Application Report
SPRAAS7F – July 2011

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

Steven Connell SYS/BIOS Kernel Team

ABSTRACT

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

This application report contains example code that can be downloaded from this link: http://www-s.ti.com/sc/techlit/spraas7.zip

Contents

1 Introduction to SYS/BIOS 6.x Legacy Support

This application note introduces DSP/BIOS 5.x users to SYS/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 SYS/BIOS 6 features.

The intention of this application note is to help you upgrade to SYS/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 SYS/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. The CCS steps in this application note apply to CCSv4.2.

Along with SYS/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 11.

1.1 What Legacy Support is Provided?

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

- The vast majority of DSP/BIOS 5 C APIs are fully supported and callable within SYS/BIOS 6, requiring no changes to existing C code.
- Tconf configuration (*.tcf) code is supported within SYS/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.
- SYS/BIOS 6 does not support all targets that were previously supported by DSP/BIOS 5. The following targets are not supported by SYS/BIOS 6: C54x, C55x, C62x, C64x, and C67x. For a full list of targets and devices that are supported, please refer to the *SYS/BIOS Release Notes*, which is in the top-level SYS/BIOS installation directory.

Expanding on these highlights, most DSP/BIOS 5 legacy C APIs still exist in SYS/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 SYS/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 SYS/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 10 and 11 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 CCS Project”
  - Section 4, “Converting Your C Code”
  - Section 5, “Migrating Memory Configurations”
  - Section 6, “Migrating Library Builds”

- **Mailbox Example.** Sections 7 through 9 demonstrate the migration process using the DSP/BIOS 5 mailbox example:
  - Section 7, "Porting the Mailbox Example to SYS/BIOS 6 Using Legacy Support"
  - Section 8, "Porting the Mailbox Example to SYS/BIOS 6 with New Modules and APIs"
  - Section 9, "Adding SYS/BIOS Tasks and Communication to a Legacy Application"
- **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 SYS/BIOS 6:

- **CCSv4 or higher**
  - CCSv4 or CCSv5 is required in order to configure SYS/BIOS using the XGCONF tool as described in Section 7.6. Steps and figures in this document use CCSv4.2.
- **XDCtools**
  - XDCtools is installed along with CCSv4. Steps in this document use XDCtools 3.20.
  - 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.
- **SYS/BIOS 6**
  - SYS/BIOS is installed along with CCSv4. Steps in this document use SYS/BIOS 6.30.
  - For setup details and instructions, please see the *SYS/BIOS 6 Getting Started Guide* that is included with the SYS/BIOS installation.
- **IPC 1.2x**
  - IPC is installed along with CCSv4.2. Steps in this document use IPC 1.21.
  - For setup details and instructions, please see the *SYS/BIOS Inter-Processor Communication (IPC) and I/O User's Guide* (SPRUGO6).
- **DSP/BIOS 5**
  - The steps in this application note are based on the "mailbox" example, which is packaged with DSP/BIOS 5. Some newer releases of DSP/BIOS 5.x did not ship with

the mailbox example. If your DSP/BIOS 5.x installation does not have the mailbox example, you can download it from http://www-s.ti.com/sc/techlit/spraas7.zip.

- **Drivers**
  - You must have already installed the appropriate drivers for your hardware configuration, if necessary. Many common drivers are installed as part of the other installations.

2 Converting Your Configuration

In SYS/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 SYS/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 7.2) is run.

SYS/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 a legacy Tconf configuration script into a RTSC configuration script and a RTSC platform package. The generated RTSC configuration script and platform package are then added to the CCSv4 project in order to build the converted program.

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).
- A repository location on your computer in which the configuration tool will store the generated RTSC platform package. This can be any location on your computer, for example "C:\myRepository".

Note: If your configuration file imports any additional configuration files (*.tci files) by means of calls to `utils.importFile()`, then you must update your CCSv4 project with the location of these files so that XDCtools can find them at configuration time. See Section 3.4, "Updating a CCSv4 Project's Search Path to Find Included Configuration Files" for how to do this.

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]
   [--pr package repository] [--pn package name] infile.tcf
```

Options:

-c	name of the output RTSC configuration file
--pr	location of the repository to contain the generated RTSC platform package
--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 SYS/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] [--pr package repository] [--pn package name] 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 B.1.

The output from the conversion tool is a RTSC platform package and a RTSC configuration script (with the same name as the Tconf script, but with the extension .tcf replaced with .cfg). You can specify the name of the generated configuration script as well as the name of the platform using the -c and --pn options, respectively.

The conversion tool makes changes like the following to the Tconf script (exact statements will vary), and saves them in the RTSC configuration script:

Old Syntax	New Syntax
`utils.loadPlatform`	`xdc.loadPackage('ti.bios.tconf')`
`prog.gen()`	`<blank>`
`<instance>.fxn = prog.extern("fxn");`	`<instance>.fxn = "fxn";`
`<instance>.arg0 = prog.decl("foo");`	`<instance>.arg0 = $externPtr("foo");`
`bios.MEM.[LOAD]TEXTSEG = prog.get("sectionname");`	`Program.sectMap[".text"].[run|load]Segment = "sectionname";`

In previous versions of the conversion tool, unsupported MEM configuration code was placed in a comment in the *.cfg file, and a warning stated that the property was no longer supported. This is no longer the case; unsupported properties are ignored when generating the *.cfg file.

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 ...
```

2.3 How Memory Map Settings Are Translated

In addition to the *.cfg file, the conversion tool generates a RTSC platform package. This platform package contains a memory map configuration and platform-specific configuration (such as clockRate and deviceName), and will be used within your CCSv4 project to build the legacy program you are converting.

The conversion tool extracts memory map information from the Tconf script and saves equivalent information in the platform package. The platform package's memory map corresponds to the memory map created by the legacy program's Tconf script. The platform package also contains information from bios.MEM and bios.GBL parameters in the Tconf script.

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 the platform package, 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.

You should use the generated RTSC platform as the RTSC Platform you specify in your CCSv4 project settings (either when creating the project, or by updating the project's build options).

The generated RTSC platform package will be saved to the location which is specified with the conversion tool's --pr command line option.

For more on memory configurations, see Section 5 of this document and Chapter 5 of the *SYS/BIOS User's Guide* (SPRUEX3)..

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
  --pr <full path to repository location> --pn myPlatform <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.catalogName:ti.catalog.c6000
        params.deviceName:DM6437
        params.clockRate:594
Target: ti.targets.C64P
Clock Rate: 594
```

This command also creates the following files:

- myConfig.cfg
- <filename>cfg.h (where <filename> is the filename of the .tcf file)
- myPlatform: a directory containing the generated RTSC platform package, located in the repository specified on the command line

The “ti.bios.conversion” utility converts the *.tcf file to a RTSC configuration file called myConfig.cfg and also creates a RTSC platform "myPlatform". This RTSC configuration script, in combination with the custom RTSC platform "myPlatform", may now be used to configure your application and build with SYS/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 or myConfig.h) already exist in the current directory, or if the platform package directory myPlatform is already populated, 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.

2.5 Memory Map Issues for C28x Users

SYS/BIOS 6 uses a larger stack than DSP/BIOS 5, as well as having a slightly larger footprint. In result, it is possible that not all code and data from a legacy DSP/BIOS 5 program will fit into the same memory map with SYS/BIOS 6. If your legacy program places a lot of code or data in internal memory, you may see link time errors when building your converted program.

If this happens, you may need to decrease the system stack size by adding the following configuration code at the end of the *.cfg file generated by the conversion tool:

```
Program.stack = 0x200;
```

You may also need to modify the placement of other memory sections, for example by moving them into external memory, in order for the program to link without errors.

For more information on placing memory sections, see Section 5.2, “Memory Segments and Sections” of this document, as well as the section on “Placing Sections into Memory Segments" in the *SYS/BIOS User's Guide*.

3 Converting and Updating Your CCS Project

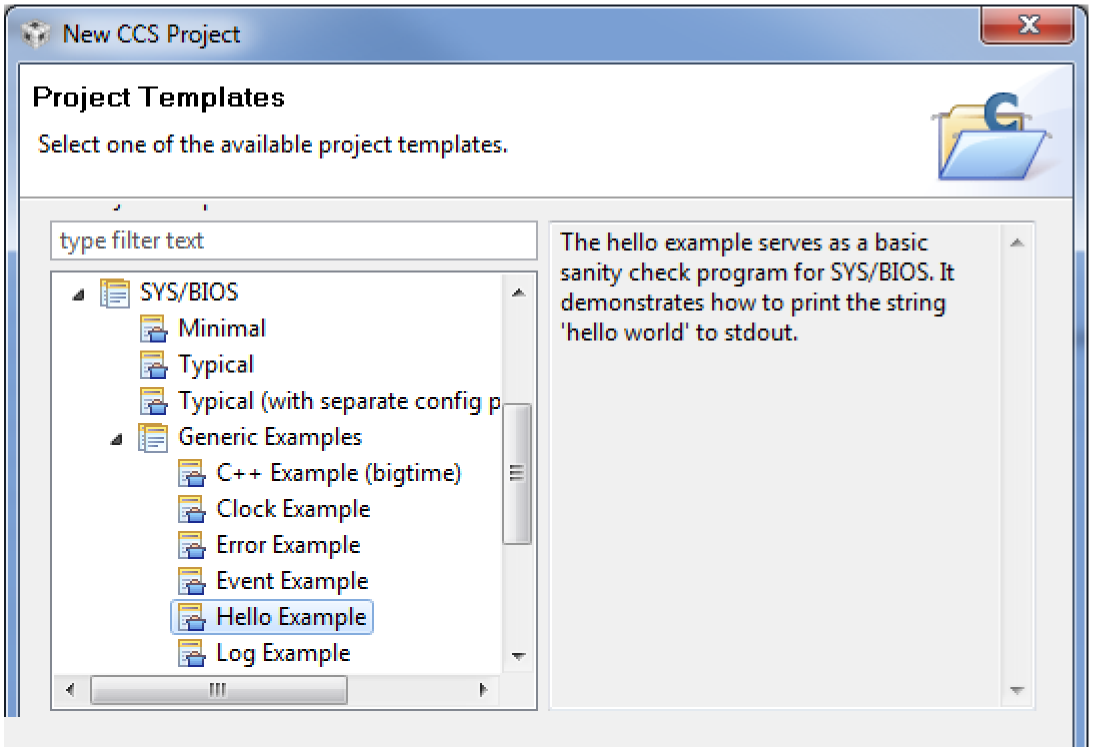
The following subsections describe changes you may need to make within your Code Composer Studio project. The CCS steps in this application note apply to CCSv4.2.

3.1 Migrating Code Composer Studio Projects

To migrate your DSP/BIOS 5 application to SYS/BIOS 6, you will need to create a new CCSv4 project. This is true even if you have already created a CCSv4 project for your DSP/BIOS 5 application.

Use the following general steps to migrate your legacy DSP/BIOS 5 application to build with SYS/BIOS 6. These steps are covered in greater detail for the mailbox example in Section 7.3, “Creating a CCSv4 Project for the Mailbox Program”.

1. Use the New CCS Project wizard in CCSv4 (**File > New > CCS Project**) to create a new project, selecting the correct hardware platform and choosing the SYS/BIOS > Generic Examples > Hello Example” project template.



The hello example serves as a basic sanity check program for SYS/BIOS. It demonstrates how to print the string 'hello world' to stdout.

2. On the “RTSC Configuration Settings” page, make sure the **RTSC Target** is specified correctly.
3. Choose a **RTSC Platform** and **RTSC Build-Profile**. For **RTSC Platform**, you should select the platform package that was created as a result of running the Conversion Tool, as described in Section 2, “Converting Your Configuration.” The steps required to choose the generated RTSC platform will be very similar to those covered in Section 3.2, “Updating a CCSv4 Project to Build with a Generated Platform Package.” For **RTSC Build-Profile**, you should choose “release” in most cases.
4. Click **Finish**. The project creation wizard creates a CCS project with the name you specified.
5. Delete the following source files from the CCS project: hello.c, common.cfg.xs, and hello.cfg.
6. Add your source file(s) to the project.
7. 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, “Converting the Mailbox Configuration Using the Conversion Tool”.
8. Configure the project build settings to match those from your CCSv3 project. You can add compiler options, assembler options, linker options, and include paths to the new project, as well as any special #defines that were set in the CCSv3 project.
9. Rebuild the project, load the program, and then run.

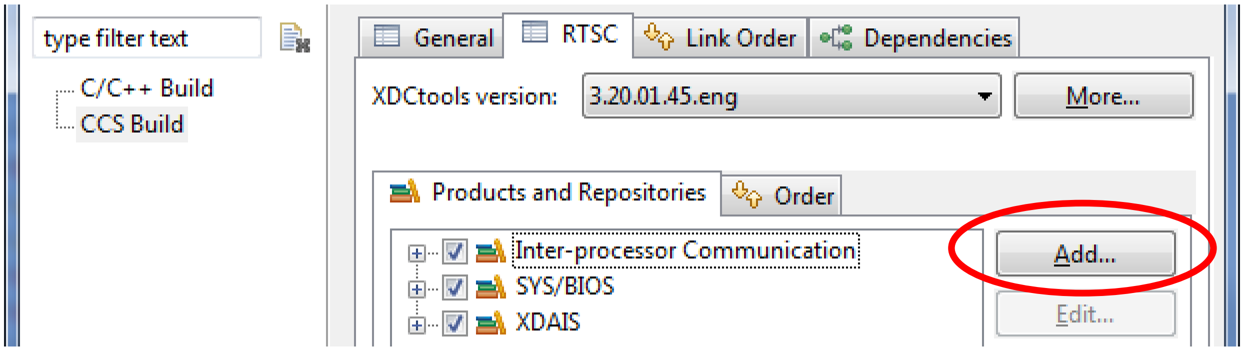
See the RTSC+Eclipse QuickStart topic in the RTSC-pedia wiki for details about creating a CCSv4 project that uses RTSC and SYS/BIOS 6 content. See the CCSv4 online help for more about converting CCSv4 projects.

3.2 Updating a CCSv4 Project to Build with a Generated Platform Package

As mentioned in Section 2.3, the conversion tool creates a RTSC platform package, which is a custom platform that is generated based on your original *.tcf file. User-created RTSC platform packages are used to create user-defined Memory segments and to change the pre-defined Memory segments that ship for a default platform. The platform package generated by the conversion tool contains settings for any memory changes in the Tconf configuration script passed to it.

In order to use a custom RTSC platform package in your build, you must select this platform under the "RTSC platform" field in the project's properties (either when creating the project, or by updating the project's properties). Use the following steps to build your project using this generated RTSC platform package:

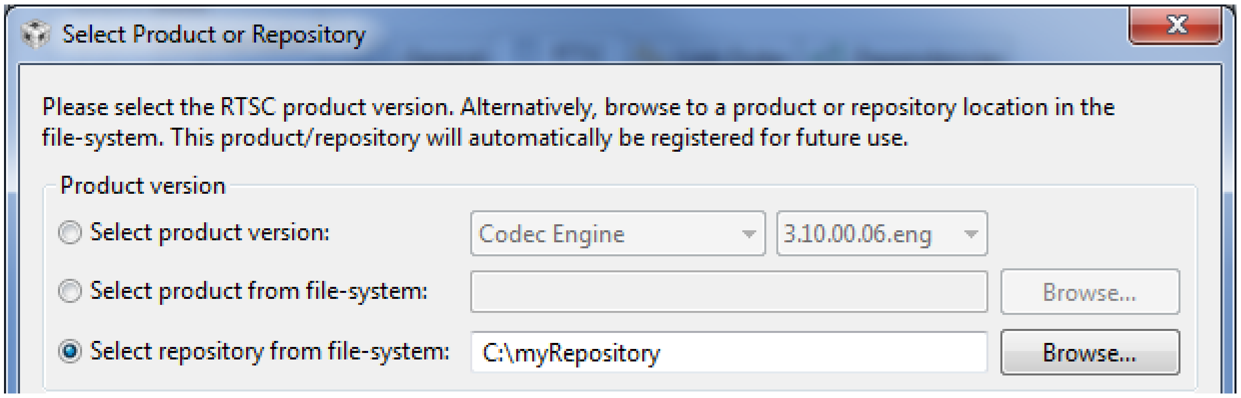
1. In the project pane, right-click on your project, and select **Build Properties**.
2. In the **CCS Build** category of the Properties dialog, choose the **RTSC** tab.
3. If you have installed more than one version of XDCtools or DSP/BIOS 6.x, make sure the version you want to use is selected.
4. Add the path to RTSC platform package repository. This allows CCSv4 to build with the RTSC platform package generated from the conversion tool. Do this by clicking the **Add** button next to the "Products and Repositories" tab.



XDCtools version: 3.20.01.45.eng

Inter-processor Communication
SYS/BIOS
XDAIS

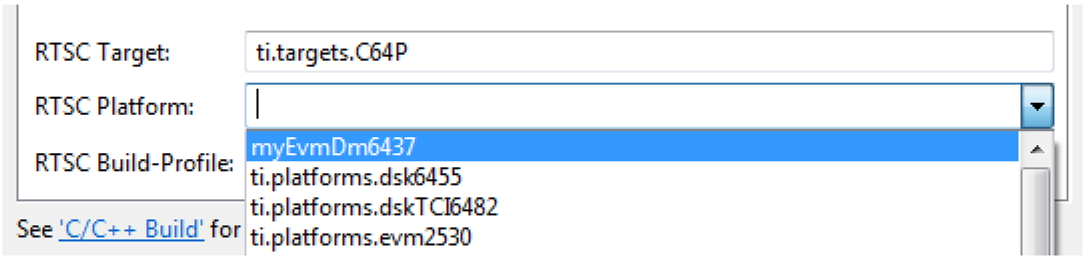
For example, to add a RTSC platform package located in the repository "C:\myRepository", enter the repository path in the "Select repository from file-system" field and click **OK**.



Please select the RTSC product version. Alternatively, browse to a product or repository location in the file-system. This product/repository will automatically be registered for future use.

Select product version: Codec Engine 3.10.00.06.eng
Select product from file-system:
[✓ Select repository from file-system:] C:\myRepository

5. In the **RTSC Platform** field, select your custom RTSC platform from the drop down list. For example, if the repository "C:\myRepository" contains a customized RTSC platform package generated from the conversion tool called "myEvmDm6437", you would see the following:



RTSC Target: ti.targets.C64P
RTSC Platform:
RTSC Build-Profile:

myEvmDm6437
ti.platforms.dsk6455
ti.platforms.dskTCI6482
ti.platforms.evm2530

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

For more information on custom platform files and how to use them in CCSv4, see Section 7.3, “Creating a CCSv4 Project for the Mailbox Program”.

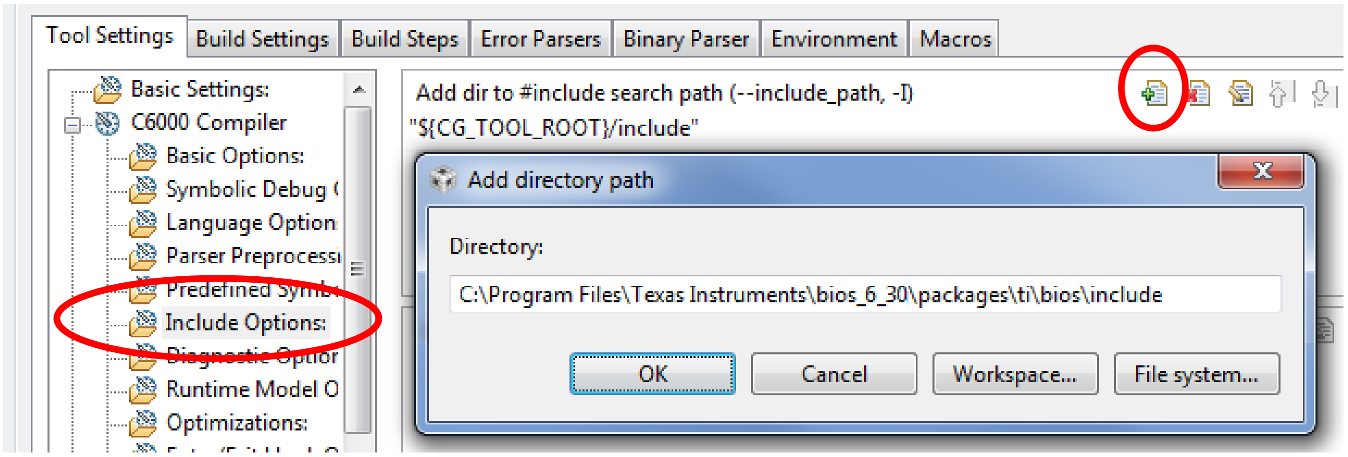
3.3 Updating the Project's Compiler Search Path

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:

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

Follow these steps to add this include file directory to the compiler search path.

1. In the CCSv4 project pane, right-click on your project and select **Build Properties**.
2. In the left pane of the project's Properties dialog, select **C/C++ Build**. You will see the build properties in the right pane.
3. In the "Configuration Settings" area, choose the **Tool Settings** tab, then click the + sign next to **C6000 Compiler** 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, -I)
"${CG_TOOL_ROOT}/include"

Directory: C:\Program Files\Texas Instruments\bios_6_30\packages\ti\bios\include

5. Click **File system** and browse to find the \packages\ti\bios\include directory of your SYS/BIOS installation and click **OK**.
6. Click **OK** in the Properties dialog to apply the settings you added.

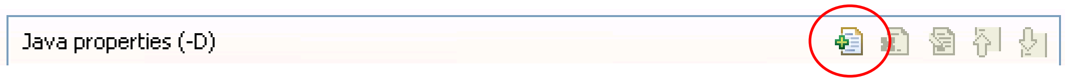
3.4 Updating a CCSv4 Project's Search Path to Find Included Configuration Files

If your configuration imports any Tconf include files (*.tci), 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 the CCSv4 project pane, right click on your project, and select **Build Properties**. Choose the **C/C++ Build** category.

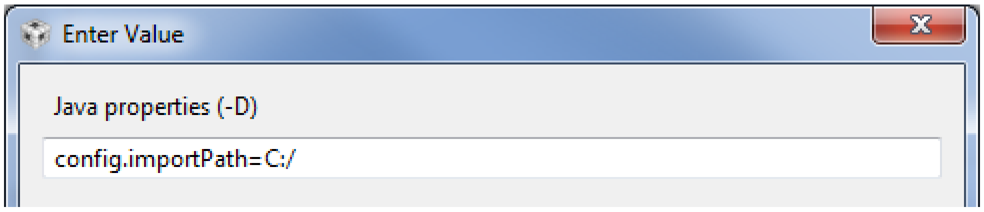
2. In the “Configuration Settings” area, choose the **Tool Settings** tab and scroll down to the **XDCtools->Advanced Options** item.
3. Click the **“+”** icon for the box that says “Java properties (-D)”.



4. 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:



```
config.importPath=C:/
```

5. Click **OK** to accept the new option.
6. Click **OK** in the Properties dialog to apply the settings you changed. 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**. Name the configuration and 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 SYS/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.

Note for C28x: As described in Section 2.5, SYS/BIOS 6 uses a larger stack than DSP/BIOS 5, as well as having a slightly larger footprint. As a result, it is possible that not all code and data from a legacy DSP/BIOS 5 program will fit into the same memory map with SYS/BIOS 6. If your legacy program places a lot of code or data in internal memory, you may see link time errors when building your converted program.

4 Converting Your C Code

Most DSP/BIOS 5 legacy C APIs still exist in SYS/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 SYS/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 SYS/BIOS 6 module counterparts. It lists both the legacy support module for identical APIs and the "new version" of that module introduced in SYS/BIOS 6. While most legacy modules are supported, we recommend that you use the new SYS/BIOS 6 modules if you are writing new code.

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

DSP/BIOS 5 Legacy Module	SYS/BIOS 6 Legacy Module	Recommended SYS/BIOS 6 Module	Legacy APIs Supported?	Support Plan
ATM	None	None	Yes	Legacy support only
BCACHE	None	ti.sysbios.hal.Cache and ti.sysbios.family.c64p.Cache	Yes	--
BUF	ti.bios.BUF	ti.sysbios.heaps.HeapBuf	Yes	--
C62	None	ti.sysbios.hal.Hwi and ti.sysbios.family.c64p.Hwi	Yes	--
C64	None	ti.sysbios.hal.Hwi and ti.sysbios.family.c64p.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	--

DSP/BIOS 5 Legacy Module	SYS/BIOS 6 Legacy Module	Recommended SYS/BIOS 6 Module	Legacy APIs Supported?	Support Plan
GBL	ti.bios.GBL	ti.sysbios.BIOS and ti.sysbios.hal.Cache	Yes*	--
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.knl.Mailbox	Yes	--
MEM	ti.bios.MEM	xdc.runtime.Memory and ti.sysbios.heaps.HeapMem	Yes	Supported in XDCtools
MPC	None	None	No	To be supported in a future release
MSGQ	ti.bios.MSGQ	ti.sdo.ipc.MessageQ	Yes	Supported in IPC
PIP	None	None	No	Will not be supported
POOL	ti.bios.POOL	xdc.runtime.Memory and ti.sysbios.heaps.Heap*	Yes	Supported in XDCtools
PRD	ti.bios.PRD	ti.sysbios.knl.Clock	Yes*	--
QUE	ti.bios.QUE	ti.sdo.utils.List	Yes	Supported in IPC
RTDX	ti.bios.RTDX	ti.sysbios.rta.Agent and RTA Tools in stop-mode, or System Analyzer and UIA	No	--
SEM	ti.bios.SEM	ti.sysbios.knl.Semaphore	Yes	--
SIO	ti.bios.SIO	ti.sdo.io.Stream	Yes	--
STS	ti.bios.STS	None	Yes	No RTA tool support. Legacy APIs allow compiling and linking.
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 SYS/BIOS 6 APIs

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

Using a combination of legacy code and new code in the same SYS/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 SYS/BIOS 6 modules in the C code and RTSC configuration.

See Sections 9 and 9.2 for examples that add SYS/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 SYS/BIOS 6 modules, you must place the new configuration code *after* the existing legacy configuration code.

If SYS/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 SYS/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 SYS/BIOS 6 and DSP/BIOS 5 names, then the SYS/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 SYS/BIOS 6 setting).

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

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

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

When you update a legacy C file to include SYS/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 SYS/BIOS 6 header file #include directives after any legacy #includes, then the SYS/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 SYS/BIOS 6, just make sure to include the SYS/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 SYS/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 SYS/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 SYS/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 SYS/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 SYS/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 SYS/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 SYS/BIOS 6 in the Memory (xdc.runtime.Memory), HeapMem (ti.sysbios.heaps.HeapMem), and Program modules.

The subsections that follow provide an overview of Memory segments and sections, Heaps, and dynamic memory allocation in SYS/BIOS 6. For more details, see Chapter 5 of the *SYS/BIOS User's Guide* (SPRUEX3).

Note for C28x: As described in Section 2.5, SYS/BIOS 6 uses a larger stack than DSP/BIOS 5, as well as having a slightly larger footprint. As a result, it is possible that not all code and data from a legacy DSP/BIOS 5 program will fit into the same memory map with SYS/BIOS 6. If your legacy program places a lot of code or data in internal memory, you may see link time errors when building your converted program.

5.1 Memory and RTSC Platforms

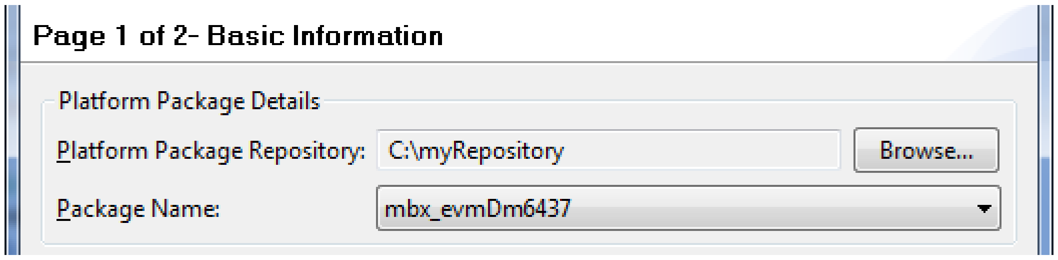
In DSP/BIOS 5, the MEM manager module was used to create new MEM segments for user applications. In SYS/BIOS 6, MEM segments have been replaced by XDCtools Memory segments, which are specified in the project's RTSC platform.

The most significant difference between MEM segments and Memory segments is how they are created. While DSP/BIOS 5 MEM segments were created or modified in the configuration script, SYS/BIOS 6 Memory segments are created or modified using the RTSC platform wizard. When you use the conversion tool, a RTSC platform with memory segments that match those in your DSP/BIOS 5 application is created automatically.

The SYS/BIOS 6 installation includes several RTSC platforms that define Memory segments. To customize these Memory segments, you can create a custom RTSC platform package and use that to build the application. To generate a custom platform package, use the conversion tool or the RTSC platform wizard.

If, after using the conversion tool to create a RTSC platform, you want to further customize its memory settings, you can use the RTSC platform wizard to edit the platform as follows:

1. In CCS v4.x, choose **Tools > RTSC Tools > Platform > Edit/View**.
2. On the Basic Information page, click **Browse** and find the location of the repository you specified when you used the conversion tool.
3. In the **Package Name** field, then select mbx_evmDm6437 (or some other RTSC platform package you want to edit). Then click **Next**.


Page 1 of 2- Basic Information

Platform Package Details

Platform Package Repository: C:\myRepository

Package Name: mbx_evmDm6437

4. Use the Device Page to change the clock speed, memory segment sizes and locations, and memory section assignments.

For more details, see Chapter 5 of the *SYS/BIOS User's Guide* (SPRUEX3) and the demos at http://rtsc.eclipse.org/docs-tip/Demo_of_the_RTSC_Platform_Wizard_in_CCSv4.

5.2 Memory Segments and Sections

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

Memory sections in SYS/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;
```

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";
```

For more details on section placement, see Chapter 5 of the *SYS/BIOS User's Guide* (SPRUEX3).

5.3 Heaps

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

All SYS/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 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 SYS/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 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 SYS/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”;
```

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

Table 2. SYS/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.

5.4 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 SYS/BIOS 6, the Memory module and the Heap modules provided by SYS/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.

SYS/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. 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);
```

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

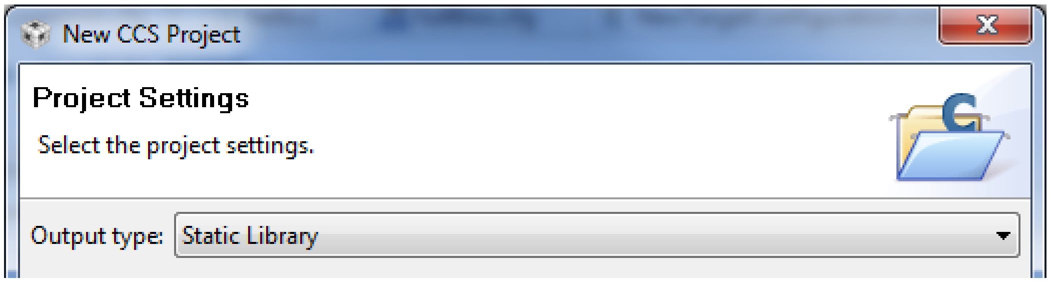
However, since library builds do not use a SYS/BIOS configuration, there are additional steps you must take to rebuild a library. You’ll need to update the library project options by adding the extra include search paths that would normally be added by the generated “compiler.opt” file.

You may rebuild your existing DSP/BIOS 5 libraries using the legacy support included with SYS/BIOS 6. To do so, you’ll need to create a new project within CCSv4. This is required because there is currently no way to automatically convert your CCSv3.x project to a CCSv4 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 CCSv4

Follow the steps to create a new CCSv4 project that uses SYS/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 **Project Settings** page in the project-creation wizard, select **Static Library** from the **Output Type** drop-down box.



Project Settings

Select the project settings.

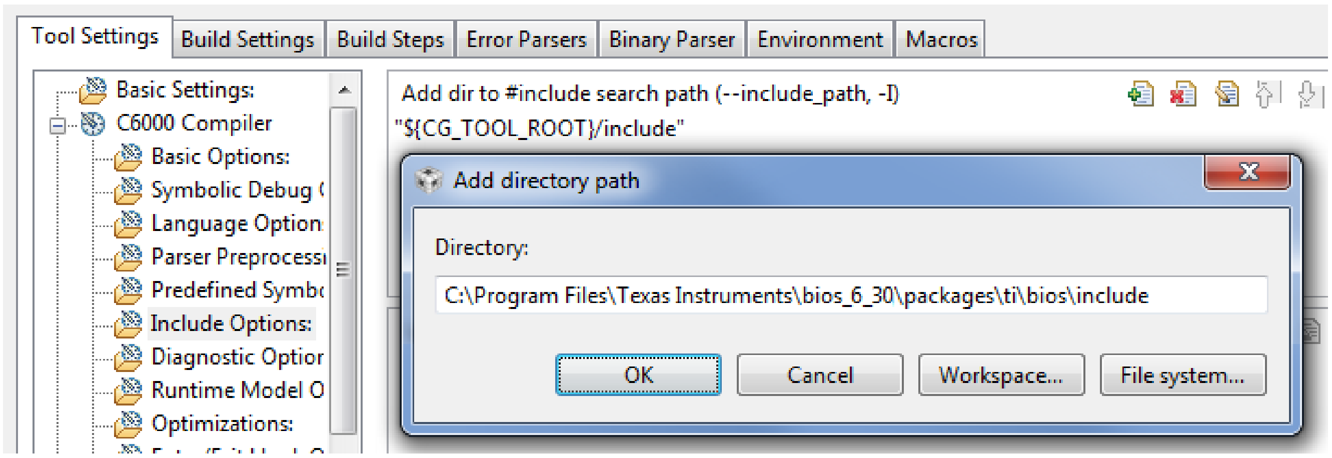
Output type: Static Library

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 CCSv4 Library Projects

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

1. In the CCSv4 project pane, right-click the library project and select **Build 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** 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, -I)
"${CG_TOOL_ROOT}/include"

Directory:
C:\Program Files\Texas Instruments\bios_6_30\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 or browse for the path to the SYS/BIOS legacy include directory in the Directory field and click **OK**.
6. For both types of library (legacy and SYS/BIOS 6 API), repeat the two previous steps. This time, add the following additional paths to the compiler's include file search path, where you replace the <*SYS/BIOS 6 Install Dir*> and <*XDCtools Install Dir*> with the actual locations:
   - <*SYS/BIOS 6 Install Dir*>/packages
   - <*XDCtools Install Dir*>/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 SYS/BIOS 6.

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

SYS/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 SYS/BIOS 6 legacy support. That is, the DSP/BIOS 5 APIs are still used, but the project is built with SYS/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 SYS/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 (mailbox.c) into the "mailbox" directory from <*DSP/BIOS 5 Install Dir*>/packages/ti/bios/examples/basic/mailbox/mailbox.c.

Note: Some newer releases of DSP/BIOS 5.x did not ship with the mailbox example. If your DSP/BIOS 5.x installation does not have the mailbox example, you can download it from http://www-s.ti.com/sc/techlit/spraas7.zip.

3. Copy the following mailbox Tconf configuration files to the "mailbox" working directory.

- packages/ti/bios/examples/basic/mailbox/mailbox.tci
- packages/ti/bios/examples/basic/mailbox/evmDM6437/mailbox.tcf
- packages/ti/bios/examples/basic/mailbox/evmDM6437/mailbox_evmDM6437_custom.tci
- packages/ti/bios/examples/common/evmDM6437_common.tci

Don't create any subdirectories in your working directory. Your working directory should simply contain the mailbox.c, mailbox.tci, mailbox.tcf, mailbox_evmDM6437_custom.tci, and evmDM6437_common.tci files when you are done copying files.

All directories shown in the previous list are relative to the location where you installed DSP/BIOS 5.x. This list shows files for the DM6437 EVM. If you substitute files for another platform, you must also change the migration steps in later sections accordingly.

7.2 Converting the Mailbox Configuration Using the Conversion Tool

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

This section demonstrates how to use the conversion tool to convert the mailbox.tcf Tconf script into the mailbox.cfg RTSC configuration script and a custom RTSC platform package.

This section uses a repository located at "C:\myRepository" to store the generated RTSC platform package. If you wish to use a different repository, replace "C:\myRepository" in these steps with the full path of your repository.

For more information on the conversion tool, see Section 2, “Converting Your Configuration”.

1. At a MS-DOS command prompt, change (`cd`) to the working 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  "C:/Program Files/Texas Instruments/<bios_install_dir>/packages"
  ti.bios.conversion --pr "C:\myRepository" --pn mbx_evmDm6437 -i mailbox.tcf
```

Running this command produces the following output:

```
Platform: ti.platforms.evmDM6437
        params.catalogName:ti.catalog.c6000
        params.deviceName:DM6437
        params.clockRate:594
Target: ti.targets.C64P
Clock Rate: 594
```

-

This command also creates the following files:

- mailbox.cfg
- mailboxcfg.h
- mbx_evmDm6437: a directory containing the generated RTSC platform package, located in the repository C:\myRepository

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 SYS/BIOS 6 form. This file is included by the C source file “mailbox.c.”

Lastly, the tool creates the directory “mbx_evmDm6437”, which contains a custom RTSC platform package. Custom RTSC platforms can be used to create user-defined Memory segments. For more information on custom RTSC platforms and how to use them, please see Section 3.2, “Updating a CCSv4 Project to Build with a Generated Platform Package”.

Known Issue: There is currently a bug (bugzilla) in XDCtools that requires a workaround in the mailbox.cfg file. This known issue will cause a build error, but the workaround is very simple.

1. Open the mailbox.cfg file in a text editor.
2. Change the following line:

```
mbx.length       = 2;
```

to be:

```
mbx.messageLength       = 2;
```

3. Save and close the file.

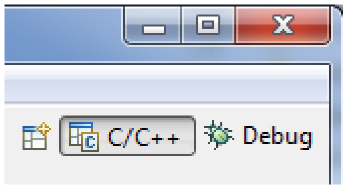
7.3 Creating a CCSv4 Project for the Mailbox Program

The **Project → Import Legacy CCSv3.3 Project** command in CCSv4 does not migrate a DSP/BIOS 5 project to the SYS/BIOS 6 project organization correctly. So, instead of migrating the existing project, it is recommended that you create a new SYS/BIOS 6 project.

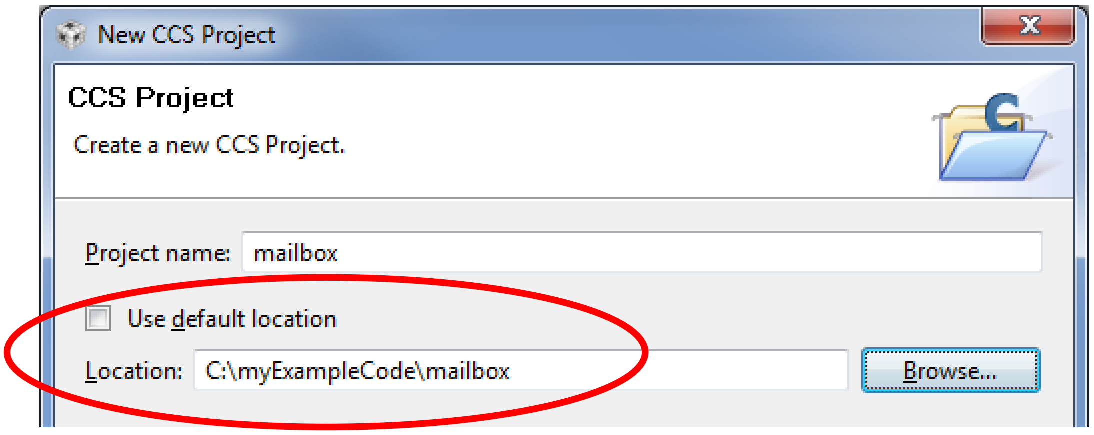
This section walks you through the CCSv4 project creation wizard to create a new project for the legacy mailbox example.

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 SYS/BIOS 6 project in CCSv4, 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 CCSv4, 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 this is unchecked, click the **Browse** button, and navigate to your working directory, choosing it for the project location.

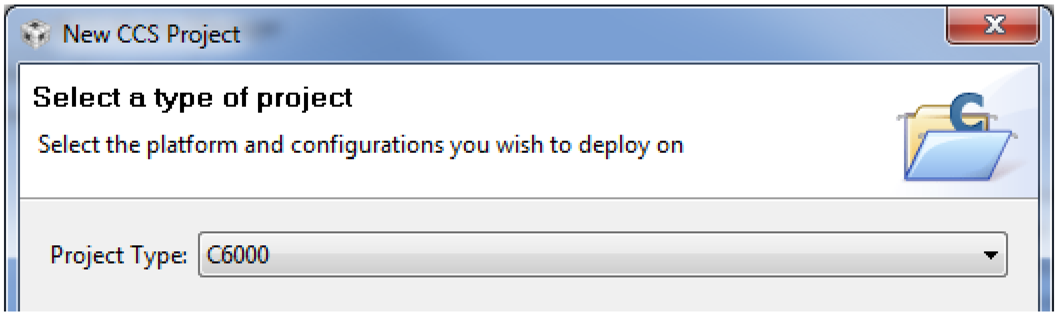


Project name: mailbox

Use default location: ☐

Location: C:\myExampleCode\mailbox

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

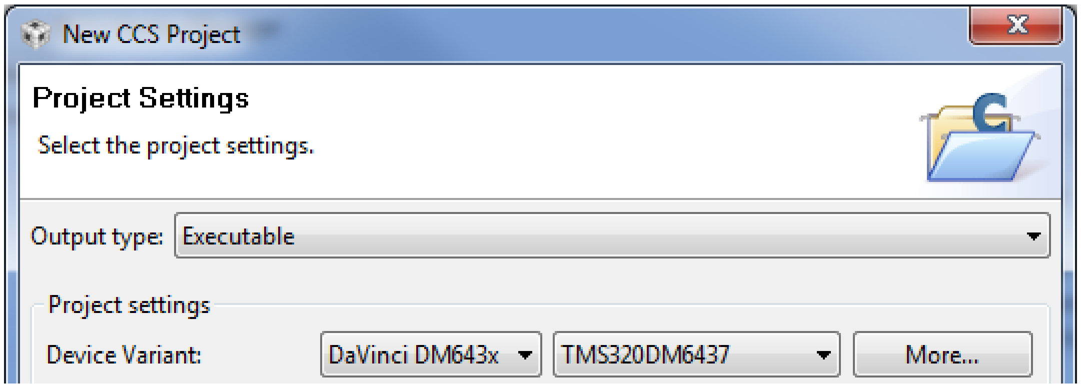

New CCS Project

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 **Output type** drop-down box. Make sure the device settings are correct for your target.


New CCS Project

Project Settings

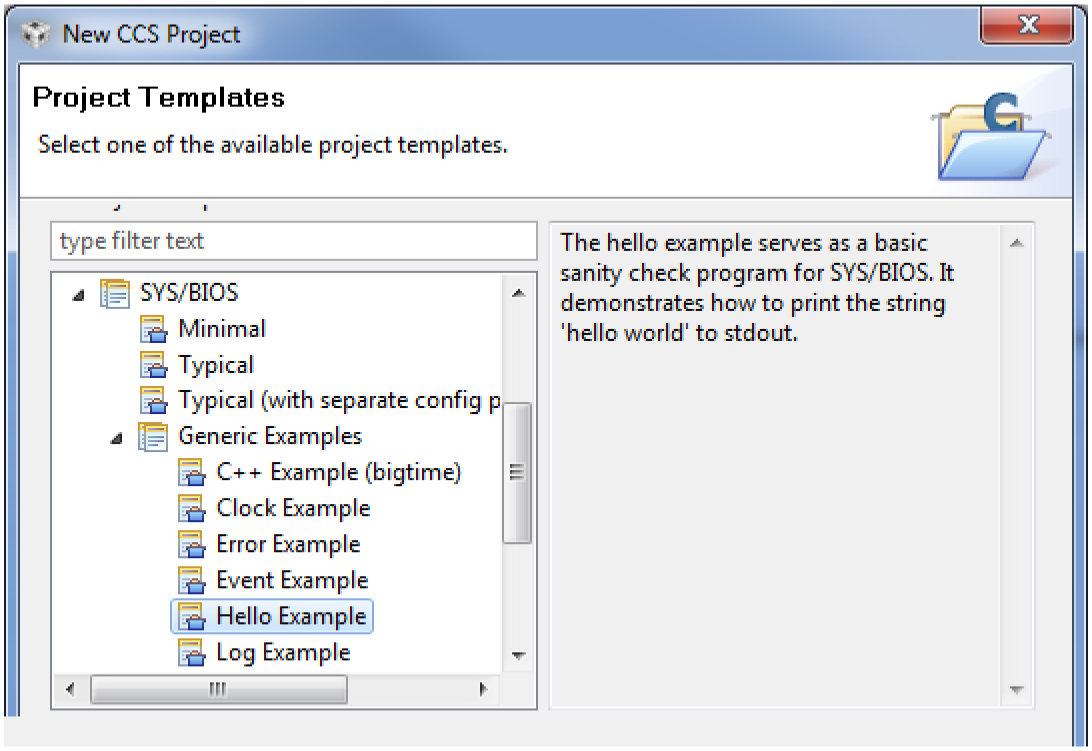
Select the project settings.

Output type: Executable

Project settings

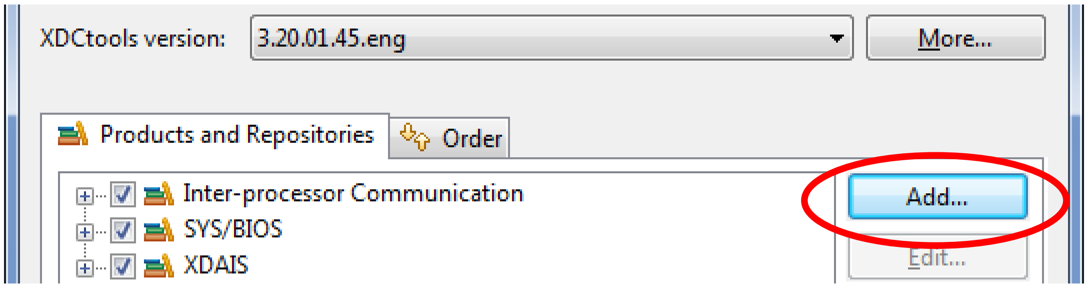
Device Variant: DaVinci DM643x TMS320DM6437

7. Click **Next** to advance to the **Project Templates** page. Select the SYS/BIOS > Generic Examples > Hello Example project template.

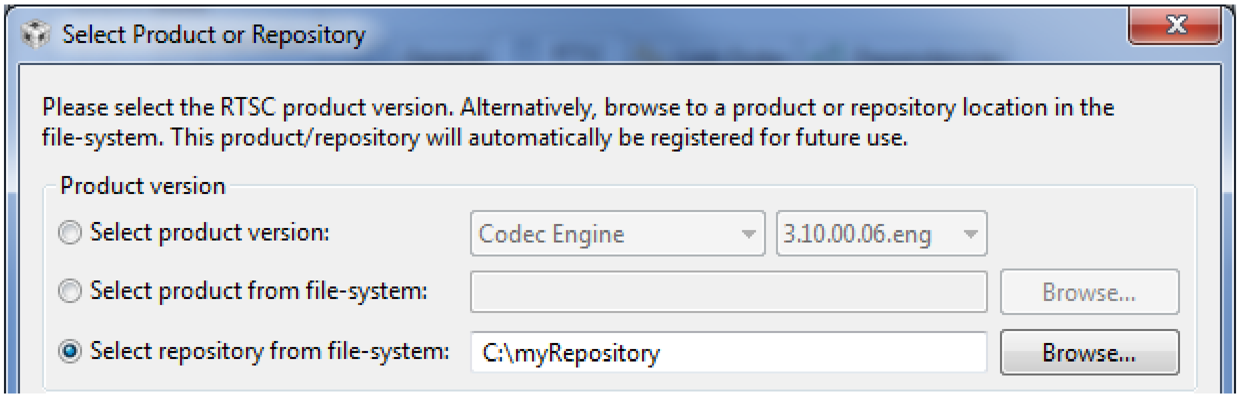


The hello example serves as a basic sanity check program for SYS/BIOS. It demonstrates how to print the string 'hello world' to stdout.

8. Click **Next** to bring the wizard to the **RTSC Configuration Settings** page. In this part of the project wizard, we'll choose the custom RTSC platform that was created by the conversion tool. In order for the wizard to find it, you need to add the repository path to the project.
9. Make sure that the latest versions you have installed of **XDCtools**, **DSP/BIOS**, and **Inter-processor Communication** are all selected on this page. These will be needed when you build the project.
10. Click the **Add** button next to "Products and Repositories."


XDCtools version: 3.20.01.45.eng

11. Choose "Select repository from file-system" and click **Browse**. Add the path to the location of the repository in which the conversion tool created a platform package. If you used the command shown in Section 7.2, the location is C:\myRepository.



Please select the RTSC product version. Alternatively, browse to a product or repository location in the file-system. This product/repository will automatically be registered for future use.

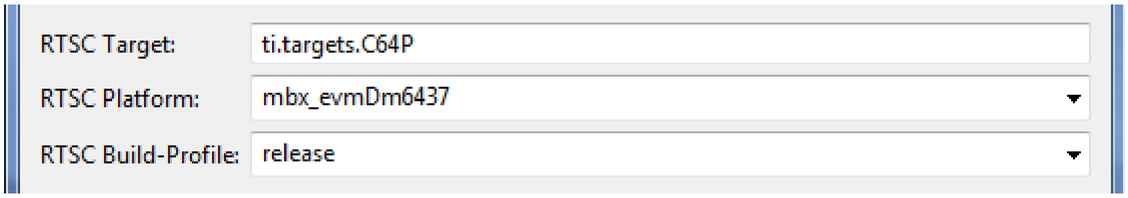
Product version

( ) Select product version: Codec Engine 3.10.00.06.eng

( ) Select product from file-system: Browse...

(●) Select repository from file-system: C:\myRepository Browse...

12. Make sure the **RTSC Target** is ti.targets.C64P.
13. Click on the **RTSC Platform** field. The wizard searches the selected repositories for C64P platforms.
14. Select the custom RTSC platform created by the conversion tool: "mbx_evmDm6437".
15. In the **RTSC Build-Profile** field, select "release".



RTSC Target: ti.targets.C64P

RTSC Platform: mbx_evmDm6437

RTSC Build-Profile: release

16. Click **Finish** to complete the project creation.
17. Notice that all the mailbox source and configuration files that exist in the working directory are automatically added to the mailbox project.
18. Right-click on unneeded files in the mailbox project and delete the following files:
    – hello.c
    – hello.cfg
    – common.cfg.xs
    – mailbox.tcf
    – mailbox.tci
    – evmDM6437_common.tci
    – mailbox_evmDM6437_custom.tci

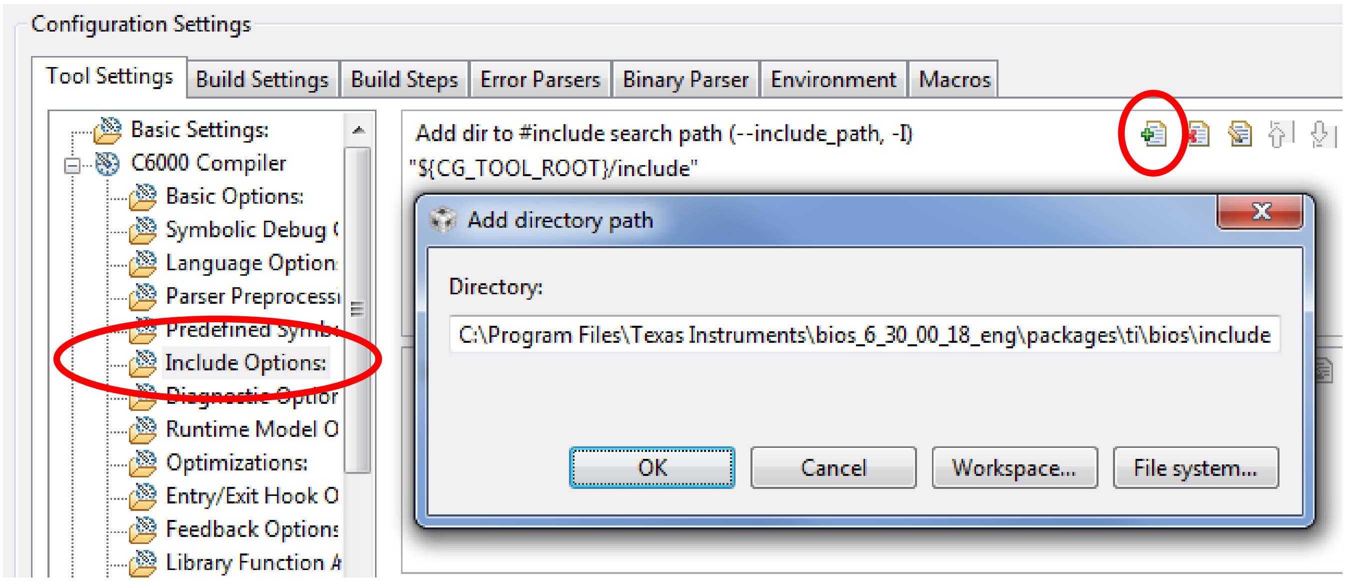
7.4 Updating the Compiler Search Path

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:

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

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

1. In the CCSv4 project pane, right-click the mailbox project and select **Build Properties**.
2. In the left pane of the mailbox project's properties window, select **C/C++ Build**.
3. In the Configuration Settings area, click the **Tool Settings** tab, then click the + sign next to **C6000 Compiler** 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)**:
5. In the **Directory** field, type or browse the file system for the path to the SYS/BIOS legacy include file directory, <*SYS/BIOS 6 Install Dir*>/packages/ti/bios/include, and click **OK**. For example:



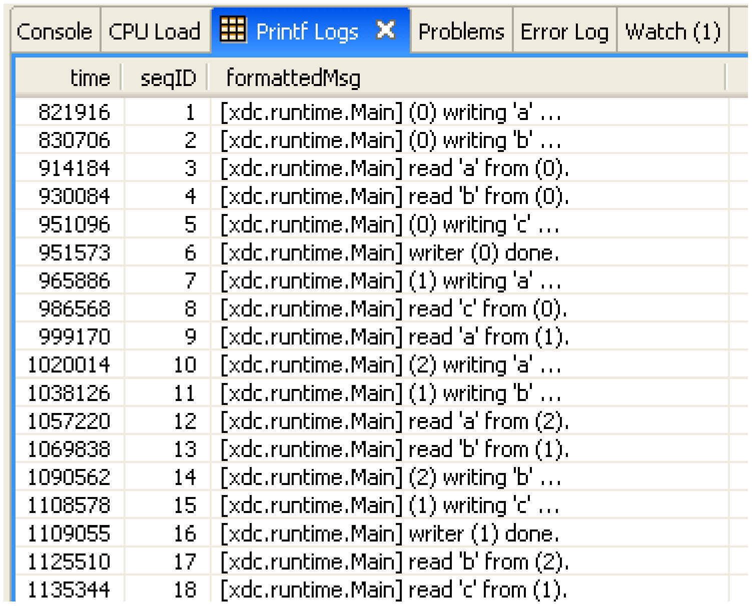
Add dir to #include search path (--include_path, -I)
"${CG_TOOL_ROOT}/include"

Directory:
C:\Program Files\Texas Instruments\bios_6_30_00_18_eng\packages\ti\bios\include

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

7.5 Building and Running the Application

1. Make sure that the active project in CCSv4 is the “mailbox” 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. If you want to use the physical DM6437 target, for **Connection**, select "Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator”. For **Device**, select "TMS320DM6437". (You can choose the “DM6437 Device Cycle Accurate Simulator, Little Endian” if you do not have a physical DM6437 target.) 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. Choose **Project → Build Active Project** to build the “mailbox” project using SYS/BIOS 6.
6. Choose **Target → Debug Active Target** to load the mailbox.out executable.
7. Switch to the Debug perspective.
8. Open the RTA log view by selecting **Tools → RTA → Printf Logs**. This opens the RTA Printf Logs pane.
9. Connect CCSv4 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).

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

Note for C28x: As described in Section 2.5, SYS/BIOS 6 uses a larger stack than DSP/BIOS 5, as well as having a slightly larger footprint. As a result, it is possible that not all code and data from a legacy DSP/BIOS 5 program will fit into the same memory map with SYS/BIOS 6. If your legacy program places a lot of code or data in internal memory, you may see link time errors when building your converted program.

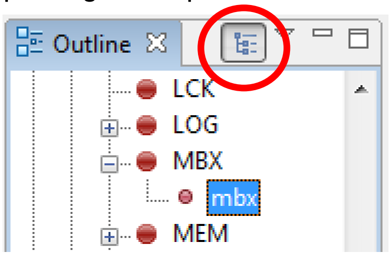
7.6 Using XGCONF to View the SYS/BIOS Configuration

You can use the XGCONF tool to view a legacy SYS/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.

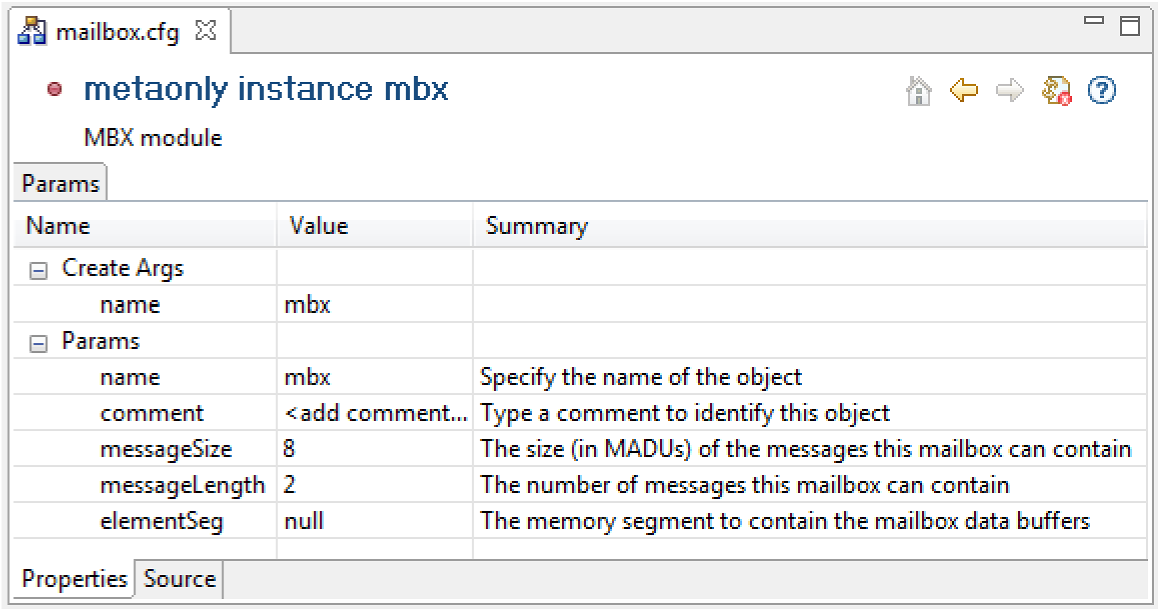
1. To open XGCONF, you must be in the **C/C++** perspective of CCSv4. Click the **C/C++** icon to switch back to that perspective.
2. Make sure the mailbox project is set to be the "Active" project. You can do this by right-clicking on the project in the "C/C++ Projects" view and selecting **Set as Active 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 file list and select **Open With → XGCONF**. It takes a few seconds for XGCONF to open. During this time, the CCS 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 by clicking the icon. Click the **+** sign next to the **ti.bios** package 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 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	Summary
⊟ Create Args		
name	mbx	
⊟ Params		
name	mbx	Specify the name of the object
comment	<add comment...	Type a comment to identify this object
messageSize	8	The size (in MADUs) of the messages this mailbox can contain
messageLength	2	The number of messages this mailbox can contain
elementSeg	null	The memory segment to contain the mailbox data buffers

7. You can use XGCONF in this way to view other properties of a SYS/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 SYS/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* SYS/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 SYS/BIOS 6 code with existing legacy DSP/BIOS 5 code.

8.1 Updating the Configuration Files

Where SYS/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 SYS/BIOS 6 are configured very differently. The following steps explain the configuration changes needed to port the mailbox application to SYS/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 SYS/BIOS 6 modules to one that uses non-legacy SYS/BIOS 6 modules.

8.1.1 Porting the mailbox.cfg File

In SYS/BIOS 6, the `useModule()` method enables the application to use a particular module, such as a SYS/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 SYS/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.knl.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 the file.

8.1.2 Porting the Contents of the evmDM6437_common.tci File

When you ran the conversion tool in Section 7.2, you used the -i option in order to migrate and copy in the code of the included *.tci files into the mailbox.cfg file. So, the code from the "evmDM6437_common.tci" file is now located within the mailbox.cfg file after a comment containing that file name. Therefore, all edits in this section will be made to the mailbox.cfg file.

The contents of the mailbox.cfg file include 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 SYS/BIOS objects and mallocs, and to configure the cache and the clock

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

1. Delete the following lines of code from the mailbox.cfg 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. RTDX is no longer supported. The Task Manager is enabled by default in SYS/BIOS 6, and the configuration code to enable RTA was already added by the conversion tool.

8.1.2.1 Converting the IRAM Heap Creation Code

Next the code in mailbox.cfg creates a heap in IRAM and maps the segment as the default heap for SYS/BIOS objects and mallocs.

In SYS/BIOS 6, heaps and segments are configured using the Program, Memory, and HeapMem modules. These modules function in SYS/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 “mailbox.cfg”:

```
/* 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 earlier in the mailbox.cfg file, 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 earlier in the mailbox.cfg file, 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 earlier in the mailbox.cfg file, assign this heap to be the default Heap for SYS/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 mailbox.cfg file:

```
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 SYS/BIOS 6, as the enabling of the cache is now handled in the RTSC platform.

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 SYS/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 SYS/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 mailbox.cfg file:

```
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 the file.

8.1.3 Porting the Contents of the mailbox.tci File

When you ran the conversion tool in Section 7.2, the -i option was used in order to migrate and copy in the code of the included *.tci files into the mailbox.cfg file. So, the code from the “mailbox.tci” file is now located within the mailbox.cfg file after a comment containing that file name. Therefore, all edits in this section will be made to the mailbox.cfg file.

The code from the “mailbox.tci” file contains configuration settings for SYS/BIOS Logs and creates SYS/BIOS Tasks for the mailbox program. While the code to configure logging in SYS/BIOS 6 is different from earlier versions, SYS/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 SYS/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 SYS/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.cfg 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 SYS/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 SYS/BIOS 6 Task module. Creating a static Task in SYS/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 the file.

8.1.4 Porting the Contents of the mailbox_evmDM6437_custom.tci File

When you ran the conversion tool in Section 7.2, the -i option was used in order to migrate and copy in the code of the included *.tci files into the mailbox.cfg file. So, the code from the "mailbox_evmDM6437_custom.tci" file is now located within the mailbox.cfg file after a comment containing that file name. Therefore, all edits in this section will be made to the mailbox.cfg file.

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

In the next steps, you will convert this configuration code to SYS/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.messageLength = 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 SYS/BIOS 6 Mailbox Configuration Using XGCONF

In Section 7.6, you saw how to use XGCONF to view the legacy SYS/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".)

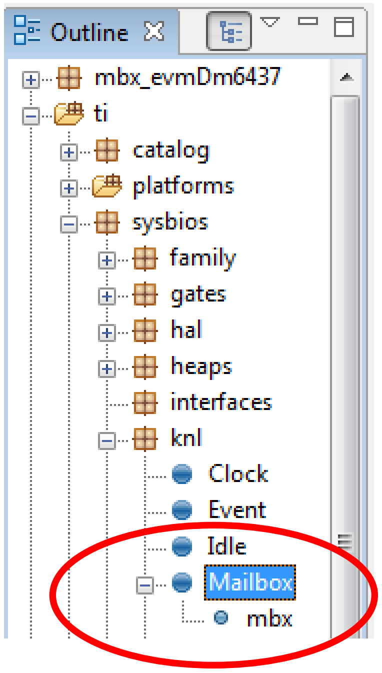
In this section, you will learn where to look in XGCONF to find SYS/BIOS 6 module settings.

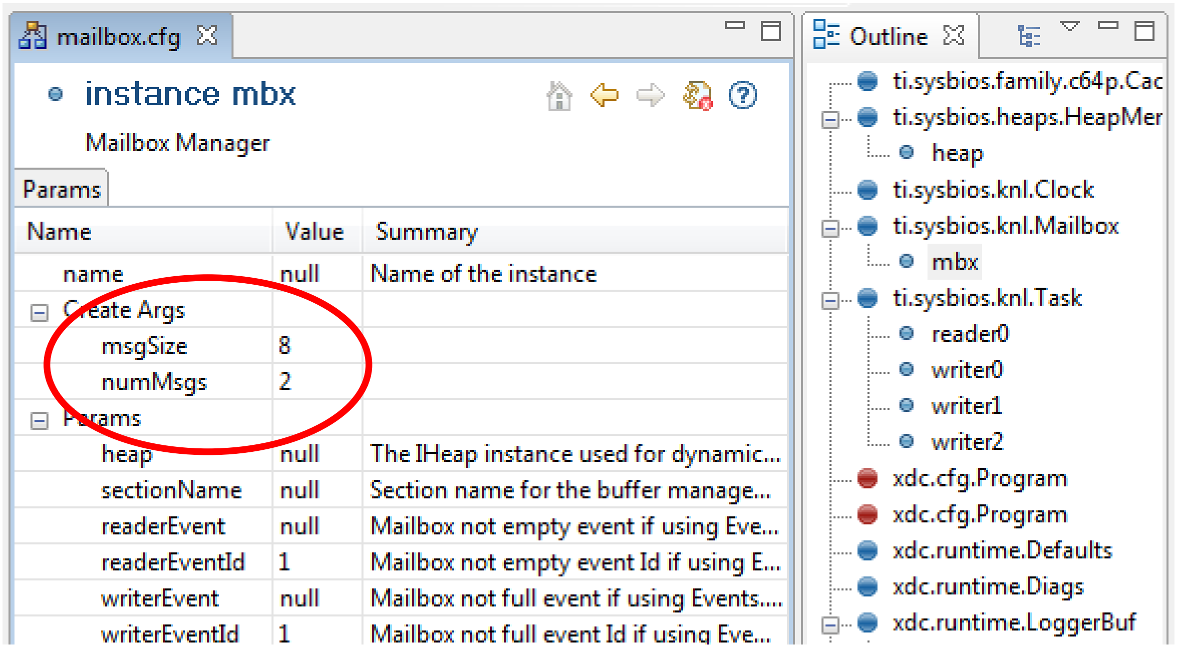
1. Make sure the mailbox project is set to be the "Active" project. You can do this by right-clicking on the mailbox project in the "C/C++ Projects" view and selecting **Set as Active Project**.
2. To open XGCONF, you must be in the **C/C++** perspective of CCS. If necessary, click the **C/C++** icon to switch back to that perspective.
3. To open XGCONF, right click the "mailbox.cfg" file in the mailbox project file list, 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.cfg”. Recall that in that section you added the following code:

```
xdc.global.Mailbox   = xdc.useModule('ti.sysbios.knl.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. In the **Outline** tab, you can use the icon to switch between two viewing modes. When the icon is highlighted, you see a tree view of all the packages available for configuration. Expand the tree in the Outline tab as shown in the figure. The tree shows that the Mailbox module is ti.sysbios.knl.Mailbox.
6. By toggling the icon off, you can change the view to show only modules that are used in the application and to list modules by their full package names.
7. 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. Notice that its properties match the statements you added to the configuration code.





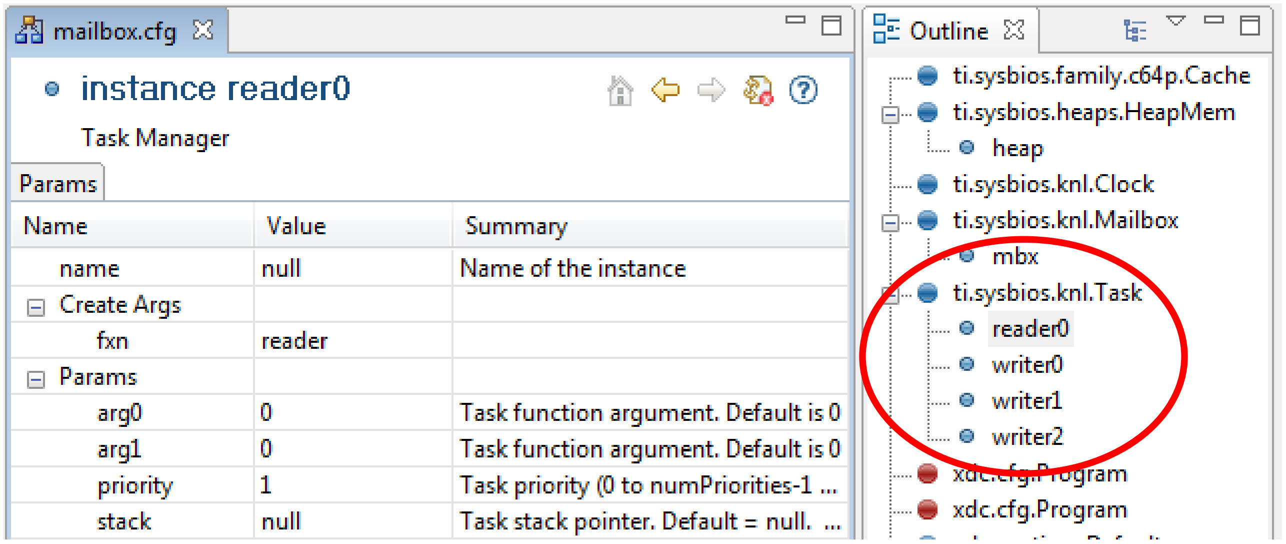
Name	Value	Summary
name	null	Name of the instance
Create Args		
msgSize	8	
numMsgs	2	
Params		
heap	null	The IHeap instance used for dynamic...
sectionName	null	Section name for the buffer manage...
readerEvent	null	Mailbox not empty event if using Eve...
readerEventId	1	Mailbox not empty event Id if using E...
writerEvent	null	Mailbox not full event if using Events....
writerEventId	1	Mailbox not full event Id if using Eve...

```
ti.sysbios.family.c64p.Cac
ti.sysbios.heaps.HeapMer
    heap
ti.sysbios.knl.Clock
ti.sysbios.knl.Mailbox
    mbx
ti.sysbios.knl.Task
    reader0
    writer0
    writer1
    writer2
xdc.cfg.Program
xdc.cfg.Program
xdc.runtime.Defaults
xdc.runtime.Diags
xdc.runtime.LoggerBuf
```

8. Next, let's look at some of the Task instances. Recall the Task instances that were created by code like the following in Section 8.1.3.2.

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

9. In the Outline pane, expand the ti.sysbios.knl.Task module. Notice there are several **Task** instances, corresponding to the instances created by the mailbox.cfg script:



Name	Value	Summary
name	null	Name of the instance
Create Args		
fxn	reader	
Params		
arg0	0	Task function argument. Default is 0
arg1	0	Task function argument. Default is 0
priority	1	Task priority (0 to numPriorities-1 ...
stack	null	Task stack pointer. Default = null. ...

You can use XGCONF in this way to view other properties of a SYS/BIOS program's configuration. All you need to remember is that the layout of the **Outline** tab reflects the layout of the SYS/BIOS 6 and XDCtools packages and modules. 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 SYS/BIOS 6, these legacy headers and C API calls must be changed to their SYS/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 SYS/BIOS 6 counterparts. For the most part, a SYS/BIOS 6 header file has a similar name to its legacy counterpart. For example, the legacy “tsk.h” corresponds to “Task.h” in SYS/BIOS 6. The SYS/BIOS 6 header files are found in the ”ti/sysbios” directory and sub-directories.

   SYS/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 SYS/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/knl/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 SYS/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/knl/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 SYS/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 SYS/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 SYS/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 SYS/BIOS 6 equivalent:

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

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

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

   with the SYS/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 SYS/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;
```

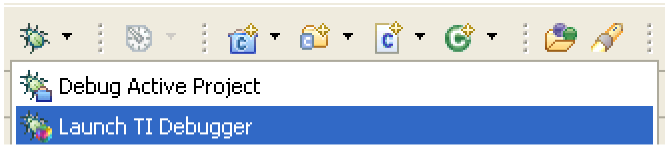
For more information on Diags, please see Section 8.2.5 for diagnostic masks used as arguments in `Log_print*()` calls, or see the online CDOC reference documentation that came with your XDCtools installation.

8.3 Building and Running the Application

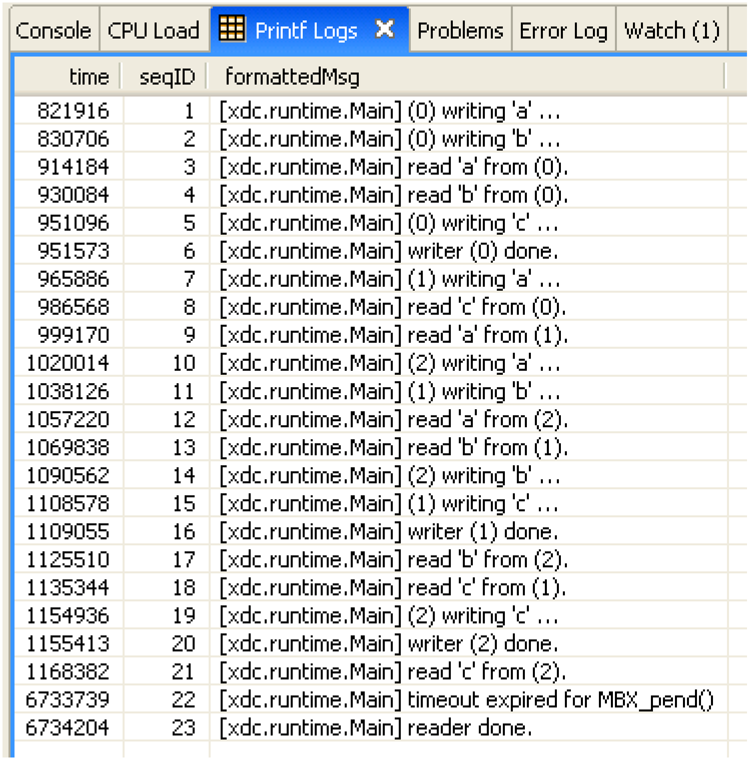
1. Make sure that the "mailbox" project is the active project. Then, from within the **C/C++** perspective, select **Project → Build Active Project** to build the mailbox application using SYS/BIOS 6.

Note for C28x: As described in Section 2.5, SYS/BIOS 6 uses a larger stack than DSP/BIOS 5, as well as having a slightly larger footprint. As a result, it is possible that not all code and data from a legacy DSP/BIOS 5 program will fit into the same memory map with SYS/BIOS 6. If your legacy program places a lot of code or data in internal memory, you may see link time errors when building your converted program.

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 CCSv4 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 SYS/BIOS Tasks and Communication to a Legacy Application

Once you have updated an DSP/BIOS 5 legacy application to build and run using SYS/BIOS 6 legacy support, you may want to expand upon that program, making use of some of the new SYS/BIOS 6 modules and code. This section shows how to add two new SYS/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 SYS/BIOS 6 Tasks, which communicate with each other via Mailbox. Section 9.2 shows how the legacy TSKs can communicate with the SYS/BIOS 6 Tasks, and vice versa.

Since this section expands on Section 4.2, “Extending Your Legacy Program with SYS/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 API, see Section 8.2.5.

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

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

The two new SYS/BIOS 6 Tasks being added are considered “new code.” As such, there are two new functions written for them, both of which use SYS/BIOS 6 APIs. Since they will communicate with one another, they will use a SYS/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 SYS/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.knl.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 SYS/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/knl/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 SYS/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 SYS/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 SYS/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 Conclusion

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

11 References

- *SYS/BIOS 6 User's Guide* (SPRUEX3 -- BIOS_INSTALL_DIR/docs/Bios_User_Guide.pdf)
- *XDCtools Release Notes* (XDC_INSTALL_DIR/xdctools_#_##_release_notes.html). Includes information about changes in each version, known issues, validation, and device support.
- *SYS/BIOS 6 Release Notes* (BIOS_INSTALL_DIR/bios_#_##_release_notes.html). Includes information about changes in each version, known issues, validation, and device support.
- *SYS/BIOS Getting Started Guide* (BIOS_INSTALL_DIR/docs/Bios_Getting_Started_Guide.pdf). Includes steps for installing and validating the installation.
- RTSC-pedia at http://rtsc.eclipse.org/docs-tip for more about RTSC and XDCtools and for demos that use RTSC-related features in CCSv4.
- RTSC+Eclipse QuickStart at http://rtsc.eclipse.org/docs-tip/RTSC+Eclipse_QuickStart for information about using RTSC in CCSv4.

Appendices

A Unsupported DSP/BIOS 5 APIs

The following table lists DSP/BIOS 5 APIs for which legacy support is currently not provided in SYS/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 SYS/BIOS 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

Module	API	Future Support?
RTDX	RTDX_CreateInputChannel	No
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 and Methods

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

B.1 Tconf prog.decl() and prog.extern() Methods

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");
```

B.2 CLK Module Properties

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

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

B.3 GBL Module Properties

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

B.4 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.5 IDL Module Properties

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

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

B.6 LOG Module Properties

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

B.7 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.8 PRD Module Properties

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

B.9 SWI Module Properties

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

B.10 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, SYS/BIOS 6.10, and SYS/BIOS 6.10 legacy support. This benchmark data is meant to help you more easily compare DSP/BIOS 5 and SYS/BIOS 6.

For any given benchmark, the result for the version that has the best score is in **bold** for clarity. If the best result is a "tie" between 2 or more 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 SYS/BIOS 6 benchmarks, please refer to:

- *SYS/BIOS 6 User's Guide* (SPRUEX3) accessible via the SYS/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 SYS/BIOS 6 benchmarks, the timer has a margin of error of +/- 8 CPU cycles. So, when comparing benchmark numbers between SYS/BIOS 6 and SYS/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 SYS/BIOS 6 and SYS/BIOS 6 legacy that differ by 8.

Benchmark	SYS/BIOS 6 APIs (CPU cycles)	SYS/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**
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

Benchmark	SYS/BIOS 6 APIs (CPU cycles)	SYS/BIOS 6 legacy APIs (CPU cycles)	DSP/BIOS 5.32 (CPU cycles)
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 Application

	SYS/BIOS 6	SYS/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

	SYS/BIOS 6	SYS/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

	SYS/BIOS 6	SYS/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

	SYS/BIOS 6	**SYS/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

	SYS/BIOS 6	**SYS/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**