Reference Frameworks for eXpressDSP Software:
RF5, An Extensive, High-Density System

Todd Mullanix, Davor Magdic, Vincent Wan,
Bruce Lee, Brian Cruickshank, Alan Campbell,
Yvonne DeGraw (technical writer)

ABSTRACT

Reference Frameworks for eXpressDSP Software are provided as starterware for
developing applications that use DSP/BIOS and the TMS320 DSP Algorithm Standard
(also known as XDAIS). Developers first select the Reference Framework Level that best
approximates their system and its future needs. Developers then adapt the framework
and populate it with eXpressDSP-compliant algorithms. Since common elements such as
memory management, device drivers, and channel encapsulation are pre-configured in
the frameworks, developers can focus on their system’s unique needs and achieve better
overall productivity.

The reference frameworks contain design-ready, reusable, C language source code for
TMS320 ‘C5000 and ‘C6000 DSPs. Developers can build on top of the framework,
confident that the underlying pieces are robust and appropriate for the characteristics
of the target application.

Reference Framework Level 5 (RF5) is intended for use in high-density applications with many
channels and algorithms. It is designed to support 1-100+ channels and many eXpressDSP-
compliant algorithms. Its total memory footprint is larger than that of lower-numbered
frameworks, but is somewhat reduced through the use of static techniques.

This application note first provides an overview of Reference Frameworks. It then explains
how to install, run, and explore Reference Framework Level 5. Other application notes are
provided to show how RF5 can be adapted to specific domain spaces, such as video and
telephony.

Code Composer Studio, DSP/BIOS, eXpressDSP, and TMS320 are among the trademarks of Texas
Instruments. See www.ti.com for a list of trademarks and registered trademarks belonging to Texas
Instruments.

Contents

1 Overview of Reference Frameworks for eXpressDSP Software ................................................. 4
  1.1 Reference Framework Target Levels ...................................................................................... 4
  1.2 Reference Framework Architecture ...................................................................................... 5
  1.3 Adapting the Reference Frameworks .................................................................................... 7
  1.4 Bill of Materials ..................................................................................................................... 8
2 Why Use Reference Framework Level 5? ................................................................................... 8
  2.1 Characteristics of RF5 Applications .................................................................................... 9
  2.2 Key Features of RF5 ............................................................................................................. 10
Figures

Figure 1. Reference Frameworks for eXpressDSP Software and Entry Points ........................................ 6
Figure 2. Video Surveillance System ...................................................................................................... 11
Figure 3. Application Processing Flow ................................................................................................. 15
Figure 4. Processing Elements in RF5 .................................................................................................. 15
Figure 5. Communication Between a Task and a Device Driver via an SIO Object ................................ 17
Figure 6. Communication Between Two Tasks via SCOM Messages .................................................. 17
Figure 7. Communication Between Cells via ICC Objects .................................................................. 18
Figure 8. Example of Task Receiving Both SCOM Messages and Control Messages ....................... 19
Figure 9. Complete RF5 Data Path ..................................................................................................... 20
Figure 10. RF5 Folder Structure ........................................................................................................... 22
Figure 11. Topology of Modules in RF5 ............................................................................................... 24
Figure 12. RF5 DSP/BIOS Configuration ............................................................................................. 30
Figure 13. Execution Graph Activity of RF5 Tasks .............................................................................. 31
Figure 14. Threads (Tasks) in RF5 ....................................................................................................... 32
Figure 15. Data Reordering by thrRxSplitRun() .................................................................................. 33
Figure 16. Effects of Priming in RF5 ................................................................................................... 34
Figure 17. The Process Thread ............................................................................................................. 35
Figure 18. Flow of Control Messages in RF5 ....................................................................................... 39
Figure 19. SCOM Usage in RF5 (Rx Side) ............................................................................................ 41
Figure 20. SCOM Function Calling Sequence ...................................................................................... 43
Figure 21. Tasks, Channels, and Cells in RF5 ...................................................................................... 43
Figure 22. Channel Infrastructure Modules ......................................................................................... 44
Figure 23. CHAN Function Calling Sequence ..................................................................................... 45
Figure 24. ICC Data Flow Example ..................................................................................................... 53
Figure 25. Scratch vs. Persistent Memory Allocation ............................................................................ 54
Figure 26. SSCR Function Calling Sequence ....................................................................................... 56
Figure 27. DIO Data Streaming Path .................................................................................................... 59
Figure 28. UTL Debugging Levels ....................................................................................................... 61
Figure 29. UTL Message Example ...................................................................................................... 62
Figure 30. Folders Containing Configuration Scripts ............................................................................ 66
Figure 31. Waiting on SCOM Messages at Different Rates on Different Queues ................................. 68
Figure 32. Waiting on SCOM Messages at Different Rates on a Single Queue ................................... 69
Figure 33. Graph of CPU Load Percentage for Various Settings ......................................................... 76

Tables

Table 1. Reference Framework Characteristics by Level ........................................................................... 5
Table 2. RF5 Application Characteristics ............................................................................................... 9
Table 3. Architecture Components ........................................................................................................ 24
Table 4. SCOM Communication Sequence Example ............................................................................ 41
Table 5. Algorithm Instantiation Module Characteristics ...................................................................... 47
Table 6. RF5 CPU Usage Statistics ......................................................................................................... 70
Table 7. RF5 Memory Footprint ............................................................................................................ 70
Table 8. RF5 Application Footprint ........................................................................................................ 73
Table 9. Calculating RF5 Basic Framework Size .................................................................................... 74
Table 10. CPU Load Percentages at Various Frame Sizes and Sample Rates ....................................... 76
Table 11. Boards Supported by Various Reference Frameworks .......................................................... 83
1 Overview of Reference Frameworks for eXpressDSP Software

In 1999, Texas Instruments introduced several DSP software development capabilities that resulted in a dramatic improvement in the way our customers could develop software for the TMS320 family of DSPs. These key software elements are:

- **Code Composer Studio**, a highly integrated DSP development environment.
- **eXpressDSP Software Technology**, which includes the following tightly knit ingredients that empower developers to tap the full potential of TI’s DSPs:
  - **DSP/BIOS**, a highly optimized, scalable, and extensible real-time software kernel.
  - **TMS320 DSP Algorithm Standard**, also known as XDAIS, which sets rules and guidelines for algorithm developers, greatly easing the burden on system integrators.
  - **A network of third-party suppliers**, who provide hundreds of eXpressDSP-compliant algorithms and software solutions for the host development environment.

DSP/BIOS and the TMS320 DSP Algorithm Standard (also known as XDAIS) are well-established core technologies. Several hundred third-party algorithms have already passed eXpressDSP-compliance testing for the ‘C5000 and ‘C6000 platforms. These span multiple application domains, including MPEG and JPEG video codecs and telecom algorithms such as G.729, GSM, DTMF, and V.90. DSP/BIOS is pervasive throughout TI DSP solutions, bringing hardware abstraction, a robust, multi-threading kernel, and real-time analysis tools.

An eXpressDSP success story at IP Unity says, “Once you’ve created the infrastructure and have integrated the first algorithm, adding more components is remarkably easy.” While this clearly demonstrates the scalability of the core concepts, it also indicates the need for higher-level DSP infrastructure content to further speed time-to-market. By providing domain-agnostic DSP framework components, TI can allow system integrators to concentrate on application-specific solutions, rather than foundation software.

The **Reference Frameworks for eXpressDSP Software** program aims to address this by providing a starterware suite to support many types of systems. A Reference Framework (RF) is defined as:

Generic DSP starterware source code using DSP/BIOS and the TMS320 DSP Algorithm Standard. Customers can adapt the framework and populate it with eXpressDSP-compliant algorithms to achieve application-specific solutions.

1.1 Reference Framework Target Levels

Software economics and complexity have changed dramatically since CCStudio, DSP/BIOS, and the TMS320 DSP Algorithm Standard were first conceived. Code sizes are usually larger. Software from many different vendors is typically integrated.

Yet, the classic, constrained, embedded DSP application is still out there—reminding us that DSP development will always value real-time performance, power, minimized code size, and cost optimization. To that end, several frameworks are required to meet such diverse needs. For example, the memory management scheme for a small, static system such as a digital hearing aid need not be as full-featured as a farm of DSPs in a telecommunications media server.
Several Reference Frameworks for eXpressDSP Software (RFs) will be produced. These frameworks will range in complexity from RF1, which is aimed at designers trying to produce extremely compact, consumer systems, to levels 5-10 with multiple algorithms, many channels, and different execution rates. The key for developers will be to pick the Reference Framework that best approximates a system and its future needs.

Table 1 compares the characteristics of Reference Frameworks available as of the release of this document.

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>RF1</th>
<th>RF3</th>
<th>RF5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute minimum memory footprint</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Static configuration</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dynamic object creation (e.g., DSP/BIOS objects)</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Static memory management</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Number of channels recommended</td>
<td>1-3+</td>
<td>1-10+</td>
<td>1-100+</td>
</tr>
<tr>
<td>Number of eXpressDSP-compliant algorithms recommended</td>
<td>1-3+</td>
<td>1-10+</td>
<td>1-100+</td>
</tr>
<tr>
<td>Uses DSP/BIOS real-time analysis</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Uses DSP/BIOS real-time analysis</td>
<td>In stop-mode only. RTDX not configured by default.</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Uses DSP/BIOS scheduling kernel</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Uses Chip Support Library</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Uses XDAIS algorithms</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Portable to other devices, ISAs, boards</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Supports multiple execution rates and priorities</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Supports thread blocking</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Implements control functionality</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Implements DSP-GPP functionality</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

All Reference Frameworks are application-agnostic. Each framework can be used for many applications, including telecommunication, audio, video, and more.

1.2 Reference Framework Architecture

In effect, a Reference Framework is an *application blueprint*. Memory management policies, thread models, and channel encapsulations are common framework elements developers construct today. By relegating these elements to the blueprint, developers can focus on their system’s needs. Developers starting new designs can build on top of the framework, confident that the underlying pieces are robust and fit the characteristics of the target application.
Reference Frameworks are not simply demonstration tools. Instead, they contain production-worthy, reusable, eXpressDSP C language source code for TMS320 'C5000 and 'C6000 DSPs. Some framework elements should be treated as binary libraries, although source code is provided for all levels. For example, the memory management in RF3 and RF5 is optimized for systems that fit that level’s description—few, if any, modifications should be required. Conversely, there are natural adaptation entry points, such as replacing the TI algorithms provided with the frameworks (VOL_TI and FIR_TI) with real intellectual property.

Figure 1. Reference Frameworks for eXpressDSP Software and Entry Points

This figure shows such example framework starterware elements as memory management and overlay policies, channel abstraction, and algorithm DMA managers. In any given Reference Framework, only the modules that suit the particular level are included. For example, it makes no sense to bundle algorithms into a channel abstraction in an RF1 application, since it targets systems with only a few algorithms and/or channels.
The architecture is analogous to that of a house. DSP/BIOS and the Chip Support Library represent the strong foundations. On top of these, the builder creates the structure of the house, laying out the rooms, fitting electricity, plumbing and other supplies, before crafting the receptacles for the owner’s appliances. This is analogous to a Reference Framework defining the application blueprint. Plugging into the receptacles are the XDAIS algorithms, which may either fit directly or require only minor modifications to the container for housing.

The malleability of the framework is analogous to adding an extra room with a few fittings such as a loft or workshop. Customers are encouraged to build on top of the framework to create application-specific solutions.

1.3 Adapting the Reference Frameworks

The most important requirement of the Reference Frameworks is that they must be relatively easy to port to customer hardware.

Each framework is packaged as a complete application on one or more Texas Instruments DSP Starter Kits (DSK) or other boards. The boards for which frameworks are supplied may be different across framework levels. For example, it makes most sense to supply the compact, static, minimal footprint RF1 on a ‘C5402 DSK, and the multi-channel, multi-algorithm frameworks of RF5 on a high-end ‘C6416 TEB. However, all Reference Frameworks will be continuously re-evaluated for porting as new DSKs reach the market.

As a rule, framework source code will be supplied in C to enable switching between the ‘C5000 and ‘C6000 Instruction Set Architectures (ISAs). Note that this has little impact on system performance since the majority of CPU cycles are typically spent in the XDAIS algorithms, which may be handcrafted in optimized assembler. The DSP/BIOS kernel is also coded largely in assembly language to minimize scheduling latencies and provide maximum performance.

Three main elements require software modifications to adapt a Reference Framework to customer hardware, as we describe in later sections:

- **Switching to other algorithms and changing the number of channels**
  This is where the application takes shape. This application note focuses primarily on this type of modification. By making this step straightforward, Reference Frameworks greatly reduce time-to-market.

- **Modifying the application to make it system-specific**
  All Reference Frameworks can be used for many applications, including telecommunication, audio, video, and more. Modifying the supplied framework application ranges from trivial to major, as the system developer sees fit. Even when major additions are made, Reference Frameworks are invaluable as foundation software.

- **Changing the driver(s) to run on end-system hardware**
  Changing the supplied driver is also an easier task as compared to some years ago. All framework drivers follow the conventions of the new DSP/BIOS I/O Driver Model detailed in the *DSP/BIOS Driver Developer’s Guide* (SPRU616). Standard conventions for hardware drivers make integration and adaptation easier in much the same way that XDAIS simplifies algorithm integration.
1.4 Bill of Materials

All Reference Frameworks will have a common "look and feel." The intention is to reduce the learning curve for customers who use more than one framework to construct several systems of varying complexity. Consistent software engineering practices have been adopted in naming conventions and style, enabling customers to quickly assess different framework levels.

Furthermore, a common Bill of Materials (BOM) is provided with each Reference Framework. At a minimum, the following items can be expected:

- Production-worthy, reusable, eXpressDSP C language source code.
- A complete, "generic" application using the Reference Framework running on a Texas Instruments DSK or TEB.
- Clear selection criteria to determine if a particular framework meets your system needs.
- A footprint budget. This enables the system integrator to quickly determine whether or not the algorithm IP and framework will fit in the chosen TMS320 DSP’s memory space.
- An instruction cycles budget. In this case customers can, for example, evaluate the number of channels that can be executed.
- Adaptation instructions detailing how to build on top of a Reference Framework for your application.

2 Why Use Reference Framework Level 5?

Reference Framework Level 5 (RF5) is intended to enable designers to create extensive applications that use numerous algorithms or channels.

In contrast to lower Reference Framework levels, RF5 uses blocking threads (tasks). As a result, RF5 can be used in applications that have complex interdependencies between threads. RF5 infrequently uses dynamic object creation, however adaptations that dynamically create threads are not precluded by design.

RF5 uses the term "cell" to describe an application wrapper for an algorithm. An RF5 "channel" can contain multiple cells, and hence multiple algorithms. RF5 provides modules to allow applications to create and control cells and channels. Additional modules support communication, synchronization, and XDAIS scratch memory sharing between cells. The algorithms used in RF5 cells are eXpressDSP-compliant algorithms, which are easily integrated.

By default, RF5 converts an incoming stereo audio signal to digital data. It processes both channels independently by applying separate filters to both channels (the user can control the filters at run-time independently for each channel). It then applies a single volume control setting to both channels. The volume setting can be controlled at run-time through a slider on the PC. Finally, it sends the output to the output codec.
Although the default application is a simple audio application, RF5 is well suited to video and other applications. The default behavior simply provides a basis for adapting RF5 to other applications. The application logic can easily be modified.

2.1 Characteristics of RF5 Applications

Typical applications that are suited for use with RF5 have extensive channel and algorithm requirements. Typically, control capability is also needed. Although memory usage is always an issue for DSP applications, applications suited to RF5 typically use DSPs with more memory than low-end DSPs.

RF5 leverages foundation software, such as DSP/BIOS, the Chip Support Library (CSL), and XDAIS. It also calls on the services of various Reference Framework modules. For example, the UTL module is used for debugging and diagnostics. Developers can either build upon the current framework or port their applications to a higher-level framework with relative ease.

Table 2 shows the characteristics of RF5 in more detail than Table 1. It should be used as a guide to determine whether or not RF5 is suitable as the basis of your end-application.

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>RF5</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static configuration</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Dynamic object creation (e.g., DSP/BIOS objects)</td>
<td>✔</td>
<td>Although the default RF5 application creates most objects statically, the framework does not preclude the dynamic creation of objects. Static configuration is preferred for simplicity and footprint reduction.</td>
</tr>
<tr>
<td>Static memory management</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Number of channels recommended</td>
<td>1-100+</td>
<td>For 1-10 channels, consider using RF3.</td>
</tr>
<tr>
<td>Number of eXpressDSP-compliant algorithms recommended</td>
<td>1-100+</td>
<td>For 1-10 algorithms, consider using RF3.</td>
</tr>
<tr>
<td>Absolute minimum memory footprint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uses DSP/BIOS real-time analysis</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Uses DSP/BIOS scheduling kernel</td>
<td>✔</td>
<td>Uses primarily TSK and HWI threads. Use of SWI threads is not precluded.</td>
</tr>
<tr>
<td>Uses Chip Support Library</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Uses XDAIS algorithms</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Portable to other devices, ISAs, boards</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Supports multiple execution rates and priorities</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Supports thread blocking</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Implements control functionality</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Implements DSP-GPP functionality</td>
<td>✔</td>
<td>Will be the key feature of RF6.</td>
</tr>
</tbody>
</table>

If your application does not match the characteristics suited to RF5, you should use a different Reference Framework. Since RF5 is the "extensive" framework, its capabilities are beyond the needs of many applications. You may want to consider using RF3 as the basis for applications with only 1 to 10 channels and algorithms and less complicated thread interaction needs.
Note, however, that RF5 still works perfectly well for even simple systems with one algorithm and one channel. In fact, the tasking and data streaming paradigms are similar to that of many general-purpose processor operating systems. As a result, the framework provides value with a familiar look and feel. The only point to recognize is that RF5 may be overkill for small systems.

### 2.2 Key Features of RF5

RF5 provides the following key features:

- **Provides a scalable channel manager.** The CHAN manager makes the application scalable to large numbers of XDAIS algorithms while minimizing the need for a large number of TSKs in the system. Algorithms can efficiently share scratch data memory and can easily be replaced via the ICELL interface.

- **TSK-based application.** In contrast, RF3 is SWI-based. The TSK module provides more scheduling flexibility than the SWI module, but carries with it more performance and memory overhead.

- **Efficient inter-task communication.** A module called SCOM is provided for simple single-direction, zero-copy data passing among tasks. Think of SCOM as a token-passer. The task that has the token can access the buffer freely.

- **Structured thread-safe control mechanism.** The Control thread interacts with the outside world, for example by reading on-board settings that can change during the course of the application, and sends control messages to other threads.

- **Easy replacement of I/O drivers.** The IOM model allows alternate mini-drivers to be connected to the DIO adapter used by RF5.

- **Allows easy debugging.** The well-defined structure of RF5 allows for fast debugging. In addition, the UTL module allows debugging levels to be changed rapidly.
2.3 Applications Suited to RF5

All Reference Frameworks are application-agnostic. Each framework can be adapted for use in many applications, including telecommunication, audio, video, and more. Figure 2 shows an example application to which RF5 is suited.

Other applications for which RF5 can be adapted include:

- 3G wireless infrastructure devices
- Video infrastructure devices, for example, security applications
- Interactive TV server
- Universal Port Switch

Additional application notes will be provided to show how RF5 can be adapted to specific application types, such as video.
3 Installing and Running the RF5 Base Application

This section describes how to build and run Reference Framework Level 5 (RF5) as it is. Later sections describe how the application works and how it can be adapted.

3.1 Preparing the Hardware

The following steps provide an overview of how to connect hardware to your host PC in order to run the default RF5 application. Although the default application uses audio inputs and outputs, RF5 is well suited for use with video or other types of signals.

For details and diagrams about board-specific steps, see the documentation provided with your board. For additional board-specific information, see the readme.txt file in the RF_DIR\driver folder for your board (for example, RF_DIR\src\teb6416pcm3002).

**NOTE**: The top-level folder of the Reference Frameworks distribution is called "referenceframeworks". The full path to this folder is called RF_DIR in this application note.

1. Shut down and power off your PC.
2. Connect the appropriate data connection cable to the board.
3. Connect the other end of the data connection cable to the appropriate port on your PC.
4. Connect an audio input device such as a microphone or the headphone output of a CD player to the audio input jack (or jacks) on the board. You can also connect the audio output of your PC sound card to the audio input of the board.
5. Connect a speaker (or speakers) or other audio output device(s) to the audio output port(s) of the board.
6. Plug the power cable into the board.
7. Plug the other end of the power cable into a power outlet.
8. Start the PC.

3.2 Preparing the Software

The following list outlines the software installation and setup steps required to run RF5. For details, see the appropriate Quick Start Guides, online help, or the readme.txt file.

1. If you have not already done so, install CCStudio 2.2 or a later version. It is recommended that you have the latest version of the CCStudio software, as it may contain important features or problem fixes.
2. Check the configuration of your parallel printer (LPT) port. Make sure the parallel port is in ECP or EPP mode and note the first address of the port. This is normally 0x378. For details on checking the parallel port configuration, see the Quick Start Guide provided with your board. If you are using an emulator such as TI XDS560, connect the cable to the 14-pin JTAG header on your board and follow the instructions the emulator vendor has provided.
3. Use the Setup Code Composer Studio application to configure the software for your board. For details, see the documentation provided with your board.

4. Download the Reference Frameworks code distribution file from the Reference Frameworks area of the DSPvillage website (www.dspvillage.com). Place this file in any location, and unzip the file. The c:\ti\myprojects folder is a suggested location for these files.

Make sure to use directory names when you unzip the file. You may need to enable an option called something similar to "Use folder names" in your zip utility.

Do not extract the zip file into a directory with a path that contains spaces such as c:\Program Files. Spaces in directory paths are not currently supported by the TI Code Generation Tools.

3.3 Building and Running the RF5 Application

The targets supported for RF5 as of the publication date are listed in Appendix E: Reference Framework Board Ports, page 83. Additional boards may be added in the future. If you want to run RF5 on a different target, you need to port hardware-dependent parts of the application.

After installing the package, you are ready to build and run the RF5 application.

1. Verify that CCStudio is using the correct startup GEL file for your board. For example, for a 'C6416 TEB board, the startup GEL file should be c:\ti\cc\gel\TEB6416.gel.

2. Within CCStudio, choose Project→Open and select the app.pjt project in RF_DIR\apps\rf5\projects\target, where target matches your board. (For example, RF_DIR\apps\rf5\projects\teb6416.)

3. Choose Project→Build to build the RF5 application. Alternatively, you can choose to just load the pre-built application located in RF_DIR\apps\rf5\projects\teb6416\Debug folder.

NOTE: If you have recently upgraded to a newer version of CCStudio, you are encouraged to run the RF_DIR\build.bat from an MS-DOS window command line. This simple batch file rebuilds all RF projects. It ensures that all modules are in sync with the latest TI Code Generation Tools. Note that you must first run c:\ti\dosrun.bat so that the paths in the build.bat file are recognized.

4. Choose File→Load Program and load the app.out file in the Debug folder.

5. Start your CD player or other audio input.

6. Choose Debug→Run (or F5). You should hear the FIR filtered audio output through the speakers connected to the target board.

NOTE: The 'C64x device on the 'C6416 TEB board does not support RTDX, which is used for real-time target to host data transfer. As a result, instrumentation data provided by DSP/BIOS is available in stop-mode only. That is, they can be seen in Code Composer Studio only if the program halts or reaches a break point.
7. Choose File→Load GEL and select the appControl.gel file from the project folder (above the Debug folder).

**NOTE:** The appControl.gel file is a GEL script file that displays sliders to control algorithm parameters. When you move these controls, the script writes values to program variables on the target.

8. Choose GEL→Process Control→Volume. A slider appears with values ranging from 0 to 200. The default value is 100. Sliding this controls the output volume on a stereo codec.

**NOTE:** On the 'C6416 TEB, the output is stereo even if the GEL control is modified to allow the volume balance to be set to the extreme left or right (effectively mono output) and the codec is configured for internal loop-back (that is, data does not go to the MCBSP). This observation was made on two separate 'C6416 TEB boards, making it unlikely that the only some boards show this problem. No workaround has been found for this problem; this is probably a limitation arising from board-specific hardware circuitry.

9. Choose GEL→Process Control→Filter1 (or Filter2). A slider appears with values ranging from 0 to 2. The default value is 1. Sliding this controls the FIR filter coefficients.

4 RF5 Overview

This section provides overviews of several aspects of RF5. Included are an overview of the application requirements, application building blocks, the folder hierarchy, the module hierarchy, and the application parameters and function call hierarchy.

4.1 Application Behavior

The application specifications for the base RF5 application are as follows. Although the default application uses audio inputs and outputs, RF5 is well suited for use with video or other types of signals. Application notes that describe how RF5 can be adapted to other application domains are listed in Section 13, References, page 71.

The application takes the incoming stereo audio signal and converts it to digital data at a given sampling rate. One sampling of the signal gives a block of two signed 16-bit integers, one for the left and one for the right channel. The application groups these blocks into frames of given size before processing them.

For processing, the application splits each incoming interleaved stereo frame into two single-channel frames. One frame contains only left-channel samples; the other contains only right-channel samples. The application processes these frames separately. To process one channel frame, the application applies a FIR filter and a volume control algorithm to it. Filter coefficients (low-pass, high-pass, or passthrough) and amplification/attenuation values may be controlled by the user via GEL, CCStudio's scripting language for accessing target memory, with a simulated slider that writes values to designated variables that the target reads periodically.
After it processes each channel, the application joins the independent channel frames back into one interleaved stereo frame. This frame is then sent to the output where the codec converts it into analog stereo signal.

![Diagram showing application processing flow]

**Figure 3. Application Processing Flow**

The number of channels in the application is a modifiable constant, which allows the application to be easily scaled to a large number of channels. The number of channels in the codec (1=mono, 2=stereo) is also a modifiable constant.

The supplied FIR XDAIS algorithm is used for filtering and the supplied VOL XDAIS algorithm is used for volume control.

### 4.2 Application Building Blocks and Structure

Let us look briefly at what data processing elements and what data communication elements we have in RF5, and how we pass control messages. Then we will see how they all fit together in the RF5 data path.

#### 4.2.1 Data Processing Elements in RF5

The four basic data processing elements in RF5 are tasks, channels, cells, and XDAIS algorithms.

![Diagram showing processing elements in RF5]

**Figure 4. Processing Elements in RF5**

At the top level is a DSP/BIOS task. A task is a collection of channels, a channel is a collection of cells, and a cell is a wrapper for a XDAIS algorithm. Each of these elements can have multiple instances. For example, there are two instances of the filtering algorithm we use, one for each channel. They use different filtering parameters and remember different history. The data describing each instance object is different, but both instances share the code that operates on the data. The same is true for cells and channels (but rarely tasks). Typically, when we talk about an element, we actually refer to an instance of that element.
A XDAIS algorithm is an off-the-shelf, reusable data processing component, that implements a certain interface (IALG). Typically it implements, via the interface, a fairly complex function, for example JPEG encoding or audio enhancement; but it can be as simple as audio signal amplification, which is the VOL algorithm that comes with RF5. XDAIS algorithms in your application can be purchased from a third party, or you can incorporate your own custom ones.

A cell is a wrapper around a XDAIS algorithm. XDAIS algorithms have standardized resource management functions (for requesting memory and DMA). However, the actual data processing function, which lies at the heart of the algorithm, has no standard naming convention. The processing functions can vary not only in function signatures (some take two or more buffers, etc.), but often also in their number. The purpose of a cell is to provide a standard interface between the algorithm and the outside world, by defining only one processing function. Each cell implements a simple ICELL interface, which defines up to four functions for a cell: open, execute, close, and control. All functions other than execute are optional.

Most cell wrappers are simple, though they can perform whatever operations are necessary in addition to what the algorithm does. The application integrator writes the cell code, (or modifies code of existing cells for their algorithms). In the future, some XDAIS algorithms vendors may supply cell wrappers as well. It is also possible to create a cell that does not contain a XDAIS algorithm.

A channel is a collection of cells, and its purpose is to execute its cells in series. Channels always perform a fixed operation—executing cells serially—so they do not need any additional code to be written. Typically several channels contain sets of cell instances that perform identical functions, possibly with different parameters.

Finally, a task is collection of channels, which executes them in series. The purpose of the task is to organize data communication at a higher level, that is by talking to device drivers, other tasks, and similar constructs. Unlike channels, tasks do have task-specific code, which the user writes. This code is usually just sending and receiving data to and from the outside world, and executing channels. A task has freedom to execute channels in whatever way it desires, which may be dictated by data flow and control information. A task can also have no channels at all.

One important feature of tasks is that they can occasionally send control messages to one another (in addition to streaming data they regularly send). For that reason, each task that runs get-data/execute-channels/send-data iterations, checks for the presence of control messages at the beginning of each iteration. If there are any, the task applies them to perhaps change its control logic, or to control the cells contained in its channels.

The term task refers to an actual DSP/BIOS object, and the term thread refers to user code that the task executes and pertinent data. We use terms task and thread interchangeably, since the supplied RF5 application uses tasks but not software interrupts (SWIs). It also uses hardware interrupts (HWIs) for high-priority event processing triggered by peripherals.

4.2.2 Data Communication Elements in RF5

We divide the elements for passing data between processing elements into task-level data communication elements and cell-level data communication elements. Rather than just using global variables to inform processing elements where the data is, we introduce structured objects for passing that information.
For task-level communication, which uses semaphore-based synchronization, we have SIO objects and SCOM messages.

- **SIO objects** interface with device drivers and tasks. These are standard DSP/BIOS objects, and are typically double-buffered:

![Diagram](image)

Figure 5. Communication Between a Task and a Device Driver via an SIO Object

Typically the task allocates two buffers for the data, and passes empty buffers to the input device driver and collects buffers full of data from it (and the other way round for output). Note that in some applications, a task may talk to a device driver without using SIO objects; instead it uses whatever format the driver prescribes.

- **SCOM messages** are user-defined, structured data objects that tasks exchange among themselves. SCOM stands for Synchronized Communication. Tasks allocate buffers that they want some other task to write data to or read data from. They need to communicate to the other task where the buffer is, but also to synchronize with it to prevent simultaneous access. To do that, they use SCOM messages as buffer descriptors, which tasks pass among themselves. In that sense, an SCOM message is like a token for a buffer it describes: the task that holds the message—the token—can read from the buffer or write to it exclusively. When finished, the writer passes the message along to the reader task and vice versa.

![Diagram](image)

Figure 6. Communication Between Two Tasks via SCOM Messages

Each task creates its own receiving SCOM queue (or more than one if necessary), and puts SCOM messages to other tasks’ receiving queues. A task only needs to know the name (a character string) of a queue it wants to put messages on. More than one task can send SCOM messages to the same queue.

For cell-level communication, we have inter-cell communication (ICC) objects and lists of those objects. The purpose of an ICC object is to describe the buffer from which a cell reads the data, or to which the cell writes the data. Each cell has one input list and one output list of those objects. Two cells in effect communicate by having the same ICC object in their lists: the cell that writes to a buffer described by an ICC object has the object in its output list, and the cell that reads the buffer has the object in its input list.
This allows for an arbitrary, and flexible, topology between cells in a channel. In fact, cells within different channels (but still in the same task) can communicate this way. More importantly, a task that receives data from another task can pass the data to its cells using ICCs, "unwrapping" the data first if necessary.

We see this in Figure 7, which shows a task's channel with three cells and various input/output and intermediate buffers:

![Figure 7. Communication Between Cells via ICC Objects](image)

Cell 1 reads its data from the task (the tasks writes into the buffer described by Cell 1's input ICC object), Cell 1 stores its output in two buffers, one read by cell 2 and one read by cell 3. Cell 3 also reads cell 2's output, and cell 3's output is finally read by the task.

ICC objects come in different, and user-definable, flavors. The simplest ICC is the one that describes a plain buffer, in terms of its address and size. This is called a linear ICC buffer. For example, in Figure 7, cell 1's input ICC would likely point to the same buffer the tasks uses for receiving data via SIO or SCOM.

The user is free to define their own ICC types, which can have pre-processing or post-processing operations the cell can call before or after accessing data described by the buffer.

### 4.2.3 Application Control in RF5

We have noted earlier that tasks can pass control messages among themselves. In baseline RF5, there is only one task that passes messages around, the control task, and only to one other task, the task that contains the FIR+VOL processing channels.

As the specifications require, the user can control the application via a GEL script that has one slider for each channel to control filter coefficients (selection among low-pass, high-pass, and passthrough filter), and one slider for volume control. For simplicity one slider controls both channels, although since the channels operate independently, the GEL script could have one slider for each channel's volume.
The GEL script follows user's actions and writes new filter/volume values in a global variable on the target. The control task periodically reads this variable, and when it detects a change, it sends a control message to the processing task. Tasks use the DSP/BIOS MBX (mailbox) module for control messages. The MBX module, unlike SCOM, makes a copy of the sender's message before placing it on the mailbox queue, and it makes a copy of the message in queue when delivering it to the recipient. For that reason it is more convenient for small asynchronous messages—as is the case with control messages. Each control message is a simple set of three 32-bit values: command, first command argument, second command argument.

In RF5, the MBX object is created dynamically in the thrProcessInit() function because the call to MBX_create can obtain and use the message size and mailbox length when creating the object.

Figure 8 shows a task that has both one SCOM queue for receiving data messages and one dynamically-created MBX object for receiving control messages. Unlike SCOM queues, a task should only have one mailbox.

![Figure 8. Example of Task Receiving Both SCOM Messages and Control Messages](image)
4.2.4 RF5 Data Path

Figure 9 shows the specific instances of various processing and communication components, connected together to form the application that is RF5. Figure 14 provides a somewhat simplified picture of the data path.

All the components of the data path are explained in separate sections, but here is a brief overview of the processing dynamics:

The data path begins with the device driver streaming the data into the RxSplit thread’s double buffer bufRx. The low-level IOM mini-driver, programs the serial port and the DMA (or EDMA on platforms that have it) to transfer and group the incoming samples into frames. The RxSplit thread does not communicate with the driver directly; instead, RxSplit owns the inStream SIO object, which interfaces to a mini-driver via another small module, DIO. Through SIO, RxSplit tells the driver where it wants the data (in one or the other half of its private buffer bufRx), and when it wants it. Because of double buffering, the mini-driver can keep pumping the data into one half of bufRx at the same time when RxSplit works to process the other half to prepare data for the Process thread.

**Figure 9. Complete RF5 Data Path**

All the components of the data path are explained in separate sections, but here is a brief overview of the processing dynamics:

The data path begins with the device driver streaming the data into the RxSplit thread’s double buffer bufRx. The low-level IOM mini-driver, programs the serial port and the DMA (or EDMA on platforms that have it) to transfer and group the incoming samples into frames. The RxSplit thread does not communicate with the driver directly; instead, RxSplit owns the inStream SIO object, which interfaces to a mini-driver via another small module, DIO. Through SIO, RxSplit tells the driver where it wants the data (in one or the other half of its private buffer bufRx), and when it wants it. Because of double buffering, the mini-driver can keep pumping the data into one half of bufRx at the same time when RxSplit works to process the other half to prepare data for the Process thread.
When it receives input frames via SIO, RxSplit then collects the scomMsgRx SCOM message from its scomRxSplit queue, block-waiting if the message is not there. The message describes two buffers owned by the processing task. To those buffers RxSplit writes separated channel data. Afterwards, it puts the scomMsgRx message back on the Process thread's receiving SCOM queue.

When the Process thread collects scomMsgRx message, that message is a signal to the Process thread that the new input frame has been separated into channels and stored into Process' private buffers bufInput[NUMCHANNELS]. These two buffers are connected to FIR cells' input ICC objects, so the thread executes its NUMCHANNELS (2) channels. In each channel, its FIR cell reads from its respective bufInput buffer, processes the data, and stores it in the bufIntermediate buffer. The VOL cell reads from that buffer and stores its result in the bufOutput part of the buffer for that channel. Thanks to ICCs, however, the cells do not need to know actual buffer names, so they can function in any channel and task.

The architecture of RF5 is such that it is easy to change the number of channels. For each channel, you can specify the cells it contains, and thus the XDAIS algorithms it executes.

When adapting RF5 for other applications, a common adaptation is to modify or replace the tskRxSplit and tskTxJoin tasks, since they may not even be required. Another common modification is to adapt the use of the control thread to the needs of the application.

4.3 Folder Hierarchy

The Reference Framework folder tree contains application sources and library modules. All the code that different RF5-based applications could possibly share has been pushed into libraries, for reusability (although you can find source code for these libraries in the RF5 tree, too). Application specific files, such as task code and cell wrappers, are in the \apps sub-tree.

You can begin to explore RF5 by examining the folder tree that contains the application and associated files. We recommend that you retain the provided structure for your development.
Figure 10 shows the folders used by RF5 and highlights some important files they contain.

### Figure 10. RF5 Folder Structure

Folders to notice include:

- **apps\rf5.** The root folder for the RF5 application. To modify RF5, make a copy of the rf5\tree at the same folder level, and modify the copy.
  - **appConfig.** Contains DSP/BIOS TextConf scripts that are generic to all platforms. These scripts are imported by the board-specific appcfg.tcf file.
  - **cells.** Contains application-side cell implementation code. Here the system integrator adds simple "glue code" to match up the XDAIS algorithm interface(s) to the framework APIs. This provides a consistent execution interface and makes it easier to group
algorithms into channels. There is one folder for each algorithm or algorithm encode/decode pair. For example, you might create a cells\mp3 folder to house cellMp3.h, cellMp3encode.c, and cellMp3decode.c.

- **projects.** Contains hardware-specific files for the RF5 application. This includes board-specific configuration files, project, GEL, and linker files. These files are placed in platform-named folders so that RF5 can be provided for multiple platforms. Targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 83.

- **threads.** Contains hardware-independent source files for the threads (tasks).

  - **include.** Contains a number of public header files used by Reference Frameworks. RF5 uses some, but not all, of these header files.

    Public header files are referenced by both algorithm and framework code. In contrast, private header files are stored with the source code that includes them and are not intended for use by other modules. Each library module has one header file in this folder.

  - **lib.** Contains a number of library files linked in with Reference Framework applications. RF5 uses some, but not all, of these libraries. Each library module has one library per DSP family in this folder. In addition, libraries are built for each target flavor. For example, RF modules are provided for both 55x small data model (.l55) and large model (.l55l).

  - **src.** Contains folders with source files for modules in the include and lib folders. The readme.txt files in each of these folders provide information about the modules and their use. Library modules typically need little or no modification.
4.4 Module Hierarchy

RF5 uses several collections of modules. Figure 11 shows the high-level framework architecture.

![Figure 11. Topology of Modules in RF5](image)

Table 3 describes the components of this architecture diagram. API function descriptions are provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Adaptation Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Framework</td>
<td>Reference Frameworks can be adapted for use in many applications, including telecommunication, audio, video, and more. Such changes are not detailed in this application note.</td>
<td>Modification likely</td>
</tr>
<tr>
<td>Threads</td>
<td>Threads perform processing. Threads are specified via the DSP/BIOS configuration and through the code run as the function for each thread. Threads are generally hardware-independent.</td>
<td>Yes, to customize for your application.</td>
</tr>
<tr>
<td>Cell wrappers</td>
<td>Application-side cell implementation code. Contains simple &quot;glue code&quot; to match the XDAIS algorithm interface to the framework APIs.</td>
<td>Typically minimal, to switch algorithms</td>
</tr>
<tr>
<td>XDAIS algorithms</td>
<td>TI's eXpressDSP-compliant implementation of the simple algorithms used by default. Algorithms may be written by you or provided by a vendor.</td>
<td>No</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Adaptation Required</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>ALGRF</td>
<td>Functions such as ALGRF_create() for creating IALG-based instances of XDAIS algorithms. Uses DSP/BIOS MEM module for dynamic memory allocation. The functions are similar to the functions in the CCStudio’s ALG module.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>CHAN</td>
<td>Manages the serial execution of XDAIS algorithms contained in cells.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>SCOM</td>
<td>Implements synchronized message passing among tasks. In RF5, tasks use it to give and take ownership of buffers so they can write to buffers or read from them exclusively.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>ICELL</td>
<td>Interface specification for encapsulating XDAIS algorithm instances in cells.</td>
<td>Interface must be implemented for each algorithm used.</td>
</tr>
<tr>
<td>ICC</td>
<td>Provides an Inter-Cell Communication mechanism to move input and output data from a cell. Describes how data flows in and out of a cell.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>SSCR</td>
<td>Manages the overlaying of on-chip scratch data memory requested by XDAIS algorithms.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>UTL</td>
<td>Provided to support debugging and diagnostics.</td>
<td>No, but source code is provided</td>
</tr>
<tr>
<td>DIO</td>
<td>Stream device adapter. The upper layer of the device driver. Provides an interface between an IOM mini-driver and an SIO data stream object. Executes DSP/BIOS calls to manage buffer transfers. This layer is hardware-agnostic. Described in SPRU616.</td>
<td>No</td>
</tr>
<tr>
<td>IOM mini-driver</td>
<td>Simple, low-level device driver interface between application threads and hardware devices. For example, “teb6416pcm3002”. Defines a set of device-agnostic APIs to interface with. Described in SPRU616.</td>
<td>Yes, to port controller to your hardware</td>
</tr>
<tr>
<td>DSP/BIOS</td>
<td>The set of modules provided as a scalable real-time kernel. This includes modules for scheduling, memory management, instrumentation, and I/O.</td>
<td>No</td>
</tr>
<tr>
<td>CSL</td>
<td>Chip Support Library. Simplifies the job of developing device drivers and interacting with peripherals.</td>
<td>No</td>
</tr>
<tr>
<td>Hardware</td>
<td>The target board, including codecs and other peripherals.</td>
<td>—</td>
</tr>
</tbody>
</table>

### 4.4.1 Rebuilding and Debugging Libraries

Source code is provided for the additional libraries used by RF5 not only so you can modify and recompile the libraries, but for debugging purposes as well. If you halt execution while within code in a library module, CCStudio asks if you want to locate and open the appropriate source file for the module. This allows you to step into module procedures and inspect internal and external variables, even if you do not intend to modify the code.

**Hint:** In CCStudio, using Options→Customize→Directories menu option, you can specify which folders CCStudio should search to locate the source file. If you specify source code folders for the modules used in RF5, CCStudio opens windows with their source code automatically as you step into a library module procedure.

A readme.txt file is provided in each library source folder. These readme.txt files list the module files, tell which frameworks use the module, and answer questions about the module.

Libraries are built with debugging enabled (-g) and no optimization. For performance reasons you may wish to rebuild the libraries using optimization switches for post-development versions of your applications.
If you rebuild a library and then rebuild the Reference Framework application, either delete the executable file (app.out) or use Project→Rebuild All in order to build with the new library. CCStudio does not currently check for dependencies on rebuilt libraries.

The Reference Frameworks distribution does not include source code for IOM device driver modules. Such files can be obtained as part of the DSP/BIOS Driver Developer's Kit (DDK). You do not need the DDK in order to run the Reference Frameworks—the driver library and public header files are included in the Reference Frameworks distribution. For details about the DDK and mini-driver development and use, see the DSP/BIOS Driver Developer's Guide (SPRU616).

4.5 Application Parameters and Function Calls Hierarchy

When you start exploring RF5, you will best understand its flow if you look at the hierarchy of function calls in it. You can easily identify where a function or a data structure is defined by looking at its name. If a name begins with \textit{MOD\_}, it is a library module defined in module \textit{mod}—meaning that its public interface is in include\textbackslash mod.h directory, and its source is in src\textbackslash mod. If a global name does not have the prefix with the underscore, it means it's an application module, and its file of origination is still determined by the first part of the name. (For example, the \textit{thrProcessRun()} function is declared in the thrProcess.h file and defined in the thrProcess.c file.)

Global symbols use a simplified Hungarian notation (for example, the Process thread's state variable, \textit{tskProcess}, is an object of type "task," which uses the TSK module), and local names usually follow less strict naming convention.

Before we examine the function call hierarchy, let us first look at two important parameters you will see throughout the code:

4.5.1 Number of Channels and Data Frame Sizes

The number of channels in the RF5 application is specified in appResources.h:

```
#define NUMCHANNELS 2
```

For some applications, different threads may have different number of channels, so their number of channels would be defined in their header files.

In appIO.h, the following constant identifies the number of channels in the codec:

```
/* The 6416TEB has a stereo codec */
#define NUMCODECCHANS 2
```

A constant in appResources.h specifies the size of data buffers used in the application. The size of a per-channel data buffer—that is, the size after the signal has been split—is defined as:

```
#define FRAMELEN 80
typedef Short Sample; // signed 16-bit integer
```

The size in bytes of the frame for one channel is expressed with the formula:

```
FRAMELEN * sizeof( Sample )
```
Similarly, the size in bytes for all the channels is expressed as:

\[ \text{NUMCODECCHANS} \times \text{FRAMELEN} \times \text{sizeof( Sample )} \]

The heartbeat of the system comes from the codec and the CSL modules used by the controller (EDMA and MCBSP for the 'C6416). Because the controller uses EDMA, the CPU does little work to receive and transmit the frames. Therefore, the larger the frame size, the less time the CPU spends on I/O, and the more time remains for processing. Larger buffer sizes, however, result in larger latencies, and consume more memory, so a compromise must be found.

For example, the 'C6416 TEB, has a stereo codec (PCM3002) with a sampling frequency of 48 kHz. Each sample is 16 bits. The calculation for how often data is processed is as follows:

\[
\frac{1 \text{ second}}{\text{frequency (Hz)}} \times \text{FRAMELEN} = \\
\frac{1}{48000} \times 80 = \\
0.00166 \text{ s} = 1.66 \text{ ms per frame}
\]

You can measure the period it takes to process one frame by looking at stsTime0 statistic (STS) BIOS object, if you choose the units for stsTime0 to be in milliseconds or microseconds. The Process thread uses the stsTime0 object and the UTL module to measure elapsed time between two iterations.
The following diagram shows the hierarchy of the important function calls in RF5 (on the 'C6416 TEB as an example, and without instrumentation calls and most of the standard DSP/BIOS functions, for simplicity.)

```
- main() /* initialize the application */
   +---- appIOInit() /* initialize application I/O */
   |      +---- TEB6416_EDMA_PCM3002_init() /* device controller init */
   |      `---- TEB6416_EDMA_PCM3002_setup() /* device controller setup */
   +---- appThreadInit() /* initialize threads and the modules threads use */
       +---- CHAN_init() /* initialize the CHAN module */
       |      |---- ALGRF_init() /* CHAN module initializes ALGRF module */
       |      |    |---- SSCR_init() /* and SSCR module */
       |      |---- ICC_init() /* initialize the ICC module */
       |      +---- SCOM_init() /* initialize the SCOM module */
       |      +---- CHAN_setup() /* setup the CHAN module (pass heap IDs etc. */
       |      |---- ALGRF_setup() /* setup ALGRF */
       |      |    |---- SSCR_setup() /* setup SSCR */
       |      +---- thrRxSplitInit() /* initialize RxSplit thread */
       |      |---- SIO_create() /* create SIO stream */
       |      |    |---- SCOM.create() /* create receiving SCOM queue */
       |      `----- thrProcessInit() /* initialize RxSplit thread */
       |          |---- SIO_create() x2 /* create two receiving SCOM queues */
       |          |    |---- setParamsAndStartChannels() /* create and init. cell objects */
       |          |    |    |---- ICC_linearCreate() /* create ICCs for each cell */
       |          |    |    |    |---- CHAN_regCell() /* register cells in each chan. */
       |          |---- thrTxJoinInit() /* initialize RxSplit thread */
       |          |    |---- SIO_create() /* create SIO stream */
       |          |    |    |---- SCOM.create() /* create receiving SCOM queue */
       |          +---- thrControlInit() /* initialize control thread */
       |              +---- thrRxSplitStartup() /* open receiving and sending SCOM queues */
       |                  |---- SIO_issue() x2 /* prime SIO stream */
       |                  +---- thrProcessStartup() /* runs continuous loop */
       |                      |---- setParamsAndStartChannels() /* runs continuous loop */
       |                      |    |---- CHAN_open() x2 /* open each channel (create algs. */
       |                      +---- thrTxJoinStartup() /* open each channel (create algs. */
       |                          +---- SIO_issue() x2 /* prime SIO stream */
       |                              +---- thrControlStartup()

- thrRxSplitRun()
   +---- SCOM.open() x2 /* open receiving and sending SCOM queues */
      `-------/* runs continuous loop */
       +---- SIO_reclaim()
       +---- SCOM.getMsg()
       +---- SCOM.putMsg()
       +---- SIO_issue()

- thrProcessRun()
   +---- SCOM.open() x4 /* open receiving and sending SCOM queues */
      |---- SCOM.putMsg() x2
      `-------/* runs continuous loop */
          +---- checkMsg()
              |    |---- VOL_cellControl() x2 or FIR_cellControl() x2
              +---- SCOM.getMsg() x2
              +---- CHAN_execute() /* one per channel */
              |    |---- cellFxns->cellExecute() /* one per cell */
              +---- SCOM.putMsg() x2
```
5 Application Configuration and Startup

Like other conventional real-time operating systems, DSP/BIOS enables an application to dynamically create objects, such as tasks and semaphores, at any time during program execution. However, in reality, many real-time applications simply create all the necessary objects at the start of the application. This wastes program memory since the code for creating objects must be present in target memory even though it is only used once.

Instead of using APIs for dynamic creation, DSP/BIOS enables developers to statically generate a configuration tailored to the needs of the application. This significantly reduces the target memory footprint by eliminating the need to code the creation logic.

Typically, a module, consisting of a header file and a library file, implements a class of objects and lets the user create as many instances of it as needed. The concept is similar to the class concept in object-oriented programming, but the API is C-based (and the underlying implementation is often in assembly, for performance). For instance, one module implements a task, and with its code the user can create as many task instances as needed.

Information about the DSP processor and DSP/BIOS objects are stored in a configuration database (.cdb).

Traditionally, DSP/BIOS used a Configuration Tool with a graphical interface for the creation and configuration of static objects. This tool is still available as part of CCStudio 2.2, and users who may want to continue using it in their workflow can ignore the remaining contents of this section. For those who are interested in a more flexible, script-based configuration tool, DSP/BIOS TextConf (tconf) now offers an alternative for configuring your application at design-time. The main benefits of using TextConf in the Reference Frameworks are as follows:

• **Easily port frameworks to new target boards and platforms.** The configuration scripts clearly separate application-specific settings from target-specific settings. This makes it easy to port applications to new target boards and platforms. Only those settings made for target-specific reasons require modification when porting to a new board.

• **Eliminate potential update issues.** The configuration database (CDB) file used with the graphical configuration tool must be updated with every new version of CCStudio. In some cases, this conversion can be problematic. Textual configuration uses scripts as source files, and eliminates this conversion altogether. These scripts are much smaller and easier to maintain than their CDB counterparts.

Configuration scripts are written in JavaScript, a powerful scripting language that has a C-like syntax, which helps to reduce the learning curve for a typical C programmer.
Reference Frameworks provide textual configuration by including a .tcf file that can be run to create the configuration database (.cdb). This .tcf file imports both application-specific scripts (for example, appInstrument.tci) and target-specific scripts (for example, appBoard.tci). These scripts contain the JavaScript code to create and configure static objects. Typically, a target-specific script first loads a particular platform and then imports sub-scripts to set up memory sections, drivers, and other target-specific objects.

More information about the configuration scripts for RF5 is provided in Section 10.2.1, Textual Configuration Scripts, page 66. Complete documentation on textual configuration can be found in the DSP/BIOS TextConf User’s Guide (SPRU007). For information on specific properties of DSP/BIOS objects, please refer to the DSP/BIOS API manual for your DSP family.

Figure 12 shows objects configured for RF5 in the DSP/BIOS Configuration Tool.
The configuration database for RF5 includes the following objects added for use by the application:

- **TSK (task) objects**: tskRxSplit, tskTxJoin, tskProcess, tskControl
- **UDEV (user device) object**: udevCodec
- **DIO (I/O device) object**: dioCodec
- **LOG object**: logTrace
- **STS (statistics) objects**: stsTime0 through stsTime9 (used by UTL module)

**NOTE:** The 'C64x device on the 'C6416 TEB board does not support RTDX, which is used for real-time target to host data transfer. As a result, instrumentation data from LOG and STS objects is available in stop-mode only. That is, the data can be seen in Code Composer Studio only if the program halts or reaches a break point.

The configuration sets properties that specify whether a module is needed in your application, and how it is used—how many instances exist, what their names are, and what parameters they have. Based on this information, appropriate source files are generated and the objects are created and initialized as specified.

RF5 uses static configuration where possible. The key reasons for this are:

- **Smaller Footprint.** Static configuration saves memory footprint versus run-time configuration, since there is no need for "init" and "create" start-up logic.

- **Less Fragmentation.** Minimizing the number of MOD_create() calls, which dynamically allocate memory for their data objects, reduces the potential for memory fragmentation. Dynamic allocations of small, varying size objects can fragment the memory pool, and impact the performance of the memory manager when scanning for a chunk large enough to satisfy the request.

- **More Visibility.** RF5’s statically created tasks show up in the DSP/BIOS Execution Graph in CCStudio. Important information regarding the task state is immediately visible, including the task state and assertions indicating a thread did not meet its real-time deadline. The Execution Graph in Figure 13 shows RF5 task states. Dynamically created objects cannot be viewed in the DSP/BIOS Execution Graph, the Kernel Object View, and other real-time analysis windows.

![Figure 13. Execution Graph Activity of RF5 Tasks](image)
This Execution Graph shows that the core activity occurs in tskProcess. Less time is spent in the bracketing tskRxSplit and tskTxJoin tasks, with an occasional "blip" in tskControl as a result of a poll for GEL variable changes. Clearly, the CPU is not highly loaded as evidenced by the time spent in KNL_sw, which runs the task scheduler.

NOTE: The 'C64x device on the 'C6416 TEB board does not support RTDX, which is used for real-time target to host data transfer. As a result, the Execution Graph is available in stop-mode only. That is, it is updated only when the program halts or reaches a break point.

6 Thread Scheduling

RF5 uses two flavors of DSP/BIOS threads: hardware threads (HWIs) and tasks (TSKs). The user's main() function, which DSP/BIOS calls in the initialization phase, is referred to as the main() thread.

In this application note, we often refer to tasks' execution threads simply as threads; HWIs are typically only the concern of device drivers.

RF5 tasks run in infinite loops of waiting for the data, processing the data, and sending the data to other threads. For synchronization, tasks use SIO objects (for talking to device drivers), SCOM objects (for sending messages between tasks), and MBX objects (for sending and receiving control messages).

![Figure 14. Threads (Tasks) in RF5](image-url)

**Key**
- task (thread)
- channel
- cell
- device driver (HWI)
- SIO object
- static buffer
- SCOM queue
- SCOM message
- MBX mailbox (control)
Note that RF5 application code does not touch semaphores directly. They are buried within SCOM, SIO, and MBX objects, and thus safer to use. Still, it is recommended that an RF5-derived application has as few tasks and as few semaphores as possible, because semaphore-based synchronization is the main source of difficult run-time problems. (A list of "50 Rules for Writing Unmaintainable Code" recommends the following as rule number 50: "Use threads with abandon.") Fortunately, the RF5 architecture is such that many channels, cells and algorithms can run under the blanket of a single task.

Usually multiple tasks, and multiple priorities, are necessary in systems that process data at different rates. For example, a telephony system may have a high-priority task for 10 ms G.729 channels, a medium-priority task for GSM channels, and a low-priority task for 30 ms G.723 channels. In RF5, all tasks have equal priority by default.

In the following sections we examine the four threads RF5 employs: RxSplit, Process, TxJoin, and Control.

6.1 The RxSplit Thread

This thread splits the signal into the number of channels specified for the application. The source code for the thread is in files thrRxSplit.h and thrRxSplit.c.

During the initialization and startup phase, this thread creates and opens an input SIO (stream I/O) object called inStream using the "/dioCodec" device. The stream's buffer size is NUMCODECCHANS * FRAMELEN * sizeof( Sample ), and the thread allocates it statically. It also creates its receiving SCOM queue, "scomRxSplit".

At run-time, task tskRxSplit calls the thrRxSplitRun() function. Before it enters its infinite loop, the thread opens—that is, gets the handles to—the two SCOM queues. Within the infinite loop, this function uses SIO_reclaim() to request a full buffer from inStream. If no buffer is ready, it blocks (waits). Then it waits for an SCOM message from the Process thread, by calling SCOM_getMsg() (again, it blocks if the message is not in the queue). The SCOM message is a simple structure, ScomBufChannels (defined in appTheads.h), which contains a pointer to each destination buffer.

Once it receives the message, the task loops through the channels and the frame elements to split the input data into separate buffers, pointed to by the SCOM message. Figure 15 shows how it reorders stereo data from the original input buffer to the output buffers.

![Figure 15. Data Reordering by thrRxSplitRun()](image-url)
It then calls SCOM_putMsg() to put the message to the Process thread's receiving queue, scomToProcessFromRx, indicating that the separated data is available. Finally, it calls SIO_issue() to send an empty buffer back to inStream.

Both RxSplit and TxJoin threads are optional. For example, mono systems or systems with simple data streams may not need these tasks. Alternatively, given the sophistication of the EDMA peripheral, this work could be performed directly in the driver code.

These tasks are used in RF5 to enable rapid prototyping of pre- and post-processing. They allow RF5 to use a generic device driver. This is an important advantage, since system integrators may not all be experienced device driver writers—if system integrators need not worry about modifying the driver, systems can be built quicker.

6.2 The TxJoin Thread

This thread is entirely similar to tskRxSplit, but it performs the reverse action by joining a split signal. Its source code is in thrTxJoin.c and thrTxJoin.h.

In the startup phase, both RxSplit and TxJoin issue their buffers to the drivers via their respective SIO objects, with one important difference: RxSplit's SIO object is an input object, so RxSplit issues empty buffers for the drivers, and waits to reclaim it later when the driver fills them with data.

TxJoin, on the other hand, has an output SIO object, so when it issues its two buffers to the driver via SIO, the driver starts outputting them immediately (the buffers are filled with zeros to indicate silence). This is priming, and TxJoin primes the transmit side to ensure continuous output and to avoid initial pops and clicks that might be heard otherwise. The effect of priming is to shift the timing diagram as shown in Figure 16 so that frames are output only at true frame boundaries. For example, with an RF5 application framesize of 80 and a 'C6416 TEB sampling frequency of 48 kHz, transmit frames are output on the next 1.67 ms boundary, regardless of how long the processing takes.

Figure 16. Effects of Priming in RF5
6.3 The Process Thread

This thread is the core of the application; it processes the data through the XDAIS algorithms. It is the main thread you will modify when adapting the RF5 application to your needs. The source code for this thread is in files thrProcess.c and thrProcess.h.

![Diagram of the Process Thread]

While the state structure for RxSplit and TxJoin threads is trivial, the ThrProcess structure (that is, its instance variable thrProcess), defined in thrProcess.h, contains more information:

```c
typedef struct ThrProcess {
    CHAN_Obj chanList[NUMCHANNELS]; // array of channel objects
    CELL_Obj cellList[NUMCHANNELS * NUMCELLS]; // array of cell objects
    Sample *bufInput[NUMCHANNELS]; // pointers to input buffers
    Sample *bufOutput[NUMCHANNELS]; // pointers to output buffers
    Sample *bufIntermediate; // pointers to intermediate buf.
    ScomBufChannels scomMsgRx; // SCOM object
    ScomBufChannels scomMsgTx; // SCOM object
} ThrProcess;
```

The actual uninitialized placeholders for channel and cell objects are part of this structure. Data buffers are allocated statically, outside this structure, because in general they may need to be aligned for cache purposes, and alignment cannot be easily controlled for structure members. The pointers to those buffers are part of the structure, though, so if you use CCStudio's View Graph feature to visually inspect, say, the intermediate buffer, you can set the address to thrProcess.bufIntermediate and have it visible at all times.

Technically, there is no need for explicit state capturing with tasks, since each task has its own stack. However, having a relevant thread's variables globally visible is valuable in debugging, and it is a good programming practice to avoid using static variables (especially if you plan on making a procedure reentrant in the future).
The thread initializes its cell array in thrProcessInit(), and creates actual XDAIS algorithms and initializes the channel array in thrProcessStartup(). A two-phased initialization is needed because in the first phase the system collects information about all the cells to calculate the minimum scratch buffer size, without creating anything. During the second phase the memory is allocated and the algorithms are created.

The bulk of the startup work is performed in setParamsAndStartChannels() function. This function knows, through its single argument, whether thrProcessInit() or thrProcessStartup() called it. That is, it knows whether it is running in the first or in the second phase of the initialization. In the first phase, it defines the contents of the cell objects and registers the cells. In the second phase, it initializes the channel objects by creating the XDAIS algorithms. But in both phases, it needs to set XDAIS parameters for the algorithms in the cells. Default XDAIS parameters are provided by the vendor in a global object, but you often need to modify some parameter fields. To avoid permanently storing slightly modified parameter objects needed only during the initialization and setup phases, this function defines parameters in both phases by copying the global parameter object to a local parameter object. Then it changes the necessary fields, calls the cell registration or channel creation function (depending on the phase). The local object is freed when the procedure finished running.

Apart from defining cells' objects and algorithm parameters, the initialization also creates SCOM objects for communication, and initializes cell intercommunication buffers (ICCs) by pointing them to the static data buffers. Figure 17 shows the connections between buffers and ICC objects.

The Process thread has two channels with two algorithms each, and it refers to its cells with a mnemonic instead of a number; it defines the following in thrProcess.h:

```c
enum {
   CELLFIR = 0,       // cell #0
   CELVOL,            // cell #1
   NUMCELLS           // total number of cells
};
```

Instead of referring to the FIR cell's index as just 0, it uses CELLFIR mnemonic, which makes code more readable.

The Process thread's NUMCHANNELS (2) channels have identical cell types, but if Process had dissimilar channels, for example, one channel with FIR, VOL algorithms, and one channel with FIR, G723Encode algorithms, it would enumerate its channels as well, and prefix cell names with respective channel names:
// enumeration example for a thread that has two channels with different types
// of cells: one channel with FIR+VOL, and one channel with FIR+ENCODE

enum {  // channel indices
    CHFILTER = 0,  // first channel (FIR + VOL)
    CHCOMPRESS,    // second channel (FIR + G723ENC)
    NUMCHANS      // total number of channels
};

enum {  // cell indices for channel FILTER
    CHFILTERCELLFIR = 0,  // cell #0 (FIR)
    CHFILTERCELLVOL,      // cell #1 (VOL)
    CHFILTERNUMCELLS      // total number of cells for this channel
};

enum {  // cell indices for channel COMPRESS
    CHCOMPRESSCELLFIR = 0,  // cell #0 (FIR)
    CHCOMPRESCCELLENC,     // cell #1 (ENCODER)
    CHCOMPRESSNUMCELLS     // total number of cells for this channel
};

The previous example shows what happens when threads in the application have possibly
different number of channels. Since this enumeration is private for each thread, each thread
would have its own constant for the number of channels. In RF5, it so happens that there are
two physical channels (left and right), so the NUMCHANNELS = 2 constant is defined in the top
application include file, appResources.h.

If the thread had different number of cells for each channel, you would declare cell lists
separately for each channel (in the previous example, with sizes CHFILTERNUMCELLS and
CHCOMPRESSNUMCELLS, respectively).

At run-time, the task performs the thrProcessRun() function. Before entering the infinite loop, the
thread opens the four SCOM queues it needs (two are the thread's own receiving queues, and
the other two are receiving queues for threads RxSplit and TxJoin). The thread, which has two
SCOM messages of type ScomBufChannels, sets one to point to the two bufInput[] buffers and
the other to point to the two (indicated by NUMCHANNELS) output buffers. Neither this thread
nor the RxSplit/TxJoin threads change the content of these messages. In that sense, each
SCOM message is like a token, and whatever task currently has the message has free access to
the buffers the message describes. Other applications may use SCOM messages whose
content changes over time as well.

The Process thread places the message describing the input buffers, scomMsgRx, on RxSplit's
receiving queue. However, the Process thread places the message describing the output
buffers, scomMsgTx, on its own receiving queue. This makes the infinite loop straightforward:
get both SCOM messages, process the data, put both SCOM messages to other threads' queues, and repeat.
The following code shows the infinite loop in the `thrProcessRun()` function.

```c
// Main loop
while (TRUE) {
    ScomBufChannels *scomMsgRx, *scomMsgTx;

    // check for control (MBX) messages (not to be confused with SCOM msgs)
    checkMsg();

    // get the message describing full input buffers from Rx
    scomMsgRx = SCOM_getMsg( scomReceiveFromRx, SYS_FOREVER );

    // get the message describing empty output buffers from Tx
    scomMsgTx = SCOM_getMsg( scomReceiveFromTx, SYS_FOREVER );

    // record the time period between two frames of data in stsTime0
    UTL_stsPeriod( stsTime0 );

    // process the data
    for( chanNum = 0; chanNum < NUMCHANNELS; chanNum++ ) {
        CHAN_Handle chanHandle = &thrProcess.chanList[chanNum];

        // Set the input ICC buffer for FIR cell for each channel
        ICC_setBuf(chanHandle->cellSet[CELLFIR].inputIcc[0],
                   scomMsgRx->bufChannel[chanNum],
                   FRAMELEN * sizeof( Sample ) );

        // Set the output ICC buffer for VOL cell for each channel
        ICC_setBuf(chanHandle->cellSet[CELLVOL].outputIcc[0],
                   scomMsgTx->bufChannel[chanNum],
                   FRAMELEN * sizeof( Sample ) );

        // execute the channel
        UTL_stsStart( stsTime1 );  // start the stopwatch
        rc = CHAN_execute( chanHandle, NULL );
        UTL_assert( rc == TRUE );
        UTL_stsStop( stsTime1 );   // elapsed time goes to this STS
    }

    // send the message describing full output buffers to Tx
    SCOM_putMsg( scomSendToTx, scomMsgTx );

    // send the message describing consumed input buffers to Rx
    SCOM_putMsg( scomSendToRx, scomMsgRx );
}
```

This loop first checks to see if the mbxProcess mailbox contains any control messages. It does not wait for a message if the mailbox is empty. Next, it calls `SCOM_getMsg()` twice to get a full input buffer from the scomToProcessFromRx SCOM queue, and an empty output buffer from the scomToProcessFromTx queue. If a message is not ready, it blocks while waiting.
Once it gets ownership of the buffers, it loops through the channels. The calls to ICC_setBuf() set the buffer and buffer size for ICC objects. These ICC calls are not necessary in single-buffered systems. Since the address is always the same, it is included in the code to allow easy migration to double buffering. For each channel, this loop then calls CHAN_execute(), which in turn causes each cell in the channel to perform its XDAIS algorithm. See Section 7.3, ICELL Interface—Cells as Algorithm Containers, page 48 for more about cells. After executing the channel, this function calls SCOM_putMsg() twice to send the full buffer to the scomProcToTx object and put the used buffer to the scomProcToRx SCOM buffer.

6.4 The Control Thread

Each thread can send and receive control messages via mailboxes. In RF5, only Process thread receives messages, and only the Control thread sends messages. Most applications will have a thread like the Control thread. Its source files are thrControl.h and thrControl.c.

The Control thread interacts with the outside world, for example reads values of various onboard hardware settings that can change during the course of the application work, and sends control messages to other threads—to the Process thread in this case. In RF5, the Control thread reads a global variable that the user changes via the GEL script. The variable contains information about channel volume and type of filters to use. Every 100 ticks the Control thread checks this variable and if it detects a change, it sends a message to the Process thread.

```c
while (TRUE) {
  checkMsg();
  process data
}
checkMsg() {
  CtrlMsg rxMsg;
  if (MBX_pend( mbxProcess, &rxMsg, 0 )) {
    if (rxMsg.cmd == MSGNEWCOEFF) {
      // set channel num. <rxMsg.arg1> to use
      // FIR coefficient set num. <rxMsg.arg2>
      FIR_cellControl(<cell>, <coefficients>);  
    } else
      ...
  }
}
```

```c
while (TRUE) {
  if externalControl for ch. #0 has changed
  txMsg.cmd = MSGNEWCOEFF; // ctrl command
  txMsg.arg1 = 0; // channel to control
  txMsg.arg2 = externalControl.firCoeff[0];
  MBX_post( mbxProcess, &txMsg, 0 );
  ...
  TSK_sleep( 100 ); // sleep for 100 ticks
}
```

Figure 18. Flow of Control Messages in RF5
Control message format is defined in appThreads.h, and consists only of three 32-bit unsigned integers: message command, first argument, and the second argument. Each recipient, the Process thread in this case, defines in its header file what kind of messages it accepts. The control thread, which imports this file, interprets the event in the outside world and sends the appropriate message (for example, "Volume changed to 65" or "Switch to high-pass filter for channel #1").

During the initialization phase, this thread sets the default volume to 100. During the startup phase, this thread sends a message to initialize the volume and FIR filter coefficients.

At run-time, the thrControlRun() function runs in an infinite loop. Within this loop, it loops through all the channels and checks the variables written to by the GEL controls for changes. If the value has been changed, it calls MBX_post() to send a message with the new value and the channel for which it was changed.

The tskControl object runs at the same priority as the other tasks. In order to allow other tasks and the idle functions to run, after each loop, it suspends itself for 100 ticks by calling TSK_sleep().

Note that this message posting style (as opposed to direct global variable modification) allows the user to give the control task any priority level. The only issue that may arise is potential starvation or unresponsiveness if the thread has too low a priority.

6.5 SCOM Module — Synchronized Communication

We mentioned earlier that RF5 tasks use the SCOM (Synchronized COMmunication) module for passing data-related messages among themselves. SCOM is a generic inter-task message-passing mechanism.

SCOM messages are user-defined structured data objects that tasks exchange among themselves. A task can pass around an SCOM message by placing it on an SCOM queue via SCOM_putMsg(), or taking it from the queue via SCOM_getMsg().

An SCOM queue is a named, semaphore-based queue (it uses the efficient DSP/BIOS QUE module), created via SCOM_create(). The name of the queue is a String. Any task that knows an SCOM queue's name can get a handle to the queue (needed for put and get message operations), by calling SCOM_open() and passing the known name. Typically, each task creates one or more SCOM queues from which it will receive messages. Any SCOM message can be placed on any SCOM queue.

An SCOM message can be any data structure that has a QUE_Elem variable as its first field. For example:

```c
struct myMsg {
    QUE_Elem   elem;
    Int        someField;
    ...
} myMsg;
```
RF5 uses SCOM in a simple way: Process task has some buffers to which it wants some data written by RxSplit task, and read by TxJoin task. The Process task needs to 1) tell the other tasks where the buffers are, and 2) ensure that two tasks do not access the same buffer at the same time. So Process creates one SCOM message for (synchronized) communication with RxSplit task, and one for the TxJoin task. The Rx message describes the buffers where RxSplit should write; similarly for the Tx side. Each SCOM message then becomes like a token for its respective buffer: the task that has the token can access the buffer freely.

![Figure 19. SCOM Usage in RF5 (Rx Side)](image)

Each SCOM message is of custom, user-defined format. In another application, an SCOM message can contain some information other than just plain buffer address, and participants in the communication can read and write over that part of the message.

Let us look at a general case where some tasks A and B want to exchange messages. Tasks A and B agree that A sends data of type MyMsg to B via SCOM queues named "scomA" for task A, and "scomB" for task B. The following steps would occur:

**Table 4. SCOM Communication Sequence Example**

<table>
<thead>
<tr>
<th>Task A Actions</th>
<th>Task B Actions</th>
</tr>
</thead>
</table>
| **1.** The initialization code for task A creates the scomA SCOM queue ("scomA" is the known name, the handle scomA is a local variable and can have any name):  
SCOM Handle scomA = SCOM_create("scomA", NULL); | 2. The initialization code for task B creates the scomB SCOM queue, from which it will receive SCOM messages:  
SCOM Handle scomB = SCOM_create("scomB", NULL); |
| **3.** Task A’s startup code gets the reference to B’s queue (after the init phase, when it knows task B has created its queue):  
SCOM Handle scomB = SCOM_open("scomB"); | 4. Task B’s startup code gets the reference to A’s queue:  
SCOM Handle scomA = SCOM_open("scomA"); |
5. Task A allocates a buffer of type MyMsg, and writes some information into it:
   ```c
   MyMsg *msg = MEM_alloc(heapId, sizeof(MyMsg), 0);
   msg->someField = someValue;
   ```

6. In a loop, task A sends the message to task B by placing it on B's queue:
   ```c
   SCOM_putMsg(scomB, msg);
   ```

7. In a loop, task B checks to see if Task A has sent a message. Since the message has been sent, Task B receives the message, and reads the information. The timeout (for example, SYS_FOREVER) specifies how long to wait for the message. Use 0 as the timeout if Task B should not wait.
   ```c
   MyMsg *msg = SCOM_getMsg(scomB, SYS_FOREVER);
   read/write msg->someField...
   ```

8. When finished processing the data described by the message (or in the message), task B sends the message back to task A:
   ```c
   SCOM_putMsg(scomA, msg);
   ```

9. Task A now gets the message, and the loop repeats:
   ```c
   msg = SCOM_getMsg(scomA, SYS_FOREVER);
   ```

Note that you most likely would never want two SCOM messages to point to the same buffer. An SCOM message is like a token for the buffer. If more that one SCOM message points to the same buffer, the token cannot grant access exclusivity.

This is the summary of functions provided by the SCOM module:

- **SCOM_init()**: Initializes the module.
- **SCOM_exit()**: Ends use of the module.
- **SCOM_create()**: Creates a new SCOM queue object.
- **SCOM_open()**: Gets a reference to an existing SCOM queue object by name.
- **SCOM_delete()**: DEALLOCATES and deletes a SCOM queue object.
- **SCOM_putMsg()**: Places SCOM message in an SCOM queue.
- **SCOM_getMsg()**: Receives SCOM message from an SCOM queue.

Figure 20 summarizes the sequence in which SCOM module functions may be called for a single SCOM queue. (The SCOM_init() function is called only once for the entire module.) Once a queue has been created by one thread, the other thread using this queue for communication performs SCOM_open().
Details about the SCOM module are provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.

7 Channel and Algorithm Infrastructure

RF5 provides a channel infrastructure that makes it easy to encapsulate XDAIS algorithms. Through this encapsulation, the application designer can easily scale the application to large numbers of channels and/or algorithms.

In the infrastructure, tasks contain channels. Each task may have its own processing frequency. A task may contain one or more channels. Each channel contains one or more cells. The cells encapsulate XDAIS algorithms and are described in Section 7.3, ICELL Interface—Cells as Algorithm Containers, page 48.

Figure 21 shows how tasks, channels, and cells are related in RF5:
The modules provided with RF5 and used in the channel infrastructure are shown in Figure 22 and described in the subsections that follow.

![Diagram of Channel Infrastructure Modules]

- **XDAIS algs.** - Off the shelf XDAIS algorithms
- **Cell wrappers** - Encapsulation of XDAIS algorithms
- **ICELL** - Cell interface
- **CHAN** - Channel module (collection of cells)
- **ICC** - Inter-Cell Communication module
- **ALGRF** - XDAIS algorithm instantiation for Reference Frameworks
- **SSCR** - Shared Scratch support

**Figure 22. Channel Infrastructure Modules**

### 7.1 CHAN Module—Channel Management

A channel is a collection of cells. The main purpose of a channel is to serially execute the cells contained by the channel. This module manages one or more channel objects. The structure of a channel object (CHAN_Obj) is as follows.

```c
typedef struct CHAN_Obj {
    ICELL_Obj *cellSet; /* set of cells in the channel */
    Uns cellCnt; /* number of cells in the cellSet */
    CHAN_State state; /* state of the channel */
    Bool (*chanControlCB)(CHAN_Handle chanHandle); /* optional control function */
} CHAN_Obj;
```

The threads usually do not define this object themselves, but initialize them via a CHAN_open() call. The last argument to CHAN_open() is the address of a channel attributes structure of type CHAN_Attrs. If it is NULL, CHAN_open() uses default attributes. If you want the channel attributes to differ from the defaults, declare a variable of type CHAN_Attrs and initialize it to the default CHAN_ATTRS. Then change individual field values in the structure as desired. Currently, the fields in the CHAN_Attrs structure are the channel state, which defaults to CHAN_ACTIVE, and a channel control callback function, which defaults to NULL. The channel control callback function, if not NULL, is called before any cells are executed.

In a typical setting, a thread has one CHAN_Obj for each channel (or an array of those if they are similar), and one ICELL_Obj for each cell (typically one array of those per channel). After the thread has initialized each ICELL_Obj (see Section 7.3, ICELL Interface—Cells as Algorithm Containers, page 48 for more information on cells), it makes a call like the following where `cell` is a pointer to the cell object, and `inputIcc/outputIcc` are the cell's ICC objects (also explained in following sections). This call calculates the cell's scratch memory requirements, and assigns the given ICC objects to the cell.

```c
CHAN_regCell( cell, inputIcc, 1, outputIcc, 1 );
```
When all the cells have been created and initialized, the thread makes a call like the following for each channel chanNum, passing the cellList list of cell objects (of size numCells) for that particular channel. The function creates all the XDAIS algorithms and calls each cell’s cellOpen function, if that function is defined for the cell.

```
CHAN_open( chanList[ chanNum ], cellList, numCells, NULL/* default attributes */ );
```

Finally, at run time, the thread makes the following call for each channel chanNum:

```
CHAN_execute( chanList[ chanNum ], NULL /* arg to cells */ );
```

Figure 23 summarizes the sequence in which CHAN module functions may be called. CHAN functions not shown here must be called after CHAN_setup() and before CHAN_exit() as appropriate.

![Figure 23. CHAN Function Calling Sequence](image)

The CHAN module provides the following functions:

- **CHAN_delete()**. Deletes a channel by freeing the CHAN_Obj for the channel.
- **CHAN_init()**. Initializes the CHAN module and the modules it uses internally (ALGRF and SSCR).
- **CHAN_setup()**. Sets up the CHAN module. This call allows you to specify which heaps should be used when the CHAN module allocates memory. It also allows you to specify how this channel should use the scratch buckets created by the SSCR module.
- **CHAN_regCell()**. Registers a cell. This function uses SSCR_prime() to determine the worst-case scratch buffer requirements for the algorithm specified in the cell. It also assigns the “in” and “out” ICC objects to the cell.
- **CHAN_create()**. Creates a channel by allocating a CHAN_Obj from the heap used by DSP/BIOS. It returns the pointer to this object.
• **CHAN_open().** Creates algorithm instances for the cells in the channel and opens the cells by calling their cellOpen() functions, if specified.

• **CHAN_execute().** If the channel is active, this function calls the chanControlCB() function for the channel if one is specified. Then, it calls the cellExecute() function for all cells contained by the channel.

• **CHAN_close().** Closes the cells in a channel and frees the algorithms opened and created by the channel in CHAN_open().

• **CHAN_getAttrs().** Gets the attribute structure for a channel, which includes the channel state and its control callback function.

• **CHAN_setAttrs().** Sets the attributes of a channel.

• **CHAN_exit().** Exits from the CHAN module.

The arg parameter to the CHAN_execute() function is passed to the cellExecute() function for each cell in the channel. This user-defined structure may be different for each channel (but must be the same for all cells in a channel). This parameter allows you to pass channel-level information. For example, the result of one algorithm could affect the processing of a later algorithm in the channel. The arg structure could communicate this result. Another example would be to use the arg parameter to store a semaphore handle used to lock the scratch buffer in the "locking mechanism" method of SSCR. Note that the arg parameter is usually a "global" argument—that is, one common to all the cells in a channel, and as such rarely used. Cells also have cell-specific cell environments, which let threads communicate with the cell directly, in a cell-specific way.

Details about the CHAN module are provided in the *Reference Frameworks for eXpressDSP Software: API Reference (SPRA147)* application note.

### 7.2 ALGRF Module—Algorithm Instantiation

The ALGRF module exists to create and delete XDAIS algorithms using the DSP/BIOS MEM memory manager. It is a Reference Framework service that simplifies the use of XDAIS components in end-applications. All XDAIS-compliant algorithms must implement the IALG interface. ALGRF uses algorithms' IALG implementations to instantiate XDAIS algorithm instances. Any XDAIS-compliant algorithm can be used with ALGRF.

User code does not need to call ALGRF functions directly; this is the job of CHAN and other library modules. One exception is the ALGRF_activate/deactivate sequence in cell wrappers: if the XDAIS algorithm for the cell implements IALG_activate/deactivate functions, the cell needs to call these two ALGRF functions.

Three modules have been provided to simplify use of the IALG interface to create algorithm objects. Higher-level Reference Frameworks (such as RF3 and RF5) use the ALGRF module to create, configure, and delete instances of XDAIS algorithms. The ALG module supplied with CCStudio is for general-purpose use, and does not use the DSP/BIOS MEM module for memory allocation. ALGMIN (used by RF1) is the smallest implementation of the three. Table 5 compares these three modules.
Table 5. Algorithm Instantiation Module Characteristics

<table>
<thead>
<tr>
<th>Provided with</th>
<th>ALGMIN</th>
<th>ALGRF</th>
<th>ALG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Frameworks</td>
<td>Reference Frameworks</td>
<td>CCStudio</td>
<td></td>
</tr>
<tr>
<td>Best used with</td>
<td>Compact, static systems</td>
<td>RF3 and higher</td>
<td>General purpose use</td>
</tr>
<tr>
<td>Memory allocation</td>
<td>Static</td>
<td>Dynamic; uses DSP/BIOS MEM module</td>
<td>Dynamic; supports both malloc() and MEM allocation.</td>
</tr>
</tbody>
</table>

Key points
- Super-compact.
- ALGMIN_new is key instantiation function.
- ALGMIN_new calls algInit but not algAlloc since buffers are pre-generated.
- Uses DSP/BIOS MEM module for dynamic allocation.
- Smaller footprint than ALG.
- Supports sharing of scratch memory.
- Supports both malloc() and DSP/BIOS MEM dynamic allocation.
- Not tuned for footprint.

Naturally these modules are mutually exclusive. Only one should be used in an application. ALGRF fits the needs of RF5 and other RF levels. It is not appropriate however for extremely compact, low-end systems such as RF1.

ALGRF has the following advantages over ALG:

- **Smaller footprint.** As a generic module ALG supports both malloc / free Run-Time Support Library and DSP/BIOS MEM_alloc / MEM_free dynamic memory allocation. ALGRF supports only DSP/BIOS allocation, which saves code-space for the designer. Additionally, ALGRF ensures that no “dead code” exists; only functions that are called are linked in to the executable.

- **Scratch Memory Support.** The following API has been introduced in ALGRF:

  ```c
  ALGRF_Handle ALGRF_createScratchSupport(IALG_Fxns *fxns, IALG_Handle parent, IALG_Params *params, Void *scratchBuf, Uns scratchSize)
  ```

  This function allocates memory requested by algorithms, except in the case where IALG_SCRATCH, internal data buffers are requested. Instead, the scratchBuf and scratchSize parameters indicate that a buffer already exists in the application, which can be reused by the current algorithm. Such controlled sharing saves precious data memory.

  The CHAN module uses this API in conjunction with the SSCR module.

- **Abstraction from DSP/BIOS heap labels.** ALGRF uses the DSP/BIOS MEM module dynamic memory allocation. A heap identifier label or memory segment name can be passed to MEM_alloc() indicating which heap to allocate from. Instead of hard-coding these labels, they are passed in via:

  ```c
  /* Configure the ALGRF module to use: *
  * 1st argument - memory for internal heap *
  * 2nd argument - memory for external heap *
  */
  ALGRF_setup( INTERNALHEAP, EXTERNALHEAP );
  ```

  This lets you control where algorithm data is allocated. For example, if you pass EXTERNALHEAP for both arguments, all algorithm data is allocated in external memory.
The ALG module will remain in CCStudio to support legacy content and enable non-DSP/BIOS, or "generic" applications. ALGRF sits side-by-side as an alternative XDAIS instantiation module.

Syntax for all ALGRF functions is provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.

ALGRF is applicable to all 'C5000 and 'C6000 targets. It can also be used independent of Reference Frameworks, like other library modules.

7.3 ICELL Interface—Cells as Algorithm Containers

The TMS320 DSP Algorithm Standard (also known as XDAIS) provides a standardized interface to algorithms. This allows you to easily integrate third-party algorithms into your application. For technical details on the TMS320 DSP Algorithm Standard, see TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352) and the TMS320 DSP Algorithm Standard API Reference (SPRU360).

Applications based on RF5 typically use a relatively large number of algorithms and/or channels. In order to simplify algorithm integration, RF5 introduces the "cell" concept. A "cell" is an application wrapper for a XDAIS algorithm. An RF5 "channel" can contain multiple cells, and hence multiple algorithms.

At the heart of the channel infrastructure is the concept of a cell. Algorithm’s run-time functions can be different. A cell is a standardized encapsulation of an algorithm. For every algorithm instance, there is a corresponding cell object. Instead of interfacing to the algorithm directly, the channel interfaces to a cell, which in turns calls the algorithms interface.

RF5 provides a cell interface called ICELL. The ICELL structures are defined by the interface. There are no ICELL module function calls.

The ICELL interface is similar to the IALG interface in the XDAIS specification. That is, the structures for the interface are specified in a header file. There must be an implementation of the structures for an algorithm. One major difference between ICELL and IALG is that algorithm providers are required to implement the IALG interface. In contrast, the application designer typically implements the ICELL interface for each algorithm used in an RF5 application to customize its usage within the particular application.

Your application must create implementations of the following ICELL structures for each algorithm:

- **Structure of type ICELL_Fxns and its functions.** This structure provides a consistent interface to algorithm execution functions, whose names and parameters are not standardized. The structure is defined RF_DIR\include\icell.h as follows:

```c
typedef struct ICELL_Fxns {
    BooI (*cellClose ) (ICELL_Handle handle);
    Int  (*cellControl)(ICELL_Handle handle, IALG_Cmd cmd, IALG_Status *status);
    BooI (*cellExecute)(ICELL_Handle handle, Arg arg);
    BooI (*cellOpen   )(ICELL_Handle handle);
} ICELL_Fxns;
```

For example, the RF_DIR\apps\rf5\cells\vol\cellVol.h and cellVol.c files implement the ICELL_Fxns structure and its functions for the VOL algorithm.
It is not required that you implement the cellClose, cellControl, and cellOpen functions. The cellExecute function is required and a sample is shown in the example that follows. The cellExecute function is called many times from the containing thread's main loop (indirectly, via CHAN_execute). The cellControl function may be called occasionally to modify some control information. The cellOpen and cellClose functions are called by CHAN_open() and CHAN_close() respectively only when the cell is created and when (if) it is destroyed.

The functions you implement typically make use of the IALG implementation provided by the algorithm and the ALGRF module to activate and deactivate the algorithm.

```c
Bool VOL_cellExecute( ICELL_Handle handle, Arg arg )
{
    IVOL_Fxns *volFxns   = (IVOL_Fxns *)handle->algFxns;
    IVOL_Handle volHandle = (IVOL_Handle)handle->algHandle;

    // activate instance object
    ALGRF_activate( handle->algHandle );

    volFxns->amplify( volHandle,
                     (XDAS_Int16 *)handle->inputIcc[0]->buffer,
                     (XDAS_Int16 *)handle->outputIcc[0]->buffer );

    // deactivate instance object
    ALGRF_deactivate( handle->algHandle );

    return ( TRUE );
}
```

- **Object of type ICELL_Obj.** This structure defines the characteristics of a cell. The structure is defined in `icell.h` as follows. You should not modify this structure definition.

```c
typedef struct ICELL_Obj {
    Int          size;     /* Number of MAU in the structure        */
    String       name;     /* User chosen name.                     */
    ICELL_Fxns  *cellFxns; /* Ptr to cell v-table function.         */
    Ptr          cellEnv;  /* Ptr to user defined cell env. struct  */
    IALG_Fxns   *algFxns;  /* Ptr to alg v-table functions.         */
    IALG_Params *algParams; /* Ptr to alg parameters.                */
    IALG_Handle  algHandle; /* Handle of alg managed by cell.       */
    Uns          scrBucketIndex; /* Scratch bucket for XDAIS scratch mem. */
    ICC_Handle  *inputIcc;  /* Array of input ICC objects          */
    Uns          inputIccCnt; /* # of ICC objects in the input array */
    ICC_Handle  *outputIcc; /* Array of output ICC objects         */
    Uns          outputIccCnt; /* # of ICC objects in the output array */
} ICELL_Obj;
```
Notice that this structure uses types defined by a number of related modules. This structure helps create relationships between the ICELL, IALG, ICC, and SSCR modules. Several elements in the ICELL_Obj structure are worthy of special note:

- **size and name.** The size is sizeof(ICELL_Obj). The name is a string that typically identifies the algorithm performed.

- **cellFxns.** This element points to the previously described structure of type ICELL_Fxns.

- **cellEnv.** This structure is user-defined. Each cell has its own cellEnv pointer, which can be used to maintain cell-specific information. Each cell may have a different structure definition. For example, if an algorithm has mutually exclusive runtime functions, such as apply1 and apply2, the cellExecute function could determine which function to execute based on a field in the cellEnv structure that the calling thread would write. Another use of the cellEnv structure might be to store DMA handles used by the cell (this is not for the algorithm’s DMA use). In the cellOpen function, the DMA channel could be allocated and stored in the cellEnv structure. Then the cellExecute function could use the DMA handle. See Section 7.5.2, Locking Mechanism Method Implementation, page 56 for an example that uses the cellEnv.

- **algFxns, algParams, and algHandle.** These elements have types defined by the IALG interface that is part of the XDAIS specification.

- **scrBucketIndex.** Generally, all cells in channels executed by tasks at the same priority level should have the same scrBucketIndex. This element is used by the SSCR module, which is described in Section 7.5, SSCR Module—Shared Scratch Memory, page 54.

- **inputIcc and outputIcc.** These elements are filled in by a call to CHAN_regCell() for this cell. The information is used by the ICC module, which is described in Section 7.4, ICC Module—Inter-Cell Communication, page 52.
For example, the `thrProcess.c` file creates an array of elements of type `ICELL_Obj` for all the cells in the application, which you should modify to integrate your algorithms. The following portion shows the declaration of the first cell.

```c
// Cell list declared in thread's structure
typedef struct ThrProcess {
    ...
    ICELL_Obj    cellList[ NUMCHANNELS * NUMCELLS ];
    ...
} ThrProcess;
ThrProcess thrProcess;

/* in the init phase: */
IFIR_Params firParams;
IVOL_Params volParams;
Int    chanNum;
ICELL_Obj *cell;
Bool   rc;
ICC_Handle  inputIcc;
ICC_Handle  outputIcc;

for (chanNum = 0; chanNum < NUMCHANNELS; chanNum++) {

    /* Setup a default cell used to initialize the actual cells */
    ICELL_Obj  defaultCell = ICELL_DEFAULT; // default cell fields
    ICELL_Obj *cell; // short-name alias used to access each cell’s fields

    // cell 0 (FIR)
    cell = &thrProcess.cellList[ (chanNum * NUMCELLS) + CELLFIR ];
    *cell = defaultCell; // initialize w/ default fields
    cell->name = "FIR"; // cell name (debug/diagnostics)
    cell->cellFxns = &FIR_CELLFXNS; // cell V-table
    cell->algFxns = (IALG_Fxns *)&FIR_IFIR; // XDAIS alg. V-table
    cell->algParams = (IALG_Params *)&firParams; // alg. parameters
    cell->scrBucketIndex = THRPROCESS_SCRBUCKET; // scratch level

    inputIcc = ICC_linearCreate( ... ) // input ICC buffer
    outputIcc = ICC_linearCreate( ... ) // output ICC buffer
    rc = CHAN_regCell( cell, &inputIcc, 1, &outputIcc, 1 ); // register

    <do similarly for remaining cells>
}
```

Details about the `ICELL` interface and related modules are provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.
7.4 ICC Module—Inter-Cell Communication

The ICC module is a small module that manages data communication among cells, and between cells and their tasks. When the cells of a channel execute their algorithms’ functions, ICCs are used instead of passing input/output buffers as parameters in the execution call. This module manages one or more ICC objects.

The structure of an ICC object (ICC_Obj) is as follows.

```c
typedef struct ICC_Obj {
    Ptr            buffer;         // Pointer to the buffer
    Uns            nmaus;          // Size of the buffer
    ICC_ObjType    objType;        // Type of ICC
} ICC_Obj;
```

The ICC object has a type. Currently, the only type for which API functions are provided is ICC_LINEAROBJ, which is simply a regular buffer. You may extend these types to support more complex types of signal data or to handle debugging and validation of the buffer contents.

Each cell has an array of input and output ICC objects. These input and output ICC objects are set up by CHAN_regCell().

The ICC module provides the following functions:

- **ICC_exit().** Exits the ICC module.
- **ICC_getBuf().** Gets the buffer and buffer size for the specified ICC object.
- **ICC_init().** Initializes the ICC module.
- **ICC_linearCreate().** Creates a linear ICC object from the DSP/BIOS segment. This function does not allocate the buffer; the caller must supply the buffer.
- **ICC_linearDelete().** Deletes the specified linear ICC object. This function does not free the buffer associated with the object.
- **ICC_setBuf().** Sets the buffer and buffer size for the specified ICC object.
Using ICC objects allows a channel to have a flexible data flow. For example, consider the data flow among tasks in Figure 24:

![Figure 24. ICC Data Flow Example](image)

There are three cells in this channel. The first cell has two outputs. By using five linear ICC objects (corresponding to the five arrows), this can easily managed. The pseudo-code to set up this data flow would be as follows:

```c
inX[0] = ICC_linearCreate( B1, sizeof( B1 ) )
outX[0] = ICC_linearCreate( B2, sizeof( B2 ) )
outX[1] = ICC_linearCreate( B3, sizeof( B3 ) )
inY[0] = outX[0]
outY[0] = ICC_linearCreate( B4, sizeof( B4 ) )
inZ[0] = outY[0]
outZ[0] = ICC_linearCreate( B5, sizeof( B5 ) )

CHAN_regCell( cellX, inX, 1, outX, 2)
CHAN_regCell( cellY, inY, 1, outY, 1)
CHAN_regCell( cellZ, inZ, 2, outZ, 1)
```

Why not just use pointers to static buffers instead of pointers to linear ICC objects that describe the static buffers? The answer is, in the future, some ICC objects may be other than just linear—they could be circular buffers or various form of "active" buffers (that is, buffers with operations assigned to them). In fact, the user can define their own ICC types if linear ICCs are not sufficient. The other reason is that the address only is often not enough. An algorithm may need to know the buffer size as well, especially if the data can shrink, and that calls for a structured description in the form of ICC.

Details about the ICC module are provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.
7.5 SSCR Module—Shared Scratch Memory

The SSCR module manages overlaying of on-chip scratch memory requested by XDAIS algorithms.

Scratch memory is memory the algorithm needs during its run-time execution functions. An algorithm can freely use scratch memory without regard to its prior contents. Persistent memory is any area of memory that can be safely written to, knowing that the contents will be unchanged between successive invocations by the application. This distinction enables optimization by overlaying scratch memory for several algorithms on the same physical memory as shown in Figure 25.

Since a XDAIS algorithm cannot block while executing a run-time execution function, scratch memory can be shared among other algorithms. However, since a higher-priority thread can interrupt a lower-priority thread, special protection must be provided for the shared memory.

In Figure 25, two instances of algorithm A, and one instance of algorithms B and C each, run at the same priority. Algorithm's persistent memory block must be preserved at all times, but the scratch segments need not. Therefore, the algorithms are set to share their scratch segment, whose size is set to the largest one needed.

The SSCR module allows you to provide protection in one of two ways:

- **Priority level.** A separate scratch buffer is used for each thread priority level that uses scratch buffers. Since DSP/BIOS is principally a priority-based (as opposed to time-sliced) pre-emptive real-time kernel, this method is safe. (See Section 7.5.1, Priority Method Implementation, page 56.)

- **Locking mechanisms.** One scratch buffer is shared by all algorithms. Locking mechanisms (such as semaphores, locks or HWI_disable() and HWI_restore() calls) must be used to protect against preemption whenever the scratch buffer is accessed. An application that uses time slicing must use a locking mechanism rather than relying on priority levels to protect scratch buffers. (See Section 7.5.2, Locking Mechanism Method Implementation, page 56.)
The priority level method provides lower latency but uses a larger total scratch buffer allocation. The application designer should decide whether to minimize latency or memory use.

The SSCR module manages one or more "bucket" objects. Each bucket maintains one scratch buffer. In the priority level method, each bucket corresponds to a priority level. In the locking mechanism method, there is only one bucket. A bucket object has the following structure:

```c
typedef struct SSCR_Bucket {
    Ptr buffer;              /* pointer to the scratch buffer */
    Uns size;                /* size of the scratch buffer */
    Uns count;               /* number of users of this bucket */
} SSCR_Bucket;
```

The SSCR module analyzes and allocates scratch buffers using a three-step process:

1. The application designer decided whether to use the priority or locking mechanisms for scratch overlay protection. Based on this choice, the designer specifies a bucket index in the ICELL_Obj structure for each cell. For example, the default RF5 application uses the THRPROCESSSSCRBUCKET bucket for all cells because there is only one task priority level for XDAIS algorithm execution.

2. The application registers a cell with CHAN_regCell(), which in turn calls SSCR_prime() to determine the worst-case scratch memory requirements.

3. The application calls CHAN_open(), which in turn calls SSCR_createBuf() for each cell in a channel to allocate the memory. If the scratch buffer for a bucket index has already been allocated for another cell, SSCR_createBuf() simply returns that buffer.

The SSCR module provides the following functions:

- **SSCR_init()**. Initializes the SSCR module.
- **SSCR_setup()**. Sets up the SSCR module.
- **SSCR_createBuf()**. Creates the scratch buffer or returns it if it is already allocated.
- **SSCR_getBuf()**. Gets the size of and a pointer to the scratch buffer.
- **SSCR_prime()**. Determines the worst-case scratch usage for an algorithm instance.
- **SSCR_deleteBuf()**. Deletes the scratch buffer.
- **SSCR_exit()**. Exits the SSCR module.
7.5.1 Priority Method Implementation

The following example shows a priority-based scratch sharing application. In this example, three threads (thrX, thrY and thrZ) run at three different priorities.

In appThreads.h, the number of scratch buckets is set as follows:

```c
enum SSCRBUCKETS { THRXSCRBUCKET = 0,
                  THRYSRBUCKET,
                  THRZSCRBUCKET,
                  SCRBUCKETS         };   // total # of scratch buckets
```

The number of scratch buffers must be specified in the CHAN_setup() call. For example, in appThreads.c:

```c
CHAN_setup( INTERNALHEAP, EXTERNALHEAP, INTERNALHEAP, SCRBUCKETS, NULL, NULL );
```

When a cell is set up, its scrBucketIndex field must be set to accordingly. For example in thrX.c:

```c
cell = &thrX.cellList[ (chanNum * NUMCELLS) + CELLFIR ];
*cell = defaultCell;
cell->name = "FIR";
...
cell->scrBucketIndex = THRXSCRBUCKET;
```

7.5.2 Locking Mechanism Method Implementation

The following example shows how an application could use only one scratch buffer through use of a locking mechanism. Again, three threads (thrX, thrY and thrZ) run at three different priorities. Note: If all the processing threads run at the same priority, no locking mechanisms are needed.
In `appThreads.h`, the number of scratch buckets is set as follows:

```c
enum SSCRBUCKETS { ONESCRBUCKET = 0,
                  SCRBUCKETS  };         // total # of scratch buckets
```

A single scratch buffer is specified in the `CHAN_setup()` call in `appThreads.c` as follows:

```c
CHAN_setup( INTERNALHEAP, EXTERNALHEAP, INTERNALHEAP, SCRBUCKETS, NULL, NULL );
```

When a cell is set up, its `scrBucketIndex` field must be set to `ONESCRBUCKET`. For example in `thrX.c`:

```c
cell = &thrX.cellList[ (chanNum * NUMCELLS) + CELLFIR ];
*cell                = defaultCell;
cell->name           = "FIR";
...
env.protectScratch   = TRUE;         //user defined structure
cell->cellEnv        = &env;
cell->scrBucketIndex = ONESCRBUCKET;
```

The previous code snippet also shows a use of the `cellEnv` pointer. By checking a field in the user-defined structure, the implementation of the cell can be more robust.

In the cell, a locking mechanism must be used. For example, if latency is not a key care-about, one can use `HWI_disable` and `HWI_restore` to block other threads from running and accessing the scratch buffer. It is up to the integrator to chose the best locking mechanism for the particular application.

```c
Bool FIR_cellExecute( ICELL_Handle handle, Arg arg )
{
    IFIR_Fxns  *firFxns   = (IFIR_Fxns *)handle->algFxns;
    IFIR_Handle firHandle = (IFIR_Handle)handle->algHandle;
    Uns interruptState;
    if (handle->cellEnv->protectScratch == TRUE) {
        interruptState = HWI_disable();
    }
    // activate instance object
    ALGRF_activate( handle->algHandle );
    firFxns->filter( firHandle,
                    (Short *)handle->inputIcc[0]->buffer,
                    (Short *)handle->outputIcc[0]->buffer );
    // deactivate instance object
    ALGRF_deactivate( handle->algHandle );
    if (handle->cellEnv->protectScratch == TRUE) {
        HWI_restore( interruptState );
    }
    return ( TRUE );
}
```
8 I/O and Drivers

RF5 uses the DIO driver adapter to communicate with the IOM mini-driver for the target board. The *DSP/BIOS Driver Developer's Guide* (SPRU616) contains details about IOM, DIO, and mini-drivers. This section provides a short overview.

The IOM driver model divides the driver into two parts: a device independent layer and a device-specific layer. This structure makes a driver simple and easy to maintain, and, importantly, separates it from application code so it can be used for different applications.

- **Class driver.** The class driver typically provides serialization and synchronization of multi-threaded I/O requests. Class drivers are device-independent. They interface with the application objects used to perform I/O. In the case of RF5, the class driver interfaces with SIO objects.

- **Mini-driver.** The class driver uses a device-specific mini-driver to operate on a particular device on behalf of the application software. A mini-driver is not unlike a device driver in UNIX, which is basically a table of a few low-level functions such as "open device," "close device," "control device," and "start transfer," along with a callback for "transfer complete."

The class driver used by RF5 is a combination of the SIO module and the DIO adapter. See Section 8.1, *DIO Module—SIO Stream Adapter*, page 58 for more information about DIO.

Several mini-driver libraries are provided with Reference Frameworks. The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 83. For information about porting mini-drivers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616) manual.

Note that RF5 encourages the use of the IOM driver model and DIO adapter but does not enforce it. For example, the *Reference Frameworks for eXpressDSP Software: An RF5 Adaptation Demonstrating 'C64x Video application note describes an RF5-based video application, which uses a simple, low-tech semaphore-based driver model.*

The Reference Frameworks distribution does not include source code for IOM device driver modules. Such files can be obtained as part of the DSP/BIOS Driver Developer's Kit (DDK). You do not need the DDK in order to run the Reference Frameworks—the driver library and public header files are included in the Reference Frameworks distribution. For details about the DDK and mini-driver development and use, see the *DSP/BIOS Driver Developer's Guide* (SPRU616).

8.1 DIO Module—SIO Stream Adapter

The RF5 application uses SIO streams as the main communication path between task threads and devices, so we use an "SIO device adapter to IOM mini-drivers," or DIO. After initialization, the application need only get data from the input stream(s), and store the data to output stream(s), without having to worry about hardware.

The DIO module defines a "class," so there can be as many instances of DIO objects as necessary. The DIO module and the underlying IOM mini-driver provide convenient and platform-independent access to streaming data. DIO is a stream adapter to IOM.
The RF5 application calls only the mini-driver's initialization function directly (via the configuration). Instead of calling mini-driver functions, the application calls the SIO functions SIO_issue and SIO_reclaim. These, in turn, call DIO_issue and DIO_reclaim because the user-defined device (UDEV) object referenced by the SIO object is configured to use DIO functions.

Figure 27 provides an overview of the data streaming path in applications that use DIO.

Figure 27. DIO Data Streaming Path
8.2 Mini-Driver

An IOM mini-driver implements basic I/O functions: opening the device, closing the device, controlling the device, taking a buffer for a transfer to or from the device, and performing a callback from the driver when the transfer is complete.

On the 'C6416 TEB, the interrupt associated with EDMA is used by the RF5 application. The function for the corresponding HWI object (HWI_INT8) runs the ISR function (C6X1X_EDMA_MCBSP_isr) for the mini-driver. This HWI object is configured programmatically by the mini-driver's initialization function. The mini-driver transfers blocks of streaming data, using the codec and CSL modules such as McBSP and EDMA.

The device controller transfers blocks of streaming data, using the codec and CSL modules such as McBSP, DMA, or EDMA. The codec code is isolated in the device controller to make porting easier. Several device controllers are provided with Reference Frameworks.

To port a driver to a different board with the same DSP chip, you need only change the codec initialization functions. For a different DSP chip, changes in the initialization and operation are required as well.

9 Instrumentation

RF5 provides several types of instrumentation you can use to monitor and debug the application.

9.1 Real-time Analysis in DSP/BIOS

DSP/BIOS provides several real-time analysis modules, including LOG for logging messages and STS for computing statistics. However, the 'C64x device on the 'C6416 TEB board does not support RTDX, which is used for real-time target to host data transfer. As a result, instrumentation data provided by DSP/BIOS is available in stop-mode only. That is, they can be seen in Code Composer Studio only if the program halts or reaches a break point.

On most other boards (that is, boards where RTDX is available), LOG and STS instrumentation data is transferred to the host PC by idle functions that run whenever no higher-priority threads are ready to run.

9.2 The UTL Module

Throughout RF5 modules you find UTL module calls—macros beginning with the UTL_ prefix. These are macros that can either be expanded to the code that performs the desired debugging function, or to nothing. In particular, the CHAN, ICC, SSCR, and SCOM modules all make use of the UTL_assert macro.

With conditional expansion of macros to code, you can reduce code size and remove unnecessary functionality in the deployment phase without having to remove development debugging/diagnostics aids.

Syntax for all UTL macros is provided in the Reference Frameworks for eXpressDSP Software: API Reference (SPRA147) application note.
You define the desired amount of debugging features by setting the UTL_DBGLEVEL flag to a desired level in your application’s Build Options. A debugging level includes the class for its level and all the classes from previous levels. You can also individually include or exclude classes regardless of the specified level. Figure 28 shows the debug levels and the features they enable:

![UTL Debugging Levels Diagram]

By default, the RF5 projects are built with UTL_DBGLEVEL set to 70, which enables all types of UTL macros.

For example, you may want some diagnostic messages printed to a LOG object while you develop your application, but not in the deployment phase. Rather than using LOG_printf() and physically removing it from the source code when building a release version (and then possibly having to put it back to debug subsequent releases), you can use the following UTL_logDebug() macro in your code:

```c
UTL_logDebug("Entered local procedure 'encrypt()'" );
```

If you set the variable UTL_DBGLEVEL to 40 or higher, the previous code expands to:

```c
LOG_printf( UTL_logDebugHandle, "Entered local procedure 'encrypt()'" );
```

(The UTL_logDebugHandle variable is a LOG_Handle that points to your default LOG object, which is initialized at startup via UTL_setLogs().) If the UTL_DBGLEVEL macro variable is set to 39 or less, the previous UTL_logDebug() macro expands to nothing.

The UTL module implements the following classes of debugging features:

- **Message severity macros.** Errors, warnings, diagnostic messages, and debugging messages can be sent through macros that expand to LOG_printf()-style calls. The macro names are UTL_logError(), UTL_logWarning(), UTL_logMessage(), and UTL_logDebug().

- **Assertion macro.** Stops execution for a failed assertion and brings the offending source code line to focus in the debugging window. The macro syntax is UTL_assert(<condition>).

---

Figure 28. UTL Debugging Levels
This function is particularly useful in a system where all initialization operations (such as algorithm instantiation) must succeed. Of course, some may fail as you debug applications. So, rather than returning various success/failure codes from initialization functions, we halt the target if a crucial operation fails.

- **Time measurement macros.** These can report execution time, time between two periodic executions of a point in the program, and phase between two periodic points. The statistics are shown in STS objects in CCStudio. The macros are UTL_stsStart() and UTL_stsStop() for execution times, UTL_stsPeriod() for measuring time between periodic executions of a point in the program, and UTL_stsPhase() for measuring phase between two periodic points. The configuration for RF5 contains ten STS objects that can be used for time/period measurements with UTL_sts* functions.

- **XDAIS algorithm diagnostics macros.** XDAIS algorithm memory requirements and heap usage can be reported by macros. The macro names are UTL_showAlgMem() and UTL_showHeapUsage().

Figure 29 shows some example calls to UTL functions and some sample UTL output messages.
10 Adapting the RF5 Application

Adapting RF5 to a custom application can range from changing its XDAIS algorithms to completely replacing all thread and cell code. Typically any adaptation also involves a device driver substitution.

This section looks at some issues that commonly arise when adapting RF5 to your needs:

- Where and how to add new algorithms and cells, channels, tasks, and boards. (Section 10.1.1, page 64)
- How to change the device driver. (Section 10.2, page 66)
- How to efficiently implement data communication between threads running at different rates. (Section 10.4, page 68)

The application-specific source code that glues the RF5 component modules together is provided and is intended as a starting point for the modifications needed by your applications. This source code provides a guideline as to how to adapt the application. You can use it when making design decisions about how to customize and modularize your application.

The following application notes provide in-depth information for some particular domain-specific adaptations:

- Using IDMA2-Based XDAIS Algorithms in eXpressDSP RF5 (SPRA842)
- A Multi-Rate Telephony System on the TEB6416 Using eXpressDSP RF5 (SPRA843)
- A Video, Networking, Audio System on the C64x NVDK Using eXpressDSP RF5 (SPRA844)
- A Multi-Channel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation (SPRA904)

10.1 Adding New Source Modules

When adding new source modules, you can begin by adding new cells and then work your way up to the threads (bottom-up approach). Or, you can begin by creating new channelless tasks that only communicate with one another, and then add channels and cells (top-down approach). If the topology of your application is similar to that of RF5, it may be better to use the bottom-up approach.

The subsections that follow provide overviews for the following types of adaptations:

- Adding New Cells and XDAIS Algorithms (Section 10.1.1, page 64)
- Adding New Channels to Threads (Section 10.1.2, page 65)
- Adding New Threads to the Application (Section 10.1.3, page 65)
- Adding New Boards to the Application (Section 10.1.4, page 66)

When beginning the adaptation process, make a copy of the RF_DIR\apps\rf5 folder tree and call it something like RF_DIR\apps\my_rf5.
10.1.1 Adding New Cells and XDAIS Algorithms

While it's unlikely you will use the FIR and VOL cells and algorithms in your application, you can use them as templates for your own cells. If you are creating your own algorithms, see the TMS320 DSP Algorithm Standard (XDAIS) documentation for information on making them XDAIS-compliant. The following steps show how to create cell wrappers for vendor-provided XDAIS-compliant algorithms (where "MYALG" represents the name of your algorithm):

1. In the RF_DIR\apps\my_rf5\cells folder, make a copy of the fir folder. Rename it myalg.
2. Copy the vendor-provided myalg.c file to this new folder. This file contains the default parameters for the algorithm. Do not modify this file for your application; instead, use dynamic, partial modification of the parameter structure, as demonstrated in the Process thread's setParamsAndStartChannels() procedure.
3. Copy the vendor-provided myalg.h file to the RF_DIR\include folder.
4. Copy the vendor-provided algorithm library files to the RF_DIR\lib folder.
5. Rename cellFir.h to cellMyalg.h in the RF_DIR\apps\my_rf5\cells\myalg folder. This file contains the public interface file for the cell.
6. Edit the cellMyalg.h file as follows:
   - Replace FIR with MYALG (case-sensitively for both uppercase and lowercase).
   - At the top of the file, #include the algorithm's myalg.h file or copy and paste typedefs and macro definitions from that file to cellMyalg.h.
   - Add a cell environment structure if you need one. A cell environment structure may be defined on a per-cell-instance basis to let a cell keep its state, and/or communicate with the thread that contains it. The thread can fill this object with instance-specific information, and can control its actions later by writing data to the environment, and getting information from the cell by reading this object.
     The cell environment may also be used if a cell needs to initialize itself with application-specific constants—for example, the width and height of a video image. Instead of including global application header files like appResources.h (or, worse, hard coding the values), it is better to make cells self-contained. So the cell environment would have fields for width and height, and the thread would set these fields to actual values before initializing the cell. This way, the cell can be reused in another application.
7. Rename cellFir.c to cellMyalg.c. This file contains the implementation of cell code.
8. Edit the cellMyalg.c file as follows:
   - Replace FIR with MYALG (case-sensitively for both uppercase and lowercase).
   - Replace the code in the execute function with code for executing the MYALG algorithm.
   - Add open, control, and close functions to the cell, if necessary, and update the MYALG_FXNS function table.
9. Add cellMyalg.c file to the CCStudio application project.
10. Update the link.cmd file as needed. For example, remove unused libraries and add new ones.

11. Update the setParamsAndStartChannels() function in the thrProcess.c file as needed.

12. Test the cell within your modified version of RF5, in a simple setting first if possible.

10.1.2 Adding New Channels to Threads

Channels are simple cell containers, so you do not need any additional code to create new channels in one or more threads. Simply extend the list of CHAN_Objs to account for the new channels and ICELL_Objs to account for the new cells.

In the default RF5 application, the following declaration is part of the ThrProcess object structure, and NUMCHANNELS is defined in appResources.h:

```
CHAN_Obj chanList[NUMCHANNELS];
```

If your threads have different numbers of channels or if your channels contain different numbers or types of cells, follow the recommendations for cell and channel index naming and enumeration described in Section 6.3, The Process Thread, page 35.

10.1.3 Adding New Threads to the Application

If adding new channels and cells to the existing Process thread is not sufficient to adapt RF5 for your application, you can add a new thread by following these steps:

1. Copy and rename one of the existing folders in the RF_DIR\apps\my_rf5\threads folder. If your new thread performs algorithms or uses channels and cells, make a copy of the Process thread's folder and rename it. If the new thread does not perform algorithms or use channels or cells, copy the RxSplit (or TxJoin) thread as a template instead.

2. If the new thread must share a complex data type (typically for the format of an SCOM message) with another thread, add that new typedef to appThreads.h.

3. Add function calls for new thread's initialization and startup to appThreads.c.

4. Add a new task to call the thread's run() function to the DSP/BIOS configuration.

5. If the thread will receive control messages, enumerate the message commands it should respond to in the thread's header file.

6. If another thread (typically the control thread) needs to send the new thread a message, have it include the new thread's header file.

7. If the new thread should have a different priority than others, make sure that is reflected both in the DSP/BIOS configuration.

8. If the new thread has cells and a different priority, update the "scratch bucket" definition. All threads of a single priority that use XDAIS algorithms fall into the same scratch "bucket." That means their algorithms can share scratch space, because they cannot preempt one another. If the new thread runs at a different priority level, you need to add a new "bucket" for the new priority. Scratch buckets are defined in the appThreads.h file.
10.1.4 Adding New Boards to the Application

You may want your application to work across multiple target boards, sharing as much source code among boards as possible. Each board has platform-dependent files in its respective subdirectory in the RF_DIR\apps\my_rf5\projects directory.

The easiest way to begin porting the application is to copy and rename an existing project, and then create a configuration suitable for the new target. Section 5, Application Configuration and Startup, page 29 describes the additions made to the configuration "seed" (template) for RF5. See Appendix C: Pitfalls to Avoid in Porting RF5 to Other Boards/Targets, page 78 for advice on using the provided DSP/BIOS TextConf script files for making configuration changes.

In addition modify the other files in the folder for your project folder as needed:

- `appIO.c` (This file may be used to initialize and set up drivers used by this project.)
- `link.cmd`
- `app.pjt` (Make sure to modify the compiler settings as needed. You may need or want to modify additional compiler and linker options as well.)

```
-d"CHIP_number"
-i"..\..\projects\target"
```

10.2 Porting the Configuration

As described in Section 5, Application Configuration and Startup, page 29, there are now two methods of configuring DSP/BIOS applications. The traditional method, graphical configuration requires that you create a new configuration file when porting to a different target. The newer method, textual configuration (tconf) allows you to edit and run scripts that define the configuration. Information about porting via textual configuration is provided in the subsection that follows.

10.2.1 Textual Configuration Scripts

Figure 30 shows which folders contain the configuration scripts for RF5.

```
Figure 30. Folders Containing Configuration Scripts
```
• **apps\rf5\appConfig.** Contains application-specific sub-scripts for use with any target. These files statically create and configure instrumentation objects, global properties, application-specific parameters, application thread object(s), and DIO objects. Typically, these scripts need no changes when porting to a new board. The following sample shows the sub-script that creates and configures DSP/BIOS LOG objects:

```tcl
/* ======== appInstrument.tcl ======== */
/* create a LOG object logTrace */
logTrace      = LOG.create( "logTrace" );
logTrace.comment = "default LOG object for application messages";
logTrace.bufLen  = 256;

/* change the size of the LOG_system object to a value large enough */
LOG_system.bufLen = 256;
```

• **apps\rf5\projects\target.** Contains target-specific scripts. The appcfg.tcf file is the top-level script file; it imports target-specific settings in appBoard.tci and the sub-scripts in RF_DIR\apps\rf5\appConfig. Other .tci files in this folder create memory sections, create UDEV driver objects, and set hardware-specific parameters such as the data word size and memory model. The following example shows the first part of appBoard.tci, which loads the appropriate platform and selects the memory model:

```tcl
/* ======== appBoard.tci ======== */
/* Platform-specific portion of the configuration database script */
/* Load the Platform file for the appropriate board */
utils.loadPlatform("Dsk5510");

/* This file makes all the default DSP/BIOS objects known, */
/* plus it defines shortcut javascript variable objects for easy */
/* access to those objects; for example, instead of */
/* prog.module("LOG").instance("LOG_system").buflen = <some value>; */
/* we can simply say */
/* LOG_system.buflen = <value>; */
/* */
/* Call model: use "large" instead of "small" */
/* GBL.MEMORYMODEL = "LARGE"; */
```

To port the configuration to a different target, first make any necessary modifications to the .tci files in your RF_DIR\apps\rf5\projects\target folder. Then, you can generate a new configuration database file (app.cdb) that matches all your edits by running the makeConfig.bat file. This file is provided for each target in the RF_DIR\apps\rf5\projects\target folder. In order to run this file, you must first run the dosrun.bat file in the CCStudio root directory to make the necessary environment settings. The commands to run these files are as follows:

```
cd c:\referenceframeworks\apps\rf5\projects\target
c:\ti\dosrun
makeconfig
```
10.3 Changing the Device Driver

RF5 uses the IOM driver model with the DIO adapter for use with SIO streams to get and send blocks of data from and to the codec. You may use any other model for exchanging data with hardware, in addition to or as a replacement for the IOM model.

If the I/O initialization and the rest of the application should share any constants, those constants should be placed in the appResources.h file.

For details on the IOM driver model, see the *DSP/BIOS Driver Developer's Guide* (SPRU616).

10.4 Implementing Data Communication Among Threads Running at Different Rates

We have seen that an SCOM message that describes a buffer acts like a token. Whichever thread has the token has unimpeded access to the data in the buffer. (In cases where buffers do not need to be aligned, the message itself can contain the data buffer.)

Threads in RF5 all run at the same priority, but the principal use of SCOM messages as tokens (with additional information possibly piggybacked) is the same for threads running at different priorities. What may be different, however, is how messages from different tasks are received.

Consider the following example: the processing task independently gets blocks of data, via SCOM messages, from a video capture task and a network image task:

Figure 31. Waiting on SCOM Messages at Different Rates on Different Queues

The tskCapture thread streams images from the video driver regularly every 50 ms. The tskNetImage thread, on the other hand, sends an image it receives from the network at a rate that varies between 30 and 1000 ms, and runs at a lower priority. Assume the processing task gets either image, processes it as needed, and sends it down the pipeline (or perhaps to an output device). In this arrangement, the Capture-to-Process path is more critical, as it has firmer deadlines.
Assuming the `tskProcess` thread has separate SCOM queues for separate senders ("scomToProcessFromCapture" and "scomToProcessFromNetImage" in this case), it cannot wait on messages for both with a SYS_FOREVER timeout. The reason is that sooner or later it will get stuck waiting on the network image (which can take as much as 20 times longer than getting a captured image), and will miss the real-time deadline for processing captured images. A solution to that problem is to wait with a SYS_FOREVER timeout on the time-critical item—messages from `tskCapture`—and with a 0 timeout on the less critical item—messages from `tskNetImage`.

We can generalize this approach by saying, if a task waits on SCOM messages coming at different rates to different SCOM queues, the task should wait with a SYS_FOREVER timeout only on the queue where the highest-priority (fastest-rate) messages arrive, and with 0 timeout for all others.

A better solution, however, is to eliminate the `tskProcess` thread's second receiving queue and have both the `tskCapture` and `tskNetImage` threads send messages to the single remaining queue. The `tskProcess` thread then waits on the queue with a SYS_FOREVER timeout, but gets a message whenever it arrives from either task. This solution provides the highest parallelism and best CPU utilization. In the previous solution, the `tskProcess` thread can block on the "scomToProcessFromCapture" queue with a SYS_FOREVER timeout, even if data from the `tskNetImage` thread has arrived. In that case, the CPU is not utilized optimally, because data is waiting to be processed, but all tasks in the system are blocked. The `tskCapture` thread is blocked until it gets data from the driver; `tskProcess` is blocked until it gets data from `tskCapture`; and `tskNetImage` is must wait for `tskProcess` to process its data.

The only change needed to make this single-queue scheme to work is to add a sender-ID field at the beginning of each SCOM message structure (of type `ScomBufChannels` and defined in `appThreads.h` in RF5), so that the recipient knows which thread sent the message, how to process it, and where to return it. In fact, senders may even use different formats for their SCOM messages if they all use a sender-ID as the first field. Then, the recipient can get the message, look at the sender-ID, and knowing the sender, cast the message pointer to the proper message format for the sender. SCOM messages are still used as tokens—one token for each buffer in the system—only now a task can receive different tokens through a single channel.
11 Performance and Footprint

The performance and footprint for each framework are required as part of the "Bill of Materials." This allows designers to determine if enough MIPS are available to run their XDAIS algorithms, and if the chip provides enough memory for the framework and algorithms.

11.1 Performance Characteristics

The performance characteristics of RF5 are shown in Table 6. The majority of the time is spent processing the algorithm, which is as it should be. The framework imposes little load on the system. In addition, the device driver accounts for the vast majority of the cycles in the algorithm-stripped CPU load percentage.

Table 6. RF5 CPU Usage Statistics

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF5 as supplied</td>
<td>6%</td>
</tr>
<tr>
<td>RF5, minus FIR and VOL algorithm processing</td>
<td>5%</td>
</tr>
</tbody>
</table>

Load values for the application minus the FIR and VOL algorithm processing were obtained by removing the algorithms and replacing them with simple direct copying of the source buffer to the destination buffer.

Measurements were made under the following conditions:

- Platform: 'C6416 TEB running at 500 MHz
- Sampling rate: 48 kHz
- Samples per frame: 80
- Optimization flags: none
- Debug flags: -g
- UTL_DBGLEVEL: 0
- All critical code and data in internal memory

11.2 Framework Footprint

Although RF5 does not aim to create a minimum-footprint application, footprint is always a concern. The actual RF5 footprint (program and data) and the space available for algorithms are shown in Table 7.

Table 7. RF5 Memory Footprint

<table>
<thead>
<tr>
<th>Configuration</th>
<th>'C5510 DSK</th>
<th>'C6416 TEB</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF5 as supplied</td>
<td>45,848 16-bit words</td>
<td>59,011 16-bit words</td>
</tr>
<tr>
<td>RF5 minus algorithms and application-specific components</td>
<td>17,328 16-bit words</td>
<td>27,993 16-bit words</td>
</tr>
</tbody>
</table>

In this calculation, the subtracted "application-specific components" include algorithms, buffers and heaps used by the algorithms, and unused stack and heap space. For more details, see Appendix A: RF5 Memory Footprint, page 73.
12 Conclusion

Reference Framework Level 5 (RF5) is intended to enable designers to create applications that use many channels and algorithms while minimizing the memory requirements by employing static configuration techniques. The key design goal for RF5 is to be a solid starting point for extensive, high-density applications. RF5 provides more flexibility than lower Reference Framework levels. For example, it channelizes XDAIS algorithms simply and efficiently, automatically calculating shared scratch memory requirements to optimize memory usage.

The framework is supplied as highly reusable C language source code to enable switching between 'C5000 and 'C6000 ISAs. It is packaged as a complete application, running on the 'C5510 DSK and 'C6416 TEB.

This application note includes framework selection criteria, a list of typical applications, explanations of how the framework functions, and references to other application notes that discuss key framework adaptations. This enables designers to quickly determine if RF5 fits their needs, and also to rapidly prototype for end-equipment.

RF5 teaches a methodology for building many systems. Its use of well-established eXpressDSP concepts make it easily reusable for the next project, consequently surpassing the benefits of a homegrown, tailored solution.

13 References

Complete documentation on DSP/BIOS can be found in TMS320 DSP/BIOS User’s Guide (SPRU423) and TMS320C5000 DSP/BIOS API Reference Guide (SPRU404). Documentation for XDAIS is provided in TMS320 DSP Algorithm Standard API Reference (SPRU360). An acquaintance with DSP/BIOS and XDAIS is needed for writing DSP applications at this level.

For additional information, see the following sources:

Product Documentation

- TMS320 DSP/BIOS User's Guide (SPRU423)
- TMS320C5000 DSP/BIOS API Reference Guide (SPRU404)
- TMS320C6000 DSP/BIOS API Reference Guide (SPRU403)
- DSP/BIOS TextConf User's Guide (SPRU007)
- DSP/BIOS Driver Developer's Guide (SPRU616)
- TMS320C54x Chip Support Library API Reference Guide (SPRU420)
- TMS320C55x Chip Support Library API Reference Guide (SPRU433)
- TMS320C6000 Chip Support Library API Reference Guide (SPRU401)
- TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352)
- TMS320 DSP Algorithm Standard API Reference (SPRU360)

Application Notes

- Reference Frameworks for eXpressDSP Software: A White Paper (SPRA094)
- Reference Frameworks for eXpressDSP Software: API Reference (SPRA147)
• Using IDMA2-Based XDAIS Algorithms in eXpressDSP RF5 (SPRA842)
• A Multi-Rate Telephony System on the TEB6416 Using eXpressDSP RF5 (SPRA843)
• A Video, Networking, Audio System on the C64x NVDK Using eXpressDSP RF5 (SPRA844)
• A Multi-Channel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation (SPRA904)
• Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System (SPRA791)
• Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System (SPRA793)
• The TMS320 DSP Algorithm Standard - A White Paper (SPRA581)

Web Resources
• www.dspvillage.com
Appendix A: RF5 Memory Footprint

The overall footprint sizes for RF5 are listed in Section 11.2, Framework Footprint, page 70. This appendix provides details on those numbers. This breakdown assists the system designer in planning an overall DSP solution.

The numbers in Table 8 were obtained on a 'C5510 DSK using the supplied version of this Reference Framework application and its libraries with no additional optimization. The total size is shown in 16-bit words so that it can easily be compared with the totals shown for other Reference Frameworks in their respective application notes.

### Table 8. RF5 Application Footprint

<table>
<thead>
<tr>
<th>Category</th>
<th>Size in 16-Bit Words (hex and decimal)</th>
<th>Run-time Code Size (for RF modules)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DARAM$heap</td>
<td>0x1000 (4096)</td>
<td></td>
</tr>
<tr>
<td>.SARAM$heap</td>
<td>0x2000 (8192)</td>
<td></td>
</tr>
<tr>
<td>.sysstack</td>
<td>0x400 (1024)</td>
<td></td>
</tr>
<tr>
<td>.stack</td>
<td>0x800 (2048)</td>
<td></td>
</tr>
<tr>
<td>UTL</td>
<td>0x276 (630)</td>
<td>0x195 (405)</td>
</tr>
<tr>
<td>SCOM</td>
<td>0xfa (250)</td>
<td>0x72 (114)</td>
</tr>
<tr>
<td>SSCR</td>
<td>0x1de (478)</td>
<td>0x0 (0)</td>
</tr>
<tr>
<td>rts</td>
<td>0x14b (331)</td>
<td>0x24 (36)</td>
</tr>
<tr>
<td>ALGRF</td>
<td>0x1db (475)</td>
<td></td>
</tr>
<tr>
<td>hwiVec</td>
<td>0x80 (128)</td>
<td></td>
</tr>
<tr>
<td>CHAN</td>
<td>0x1a3 (419)</td>
<td>0x49 (73)</td>
</tr>
<tr>
<td>FIR</td>
<td>0x116 (278)</td>
<td>0xc2 (194)</td>
</tr>
<tr>
<td>VOL</td>
<td>0xca (202)</td>
<td>0x80 (128)</td>
</tr>
<tr>
<td>CELL</td>
<td>0xc8 (200)</td>
<td>0xa2 (162)</td>
</tr>
<tr>
<td>ICC</td>
<td>0x21 (33)</td>
<td>0x0 (0)</td>
</tr>
<tr>
<td>application</td>
<td>0xd58 (3416)</td>
<td></td>
</tr>
<tr>
<td>driver</td>
<td>0x588 (1416)</td>
<td>0x2ca (714)</td>
</tr>
<tr>
<td>taskStacks</td>
<td>0x1900 (6400)</td>
<td></td>
</tr>
<tr>
<td>CSL</td>
<td>0xc14 (3092)</td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td>0x26d (621)</td>
<td></td>
</tr>
<tr>
<td>STS</td>
<td>0xa3 (163)</td>
<td></td>
</tr>
<tr>
<td>RTDX</td>
<td>0x806 (2054)</td>
<td></td>
</tr>
<tr>
<td>DSP/BIOS kernel</td>
<td>0x26ae (9902)</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0xb318</strong> (45848)</td>
<td></td>
</tr>
</tbody>
</table>

However, components such as the FIR algorithm are not included in your end-application. Therefore, to determine whether your XDAIS algorithms can fit in a certain DSP’s on-chip memory, we first need to subtract the size of components to be removed. Table 9 shows how the total size of the basic framework code and data is calculated by subtracting various application-specific portions:
Table 9. Calculating RF5 Basic Framework Size

<table>
<thead>
<tr>
<th>Total Size</th>
<th>45,848 16-bit words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused internal heap</td>
<td>minus 3300</td>
</tr>
<tr>
<td>Unused external heap</td>
<td>minus 8088</td>
</tr>
<tr>
<td>Application buffers</td>
<td>minus 1040</td>
</tr>
<tr>
<td>Heaps used by algorithms</td>
<td>minus 365</td>
</tr>
<tr>
<td>VOL algorithm</td>
<td>minus 202</td>
</tr>
<tr>
<td>FIR algorithm</td>
<td>minus 278</td>
</tr>
<tr>
<td>FIR coefficient table</td>
<td>minus 96</td>
</tr>
<tr>
<td>Cell algorithm code</td>
<td>minus 200</td>
</tr>
<tr>
<td>Unused stack and SysStack</td>
<td>minus 2622</td>
</tr>
<tr>
<td>Unused task stack</td>
<td>minus 5901</td>
</tr>
<tr>
<td>LOG object for debugging</td>
<td>minus 621</td>
</tr>
<tr>
<td>STS object for debugging</td>
<td>minus 163</td>
</tr>
<tr>
<td>UTL debugging module</td>
<td>minus 630</td>
</tr>
<tr>
<td>RTDX module</td>
<td>minus 2054</td>
</tr>
<tr>
<td>All cinit records</td>
<td>minus 2960</td>
</tr>
<tr>
<td><strong>Resulting Basic Framework Size</strong></td>
<td><strong>17,328 16-bit words</strong></td>
</tr>
</tbody>
</table>

The .cinit records are subtracted from the footprint here because in a production application, the .cinit records would generally reside only in ROM, thus not affecting RAM requirements.

The result is a total of 17,328 words on the 'C5510, which is the basic footprint for the supplied Reference Framework Level 5.

When calculating whether your algorithms can fit on a particular DSP, use the memory numbers provided by your algorithm vendors. XDAIS algorithm vendors must supply the following footprint sizes:

- Persistent / Scratch Data Memory (Rule 19)
- Stack Space Memory (Rule 20)
- Static Data Memory (Rule 21)
- Program Memory (Rule 22)

This data is provided for the FIR algorithm in SPRA791 and for the VOL algorithm in SPRA793.
Appendix B: Comparing RF3 and RF5 Performance

This appendix lists the results and conclusions from a performance comparison of RF3 and RF5.

Performance Comparison Setup

Comparison tests were run on the same board at different frequencies and different frame sizes.

Both RF3 and RF5 were ported to the DSK6711 with the AIC23 EVM for the tests. The clock speed of the C6711 is 150 MHz. The AIC23 codec is stereo and the frequency is configurable. For this document, a frame is comprised of N left and right 16-bit samples. So a frame size of 32 means the frame has 32 left 16-bit samples and 32 right 16-bit samples.

CCStudio 2.12 and Reference Frameworks v2.0 were used for the tests.

As part of porting RF3 and RF5 to the DSK6711 with the AIC23 EVM, the following changes were made to both applications to give more realistic benchmark numbers:

- All libraries and applications were compiled with –o2 optimization.
- Set UTL_DBGLEVEL=0 in libraries and applications that used UTL calls.
- Configured L2 for 3-way cache.
- All SWIs in RF3 had the same priority. All TSKs in RF5 had the same priority.
- The following items were in internal memory:
  - DSP/BIOS Objects
  - All entries in the BIOS Data tab (includes .stack)
  - .far and .bss
  - PIP and SIO buffers
  - task stacks (RF5).

The following additional changes were made to RF5 to allow placement of the above items into internal memory:

- Reduced task stack size to 0x800.
- Reduced system stack size to 0x130.
- Reduced INTERNALHEAP size to 0x1000.

The CPU Load Graph was used to collect all values. To make the CPU Load Graph more accurate, “Disable All” was done on the RTA Control Panel.

Note that a small, fixed amount of CPU overhead exists for the tests. Included in this overhead are the application control, system clock, and RTDX processing.
Performance Comparison Test Results

To obtain CPU load values for the RF3 and RF5 frameworks, the FIR and VOL algorithms commented out for the tests. The splitting and joining of the stereo samples was also removed.

The resulting numbers represent the framework-only overhead. This includes drivers, application thread context-switches, DSP/BiOS context-switches (for example, HWI dispatcher and IDL), inter-thread communication, and execution logic.

Table 10. CPU Load Percentages at Various Frame Sizes and Sample Rates

<table>
<thead>
<tr>
<th>kHz</th>
<th>Samples/Frame</th>
<th>Frames/Sec</th>
<th>RF3</th>
<th>RF5</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>128</td>
<td>62</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>125</td>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>250</td>
<td>2.0</td>
<td>3.1</td>
</tr>
<tr>
<td>44.1</td>
<td>128</td>
<td>345</td>
<td>2.4</td>
<td>3.2</td>
</tr>
<tr>
<td>44.1</td>
<td>64</td>
<td>689</td>
<td>5.1</td>
<td>5.6</td>
</tr>
<tr>
<td>44.1</td>
<td>32</td>
<td>1378</td>
<td>8.3</td>
<td>11.0</td>
</tr>
<tr>
<td>96</td>
<td>128</td>
<td>750</td>
<td>5.5</td>
<td>6.1</td>
</tr>
<tr>
<td>96</td>
<td>64</td>
<td>1500</td>
<td>8.8</td>
<td>11.2</td>
</tr>
<tr>
<td>96</td>
<td>32</td>
<td>3000</td>
<td>15.4</td>
<td>22.3</td>
</tr>
</tbody>
</table>

With data processing removed, the lines in Figure 33 overlap at different frequencies for RF3 (and similarly for RF5). So running at 44.1 kHz with a frame size of 32 (1378 frames/second) is basically the same as running at 96 kHz with a frame size of 64 (1500 frames/second) from a framework-only perspective.

Figure 33. Graph of CPU Load Percentage for Various Settings
CPU load percentages may be converted to time values by multiplying by the real-time deadline. For example, on a G.729-based system with a 10 ms real-time deadline and an 8 kHz telephony sample rate, the RF5 overhead for 80-sample buffers is 1.4% CPU load (by interpolating from the graph). This translates to 0.14 ms (10 ms * 1.4% = .14 ms), which leaves 9.86 ms to execute the G.729 algorithms and other application processing.

**Performance Comparison Conclusions**

While the test results cited here are for the generic RF3 and RF5 audio applications on the C6711 DSK, several generalizations can be made about RF3 and RF5:

- RF3 and RF5 frameworks are frame-size independent. The CPU loads are approximately the same when the frames/second are the similar (for example, 44.1 kHz at 64 samples per frame and 96 kHz at 128 samples per frame).

- At lower frequencies (for example, 8 kHz on the C6711 DSK) there is not a significant difference between RF3 and RF5 from a performance standpoint.

- At higher frequencies (for example, 96 kHz on the C6711 DSK), attempts should be made to minimize the number of SWIs and TSKs to avoid context switches and inter-thread communication.

- At higher frequencies (for example, >96 kHz on the C6711 DSK), moving some amount of data processing (e.g. splitting and joining) to DMA is advised.

- The SIO/TSK modules have more overhead than the PIP/SWI modules.

- When collecting CPU Load values, you should use “Disable All” on the RTA Control Panel. This will give a more accurate CPU Load Graph value. On RF5 (framework-only, with a single thread) running at 96 kHz with a frame size of 32, the value was 17.6% with everything enabled on the RTA Control Panel, and 14.3% when everything was disabled.

- A separate test involved placing critical code sections explicitly in internal memory instead of relying on the cache. There was minimal improvement for RF3 and a very slight improvement for RF5. The conclusion is that for these applications, there was little L2 cache thrashing.
Appendix C: Pitfalls to Avoid in Porting RF5 to Other Boards/Targets

Reference Frameworks are consciously designed with portability in mind. Although initially provided as a complete application on the ‘C6416 TEB, users should be able to port RF5 to their ISA and board with a minimum of work. Steps taken to ensure this when designing RF5 include:

- The framework and all its modules are 100% C source code based.
- Size computations are typically done on a framework-defined "Sample" type. For example the size in bytes for a frame is FRAMELEN * sizeof( Sample ).
- Framework modules are supplied and tested on C54 near and far models, C55 small and large data models, C62, and C64 architectures.

However, some pitfalls still exist. The most common issues to avoid are described in the following list:

- **Porting the configuration.** It can be a tedious, and more critically, error-prone task to create a DSP/BIOS configuration file (.cdb) by copying object settings from a supported board to your own hardware platform. To ease this process, DSP/BIOS TextConf scripts (.tcf and .tci) are provided with RF5. When porting to another target, you may modify the appBoard.tci script and the scripts it imports for memory and codec configuration. Then, run the script to generate a new configuration file. For details on DSP/BIOS TextConf, see the DSP/BIOS TextConf User's Guide (SPRU007).

- **DMA to external memory.** Section 3.4.1 of the C54x Enhanced Peripherals User Guide (SPRU302) states that the ‘C5402 DMA supports internal accesses only. If you wish to port RF5 to a C54 board, you must place the bufRx[] and bufTx[] driver buffer data in internal memory. This can be done as in the following code example:

  ```c
  #pragma DATA_SECTION(bufRx, "INTPROCBUFF")
  
  The linker command file must then specify the placement as follows:

  ```c
  SECTIONS
  |
  \ . INTPROCBUFF: {}
  \ => IDATA
  ```

  Alternatively, simply assign the entire .bss data section to internal memory via the MEM properties specified via that DSP/BIOS Configuration Tool.

- **Cache coherency issues.** The ‘C6416 is blessed with 1 MB of fast on-chip RAM. The ‘C6711 however, has only 64 KB of internal memory. The L1, L2 cache mechanism (present in both ‘C6x1x and ‘C64x architectures) brings off-chip execution speeds close to internal benchmarks, however some care must be taken. Thankfully, the c6x1x_edma_mbsp driver cleans and flushes the cache if necessary. However, the user is still responsible for aligning bufRx[] and bufTx[] on a 128-byte boundary, and ensuring their size is a multiple of 128 bytes as follows:
// mark buffer as external and align it on 128-byte boundary
#pragma DATA_SECTION(bufRx, "EXTERNCBUFF")
#pragma DATA_ALIGN(bufRx, 128)

// Buffers and the SIO object for the input device
#define RXFRAMELEN 256 // frameLen a multiple of 128
#define NUMINPUTBUFFERS 2 // double buffering for SIO
static Sample bufRx[NUMINPUTBUFFERS][NUMCODECHANS * RXFRAMELEN];

To reiterate, this is required only if the buffers are placed in external memory. You can avoid the problem altogether by placing them in internal memory.

- **TLC320ADxxx codecs and the least-significant-bit problem.** The AD50 and AD535 codecs on the 'C5402 DSK and 'C6711 DSK boards, require that you clear the lowest bit in each sample output to the codec. If you fail to do this, the codec interprets the data as a command when the LSB equals 1. The codec then goes into a state that the DSP code is not expecting. To workaround this, simply add the following code:

```c
/*
 * Mask off the low bit for compatibility with codecs
 * that interpret a low bit of '1' as a command flag.
 */
for (i = 0; i < FRAMELEN; i++) {
    /* AD50 and AD535 codecs must have LSB set to 0 */
    dstBuf[i] = srcBuf[i] & 0xfffe;
}
```

- **Callback arguments and 'C55x large data model arguments.** On most TI ISAs (the 'C54x for example), there is pointer-scalar equivalence. That is, an argument passed to say, a DSP/BIOS task, can be treated as a generic argument of type Void * because a 16-bit value of type integer could not be distinguished from a value of type Void *. In addition, the location of the passed parameter remains constant. The 'C55x has a more efficient allocation strategy, which differentiates pointer quantities from integer ones. However, the impact on task arguments for example, is that the user must treat types precisely, as follows:

```c
Void rf5Task(Arg id_arg)
{
    Int id = ArgToInt(id_arg);
    ...
}
```

Note that this code is still ISA-portable. ArgToInt is a macro that does its required job on 'C55x and expands to a simple cast on all other architectures.

- **RTDX support.** The 'C6416 PG 1.0 silicon used on the TEB does not support RTDX. Therefore, DSP/BIOS real-time analysis windows such as the CPU Load Graph cannot be used. Users porting RF5 to 'C55x architectures should also be aware that the use of RTDX in DMA-driven applications is not advised. Bug SDSsq20116 (AIC27 Codec transfers over DMA in 'C5510 EVM halts when RTDX is turned on) has been committed for CCS 2.2.

Note that DSP/BIOS Execution Graph, LOG and Statistics displays can still be viewed in stop mode when RTDX is not used. In stop mode, the analysis windows are updated only when the target reaches a breakpoint or is halted.
• Allocating from address 0x0. On the 'C6x1x and 'C64x DSP architectures, internal memory starts at address 0x0. Thus, when porting RF5 to another architecture—for example, the 'C6711 DSK—you should take precautions against dynamic memory allocation at address 0x0. Otherwise, a "valid" buffer beginning at address 0x0 is indistinguishable from a "bad" NULL pointer. Bug SDSsq27062 (DSP/BIOS should not allocate IRAM heap from address 0x0) has been filed to address this.

The workaround for this issue is simple. Put something else in the internal memory section in addition to the heap. For example, place the system stack in the internal memory section.
Appendix D: Converting RF5 from LIO to IOM Drivers

In this release of Reference Frameworks, RF5 has been modified to use the IOM-based mini-driver model described in the *DSP/BIOS Driver Developer's Guide* (SPRU616) manual. Previously, RF5 used the LIO driver model described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note, which is now obsolete.

If you created an RF5-based application using an earlier release of Reference Frameworks, you are encouraged to convert to IOM-based mini-drivers. To perform such a conversion, follow these steps.

1. Copy the driver-related files for your platform from `RF_DIR`\include and `RF_DIR`\lib to the corresponding folders for your application. For example, copy dsk5510_dma_aic23.h and dsk5510_dma_aic23.l55 if you are using the AIC23 codec on the 'C5510 DSK. You do not need to copy folders in `RF_DIR`\src, since source files for drivers are no longer provided with Reference Frameworks. For source files, download the DSP/BIOS Device Driver Developer's Kit.

2. Copy the appIO.h file for your platform from `RF_DIR`\apps\rf5\projects\target to the corresponding folder for your application. Remove the old `RF_DIR`\apps\rf5\threads\main\appIO.h file from the project. The new appIO.h file contains a target-specific setting for the NUMCODECCHANS constant.

3. Add the following line to `RF_DIR`\apps\rf5\threads\main\appResources.h:

   ```c
   #include appIO.h
   ``

4. Modify `RF_DIR`\apps\rf5\projects\target\appIO.c to contain the following code. The header file includes and initialization previously performed by this file are now performed by the mini-driver initialization function. In the next step, you will configure a user-defined device object to call this function.

   ```c
   #include <std.h>
   Void appIOInit() {
   /*
   * This function can be used to initialize and setup drivers
   * used by this project.
   */
   }
   ```

5. Open the configuration file (app.cdb) for your project and make the following changes:

   - In the Scheduling folder, locate the HWI object or objects that called driver ISR functions (for example, C6X1X_EDMA_MCBSP_isr). Some targets used one HWI object and others used two. This part of the configuration is no longer needed because the mini-driver configures its ISR functions programmatically in the initialization function. For each object used, do the following:

     - In the General tab, set the function property to HWI_unused.
     - In the Dispatcher tab, remove the checkmark from the Use Dispatcher property.
– In the Input/Output folder, create a User-Defined Device under the Device Drivers category. Rename the new object "udevCodec".

  - Device Drivers
  - User-Defined Devices
  - udevCodec

– Set the properties of the udevCodec object to values similar to those listed below. If you are using a different target, use the initialization function and function table pointer for the mini-driver you are using. These names can be found in the header file for your mini-driver, which is in the `RF_DIR\include` folder.

  - init function: _TEB6416_EDMA_PCM3002_init
  - function table ptr: _TEB6416_EDMA_PCM3002_FXNS
  - function table type: IOM_Fxns
  - device id: 0x0
  - device params ptr: _TEB6416_DEVPARAMS
  - device global data ptr: 0x0

  The mini-driver’s initialization function (for example, TEB6416_EDMA_PCM3002_init) configures the mini-driver’s ISR functions programmatically by calling HWI_dispatchPlug.

– In the Input/Output folder, create a DIO object under the Device Drivers category. Rename the new object "dioCodec". Set the following properties for this object.

![dioCodec Properties](image)

  - Save your changes to the configuration.

6. Copy the `RF_DIR\apps\rf5\projects\target\target_devParams.c` file to the corresponding folder of your project. Add this source file to your project in CCStudio.

7. Remove the library for the DLIO module from your linker command file (`RF_DIR\apps\rf5\projects\target\link.cmd`) or project.
Appendix E: Reference Framework Board Ports

As of the publication date of this application note, the board-specific portions of the Reference Framework applications have been ported to the following boards:

Table 11. Boards Supported by Various Reference Frameworks

<table>
<thead>
<tr>
<th>Board</th>
<th>Codec</th>
<th>RF1</th>
<th>RF3</th>
<th>RF5</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5402 DSK</td>
<td>TLC320AD50 mono codec</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Large installed base.</td>
</tr>
<tr>
<td>C5416 DSK</td>
<td>PCM3002 stereo codec</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5510 DSK</td>
<td>TLV320AIC23 stereo codec</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>All three frameworks support DSK 5510.</td>
</tr>
<tr>
<td>C6x11 DSK</td>
<td>TLC320AD535 mono codec</td>
<td>✔</td>
<td></td>
<td></td>
<td>Large installed base.</td>
</tr>
<tr>
<td>C6416 TEB</td>
<td>PCM3002 stereo codec</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Boards may be obtained at the DSPvillage eStore (http://dspestore.ti.com).

Additional device controllers may be added in the future. See the readme.txt files in the Reference Frameworks folder tree for updated information. For the latest information regarding issues you may encounter, see the Release Notes provided in the Reference Frameworks area of the DSPvillage website (http://www.dspvillage.com).

For details about how to implement device controllers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616).

Reference Framework configurations refer directly to the mini-driver's initialization function only. It is the job of the DIO adapter to use other mini-driver functions. This lets the application read blocks of data from the codec using simple SIO stream API functions.
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated