code that configure the LOG_system Log. This Log buffer is not used in DSP/BIOS 6.

```
/* Set the buffer length of LOG_system buffer */
bios.LOG_system.bufLen = 512;
```

8.1.3.2 Converting the Task Configuration Code

The DSP/BIOS 5 TSK module corresponds to the DSP/BIOS 6 Task module. Creating a static Task in DSP/BIOS 6 is similar to creating a static TSK in DSP/BIOS 5, and is fairly straightforward.

1. Delete the legacy TSK configuration code:

```
/*
 * Create and initialize four TSKs
 */
var reader0;
reader0          = bios.TSK.create("reader0");
reader0.priority = 1;
reader0["fxn"]   = prog.extern("reader");

var writer0;
writer0          = bios.TSK.create("writer0");
writer0.priority = 1;
writer0["fxn"]   = prog.extern("writer");
writer0.arg0     = 0;

var writer1;
writer1          = bios.TSK.create("writer1");
writer1.priority = 1;
writer1["fxn"]   = prog.extern("writer");
writer1.arg0     = 1;

var writer2;
writer2          = bios.TSK.create("writer2");
writer2.priority = 1;
writer2["fxn"]   = prog.extern("writer");
writer2.arg0     = 2;
```

2. Add the following Task configuration code:

```
/*
 * Create and initialize four Tasks
 */
var reader0Params = new Task.Params();
reader0Params.priority = 1;
var reader0 = Task.create('&reader', reader0Params);

var writer0Params = new Task.Params();
writer0Params.arg0 = 0;
writer0Params.priority = 1;
var writer0 = Task.create('&writer', writer0Params);

var writer1Params = new Task.Params();
writer1Params.arg0 = 1;
writer1Params.priority = 1;
var writer1 = Task.create('&writer', writer1Params);

var writer2Params = new Task.Params();
writer2Params.arg0 = 2;
writer2Params.priority = 1;
var writer2 = Task.create('&writer', writer2Params);
```

3. Save and close the file.

8.1.4 Porting the mailbox_evmDM6437_custom.tci File

The mailbox_evmDM6437_custom.tci file contains configuration settings for the legacy MBX module. The DSP/BIOS 5 MBX module corresponds to the DSP/BIOS 6 Mailbox module, and creating a static Mailbox in DSP/BIOS 6 is similar to creating a static MBX in DSP/BIOS 5.

In the next steps, you will convert this configuration code to DSP/BIOS 6 and remove unneeded legacy code.

1. Delete the legacy MBX configuration code:

```
/* Create a mbx */

var mbx;
mbx              = bios.MBX.create("mbx");
mbx.messageSize = 8;
mbx.length      = 2;
```

2. Add the following Mailbox configuration code:

```
var mbxParams = new Mailbox.Params(); // Use a default Mailbox Params
var messageSize = 8;
var length = 2;
Program.global.mbx = Mailbox.create(messageSize, length, mbxParams);
```

3. Save and close the file.

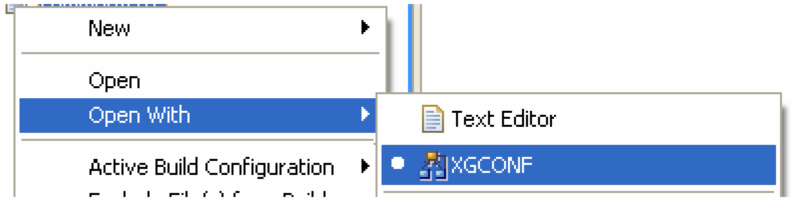
8.1.5 Viewing the DSP/BIOS 6 Mailbox Configuration Using XGCONF

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

In Section 7.7, you saw how to use XGCONF to view the legacy DSP/BIOS 6 configuration of the mailbox program. Now that the mailbox.cfg file has been updated to use non-legacy configuration code, you can again view it graphically with XGCONF, although you'll have to look in different places to see what our configuration contains. (Previously the mailbox configuration settings were found under “ti.bios”. Now they'll be found under “ti.sysbios”.)

In this section, you will learn where to look in XGCONF to find DSP/BIOS 6 module settings.

1. Make sure the project that contains the RTSC configuration file, “mailbox_configuration”, is set to be the "Active" project. You can do this by right-clicking on the mailbox_configuration project in the "C/C++ Projects" view and selecting **Set as Active Project**.
2. To open XGCONF, you must be in the **C/C++** perspective of CCStudio. If necessary, click the **C/C++** icon to switch back to that perspective.
3. To open XGCONF, right click the “mailbox.cfg” file in the project pane, and then select **Open With → XGCONF**:

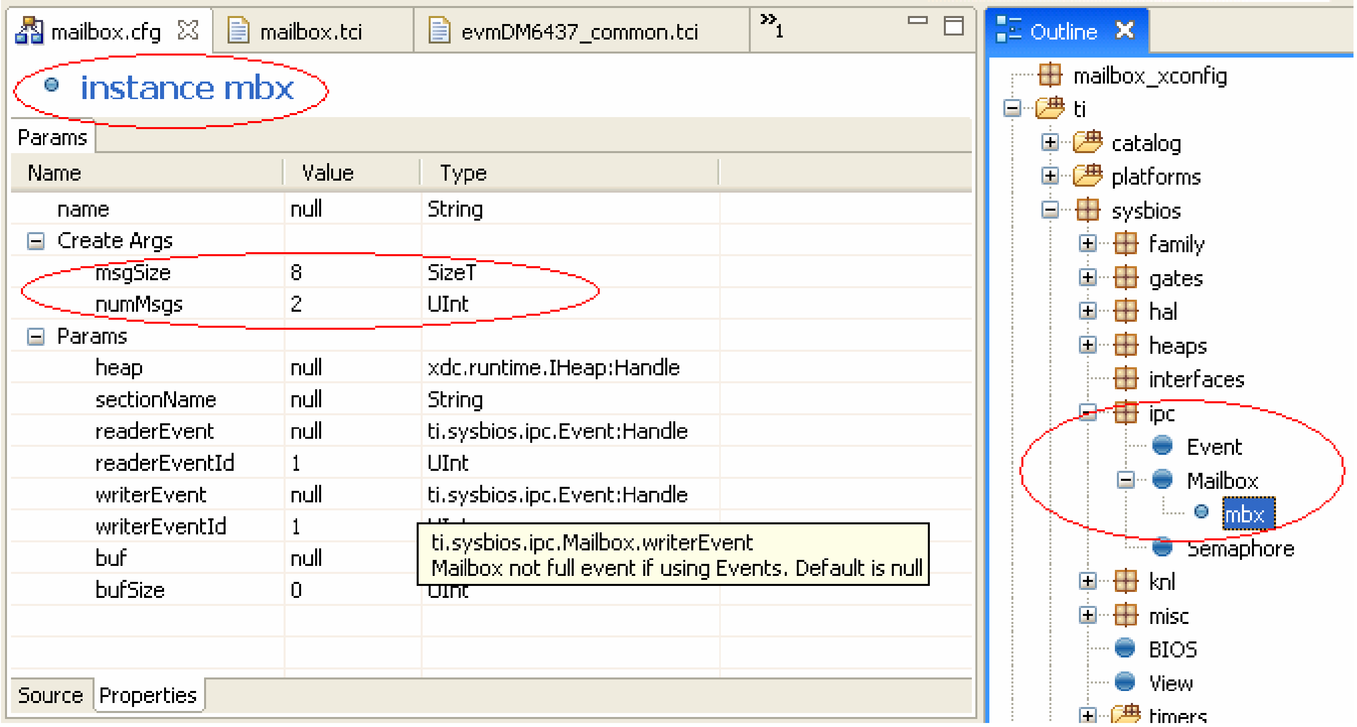


4. Recall the configuration code you added in Section 8.1.4 to create a mailbox instance in the file “mailbox_evmDM6437_custom.tci”. Recall that in that section you added the following code:

```
xdc.global.Mailbox   = xdc.useModule('ti.sysbios.ipc.Mailbox');
. . .
var mbxParams = new Mailbox.Params(); // Use a default Mailbox Params
var messageSize = 8;
var length = 2;
Program.global.mbx = Mailbox.create(messageSize, length, mbxParams);
```

5. You can also see the result of this configuration code in XGCONF, however, it will be in a different place—under the “ti.sysbios.ipc.Mailbox” module. Click the **+** sign next to the **ti.sysbios.ipc** package in the Outline pane to expand the view of this package.

6. In the **Outline** pane, toggle the full tree view of the modules on by clicking the icon. Expand the tree in the Outline page as shown in the following figure. Next, click the **+** sign next to the module **Mailbox**. Notice that there is an instance under **Mailbox** called **mbx**. This is the instance created by the RTSC configuration code. Click on **mbx** to select it. You will see the properties for this Mailbox instance in the **Properties** pane to the left, under **Create Args**. Notice that its properties match what was set in the configuration code:



instance mbx

Params

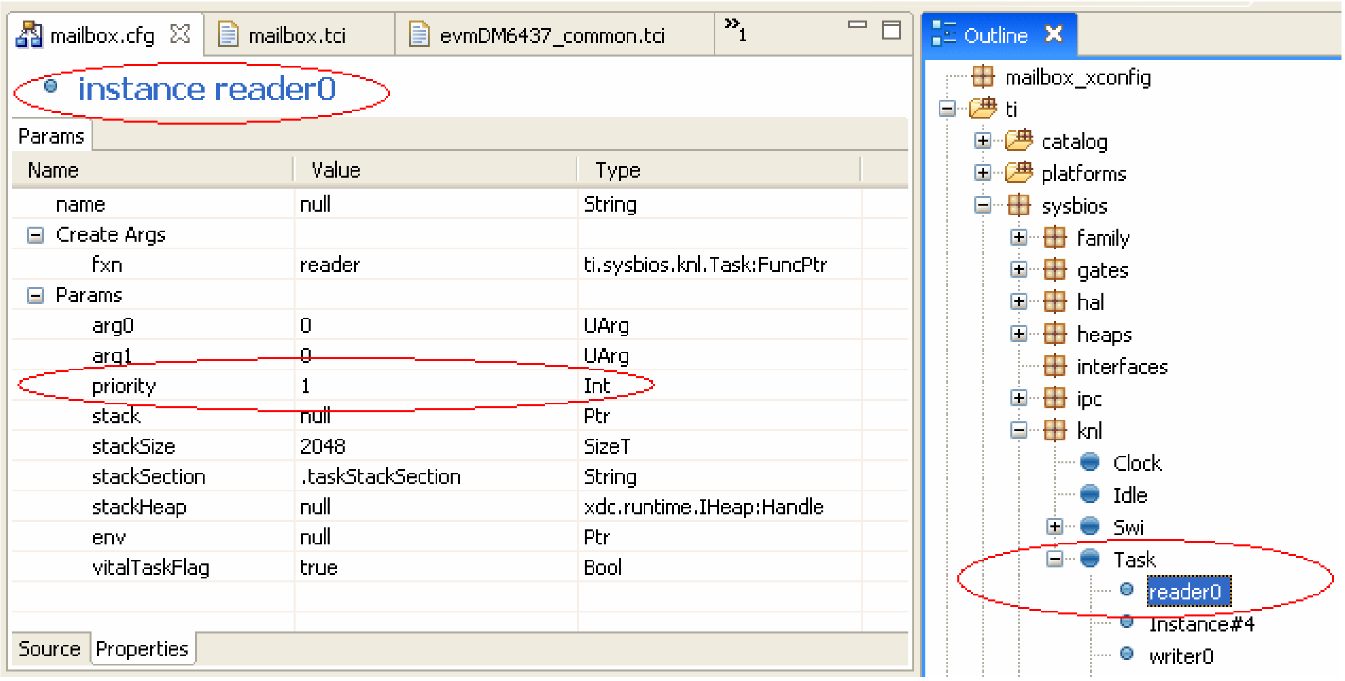
Name	Value	Type
name	null	String
⊟ Create Args		
msgSize	8	SizeT
numMsgs	2	UInt
⊟ Params		
heap	null	xdc.runtime.IHeap:Handle
sectionName	null	String
readerEvent	null	ti.sysbios.ipc.Event:Handle
readerEventId	1	UInt
writerEvent	null	ti.sysbios.ipc.Event:Handle
writerEventId	1	
buf	null	
bufSize	0	UInt

ti.sysbios.ipc.Mailbox.writerEvent
Mailbox not full event if using Events. Default is null

7. Next, let’s look at some of the Task instances. Recall the Task instances that were created by the following code from in Section 8.1.3.2.

```
var reader0Params = new Task.Params();
reader0Params.priority = 1;
var reader0 = Task.create('&reader', reader0Params);
```

8. In the Outline pane, expand the tree as shown in the following figure**.** Notice that there are several instances found under **Task**, all corresponding to the instances created by the mailbox.cfg script:



Name	Value	Type
name	null	String
⊟ Create Args		
fxn	reader	ti.sysbios.knl.Task:FuncPtr
⊟ Params		
arg0	0	UArg
arg1	0	UArg
priority	1	Int
stack	null	Ptr
stackSize	2048	SizeT
stackSection	.taskStackSection	String
stackHeap	null	xdc.runtime.IHeap:Handle
env	null	Ptr
vitalTaskFlag	true	Bool

You can use XGCONF in this way to view other properties of a DSP/BIOS program's configuration. All you need to remember is that the layout of the **Outline** tab reflects the layout of the DSP/BIOS 6 and XDCtools packages and modules. Just find the module of interest in the Outline tab, and you can see the end result of the RTSC configuration script for that module.

8.2 Updating the C Source File

The C source file "mailbox.c" contains legacy API calls and includes legacy header files. In order to port the source file to DSP/BIOS 6, these legacy headers and C API calls must be changed to their DSP/BIOS 6 equivalents.

8.2.1 Replacing the Legacy Header Files

1. Open the "mailbox.c" file for editing and find the `#include` statements near the top of the file. The mailbox.c file includes the following legacy header files:

```
#include <std.h>
#include <log.h>
#include <mbx.h>
#include <tsk.h>
```

2. The first step in converting these `#include` statements is to determine their DSP/BIOS 6 counterparts. For the most part, a DSP/BIOS 6 header file has a similar name to its legacy counterpart. For example, the legacy “tsk.h” corresponds to “Task.h” in DSP/BIOS 6. The DSP/BIOS 6 header files are found in the ”ti/sysbios” directory and sub-directories.

   DSP/BIOS 6 requires that `#include` statements use the package path of the header files, so it is important to note this path when locating the header files. The DSP/BIOS 5 header files included in the mailbox.c source file map to DSP/BIOS 6 header files as follows:

```
std.h   ----> xdc/std.h
log.h   ----> xdc/runtime/System.h, xdc/runtime/Log.h
mbx.h   ----> ti/sysbios/ipc/Mailbox.h
tsk.h   ----> ti/sysbios/knl/Task.h
```

3. In addition, the BIOS header file must be included in the application, so you will need:

```
ti/sysbios/BIOS.h
```

4. Delete the existing `#include` statements from the “mailbox.c” file, and replace them with the new list of DSP/BIOS 6 header files:

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Log.h>
#include <xdc/runtime/Diags.h>
#include <ti/sysbios/ipc/Mailbox.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/BIOS.h>
#include <xdc/cfg/global.h>
```

For more information on mapping legacy modules to DSP/BIOS 6 and XDCtools modules, see Section 4.1, “Legacy Module Mappings and API Guide”.

8.2.2 Updating the main() Function to Call BIOS_start()

All DSP/BIOS 6 programs must call the `BIOS_start()` function. The `BIOS_start()` function should be called within `main()` after all other user initialization code. Add the following code to the `main()` function of mailbox.c:

```
BIOS_start();
```

8.2.3 Replacing the Legacy MBX_post() and MBX_pend() Calls

The DSP/BIOS 5 `MBX_pend()` and `MBX_post()` APIs correspond to the DSP/BIOS 6 `Mailbox_pend()` and `Mailbox_post()` APIs. Make the following changes to the mailbox.c file to use the Mailbox module APIs:

1. In the “mailbox.c” file, go to the `reader()` function and replace the following legacy code:

```
if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
```

   with the DSP/BIOS 6 equivalent:

```
if (Mailbox_pend(mbx, &msg, TIMEOUT)== 0) {
```

2. In the “mailbox.c” file, find the `writer()` function and replace the following line of legacy code:

```
MBX_post(&mbx, &msg, TIMEOUT);
```

with the DSP/BIOS 6 equivalent:

```
Mailbox_post(mbx, &msg, TIMEOUT);
```

8.2.4 Replacing the Legacy ArgToInt() Macro Call

The `ArgToInt()` macro is defined in the legacy std.h file. To remove this legacy dependency, replace it with a simple cast.

In the “mailbox.c” file, go to the `writer()` function and replace the following line of legacy code:

```
Int id =    ArgToInt (id_arg);
```

with:

```
Int id =    (Int)id_arg;
```

8.2.5 Replacing the Legacy LOG_printf() API Calls with Log_print() Calls*

Section 8.1.3.1 showed you how to create a trace logger for the application in the RTSC configuration file with the XDCtools LoggerBuf module. Now you will modify the C code to use the new logging API, `Log_print*()`, where “*” is a place holder for the number of values to print. For example, if a print statement does not print any arguments use `Log_print0()`; if a print statement prints one value use `Log_print1()`, and so on.

The `Log_print*()` statement does not specify a log handle, but specifies a Log event diagnostic argument instead. The Log event diagnostic should match what was specified in the configuration script; in this case it was `diags_USER2`. For more information on diagnostics masks, see Section 8.2.7.

1. In the “mailbox.c” file, go to the `reader()` function and find the `LOG_printf()` statements. Notice the number of variables that each print statement prints; there are `LOG_printf()` calls that print zero arguments, and one that prints two arguments. These calls must be replaced with the appropriate `Log_print*()` call, depending on the number of arguments being printed.
2. The first `LOG_printf()` call in the `reader()` function does not print any arguments. Therefore, it should be substituted with the `Log_print0()` API. Replace the following line of code in the `reader()` function:

```
LOG_printf(&trace, "timeout expired for MBX_pend()");
```

with the new API call:

```
Log_print0(Diags_USER2, "timeout expired for MBX_pend()");
```

Note that Diags_USER2 is specified as the first argument to the new function, not a LOG buffer handle as in the legacy API. This sets the diagnostics mask for our logging to USER2, so that all Log_print*() statements specifying Diags_USER2 will have their output displayed when the application is run.

3. The second LOG_printf() call in the reader() function prints two values, so replace it with the Log_print2() API. Replace the following line of code in the reader() function:

```
LOG_printf(&trace, "read '%c' from (%d).", msg.val, msg.id);
```

with the new API call:

```
Log_print2(Diags_USER2, "read '%c' from (%d).", msg.val, msg.id);
```

4. Replace the following line of code in the reader() function:

```
LOG_printf(&trace, "reader done.");
```

with the new API call:

```
Log_print0(Diags_USER2, "reader done.");
```

5. Go to the writer() function and replace the following line of code:

```
LOG_printf(&trace, "(%d) writing '%c' ...", id, (Int)msg.val);
```

with the new API call:

```
Log_print2(Diags_USER2, "(%d) writing '%c' ...", id, (Int)msg.val);
```

Also replace:

```
LOG_printf(&trace, "writer (%d) done.", id);
```

with:

```
Log_print1(Diags_USER2, "writer (%d) done.", id);
```

6. Save and close the file.

8.2.6 Optional: Replacing the TSK_yield() Legacy API Call

The mailbox example contains a call to TSK_yield() that is commented out. If you wish to uncomment this call, it must first be converted to a DSP/BIOS 6 call.

1. Open the “mailbox.c” file for editing.
2. Go to the writer() function and replace the following legacy API call:

```
TSK_yield()
```

with:

```
Task_yield()
```

3. Save and close the file.

8.2.7 More About Diagnostics Masks (Diags)

As shown in Section 8.2.5, the `Log_print*()` APIs use a diagnostics mask as the first argument. As mentioned previously, this mask controls the logging output displayed for the program at run time for greater control over the output of the program.

There are different types of diagnostics for each module—any of which may be enabled or disabled—that categorize the logs in the application. This makes it easy to turn off all logging and print statements for an application, or disable a subset of the print output in the application.

To see the logging results, the diagnostics set for each of the `Log_print*()` statements must match what was specified in the application's RTSC configuration. For the mailbox application, this was `diags_USER2`, so that should be the first argument to all `Log_print*()` calls.

For example, all of the logging output generated by the `Log_print*()` statements may be disabled by adding the following line of configuration code and rebuilding the application:

```
Main.common$.diags_USER2 = Diags.RUNTIME_OFF;
```

To turn the logging output back on, change this line of code to:

```
Main.common$.diags_USER2 = Diags.RUNTIME_ON;
```

Another way to change the output is by changing the diagnostics mask in a `Log_print*()` statement. For example, you could change the `Log_print*()` calls in the `writer()` function to specify the `Diags_USER1` mask, instead of `Diags_USER2`.

Specifically, the print statements in the `writer()` function would change to:

```
Log_print2(Diags_USER1, "(%d) writing '%c' ...", id, (Int)msg.val);
```

and:

```
Log_print1(Diags_USER1, "writer (%d) done.", id);
```

If you have made the changes to the mailbox.c file (don't forget to rebuild), only the program output for the `reader()` function print statements would be displayed. Since the diagnostics for USER1 were never enabled in the configuration, changing the diagnostics mask of the `writer()` function's print statements to USER1 disabled them. Adding the following line of configuration code to mailbox.tci would enable the print output for USER1, and allow the output of the `writer()` function to be displayed:

```
Main.common$.diags_USER1 = Diags.RUNTIME_ON;
```

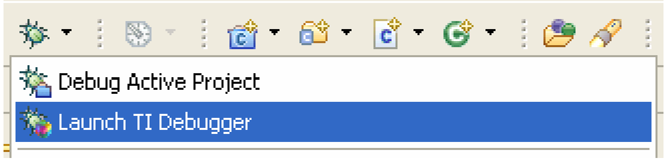
For more information on Diags, please see Section 8.2.5 for diagnostic masks used as arguments in `Log_print*()` calls, or see the online CDOC reference documentation that came with your XDCtools installation.

8.3 Building and Running the Application

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

1. Make sure that the “mailbox” project is the active project. Then, from within the **C/C++** perspective, select **Project → Build Active Project** to build the mailbox application using DSP/BIOS 6.

2. Enter the **Debug** perspective in order to load the mailbox.out executable. If you have not used it before, you can open this perspective by choosing **Launch TI Debugger** from the drop-down arrow next to the debug icon:

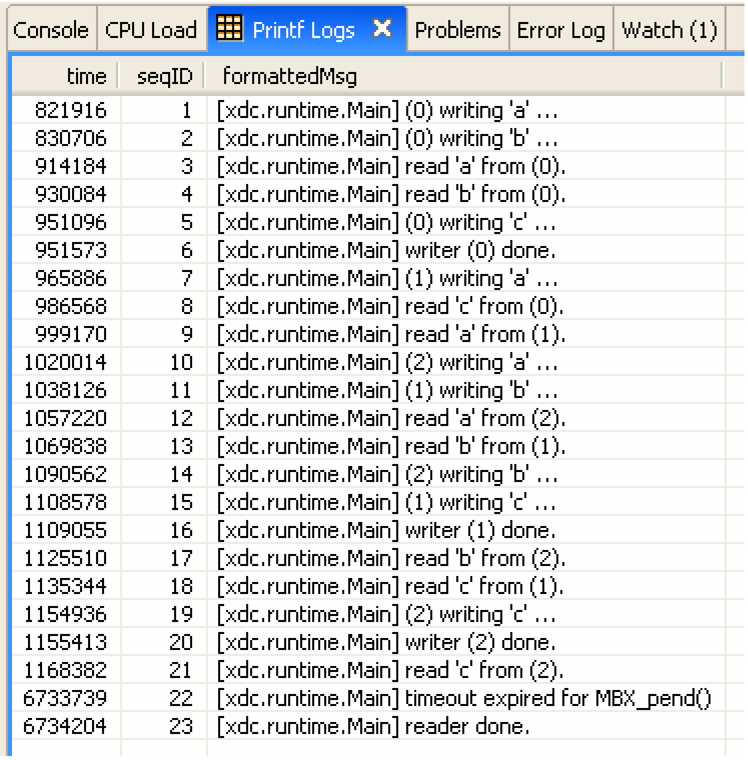


3. Connect CCStudio 4 to the DM6437 EVM board by selecting **Target → Connect Target**.

4. Reset the board by selecting **Run → Reset → CPU Reset**.

5. Load the program by selecting **Target → Load Program** and choosing the application executable “<*working directory path*>/mailbox/Debug/mailbox.out”.

6. Run the application by selecting **Run → Run** or pressing the **F8** key. Output is displayed in the RTA Printf Logs pane, as shown in the following picture.



time	seqID	formattedMsg
821916	1	[xdc.runtime.Main] (0) writing 'a' ...
830706	2	[xdc.runtime.Main] (0) writing 'b' ...
914184	3	[xdc.runtime.Main] read 'a' from (0).
930084	4	[xdc.runtime.Main] read 'b' from (0).
951096	5	[xdc.runtime.Main] (0) writing 'c' ...
951573	6	[xdc.runtime.Main] writer (0) done.
965886	7	[xdc.runtime.Main] (1) writing 'a' ...
986568	8	[xdc.runtime.Main] read 'c' from (0).
999170	9	[xdc.runtime.Main] read 'a' from (1).
1020014	10	[xdc.runtime.Main] (2) writing 'a' ...
1038126	11	[xdc.runtime.Main] (1) writing 'b' ...
1057220	12	[xdc.runtime.Main] read 'a' from (2).
1069838	13	[xdc.runtime.Main] read 'b' from (1).
1090562	14	[xdc.runtime.Main] (2) writing 'b' ...
1108578	15	[xdc.runtime.Main] (1) writing 'c' ...
1109055	16	[xdc.runtime.Main] writer (1) done.
1125510	17	[xdc.runtime.Main] read 'b' from (2).
1135344	18	[xdc.runtime.Main] read 'c' from (1).
1154936	19	[xdc.runtime.Main] (2) writing 'c' ...
1155413	20	[xdc.runtime.Main] writer (2) done.
1168382	21	[xdc.runtime.Main] read 'c' from (2).
6733739	22	[xdc.runtime.Main] timeout expired for MBX_pend()
6734204	23	[xdc.runtime.Main] reader done.

7. Halt the program by selecting **Run → Halt**.

9 Adding DSP/BIOS Tasks and Communication to a Legacy Application

Once you have updated an DSP/BIOS 5 legacy application to build and run using DSP/BIOS 6 legacy support, you may want to expand upon that program, making use of some of the new DSP/BIOS 6 modules and code. This section shows how to add two new DSP/BIOS 6 Tasks to a hypothetical legacy program that already contains two legacy TSKs.

Suppose a legacy DSP/BIOS 5 program contains two TSKs that run the following “reader” and “writer” functions, which communicate via MBX. Section 9.1 shows how to add two new DSP/BIOS 6 Tasks, which communicate with each other via Mailbox. Section 9.2 shows how the legacy TSKs can communicate with the DSP/BIOS 6 Tasks, and vice versa.

Since this section expands on Section 4.2, “Extending Your Legacy Program with DSP/BIOS 6 APIs”, you should read that section first.

```
/* ======== reader ======== */
Void reader(Void)
{
   MsgObj      msg;
   Int         i;

   for (i=0; ;i++) {
      /* wait for mailbox to be posted by writer() */
      if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
         Log_print0(Diags_USER2, "BIOS5 reader: timeout expired for MBX_pend()");
         break;
      }
      Log_print2(Diags_USER2, "BIOS 5 reader: read '%c' from BIOS 5 writer (%d).",
             msg.val, msg.id);
   }
   Log_print0(Diags_USER2, "BIOS 5 reader done.");
}

/* ======== writer ======== */
Void writer(Arg id_arg)
{
   MsgObj      msg;
   Int         i;
   Int id =    ArgToInt (id_arg);

   for (i=0; i < NUMMSGS; i++) {
      /* fill in value */
      msg.id = id;

      /* Send message to BIOS 5 MBX. Use legacy API to communicate with legacy code */
      msg.val = i % NUMMSGS + (Int)('a');
      MBX_post(&mbx, &msg, TIMEOUT);
      Log_print2(Diags_USER2, "BIOS 5 writer: (%d) using MBX_post to write '%c' to the
BIOS 5 MBX ...", id, (Int)msg.val);
   }

   Log_print1(Diags_USER2, "BIOS 5 writer: (%d) done.", id);
}
```

Notice that although this code is “old” legacy code, it has been updated to use the “new” API `Log_print*()`. This causes the output of the program to be written into the same log buffer and displayed in order. For more information on this, see Section **Error! Reference source not found.**.

9.1 Adding Two New DSP/BIOS 6 Tasks to a Legacy Program

This section shows how you can expand the program to add two new Tasks.

The two new DSP/BIOS 6 Tasks being added are considered “new code.” As such, there are two new functions written for them, both of which use DSP/BIOS 6 APIs. Since they will communicate with one another, they will use a DSP/BIOS 6 Mailbox object and APIs. All of this conforms to the rules of communication described in Section 4.2.

1. In order to add two new DSP/BIOS 6 Tasks and a Mailbox object to an existing application, you would first add RTSC configuration code similar to the following to the *.cfg file. The following code statically configures two new Tasks and a Mailbox, and must be added *after* all existing legacy configuration code in order to work correctly (see Section 4.2.1 for details).

```
/* create two new BIOS 6 Tasks */
xdc.global.Task = xdc.useModule("ti.sysbios.knl.Task");
var tskParams = new Task.Params();
tskParams.priority = 1;
Task.create('&bios6Reader', tskParams);
Task.create('&bios6Writer', tskParams);

/* create a BIOS 6 Mailbox for communication between BIOS 6 Tasks */
xdc.global.Mailbox = xdc.useModule("ti.sysbios.ipc.Mailbox");
var mbxParams = new Mailbox.Params();
var messageSize = 8;
var length = 2;
Program.global.bios6Mailbox = Mailbox.create(messageSize, length, mbxParams);
```

2. Since this example mixes legacy and new code, you must include the DSP/BIOS 6 header files before the existing DSP/BIOS 5 legacy headers in the existing legacy C file (note that the order of XDCtools header files is not important). You would add code similar to the following lines shown in bold to include necessary header files for the new Task and Mailbox code. All of the #include directives needed are shown, so that the required ordering is clear:

```
/* header files for new code must come before legacy header files */
#include <ti/sysbios/Bios.h>
#include <ti/sysbios/ipc/Mailbox.h>
#include <ti/sysbios/knl/Task.h>

#include <xdc/runtime/LoggerBuf.h>
#include <xdc/runtime/Log.h>
#include <xdc/runtime/Diags.h>
#include <xdc/runtime/System.h>
#include <xdc/cfg/global.h>

/* existing legacy header files */
#include <std.h>
#include <mbx.h>
#include <tsk.h>
```

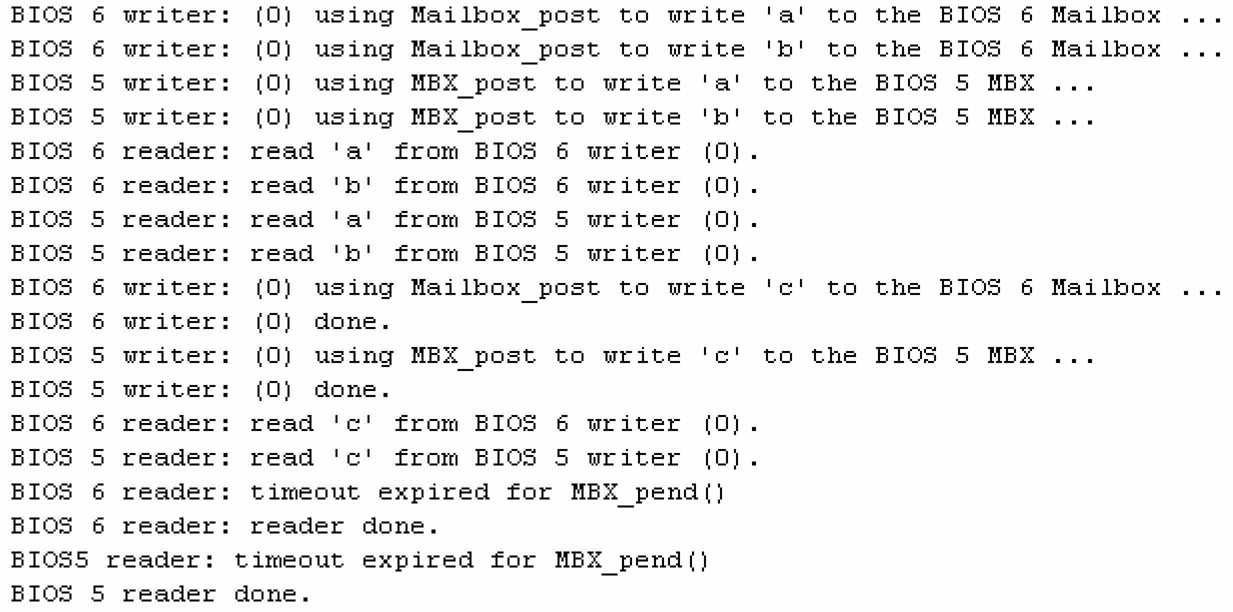
3. The following new Task functions could be used for the new DSP/BIOS 6 Tasks:

```
/*
 *  ======== bios6Reader ========
 */
Void bios6Reader(Arg id_arg)
{
    MsgObj      msg;
    Int         i;

   for (i=0; ;i++) {
      /* wait for mailbox to be posted by writer() */
      if (Mailbox_pend(bios6Mailbox, &msg, TIMEOUT) == 0) {
         Log_print0(Diags_USER2, "BIOS 6 reader: timeout expired for MBX_pend()");
         break;
      }

      Log_print2(Diags_USER2, "BIOS 6 reader: read '%c' from BIOS 6 writer (%d).",
            msg.val, msg.id);
   }

   Log_print0(Diags_USER2, "BIOS 6 reader: reader done.");
}

/*
 *  ======== bios6Writer ========
 */
Void bios6Writer(Arg id_arg)
{
   MsgObj      msg;
   Int         i;
   Int id =    (Int)id_arg;

   for (i=0; i < NUMMSGS; i++) {
      /* fill in value */
      msg.id = id;
      msg.val = i % NUMMSGS + (Int)('a');

      /* Send msg to BIOS 6 Mailbox. Use new BIOS 6 API to communicate
       * with non-legacy BIOS 6 code. */
      msg.val = i % NUMMSGS + (Int)('a');
      Mailbox_post(bios6Mailbox, &msg, TIMEOUT);
      Log_print2(Diags_USER2, "BIOS 6 writer: (%d) using Mailbox_post to write '%c'
to the BIOS 6 Mailbox ...", id, (Int)msg.val);
   }

   Log_print1(Diags_USER2, "BIOS 6 writer: (%d) done.", id);
}
```

4. To properly run a DSP/BIOS 6 program, you would also need to add the following code to the `main()` function:

```
BIOS_start();
```

Building and running should yield output similar to the following for this hypothetical example:



```
BIOS 6 writer: (0) using Mailbox_post to write 'a' to the BIOS 6 Mailbox ...
BIOS 6 writer: (0) using Mailbox_post to write 'b' to the BIOS 6 Mailbox ...
BIOS 5 writer: (0) using MBX_post to write 'a' to the BIOS 5 MBX ...
BIOS 5 writer: (0) using MBX_post to write 'b' to the BIOS 5 MBX ...
BIOS 6 reader: read 'a' from BIOS 6 writer (0).
BIOS 6 reader: read 'b' from BIOS 6 writer (0).
BIOS 5 reader: read 'a' from BIOS 5 writer (0).
BIOS 5 reader: read 'b' from BIOS 5 writer (0).
BIOS 6 writer: (0) using Mailbox_post to write 'c' to the BIOS 6 Mailbox ...
BIOS 6 writer: (0) done.
BIOS 5 writer: (0) using MBX_post to write 'c' to the BIOS 5 MBX ...
BIOS 5 writer: (0) done.
BIOS 6 reader: read 'c' from BIOS 6 writer (0).
BIOS 5 reader: read 'c' from BIOS 5 writer (0).
BIOS 6 reader: timeout expired for MBX_pend()
BIOS 6 reader: reader done.
BIOS5 reader: timeout expired for MBX_pend()
BIOS 5 reader done.
```

9.2 Communicating Between the Existing Legacy TSKs and the New Tasks

The example in the subsections that follow shows how to update our hypothetical example to communicate between legacy TSK functions and new Task functions.

This section shows how to update the code in Section 9.1 to allow the legacy TSKs to communicate with the new Tasks, and vice versa.

9.2.1 Update the bios6Writer() Task to Post a Message to the Legacy reader() TSK

First, let's see how you could change the new `bios6Writer()` Task to communicate with the legacy `reader()` TSK. This is an example of new code communicating with legacy code.

1. Look at the legacy TSK `reader()` function in Section 9. Notice that it is waiting for a message via:

```
MBX_pend(mbx, ...);
```

To update the `bios6Writer()` Task to send a message to the legacy `reader()` TSK, we must use code that matches that of the legacy code in the `reader()` function. This means the following legacy call must be made from within the `bios6Writer()` Task, which uses the *legacy* MBX object:

```
MBX_post(mbx, ...);
```

2. To update the `bios6Writer()` function to send the legacy `reader()` TSK a message, you could add the code shown in bold. (The entire updated function is shown for clarity; new additions are shown in **bold**.)

```
/* ======== bios6Writer ======== */
Void bios6Writer(Arg id_arg)
{
   MsgObj      msg;
   Int         i;
   Int id =    (Int)id_arg;

   for (i=0; i < NUMMSGS; i++) {
      /* fill in value */
      msg.id = id;
      msg.val = i % NUMMSGS + (Int)('a');

      if (i == NUMMSGS - 1) {
         /* Send message to BIOS 5 MBX. Use legacy API to talk with old code. */
         msg.val = '6';
         MBX_post(&mbx, &msg, TIMEOUT);
         Log_print2(Diags_USER2, "BIOS 6 writer: (%d) using MBX_post to write '%c'
to the BIOS 5 MBX ...", id, (Int)msg.val);
      }
      else {
         /*
          * Send msg to BIOS 6 Mailbox. Use new BIOS 6 API to communicate
          * with non-legacy BIOS 6 code:
          */
         msg.val = i % NUMMSGS + (Int)('a');
         Mailbox_post(bios6Mailbox, &msg, TIMEOUT);
         Log_print2(Diags_USER2, "BIOS 6 writer: (%d) using Mailbox_post to write
'%c' to the BIOS 6 Mailbox ...", id, (Int)msg.val);
      }
   }

   Log_print1(Diags_USER2, "BIOS 6 writer: (%d) done.", id); }
}
```

3. Look at the legacy `reader()` function in Section 9. To handle receiving of the message from the `bios6Writer()` Task you would add the new code is shown in **bold**.

```
/*
 *  ======== reader ========
 */
Void reader(Void)
{
    MsgObj      msg;
    Int         i;

    for (i=0; ;i++) {

        /* wait for mailbox to be posted by writer() */
        if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
           Log_print0(Diags_USER2, "BIOS5 reader: timeout expired for MBX_pend()");
           break;
        }

        /* print value */
        if (msg.val == '6') {
           Log_print2(Diags_USER2, "BIOS 5 reader: read '%c' from BIOS 6 writer
(%d).", msg.val, msg.id);
        }
        else {
           Log_print2(Diags_USER2, "BIOS 5 reader: read '%c' from BIOS 5 writer
(%d).", msg.val, msg.id);
        }
    }

    Log_print0(Diags_USER2, "BIOS 5 reader done.");

}
```

9.2.2 Updating the Legacy writer() TSK to Post a Message to the bios6Reader() Task

Now, let's see how we would change the legacy `writer()` TSK to communicate with the new `bios6Reader()` Task. This is an example of old code communicating with new DSP/BIOS 6 code.

1. Look at the `bios6Reader()` function in Section 9.1. Notice that it waits for a message via:

```
Mailbox_pend(bios6Mailbox, ...);
```

To update the legacy `writer()` TSK to send a message to new the `bios6Reader()` Task, we again must use code that matches what's used by the entity that we want to communicate with. So, the `writer()` TSK must use the new Mailbox module APIs to communicate with `bios6Reader()`, using the *new* Mailbox object:

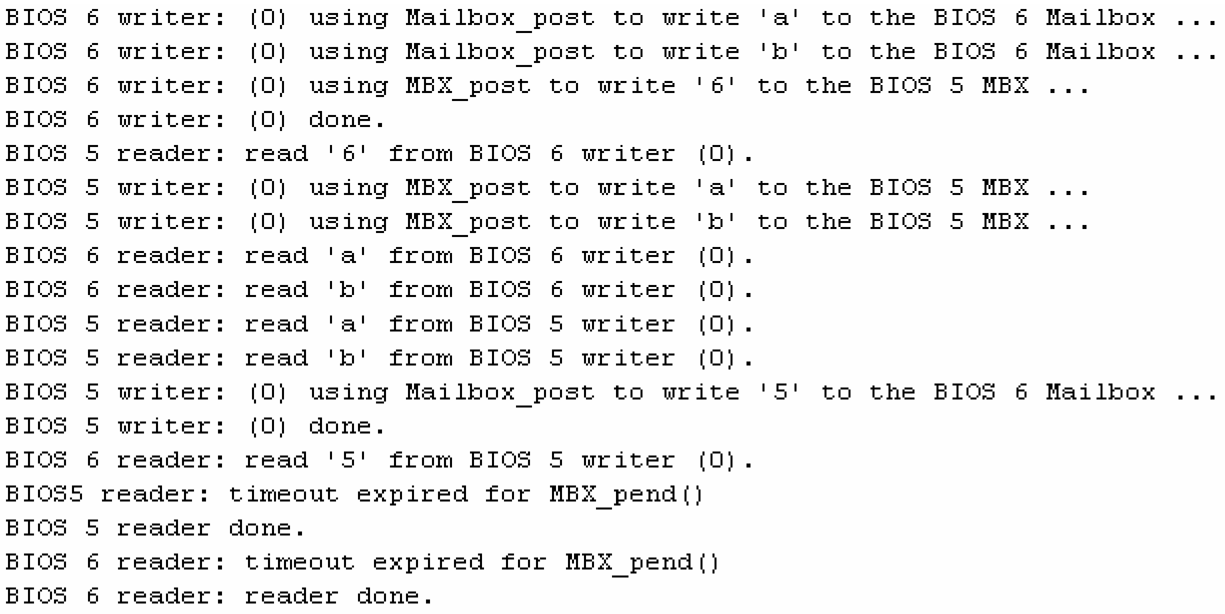
```
Mailbox_post(bios6Mailbox, ...);
```

2. To update the `writer()` function to send the `bios6Reader()` Task its last message, the following code shown in **bold** could be added:

```
/* ======== writer ======== */
Void writer(Arg id_arg)
{
    MsgObj      msg;
    Int         i;
    Int id =    ArgToInt (id_arg);

    for (i=0; i < NUMMSGS; i++) {
        /* fill in value */
        msg.id = id;
        if (i == NUMMSGS - 1) {
            /* Send msg to BIOS 6 Mailbox. Use new BIOS 6 API to communicate
             * from legacy code to new BIOS 6 code. */
            msg.val = '5';
            Mailbox_post(bios6Mailbox, &msg, TIMEOUT);
            Log_print2(Diags_USER2, "BIOS 5 writer: (%d) using Mailbox_post to
write '%c' to the BIOS 6 Mailbox ...", id, (Int)msg.val);
        }
        else {
            /* Send message to BIOS 5 MBX. Use legacy API to communicate with
             * legacy code. */
            msg.val = i % NUMMSGS + (Int)('a');
            MBX_post(&mbx, &msg, TIMEOUT);
            Log_print2(Diags_USER2, "BIOS 5 writer: (%d) using MBX_post to write
'%c' to the BIOS 5 MBX ...", id, (Int)msg.val);
        }
    }
    Log_print1(Diags_USER2, "BIOS 5 writer: (%d) done.", id);
}
```

9.2.3 Build and Run

You should see output similar to the following for this example:



```
BIOS 6 writer: (0) using Mailbox_post to write 'a' to the BIOS 6 Mailbox ...
BIOS 6 writer: (0) using Mailbox_post to write 'b' to the BIOS 6 Mailbox ...
BIOS 6 writer: (0) using MBX_post to write '6' to the BIOS 5 MBX ...
BIOS 6 writer: (0) done.
BIOS 5 reader: read '6' from BIOS 6 writer (0).
BIOS 5 writer: (0) using MBX_post to write 'a' to the BIOS 5 MBX ...
BIOS 5 writer: (0) using MBX_post to write 'b' to the BIOS 5 MBX ...
BIOS 6 reader: read 'a' from BIOS 6 writer (0).
BIOS 6 reader: read 'b' from BIOS 6 writer (0).
BIOS 5 reader: read 'a' from BIOS 5 writer (0).
BIOS 5 reader: read 'b' from BIOS 5 writer (0).
BIOS 5 writer: (0) using Mailbox_post to write '5' to the BIOS 6 Mailbox ...
BIOS 5 writer: (0) done.
BIOS 6 reader: read '5' from BIOS 5 writer (0).
BIOS5 reader: timeout expired for MBX_pend()
BIOS 5 reader done.
BIOS 6 reader: timeout expired for MBX_pend()
BIOS 6 reader: reader done.
```

10 Building the Mailbox Example Using a Custom Platform File

As discussed previously, a custom platform file must be used to create or redefine Memory segments for a platform. In DSP/BIOS 6, Memory segments are no longer created and updated in the configuration file.

The subsections that follow provide an overview of a custom platform file for the evmDM6437, show how to use this file to add new memory sections to the program, and detail the steps to build the mailbox example using this custom platform. You will generate the custom “evmDM6437.xs” platform file, build the mailbox example using the custom platform file, and update the file to redefine Memory sections.

10.1 The Generated evmDM6437.xs File

In Section 7.2.2, “Converting the Mailbox Configuration Using the Conversion Tool”, you may have used the conversion tool to convert the “mailbox.tcf” file to the “mailbox.cfg” RTSC configuration file. One of the byproducts of running the conversion tool is a generated custom platform file. If you did not perform the steps in Section 7.2.2 (it was an optional section), then please do so now in order to generate a custom platform file for your hardware.

When the conversion tool is run on a Tconf script, it parses the Tconf script’s configuration and finds all of the Memory segment settings. It creates the “evmDM6437.xs” custom platform file and puts the memory information into this file in a RTSC platform format. The custom platform can be used to build the application with all the memory segment settings of the original Tconf configuration.

Open the “evmDM6437.xs” file and look at the code. Notice that it defines an array called `memory[]`, and then defines each element of that array with Memory segment information.

```
var memory = [];
memory[0] = ["DDR2",
{
    name: "DDR2",
    base: 0x80000000,
    len: 0x8000000,
    space: "code/data"
}];
...
```

Notice that the Memory segments defined in this file are the same as in the original DSP/BIOS 5 mailbox example’s Tconf configuration for the evmDM6437 platform. The conversion tool simply adds the same Memory segments that existed in the DSP/BIOS 5 application.

Scroll down to the end of the file to see the platform definition:

```
Build.platformTable['ti.platforms.generic:plat'] = {
   clockRate: 594,
   catalogName: 'ti.catalog.c6000',
   deviceName: 'TMS320CDM6437',
   customMemoryMap: memory
};
```

This code adds the platform “ti.platforms.generic:plat” to the build system, setting its clock rate, catalog name, device name, and memory map. Notice that the `memory[]` array defined here is passed as the “customMemoryMap” argument.

For convenience, the contents of the file “evmDM6437.xs” have been provided in Section 10.5.

10.2 Building the Mailbox Example Using the Custom evmDM6437.xs Platform File

10.2.1 Observing the Existing Memory Segments of the Mailbox Program

1. Open the generated linker command file for the mailbox example “mailbox_p64p_x.xdl”. This file is generated in the Debug\configPkg\package\cfgl subdirectory of the working directory.
2. Scroll down in the file and observe the “Memory” section, which shows the Memory segments defined for the program. It should look like the following code. After the project is rebuilt using the custom platform file, the Memory segments will be redefined.

```
MEMORY
{
    IRAM (RWX) : org = 0x10800000, len = 0x20000
    L1DSRAM (RW) : org = 0x10f04000, len = 0xc000
    DDR2 : org = 0x80000000, len = 0x8000000
    SRAM : org = 0x42000000, len = 0x200000
}
```

3. Close the mailbox_p64p_x.xdl file.

10.2.2 Setting up config.bld to Load the Custom Platform File

1. In the working directory, create a new file called “config.bld” and open it for editing using your favorite text editor.
2. Add the following line of code to the config.bld file. This tells the build system to look in the custom platform file “evmDM6437.xs” when trying to find the custom platform “ti.platforms.generic:plat”:

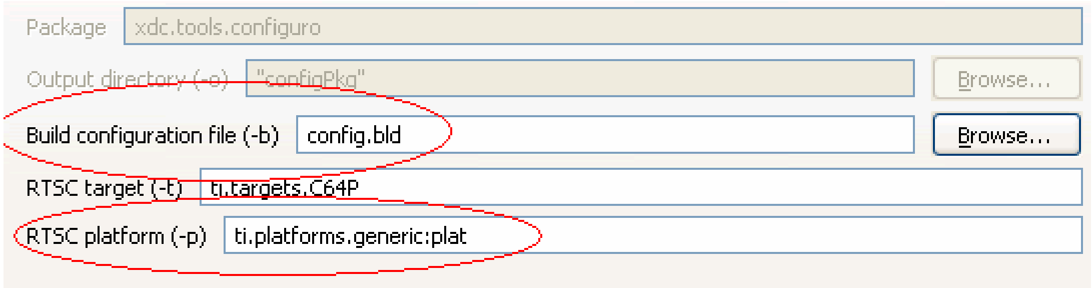
```
xdc.loadCapsule("evmDM6437.xs");
```

3. Save and close the new config.bld file.

10.2.3 Updating the Mailbox Project to Load the Custom Platform and to Use Config.bld

Note: This section applies only to users who are migrating to Code Composer Studio **4**.

1. In CCStudio 4, if it is not open already, open the mailbox project.
2. Make sure that you are in the **C/C++ Perspective** by clicking the button with that label.
3. In the project pane, right click on your project, and select **Build Properties**.
4. Under **Configuration Settings**, click the **Tool Settings** tab, and then click the **+** sign next to **XDCtools v3.15** to expand the list of XDCtools options. Click **Basic Options**.
5. For **Build configuration file (-b)**, specify “config.bld”. For **RTSC platform (–p)**, specify “ti.platforms.generic:plat”:



Package: xdc.tools.configuro

Output directory (-o): "configPkg"

Build configuration file (-b): config.bld

RTSC target (-t): ti.targets.C64P

RTSC platform (-p): ti.platforms.generic:plat

6. Click **OK** to save the project settings.
7. Select **Project-->Build Active Project** to rebuild the project using the custom platform file.

10.2.4 Observing the New Memory Segments Created by the Custom Platform File

1. Open the generated linker command file for the mailbox example “mailbox_p64p_x.xdl”
2. Again, scroll down into the file and observe the “Memory” section, which shows the Memory segments defined for the program. It should now contain the following code. Notice that there are now two additional Memory segments, “CACHE_L1P” and “CACHE_L1D”. These segments were defined in the custom platform file “evmDM6437.xs”, and did not exist in the default platform for the evmDM6437 that came with your installation:

```
MEMORY
{
    DDR2 : org = 0x80000000, len = 0x8000000
    IRAM : org = 0x10800000, len = 0x20000
    CACHE_L1D : org = 0x10f10000, len = 0x8000
    SRAM : org = 0x42000000, len = 0x200000
    CACHE_L1P : org = 0x10e08000, len = 0x8000
    L1DSRAM : org = 0x10f04000, len = 0xc000
}
```

10.3 Updating the evmDM6437.xs File to Redefine Memory Segments

You may use the custom platform file to redefine Memory segments as needed. This section describes how to shorten an existing Memory segment, IRAM, to make room for and create a new Memory segment, newIRAM.

1. Open the evmDM6437.xs platform file.
2. Redefine IRAM to be half its original size by setting its length to 0x10000:

```
memory[1] = ["IRAM",
 {
       name:  "IRAM",
       base:  0x10800000,
       len:   0x10000,        //previously set to 0x20000
     space: "code/data"
 }];
```

3. Define a new section called newIRAM in the evmDM6437.xs file. Add this code after the “L1DSRAM” definition:

```
memory[6] = ["newIRAM",
 {
       name:  "newIRAM",
       base:  0x10810000,
       len:   0x10000,
       space: "code/data"
 }];
```

4. Rebuild the project.
5. Open the generated linker command file for the mailbox example “mailbox_p64p_x.xdl”. Observe the new Memory segment “newIRAM”.

```
MEMORY
{
    DDR2 : org = 0x80000000, len = 0x8000000
    IRAM : org = 0x10800000, len = 0x10000
    CACHE_L1D : org = 0x10f10000, len = 0x8000
    SRAM : org = 0x42000000, len = 0x200000
    CACHE_L1P : org = 0x10e08000, len = 0x8000
    L1DSRAM : org = 0x10f04000, len = 0xc000
    newIRAM : org = 0x10810000, len = 0x10000
}
```

The program has now been configured and built with a new set of Memory segments.

10.4 Updating Platform Definitions to Load Code and Data in Different Segments

Another benefit of defining a custom platform file is that it allows you to specify a Memory segment for all code and data sections to load into. You can use the following values to specify the Memory segment for loading all code and data sections of your program:

- codeMemory
- dataMemory

Update the evmDM6437.xs platform file to change the default Memory segments for loading the code and data sections for the mailbox program. Add the following lines in **bold** to the file:

```
Build.platformTable['ti.platforms.generic:plat'] = {
   clockRate: 594,
   catalogName: 'ti.catalog.c6000',
   deviceName: 'TMS320CDM6437',
   customMemoryMap: memory,
   codeMemory: "SRAM",
   dataMemory: "IRAM",
};
```

Rebuild the mailbox project, and open the generated linker command file for the mailbox example “mailbox_p64p_x.xdl”. This file is generated in the Debug\configPkg\package\cfg\ subdirectory of the working directory.

Scroll to the end of the file, and you will see that the code and data sections are now assigned to load into SRAM and IRAM, respectively:

```
SECTIONS
{
    /* previously, these were all set to load into “DDR2” */

    .text: load >> (SRAM)
    .switch: load >> (IRAM)
    .stack: load > (IRAM)
    .vecs: load >> (SRAM)
    .taskStackSection: load >> (IRAM)
    .args: load > (IRAM align 0x4), fill = 0 { _argsize = 0x200; }
    xdc.noload: load >> (IRAM), type = NOLOAD
    .sysmem: load > (IRAM)
    .far: load >> (IRAM)
    myHeap: load >> (IRAM)
    .data: load >> (IRAM)
    .cinit: load > (IRAM)
    .bss: load > (IRAM)
    .const: load >> (IRAM)
    .pinit: load > (IRAM)
    .cio: load >> (IRAM)
}
```

10.5 Contents of the Originally Generated evmDM6437.xs File

```
var memory = [];

memory[0] = ["DDR2",
 {      name: "DDR2",
        base: 0x80000000,
        len: 0x8000000,
        space: "code/data"
 }];

memory[1] = ["IRAM",
 {      name: "IRAM",
        base: 0x10800000,
        len: 0x20000,
        space: "code/data"
 }];

memory[2] = ["CACHE_L1D",
 {      name: "CACHE_L1D",
        base: 0x10f10000,
        len: 0x8000,
        space: "Cache"
 }];

memory[3] = ["SRAM",
 {      name: "SRAM",
        base: 0x42000000,
        len: 0x200000,
        space: "code/data"
 }];

memory[4] = ["CACHE_L1P",
 {      name: "CACHE_L1P",
        base: 0x10e08000,
        len: 0x8000,
        space: "Cache"
 }];

memory[5] = ["L1DSRAM",
 {      name: "L1DSRAM",
        base: 0x10f04000,
        len: 0xc000,
        space: "data"
 }];

Build.platformTable['ti.platforms.generic:plat'] = {
 clockRate: 594,
 catalogName: "ti.catalog.c6000",
 deviceName: "TMS320CDM6437",
 customMemoryMap: memory
};
```

11 Conclusion

After following the example in this document, you should have a better understanding of DSP/BIOS 6 and changes from DSP/BIOS 5. Although the steps were tailored for the DSP/BIOS mailbox example for a specific platform, you should be able to migrate *any* application for *any* supported platform to DSP/BIOS 6 by following its example.

12 References

- *TMS320 DSP/BIOS 6 User's Guide* (SPRUEX3).
- *XDCtools Release Notes* (XDC_INSTALL_DIR/release_notes.html). Includes information about changes in each version, known issues, validation, and device support.
- *DSP/BIOS 6 Release Notes* (BIOS_INSTALL_DIR/release_notes.html). Includes information about changes in each version, known issues, validation, and device support.
- *DSP/BIOS Getting Started Guide* (BIOS_INSTALL_DIR/docs/ Bios_Getting_Started_Guide.doc). Includes steps for installing and validating the installation.
- *XDCtools Getting Started Guide* (XDC_INSTALL_DIR/docs/ XDCtools_Getting_Started_Guide.pdf). Steps for installing and validating the installation.
- RTSC-pedia at http://rtsc.eclipse.org/docs-tip for more about RTSC and XDCtools.
- RTSC+Eclipse QuickStart at http://rtsc.eclipse.org/docs-tip/RTSC+Eclipse_QuickStart for information about using RTSC in CCStudio 4.

Appendices

A Unsupported DSP/BIOS 5 APIs

The following table lists DSP/BIOS 5 APIs for which legacy support is currently not provided in DSP/BIOS 6. The table indicates whether there are plans to support each API in a future release. For a list of correspondences between DSP/BIOS 5 and 6 modules, see Section 4.1.

Table 3. Unsupported DSP/BIOS 5 APIs

Module	API	Future Support?
DGS, DHL, DST	Dxx_close	No
DGS, DHL, DST	Dxx_ctrl	No
DGS, DHL, DST	Dxx_idle	No
DGS, DHL, DST	Dxx_init	No
DGS, DHL, DST	Dxx_issue	No
DGS, DHL, DST	Dxx_open	No
DGS, DHL, DST	Dxx_ready	No
DGS, DHL, DST	Dxx_reclaim	No
GBL	GBL_getVersion	No
HST	HST_getpipe	No
HWI	HWI_enter	No
HWI	HWI_exit	No
MPC	MPC_getPA	Yes
MPC	MPC_getPageSize	Yes
MPC	MPC_getPrivMode	Yes
MPC	MPC_setBufferPA	Yes
MPC	MPC_setPA	Yes
MPC	MPC_setPrivMode	Yes
PIP	PIP_alloc	No
PIP	PIP_free	No
PIP	PIP_get	No
PIP	PIP_getReaderAddr	No
PIP	PIP_getReaderNumFrames	No
PIP	PIP_getReaderSize	No
PIP	PIP_getWriterAddr	No
PIP	PIP_getWriterNumFrames	No
PIP	PIP_getWriterSize	No
PIP	PIP_peek	No
PIP	PIP_put	No
PIP	PIP_reset	No
PIP	PIP_setWriterSize	No
PRD	PRD_tick	No
RTDX	RTDX_channelBusy	No
RTDX	RTDX_CreateInputChannel	No

Module	API	Future Support?
RTDX	RTDX_CreateOutputChannel	No
RTDX	RTDX_disableInput	No
RTDX	RTDX_disableOutput	No
RTDX	RTDX_enableInput	No
RTDX	RTDX_enableOutput	No
RTDX	RTDX_isInputEnabled	No
RTDX	RTDX_isOutputEnabled	No
RTDX	RTDX_read	No
RTDX	RTDX_readNB	No
RTDX	RTDX_sizeofInput	No
RTDX	RTDX_write	No
TSK	TSK_deltatime	No
TSK	TSK_getsts	No
TSK	TSK_resetTime	No
TSK	TSK_settime	No
TSK	TSK_tick	No

B Unsupported DSP/BIOS 5 Tconf Properties

The following DSP/BIOS 5 Tconf configuration properties are no longer supported. If your RTSC configuration configures one of these properties, a warning message is provided when you build the application.

B.1 CLK Module Properties

```
CLK.CONFIGURETIMER
CLK.PRD
CLK.WHICHHIRESTIMER
CLK.FIXTDDR
CLK.TCRTDDR
CLK.POSTINITFXN
CLK.CONONDEBUG
CLK.STARTBOTH
```

```
CLK.<Instance>.order
```

B.2 GBL Module Properties

```
GBL.C641XCCFGL2MODE
GBL.C641XCONFIGUREL2
```

B.3 HWI Module Properties

```
HWI.RESETVECTOR
HWI.RESETVECTORADDR
HWI.EXTPIN4POLARITY
HWI.EXTPIN5POLARITY
HWI.EXTPIN6POLARITY
HWI.EXTPIN7POLARITY
```

```
HWI.<Instance>.monitor
HWI.<Instance>.addr
HWI.<Instance>.dataType
HWI.<Instance>.operation
```

B.4 IDL Module Properties

```
IDL.AUTOCALCULATE
IDL.LOOPINSTCOUNT
```

```
IDL.<Instance>.calibration
IDL.<Instance>.order
```

B.5 LOG Module Properties

```
LOG.<Instance>.dataType
LOG.<Instance>.format
```

B.6 MEM Module Properties

```
MEM.GBLINITSEG
MEM.TRCDATASEG
MEM.SYSDATASEG
MEM.OBJSEG
MEM.BIOSSEG
MEM.SYSINITSEG
MEM.HWISEG
MEM.VECSEG
MEM.RTDXTESTSEG
MEM.USERCOMMANDFILE
MEM.ENABLELOADADDR
MEM.LOADBIOSSEG
MEM.LOADSYSINITSEG
MEM.LOADGBLINITSEG
MEM.LOADTRCDATASEG
MEM.LOADTEXTSEG
MEM.LOADSWITCHSEG
MEM.LOADCINITSEG
MEM.LOADPINITSEG
MEM.LOADCONSTSEG
MEM.LOADECONSTSEG
MEM.LOADHWISEG
MEM.LOADHWIVECSEG
MEM.LOADRTDXTEXTSEG
```

```
MEM.<Instance>.base
MEM.<Instance>.len
MEM.<Instance>.space
```

B.7 PRD Module Properties

```
PRD.<Instance>.arg1
PRD.<Instance>.order
```

B.8 SWI Module Properties

```
SWI.<Instance>.order
```

B.9 TSK Module Properties

```
TSK.<Instance>.manualStack
TSK.<Instance>.allocateTaskName
TSK.<Instance>.order
```

C Performance Benchmarks

This appendix shows a side-by-side comparison of timing and sizing benchmarks between DSP/BIOS 5.32, DSP/BIOS 6.10, and DSP/BIOS 6.10 legacy support. This benchmark data is meant to help you more easily compare DSP/BIOS 5 and DSP/BIOS 6.

For any given benchmark, the result for the version of DSP/BIOS that has the best score is in **bold** for clarity. If the best result is a “tie” between 2 or more DSP/BIOS versions, then the results for all “tied” versions are in bold.

For more information on DSP/BIOS 5 benchmarks, please refer to the following documents:

- *DSP/BIOS Benchmarks* (SPRAA16D)
- *DSP/BIOS Sizing Guidelines for TMS320C2000/C5000/C6000 DSPs* (SPRA772A)

For more information on DSP/BIOS 6 benchmarks, please refer to:

- *TMS320 DSP/BIOS 6 User’s Guide* (accessible via the DSP/BIOS 6 release notes)

C.1 C64 Timing Benchmarks

The following timing benchmarks were run using the C64x Functional Simulator for Little Endian. Note that for DSP/BIOS 6 benchmarks, the timer has a margin of error of +/- 8 CPU cycles. So, when comparing benchmark numbers between DSP/BIOS 6 and DSP/BIOS 6 legacy, numbers that differ by 8 or fewer CPU cycles should be considered equivalent. For some cases this results in a “tie” for the results between DSP/BIOS 6 and DSP/BIOS 6 legacy that differ by 8.

Benchmark	DSP/BIOS 6 APIs (CPU cycles)	DSP/BIOS 6 legacy APIs (CPU cycles)	DSP/BIOS 5.32 (CPU cycles)
Hwi_enable	**16**	**8**	**16**
Hwi_disable	**8**	**8**	24
Hwi_prolog	96	96	**64**
Hwi_epilog	112	112	**64**

Benchmark	DSP/BIOS 6 APIs (CPU cycles)	DSP/BIOS 6 legacy APIs (CPU cycles)	DSP/BIOS 5.32 (CPU cycles)
Int-to-Task	**320**	**320**	568
Int-to-Swi	216	208	**192**
Swi_enable	**48**	**48**	72
Swi_disable	**8**	**16**	24
Swi_post, again	40	40	**32**
Swi_post, no switch	80	72	**56**
Swi_post, switch	136	128	**120**
Task_create, no switch	840	840	**664**
Task_setPri, no switch	**160**	**152**	272
Task_yield	**152**	**144**	232
Semaphore_post, no task	**32**	**32**	**32**
Semaphore_post, switch	**168**	**168**	264
Semaphore_pend, no switch	40	48	**24**
Semaphore_pend, switch	**176**	**176**	232
CLK_gethtime / xdc_runtime_Timestamp_get32	**-**	**24**	64
CLK_getltime / Clock_getTicks	**8**	**8**	16

C.2 C64 Sizing Benchmarks

The following size benchmarks were built for C64x little-endian mode.

Table 4. Sizes for the Basic DSP/BIOS Application

	DSP/BIOS 6	DSP/BIOS 6 legacy	DSP/BIOS 5
Code (8-bit bytes)	3328	7168	**1984**
Initialized Data (8-bit bytes)	520	637	**56**
Uninitialized Data (8-bit bytes)	1900	2530	**1304**
C initialization (8-bit bytes)	1076	2100	**916**
Total Size	6824	12435	**4260**

Table 5. Sizes for the Swi Static Module Application

	DSP/BIOS 6	**DSP/BIOS 6 legacy**	**DSP/BIOS 5**
Code (8-bit bytes)	**6080**	10112	6144
Initialized Data (8-bit bytes)	530	633	**72**
Uninitialized Data (8-bit bytes)	2212	2742	**1784**
C initialization (8-bit bytes)	1500	2332	**1308**
Total Size	10322	15819	**9308**

Table 6. Sizes for the Semaphore Static Module Application

	DSP/BIOS 6	**DSP/BIOS 6 legacy**	**DSP/BIOS 5**
Code (8-bit bytes)	**11872**	14752	11936
Initialized Data (8-bit bytes)	564	613	**223**
Uninitialized Data (8-bit bytes)	4652	6522	**4456**
C initialization (8-bit bytes)	**2612**	3116	3340
Total Size	**19700**	25003	19955

Table 7. Sizes for the Memory Static Module Application

	DSP/BIOS 6	**DSP/BIOS 6 legacy**	**DSP/BIOS 5**
Code (8-bit bytes)	14240	17280	**13344**
Initialized Data (8-bit bytes)	652	661	**231**
Uninitialized Data (8-bit bytes)	8872	10684	**8620**
C initialization (8-bit bytes)	**2780**	3236	3468
Total Size	26544	31861	**25663**

Table 8. Sizes for the Dynamic Semaphore Module Application

	DSP/BIOS 6	**DSP/BIOS 6 legacy**	**DSP/BIOS 5**
Code (8-bit bytes)	17504	18400	**15968**
Initialized Data (8-bit bytes)	820	669	**235**
Uninitialized Data (8-bit bytes)	8872	10692	**8632**
C initialization (8-bit bytes)	**2780**	3268	3500
Total Size	29976	33029	**28335**