Application Report
SPRA796A – April 2004

Reference Frameworks for eXpressDSP Software: RF6, A DSP/BIOS Link-Based GPP-DSP System

Todd Mullanix, Vincent Wan, Arnie Reynoso,
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 6 (RF6) extends the RF5 application by adding a link to a General Purpose Processor (GPP). It retains the same support for high-density applications with many channels and algorithms provided by RF5.

RF6 leverages DSP/BIOS Link, the new standard in TI DSP-GPP communication. The framework defines a methodology for using DSP/BIOS Link both for data streaming (such as from a GPP filesystem to the DSP) and for control and status messaging (such as to send button-press events).

RF6 is initially supplied on OMAP 591x platforms that consist of a dual-core device integrating a TI-enhanced ARM and a 'C55x DSP.

This application note first provides an overview of Reference Frameworks. It then explains how to install, run, explore, and adapt Reference Framework Level 6.

Contents

Figures

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 Code Composer Studio, 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 characteristics of Reference Frameworks available as of the release of this document.

Table 1. Reference Framework Characteristics by Level

Design Parameter	RF1, Compact	RF3, Flexible	RF5, Extensive	RF6, Connected
Absolute minimum memory footprint	✓	✗	✗	✗
Static configuration	✓	✓	✓	✓
Dynamic object creation (e.g., DSP/BIOS objects)	✗	✗	✓[1]	✓[1]
Static memory management	✓	✓	✓	✓
Dynamic memory allocation	✗	✓	✓	✓
Number of channels recommended	1-3+	1-10+	1-100+	1-100+
Number of eXpressDSP-compliant algorithms recommended	1-3+	1-10+	1-100+	1-100+
Uses DSP/BIOS real-time analysis	✓[2]	✓	✓	✓
Uses DSP/BIOS scheduling kernel	✗[3]	✓[4]	✓	✓
Uses Chip Support Library	✓	✓	✓	✓[5]
Uses XDAIS algorithms	✓	✓	✓	✓
Portable to other devices, ISAs, boards	✓	✓	✓	✓
Supports multiple execution rates and priorities	✗	✓[6]	✓	✓
Supports thread blocking	✗	✗	✓	✓
Implements control functionality	✗	✓	✓	✓
Implements DSP-GPP functionality	✗	✗	✗	✓

Notes:

1. Dynamic object creation is supported in RF5 and RF6, but static configuration is preferred for simplicity and footprint reasons.
2. DSP/BIOS Real-Time Analysis is used in RF1 in stop-mode only. RTDX is not configured by default.
3. Only HWI and IDL threads are used in RF1.
4. TSK threads are not used in RF3.
5. The DSP-side RF6 application uses an IOM codec driver that leverages the new CSL 3.x software architecture.
6. RF3 supports only a single rate per channel.

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.

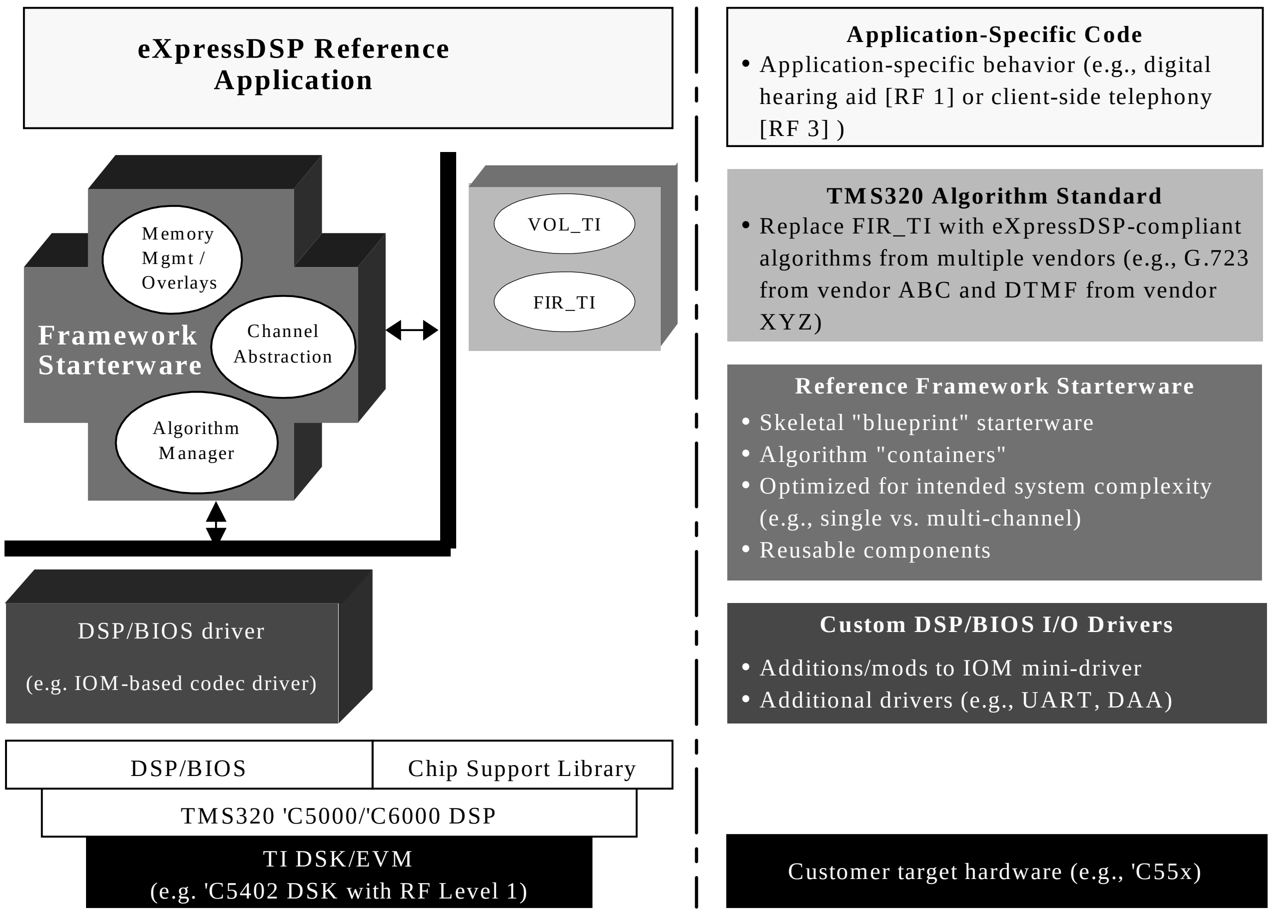

eXpressDSP Reference Application
Memory Mgmt / Overlays
Framework Starterware
Channel Abstraction
Algorithm Manager
VOL_TI
FIR_TI
DSP/BIOS driver
(e.g. IOM-based codec driver)
DSP/BIOS
Chip Support Library
TMS320 'C5000/'C6000 DSP
TI DSK/EVM
(e.g. 'C5402 DSK with RF Level 1)
Application-Specific Code
• Application-specific behavior (e.g., digital hearing aid [RF 1] or client-side telephony [RF 3])
TMS320 Algorithm Standard
• Replace FIR_TI with eXpressDSP-compliant algorithms from multiple vendors (e.g., G.723 from vendor ABC and DTMF from vendor XYZ)
Reference Framework Starterware
• Skeletal "blueprint" starterware
• Algorithm "containers"
• Optimized for intended system complexity (e.g., single vs. multi-channel)
• Reusable components
Custom DSP/BIOS I/O Drivers
• Additions/mods to IOM mini-driver
• Additional drivers (e.g., UART, DAA)
Customer target hardware (e.g., 'C55x)

Figure 1. Reference Frameworks for eXpressDSP Software and Entry Points

Figure 1 shows the architecture of a Reference Framework. The boxes on the left show the supplied framework components. For each component, there are entry points you can use to modify the reference application. The right column contains boxes with corresponding shades of gray that describe modifications that can be made to the supplied components. These include application behavior changes, algorithm replacement, driver modification, and hardware modification.

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 DSK. 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 development platform.
- 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.
- An API Reference Manual for new module libraries introduced in the Reference Frameworks.

2 Why Use Reference Framework Level 6?

Reference Framework Level 6 (RF6) is intended to enable designers to create applications on systems containing a DSP connected to a General Purpose Processor (GPP).

RF6 is an extension to RF5, which provides support for a more extensive set of channels and algorithms than lower-numbered Reference Frameworks. RF5 and RF6 can be used in applications that have complex interdependencies between threads. The addition of a GPP to the system in RF6 makes the need for support of interdependencies even more important.

This application note describes the extensions made to RF5 to create RF6. For details how both frameworks handle channels, cells, algorithms, and thread synchronization, see the application note for RF5, SPRA795.

By default, RF6 processes an incoming stereo audio signal on the DSP, then sends the data to the GPP, which sends the data back to the DSP for further processing. Finally, it sends the output to the output codec. A control thread on the GPP controls algorithm parameters such as volume and high-/low-pass filtering on the DSP.

RF6 is structured to be easily adaptable. For example, for “decode only” systems such as Internet Audio Players, the developer can chop out the A/D input and the DSP to GPP channel, integrate a XDAIS decoder, and have a simple prototype up and running.

Although the default application is a simple audio application, RF6 is well suited for video and other applications. The default behavior simply provides a basis for adapting RF6 to other applications. The application logic can easily be modified.

2.1 Key Features of RF6

RF6 provides the following key feature that is not part of RF5:

- **DSP-GPP connectivity.** The GPP can control DSP activities and perform additional processing. Bootloading the DSP, data streaming, and messaging are all supported.

In addition, RF6 contains the following key features that are part of RF5:

- **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.
- **Structured thread-safe control mechanism.** The Control thread interacts with the outside world, for example by parsing control messages from the GPP and notifying the other DSP threads.
- **Easy replacement of I/O drivers.** The IOM model allows alternate mini-drivers to be connected to the DIO adapter used by RF6. Note that DSP/BIOS Link is also implemented as an IOM driver on the DSP side. This allows drivers to be plugged in interchangeably.

2.2 Characteristics of RF6 Applications

Table 2 shows the characteristics of RF6 in more detail than Table 1. It should be used as a guide to determine whether or not RF6 is suitable as the basis of your end-application. However, if your DSP application must connect to a GPP, use RF6 as the foundation even if your algorithm and channel needs are more modest than the extensive capabilities supported by RF6.

Table 2. RF6 Application Characteristics

Design Parameter	RF6, Connected	Notes
Static configuration	✓	
Dynamic object creation (e.g., DSP/BIOS objects)	✓	Although the default RF6 application creates most objects statically, the framework does not preclude the dynamic creation of objects. Static configuration is preferred for simplicity and footprint reduction.
Static memory management	✓	
Dynamic memory allocation	✓	
Number of channels recommended	1-100+	
Number of eXpressDSP-compliant algorithms recommended	1-100+	
Absolute minimum memory footprint	✗	
Uses DSP/BIOS real-time analysis	✓	
Uses DSP/BIOS scheduling kernel	✓	Uses primarily TSK and HWI threads. Use of SWI threads is not precluded.
Uses Chip Support Library	✓	The DSP-side IOM codec driver leverages the new CSL 3.x software architecture.
Uses XDAIS algorithms	✓	
Portable to other devices, ISAs, boards	✓	
Supports multiple execution rates and priorities	✓	
Supports thread blocking	✓	
Implements control functionality	✓	
Implements DSP-GPP functionality	✓	This is the key feature of RF6.

RF6 leverages foundation software, such as DSP/BIOS, the Chip Support Library (CSL), and XDAIS. It leverages DSP/BIOS Link, which enables DSP-GPP connectivity, by providing services such as messaging and data streaming between the DSP and the GPP. It also calls on the services of various Reference Framework modules. For example, the UTL module is used for debugging and diagnostics.

2.3 Applications Suited to RF6

Typical applications suited for use with RF6 are devices that play or record audio, visual, or other signals. They also give users more extensive control over the functions being performed than typical DSP-only applications. A separate GPP is used for to provide such control.

Although memory usage is always an issue for DSP applications, applications suited to RF6 typically use DSPs with more memory than low-end DSPs.

All Reference Frameworks are application-agnostic. Each framework can be adapted for use in many applications, including telecommunication, audio, video, and more. Applications for which RF6 can be adapted include:

- Personal data assistants (PDAs), including those with audio record and playback
- Mobile phones, including those with games and cameras
- Telematics applications such as hands-free audio and voice systems,
- Wireless devices with support for Bluetooth and Wi-Fi
- Digital still and video cameras
- Advanced speech applications (text-to-speech, speech recognition)
- MP3 players
- Various data processing devices, including fax, encryption/decryption, and authentication

There are a growing number of small devices that combine two or more functions like these. RF6 is suited as foundation software for many devices with increasing consumer demand.

Figure 2 shows a block diagram for a distributed speech-enabled application. The application runs on the GPP processor, while the compute-intensive speech recognizer and portions of the text-to-speech (TTS) software run on the DSP.

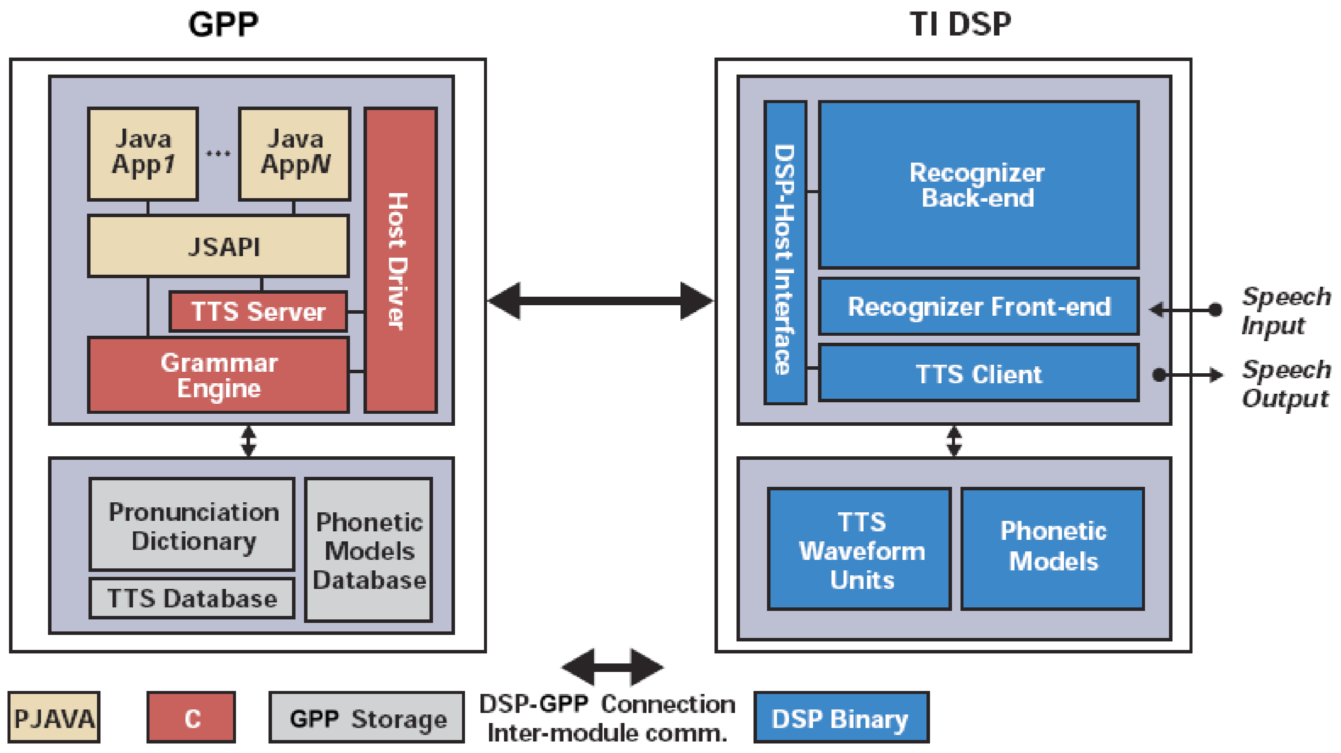


Figure 2. Speech-Enabled Application System

2.4 What RF6 Is and Is Not

When selecting a Reference Framework as the basis for an application, it is important to know both what is included and what is not included. Table 3 should help you determine whether RF6 is suitable as the blueprint for your end application.

Table 3. RF6 High-Level Criteria

RF6 Is	RF6 Is Not
Connected framework based on DSP/BIOS Link and RF5 (see below)	A framework that, like DSP/BIOS Bridge, allows DSP content to be downloaded dynamically to the target
Uses DSP/BIOS Link for GPP–DSP communication	Dynamic loading
Minimum set of GPP APIs (not DSP/BIOS Bridge)	Extensive GPP APIs like Resource Manager and Node Database
GPP and DSP Messaging APIs for Control & Status threads	Code overlays
Includes reference DSP and GPP applications on OMAP591x Target hardware with audio codec IOM driver	Hard-wired to the OMAP591x architecture (see below)
Leverages textual configuration (TCONF) for ease of porting to other ISAs	Supplied with any GUI
Leverages Catalog OMAP application notes as Target hardware, Linux, OMAP references (see below)	A complete “Getting Started” suite for OMAP applications.
Uses OMAP Chip Support Library v3.x on DSP-side (see below)	Targeted at C54x platform
	A design that minimizes footprint (as in RF1)
	Bundled with Code Composer Studio

Some of these criteria warrant further explanation:

- **Is a connected framework based on DSP/BIOS Link and RF5.** RF6 introduces few new modules. RF6 is essentially RF5 plus DSP/BIOS Link and a more rigorous messaging scheme. RF6 intentionally does not include some of the more advanced features in DSP/BIOS Bridge for wireless applications. For example, DSP/BIOS Bridge enables new code to be dynamically loaded at runtime. While there is nothing to prevent system integrators from adding this to RF6, no support exists in the base package.

- **Is not hardwired to the OMAP591x architecture.** RF6 is largely based on RF5, which currently runs on several C55x and C6000 platforms. Therefore, the target-side RF6 code is not hard-wired to the OMAP591x architecture.

  A port to another platform (for example a C64x platform) would require no changes to the DSP source code. Instead, the system integrator would simply link in new DSP/BIOS Link and codec IOM drivers, and modify the application configuration.

- **Leverages Catalog OMAP application notes as target hardware, Linux, OMAP references.** As part of the RF6 project plan, a partnership with the TI Catalog OMAP Applications Group was established. Additional documents have been created to support the tools needed to run RF6, such as the Linux kernel and the bootloader. See Section 3.3, *Recommended Reading* for details.

- **Uses the OMAP Chip Support Library v3.x on DSP-side.** This is perhaps somewhat misleading. Neither the RF6 example application nor its modules use the Chip Support Library (CSL) directly. Interaction is confined to the DSP/BIOS IOM codec device driver as shown in Figure 3. All IOM drivers leverage the Chip Support Library where applicable.

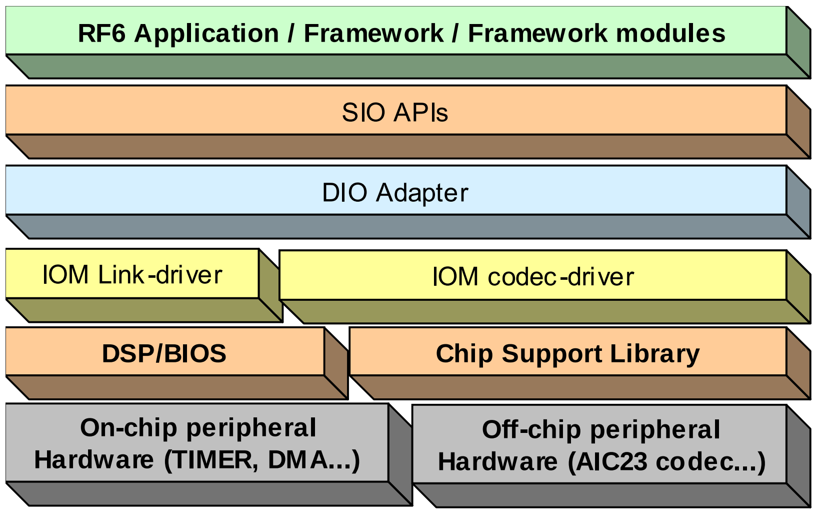

RF6 Application / Framework / Framework modules
SIO APIs
DIO Adapter
IOM Link-driver
IOM codec-driver
DSP/BIOS
Chip Support Library
On-chip peripheral Hardware (TIMER, DMA...)
Off-chip peripheral Hardware (AIC23 codec...)

Figure 3. Chip Support Library in the System Hierarchy

Note that the CSL APIs have adopted a new software architecture, which is used in the IOM codec driver used by RF6. This is important to you only if you are porting an existing C55x IOM driver codebase to the OMAP591x. The new CSL 3.x architecture is not backward compatible with previous CSL APIs. The new APIs provide several improvements, including better resource management, unified error handling, and well-defined interfaces.

3 RF6 Minimum System Requirements and Prerequisites

Installing and running the application is more involved for RF6 than for other Reference Frameworks. There are more processors (the DSP and the GPP), more development hosts (a Windows PC and a Linux box), and significantly more tooling. It is therefore essential to precisely define the minimum system requirements for running and leveraging RF6.

Details regarding each of the components can be found in the documents referenced in Table 4. For example, instructions for installing DSP/BIOS Link in MontaVista Linux are available in the *DSP/BIOS Link for MontaVista Linux (MVL) User's Guide*.

For further assistance in setup, please contact your OMAP Technology Center (OTC) provider or local TI representative.

3.1 Hardware

Figure 4 shows the typical hardware setup used for the provided implementation of RF6 on the OMAP platform.

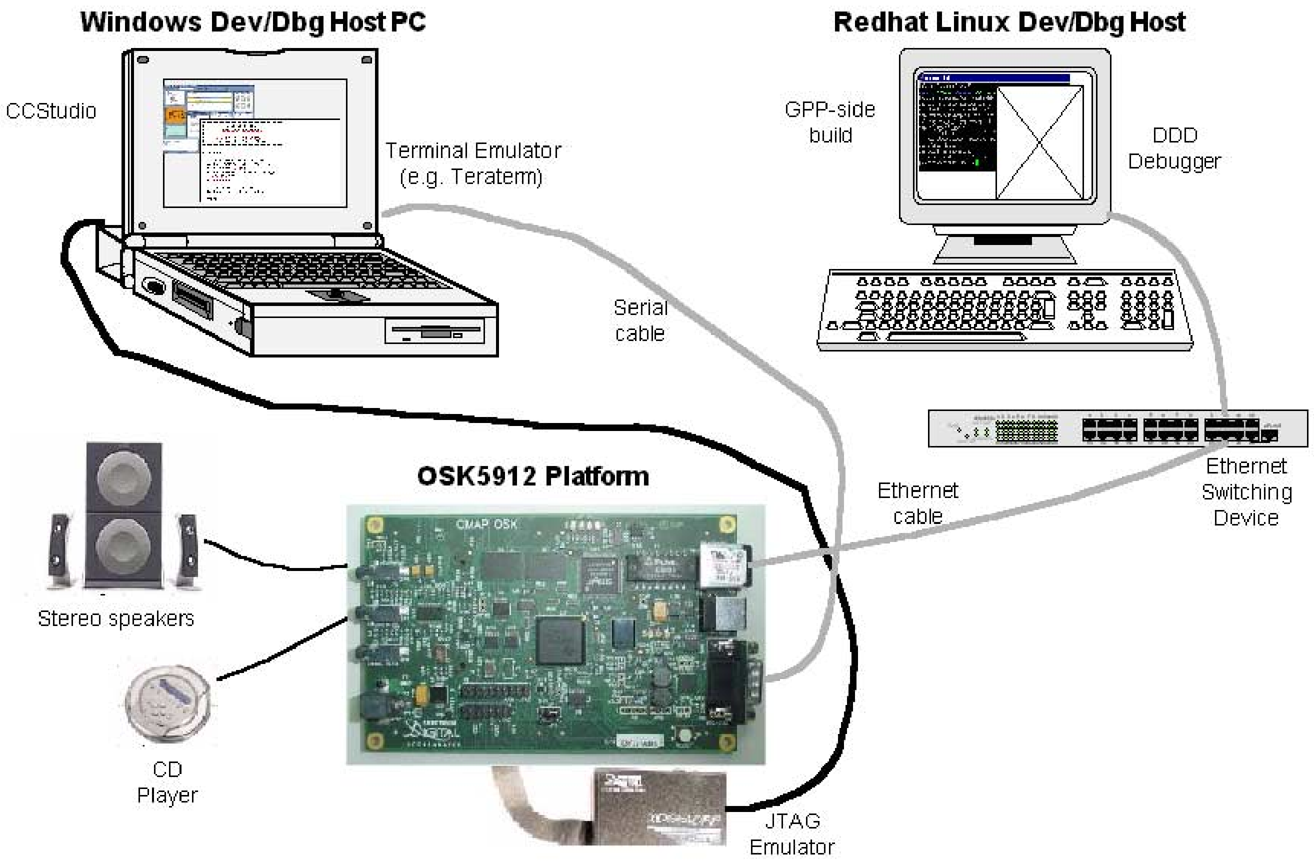


Figure 4. Typical Hardware Setup (OSK 5912 Shown)

This hardware setup consists of the following platforms:

- **Windows Development/Debug Host.** This machine should be a PC running Microsoft Windows NT, 2000, or XP. It is used to build, run, and debug the DSP-side content using Code Composer Studio. The machine must have a free COM port for terminal emulation.
- **Red Hat Linux Development/Debug Host.** This may be either a dedicated Linux host workstation or a Linux Virtual machine running under Windows. It should run Red Hat Linux version 7.2 or greater. VmWare (www.vmware.com) is one such package that has been successfully used with RF6. This host is where you build the GPP-side of your RF6-based applications. The Ethernet connection allows the target board to boot up into the Linux kernel and mount its root filesystem—for example, using the Network File System (NFS). At production time, you will likely put the target filesystem, including the kernel and necessary utilities (for example, ls and pwd) into Flash memory. This enables a standalone boot, eliminating the need for Ethernet.
- **OMAP platform.** One of the following boards:

- **Innovator5910 EVM ES 2.6 or greater.** This board contains both an ARM9TDMI processor and a TMS320C55x DSP. The version specified has Rev C or greater of the processor module (OMAP1510C) and Rev D or greater of the Innovator Ethernet Break-out-Board.
- **OMAP Starter Kit OSK5912.** This board contains both an ARM926EJ-S processor and a TMS320C55x DSP.

The hardware connections shown in Figure 4 are as follows:

- **JTAG Emulator (e.g. XDS510, XDS560, XDS510PP+).** Connects the DSP to the PC running Windows for connection to Code Composer Studio.
- **Ethernet cable.** Connects the target board to the Linux file system.
- **9-pin null modem serial cable.** Connects the GPP to the PC running a terminal emulator.

3.2 Software

The following software should be installed on the platforms used for RF6:

- **Windows Development/Debug Host.**
  - Microsoft Windows NT, 2000, or XP
  - Code Composer Studio for OMAP v2.20.18 or greater
  - A terminal emulator, for example Tera Term Pro 2.3 or greater. Tera Term is a freeware terminal emulation program with the ability to transfer files in binary format. It is available at http://hp.vector.co.jp/authors/VA002416/teraterm.html.
- **Red Hat Linux Development/Debug Host.**
  - Red Hat Linux version 7.2 or greater
  - MontaVista Linux Professional v3.1 or greater, which includes the Linux kernel, bootloader, etc.
  - The latest recommended patches from the MontaVista Zone (http://support.mvista.com). These range from kernel patches to the latest TI Innovator/OSK Kit Linux Support Package (LSP).
  - Network services such as telnetd, ftpd, and nfsd should be configured and available.
  - DSP/BIOS Link v1.10.01 or greater

The necessary header files or library components for the following software are supplied with RF6. However, for source-level debugging, we recommend that you download the full packages from TI DSPVillage.

- Driver Development Kit DDK v1.1 or greater
- Chip Support Library CSL v3.00.02 or greater
- DSP/BIOS Link v1.10.01 or greater
- MSGQ 1.01 or greater

Any additional requirements are listed in Section 4, *Installing, Building, and Running the RF6 Base Application*.

3.3 Recommended Reading

Before commencing with RF6, we strongly recommend that you become familiar with some of the documents in Table 4. A detailed reading is not required, but some understanding of the concepts will make work with RF6 go much more smoothly.

For links to access these resources, see the OMAP5912 website at http://omap.spectrumdigital.com.

Table 4. Recommended Reading Before Using RF6

Document	Available from	Brief Description / Benefit
MontaVista® Linux® Professional Edition for the Texas Instruments Innovator™ Development Kit for OMAP™ 1510 and OMAP™ 5910	MontaVista	Configuring your Linux box. Booting the Innovator.
OMAP5910 Dual-Core Processor Data Manual (SPRS197)	TI DSPVillage	
Innovator™ Development Kit for the Texas Instruments OMAP™ Platform Deluxe Model User's Guide	Productivity Systems Inc	Innovator hardware setup
Building and Installing Open Source Linux for the OMAP Innovator	OMAP Catalog application report	Bootloader installation information
How to Create, Build, and Debug a Linux Application on OMAP Innovator (SPRA395)	OTCs	GPP-side Arm9 Linux toolchain usage.
PlayWave Application Using DSP/BIOS Link	OTCs	Describes a DSP/BIOS Link based application. Data Display Debugger (ddd) usage on a streaming application.
DSP/BIOS Link for MontaVista Linux (MVL) User's Guide (LNK_022_USR)	Part of the DSP/BIOS Link installed documentation	Overview of DSP/BIOS Link. Install, build, and run of DSP-GPP Link, including example walkthrough.
DSP/BIOS Link web page: http://www.ti.com/bioslink	TI DSPVillage	
OMAP Starter Kit for the OMAP5912 (OSK5912) User Guide	Spectrum Digital	OSK hardware setup
OMAP5912 Applications Processor Data Manual (OMAP5912.pdf)	TI DSPVillage	Datasheet for OMAP5912

3.4 Running the Loop GPP Example

Finally, you should follow the steps in the *DSP/BIOS Link for MVL User's Guide* (LNK_022_USR) to get the "loopgpp" example running. Running this example will confirm that you have set up the hardware and much of the software correctly.

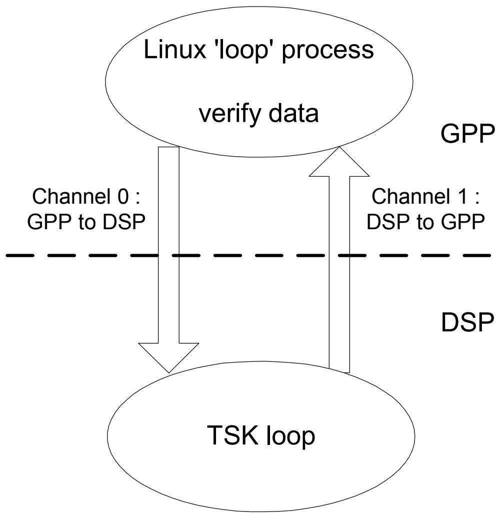


Figure 5. Simplified View of DSP/BIOS Link 'loopgpp' Example

The sample LOOP application illustrates basic data streaming concepts using DSP/BIOS Link. It transfers data between a thread running on the GPP and a DSP/BIOS task running on the DSP.

Once you have successfully run the loopgpp example, you will feel more comfortable with the hardware setup, build and debug environments, as well as DSP/BIOS Link.

4 Installing, Building, and Running the RF6 Base Application

This section describes how to install, build, and run Reference Framework Level 6 (RF6), which performs audio playback.

It is assumed that you have created a setup that fulfills all the prerequisites described in Section 3, *RF6 Minimum System Requirements and Prerequisites*. These instructions are specific to the following setup:

- Innovator 5910 or OSK5912 board. In this section, the term "target board" is used to denote either an Innovator or OSK. Where differences exist, they are noted.
- MontaVista Linux Professional 3.1 or greater on the Linux host machine
- Code Composer Studio 2.20.18 or greater on the target debug host

Ports to other platforms should follow the same flow, so you should be familiar with this section even if you plan to port the application.

For troubleshooting information about RF6, search the DSP KnowledgeBase on the Texas Instruments website for "RF6".

4.1 Preparing the Hardware

The following steps provide an overview of how to connect the hardware used to run the RF6 application. For details about board-specific steps, see the documentation provided with your board. For additional board-specific information, see the resources listed in Section 3.3, *Recommended Reading*.

1. Make sure the board and computers are connected as shown in Figure 6. This includes connecting it to your PC via a serial connection and setting up a file system accessible by the target board through Ethernet.
2. Make sure the board is powered-down.
3. Connect an audio input device such as a microphone or the headphone output of a CD player to the audio input jack on the board as shown in Figure 6. Or, you can connect the audio output of your PC sound card to the audio input of the board. Make sure you use the audio cable provided with your target board.
4. Connect speaker(s) or other audio output device(s) to the audio output port of the board as shown in Figure 6. (Do NOT connect them to the microphone port!)
5. Power up the board.

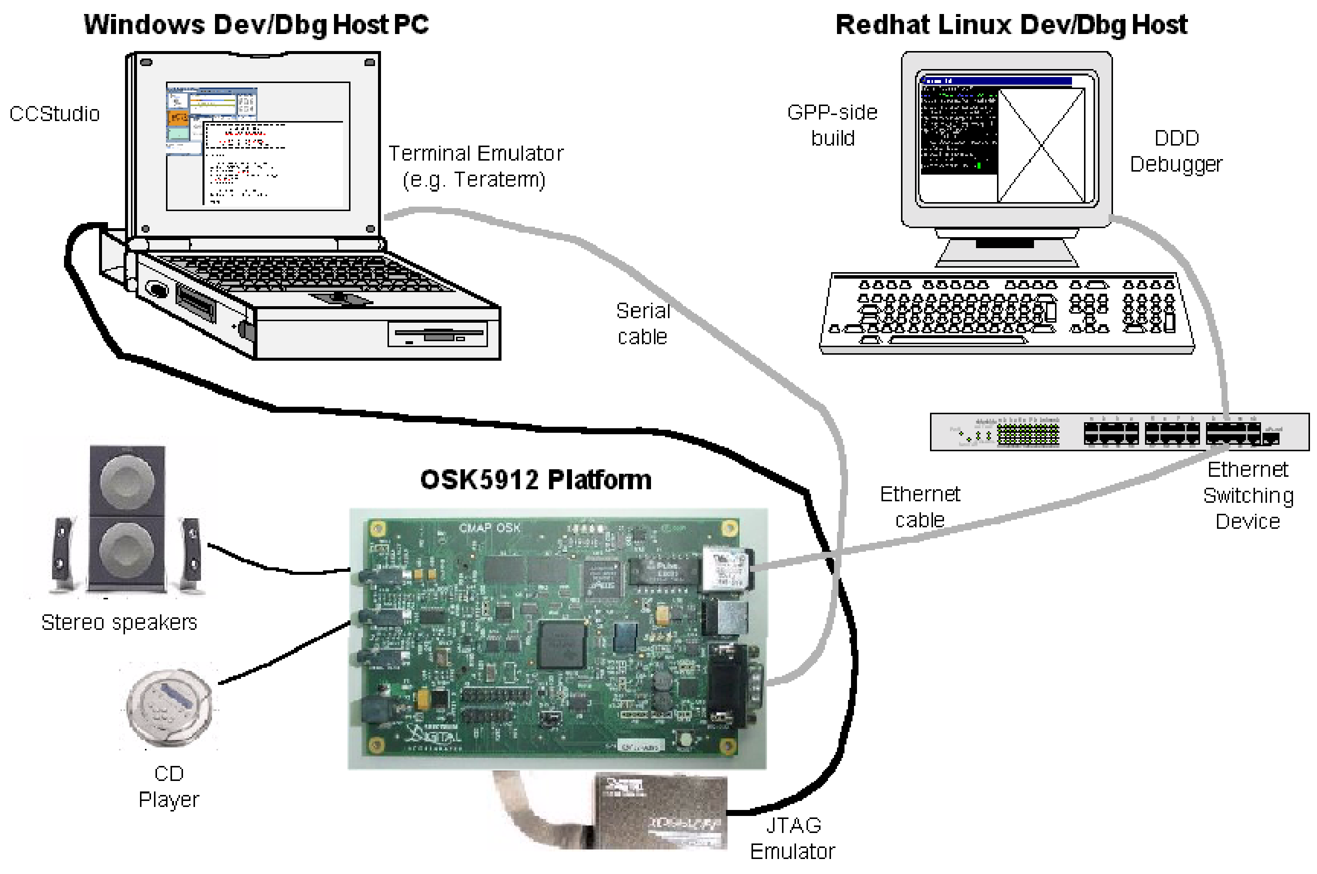


Figure 6. Hardware Device Connections

4.2 Installing the Software

Begin by downloading the Reference Frameworks code distribution files from the OMAP5912 website. The Reference Framework Level 6 code package is delivered as two separate files:

- **rf_v<*version*>.zip.** A zip file containing the DSP-side source code for the RF6 application and the Reference Framework modules. Also includes Reference Frameworks 1, 3, and 5.
- **rf_gpp_v<*version*>.tar.gz.** A tar file containing the GPP-side minimal source code for the RF6 application and associated modules.

Why is the package split in this fashion? The primary reason is that the home for each of these deliverables may be quite different. The DSP-side content typically sits on a PC running Windows while the GPP-side code is on an NFS mount of a Linux host. These two file systems may be entirely separate.

4.2.1 Linux Workstation

1. Make a directory with a name of your choice:

```
[>] cd ~/montavista
[>] mkdir –p xyzdir
```

2. Go into the new directory:

```
[>] cd xyzdir
```

3. Untar the GPP-side code in a directory of your choice:

```
[>] tar xvzf rf_gpp_v<version>.tar.gz
```

   where <version> is the version number in the file you downloaded.

4.2.2 Windows Workstation

1. If you have not already done so, install Code Composer Studio for OMAP 2.20.18 or a later version. It is recommended that you have the latest version of the Code Composer Studio software, as it may contain important features or problem fixes.

2. Unzip the DSP-side code. A suggested location for these files is the <$(INSTALL_DIR)>\myprojects folder. <$(INSTALL_DIR)> stands for the Code Composer Studio installation directory, which is c:\ti by default.

   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.

NOTE: The top-level folder of the Reference Frameworks DSP-side distribution is called "referenceframeworks". The full path to this folder is called *RF_DIR* in this application note. If you unzip the distribution as suggested in c:\ti\myprojects, *RF_DIR* is c:\ti\myprojects\referenceframeworks

3. If you have a version of DSP/BIOS Link newer than v1.10.01, you might want to point RF6 to the newer version of the DSP/BIOS Link DSP-side library (for example, dsplink.l55l). There are two ways to accomplish this:

   – Copy the newer DSP/BIOS Link DSP-side archives (for example, dsplink.l55l) to the Reference Frameworks folders. For example, you might copy the archives from c:\dsplink\dsp\lib\debug\ to *RF_DIR*\lib\debug.

   – Change the linker command file (link.cmd) to point to the new archives. For example:

   ```
   –l c:\myNewDSPLink\dsp\lib\debug\dsplink.l55l
   –l c:\myNewDSPLink\dsp\lib\debug\dsplinkmsg.l55l
   ```

   Reference Frameworks includes the DSP-side libraries of DSP/BIOS Link to allow you to build the application out of the box. There is currently no well-established way to define symbolic links in Code Composer Studio.

4. You should also update the header files using one of the two methods in step 3.

The remaining steps are necessary for debugging RF6-based DSP code. In theory, developers could work without Code Composer Studio since RF6 does not require it—code is loaded via DSP/BIOS Link on the ARM-side. In practice, however, all applications need debugging, and Code Composer Studio is the tool to use for debugging the DSP-side application.

5. Obtain the appropriate GEL file for your target from the OSK website. Ensure that the line for the GEL_Reset() call is either commented out or removed from the GEL file. This change is necessary because when Code Composer Studio is launched, GEL_Reset() moves the Program Counter to the beginning of the code. This causes c-init record processing and other initialization to be repeated, which could cause problems (SDSsq33006).

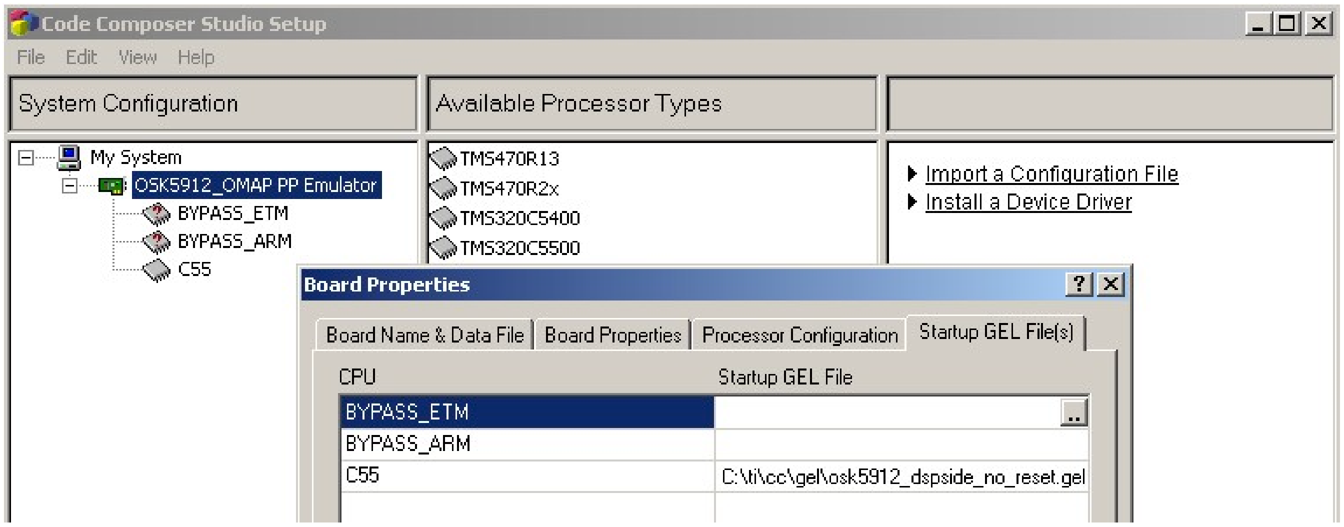
   ```
   /* GEL_Reset(); */
   ```

6. Use Code Composer Studio Setup, to configure your emulator as follows. (Figure 7 shows configuration of the OSK5912.)

   – Bypass 8-bit

   – Bypass 4-bit

   – C55 (TMS320C5500). Use the appropriate GEL file for your target board.



CPU	Startup GEL File
BYPASS_ETM	
BYPASS_ARM	
C55	C:\ti\cc\gel\osk5912_dspside_no_reset.gel

Figure 7. Code Composer Studio Setup Settings

NOTE: The 4-bit bypass is used instead of TMS470R2 (ARM9), thus bypassing the ARM in the JTAG scan chain. This disables the ability to debug the ARM9 from Code Composer Studio. However, this is not a major issue since there are a number of feature-rich debuggers on the GPP-side, such as Data Display Debugger, a graphical frontend to the well-known gdb debugger.

7. Save the configuration and exit Code Composer Studio Setup.

4.3 Build Procedure

This section describes how to build both the DSP-side and GPP-side RF6 base applications.

4.3.1 Building the Linux Kernel

Prior to building the GPP-side of the RF6 application, it is a good idea to check that you have built your Linux kernel with sufficient options to run the default RF6 application. On the OSK5912, this basically means including audio and I2C support. (The Arm-side configures the AIC23 audio codec by setting such things as the sample rate.)

For example, RF6 was tested with a kernel configuration that included support for the following options. You may have additional kernel options selected.

- System Type -> (OSK5912) TI OMAP Implementations
  – (192Mhz) OMAP161x Frequency Selection
- General Setup -> Pre-emptible kernel
- Networking Options -> DHCP Support
- Character Devices -> I2C support
  – I2C support

- I2C TPS65010 Power Controller Interfaces
- OMAP1610 I2C Adapter

- Sound
  - Sound Support
  - OMAP Sound drivers
  - AIC23 stereo audio

4.3.2 Building the GPP-Side Application on a Linux Workstation

After you untar the Linux package, you can build the GPP-side of the RF6 application. The following steps describe how to build the RF6 application.

1. Open Rules.make using your favorite text editor. The Rules.make file at the top level of the folder hierarchy is a central place to specify paths. It is used instead of scripts to avoid different shell (ksh, csh, bash, ...) nuances.

2. Modify the DSPLINK_DIR path to point to your DSP/BIOS Link installation. The local makefile for the RF6 GPP application defines $(RF_ROOT_DIR) to point to the top-level rf_gpp directory (where Rules.make is located). For example:

```
DSPLINK_DIR := $(RF_ROOT_DIR)/dsplink_v1_10_mvlpro31
```

3. Verify that the path and prefix to the cross compiler tools are correct. This ensures that the compiler, linker and archiver can be found. For example:

```
CROSS_COMPILER := /exports/montavista/pro/devkit/arm/v4t_le/bin/arm_v4t_le-
```

4. In apps/rf6/Linux/OMAP, build the GPP application with the following command:

```
[>] make
```

   The GPP application should build without errors. Both debug and release versions should be generated as a result.

   If the build fails, it may be because the dsplink library was not built, was built incorrectly (for example, the MSGQ component was not enabled), or the Rules.make DSPLINK_DIR variable was not set to point to the Link GPP code. Remember that RF6 does not ship a prebuilt dsplink library and kernel module on the GPP-side to ease potential versioning issues. Hence you first need to build dsplink.lib by following the instructions in the *DSP/BIOS Link for MontaVista Linux (MVL) User's Guide* (LNK_022_USR).

5. Copy the resulting executable, rf6_gpp, from /bin/rf6/Linux/OMAP/Debug to the NFS-mounted target file system. Use a directory of your choice. (You will put the DSP-side application in the same directory.) For example type the following command at a single command prompt:

```
[>] cp ~/montavista/xyzdir/rf_gpp/bin/rf6/Linux/OMAP/Debug/rf6_gpp
/exports/mvlpro31_knl_fsys/filesys/home/myname/rf6dir/
```

4.3.3 Building the DSP-Side Application on a Windows Workstation

After you unzip the Windows package, you can build the DSP-side of the RF6 application.

1. Use the timake command to rebuild the DSP-side application from the Windows command prompt:

```
[>] cd c:\ti\myprojects\referenceframeworks\apps\rf6\projects\target
[>] timake app.pjt DEBUG
```

NOTE: You must first run <$(INSTALL_DIR)>\dosrun.bat so that the timake command and the TI Code Generation Tools are found.

2. Copy the resulting DSP-side application (app.out) from apps\rf6\projects\target\debug to the target's file system, in the same directory where the GPP-side application is located. (For example, to ~/montavista/filesys/home/myname/rf6dir.)

NOTE: If you recently upgraded to a newer version of Code Composer Studio, you should run *RF_DIR*\buildConfig.bat followed by *RF_DIR*\build*<arch>*.bat from an MS-DOS command line (where *<arch>* is 5000 or 6000). These simple batch files rebuild *all* Reference Frameworks projects. This ensures that all modules and DSP/BIOS configuration files are in sync with the latest TI Code Generation Tools and DSP/BIOS content. Note that you must first run <$(INSTALL_DIR)>\dosrun.bat.

4.4 Running the Application in MontaVista Linux

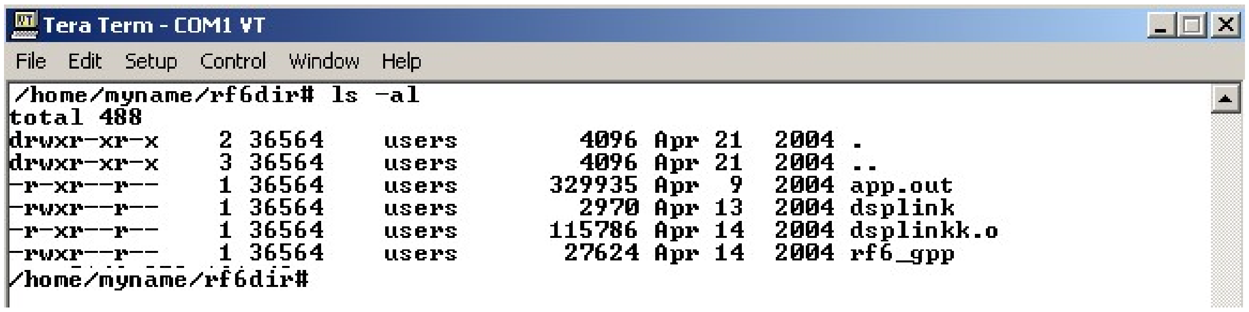
This section assumes you have already set up the MontaVista Linux operating system on the target platform. On the NFS-mounted target file system, follow this procedure, which assumes you copied both the GPP-side and DSP-side RF6 applications to /home/myname/rf6dir:

1. Copy the following files to the /home/myname/rf6dir directory using Linux:
   - The dsplink load/unload script from your DSP/BIOS Link installation.
   - The dsplinkk.o Linux kernel module from your DSP/BIOS Link installation (choose either the debug or release version).

   For example:

```
[>] cd /home/myname/rf6dir
[>] cp ~/montavista/dsplink/etc/target/scripts/Linux/dsplink .
[>] cp ~/montavista/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o .
```

   At this point, you should have files similar to those in Figure 8 stored on the file system of the target board.



```
/home/myname/rf6dir# ls -al
total 488
drwxr-xr-x    2 36564    users        4096 Apr 21  2004 .
drwxr-xr-x    3 36564    users        4096 Apr 21  2004 ..
-r-xr--r--    1 36564    users      329935 Apr  9  2004 app.out
-rwxr--r--    1 36564    users        2970 Apr 13  2004 dsplink
-r-xr--r--    1 36564    users      115786 Apr 14  2004 dsplinkk.o
-rwxr--r--    1 36564    users       27624 Apr 14  2004 rf6_gpp
/home/myname/rf6dir#
```

Figure 8. Target Board Files After Copying

2. Reboot the target board and then start the Linux kernel via the bootloader.
3. At the prompt of the target board file system, enter the following commands:
   - Go to the directory containing the application:

```
[>] cd /home/myname/rf6dir
```

   - Ensure that the appropriate file permissions have been set to load the various modules and execute RF6.

```
[>] chmod -R u+rwx *
```

   - Load the DSP/BIOS Link kernel module. (Do not forget the trailing dot, which is an argument specifying the directory location of the DSP/BIOS Link kernel module.)

```
[>] ./dsplink load .
```

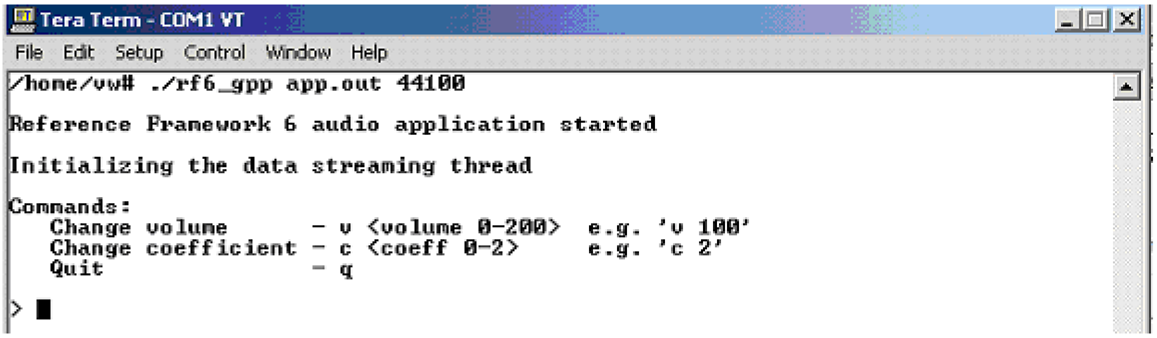
   - Run the rf6_gpp user-mode process, which loads the DSP executable app.out with the AIC23 codec running at 44.1 kHz, and starts full data streaming through DSP/BIOS Link.

```
[>] ./rf6_gpp app.out 44100
```

   The last parameter is the frequency in Hz. Any frequency supported by the audio codec is allowed (for example, 8000, 16000, or 44100).

NOTE: If you are using the XDS510PP+ emulator, open SDConfig.exe and click the red “R” icon in the toolbar to reset the SD emulator. (The emulator holds the DSP in reset when the latter is connected via JTAG.)

4. Start your CD player or other audio input device.
5. You should hear the FIR filtered audio output through the speakers connected to the target board.
6. You will see the application’s GPP-side command menu:



```
/home/vw# ./rf6_gpp app.out 44100

Reference Framework 6 audio application started

Initializing the data streaming thread

Commands:
   Change volume      - v <volume 0-200>  e.g. 'v 100'
   Change coefficient - c <coeff 0-2>     e.g. 'c 2'
   Quit               - q

> █
```

Figure 9. RF6 GPP-Side Application Menu

Within this application, you can send either filter coefficients change commands or volume change commands. Table 5 shows the acceptable commands:

Table 5. GPP-Side Commands

Command	Use	Acceptable value range
c <value>	Change FIR filter coefficients	0 (low-pass filter) 1 (high-pass filter) 2 (all-pass filter)
v <value>	Change volume	0 to 200 (% of input volume)
q	Quit application gracefully	

When you have finished running the application, use the q command to quit. You can re-run the application as many times as you like using the following command:

```
[>] ./rf6_gpp app.out 44100
```

4.5 Debugging the DSP-Side Application with Code Composer Studio

Once you have started running the GPP-side and DSP-side applications, you can connect to the DSP-side application using Code Composer Studio.

To use Code Composer Studio with the already-running program, follow these steps:

1. Configure Code Composer Studio Setup as described in Section 4.2.2, *Windows Workstation*.
2. Run the RF6 application as described in Section 4.4, *Running the Application in MontaVista Linux*.
3. When the application is running and waiting for the next command, launch Code Composer Studio. You might encounter a few error messages that say “Trouble reading target CPU memory". Simply dismiss them by clicking Cancel. (SDSsq33006)
4. In Code Composer Studio, choose **Project→Open** and select the DSP-side project from the *RF_DIR*\apps\rf6\projects\target folder.

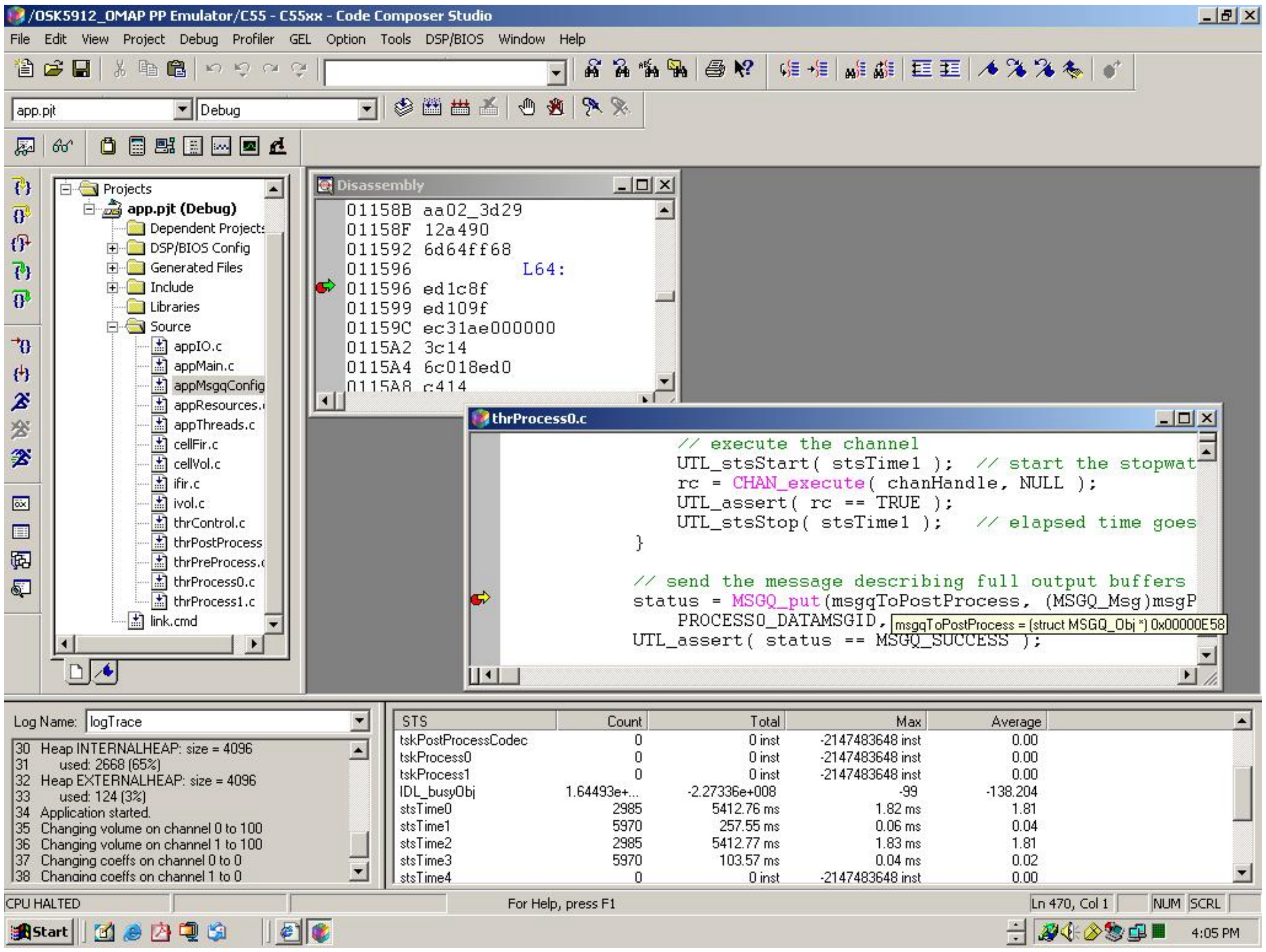
5. In Code Composer Studio, choose **File→Load Symbols→Load Symbols Only**. Note that you do NOT load the program here, since it has already been loaded by the GPP-side application via DSP/BIOS Link.

6. In the Load Symbols dialog, select the DSP executable that is being run (for example, app.out). This informs Code Composer Studio about the program loaded on the target, and allows Code Composer Studio to make use of symbolic debugging.

7. At this point, you may debug the DSP executable as you would other DSP programs. For example, you can set breakpoints, run, halt, watch variables, view LOG messages and STS statistics. Within Code Composer Studio, you can use the same DSP/BIOS Analysis Tools you use with non-GPP connected applications.

8. You can verify the data rate by following these steps:
   - Choose **DSP/BIOS→Statistics View** to open the statistics analysis tool.
   - Right-click on the Statistics View area and choose Property Page.
   - In the Units tab, set the unit for the stsTime0 object to milliseconds. (This object collects statistics for the thrProcess0 task.)
   - Notice that the average for stsTime0 average is 1.81 ms. This is consistent with the 44.1 kHz sample rate and the 80-sample-size buffers.

9. When you have finished debugging, leave the DSP running free (F5) and close Code Composer Studio. This is necessary so that the GPP-side application can shutdown cleanly when you use the quit ("q") command.



```
01158B aa02_3d29
01158F 12a490
011592 6d64ff68
011596          L64:
011596 ed1c8f
011599 ed109f
01159C ec31ae000000
0115A2 3c14
0115A4 6c018ed0
0115A8 c414
```

```
            // execute the channel
            UTL_stsStart( stsTime1 );  // start the stopwat
            rc = CHAN_execute( chanHandle, NULL );
            UTL_assert( rc == TRUE );
            UTL_stsStop( stsTime1 );   // elapsed time goes
        }

        // send the message describing full output buffers
        status = MSGQ_put(msgqToPostProcess, (MSGQ_Msg)msgP
            PROCESS0_DATAMSGID,
        UTL_assert( status == MSGQ_SUCCESS );
```

```
msgqToPostProcess = (struct MSGQ_Obj *) 0x00000E58
```

Log Name: logTrace

```
30  Heap INTERNALHEAP: size = 4096
31      used: 2668 (65%)
32  Heap EXTERNALHEAP: size = 4096
33      used: 124 (3%)
34  Application started.
35  Changing volume on channel 0 to 100
36  Changing volume on channel 1 to 100
37  Changing coeffs on channel 0 to 0
38  Changing coeffs on channel 1 to 0
```

STS	Count	Total	Max	Average
tskPostProcessCodec	0	0 inst	-2147483648 inst	0.00
tskProcess0	0	0 inst	-2147483648 inst	0.00
tskProcess1	0	0 inst	-2147483648 inst	0.00
IDL_busyObj	1.64493e+...	-2.27336e+008	-99	-138.204
stsTime0	2985	5412.76 ms	1.82 ms	1.81
stsTime1	5970	257.55 ms	0.06 ms	0.04
stsTime2	2985	5412.77 ms	1.83 ms	1.81
stsTime3	5970	103.57 ms	0.04 ms	0.02
stsTime4	0	0 inst	-2147483648 inst	0.00

CPU HALTED   For Help, press F1   Ln 470, Col 1   NUM SCRL

Figure 10. Debugging RF6 in Code Composer Studio

4.6 Rebuilding and Debugging DSP-Side Libraries

Source code is provided for the additional libraries used by RF6 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, Code Composer Studio 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 Code Composer Studio, using **Options→Customize→Directories** menu option, you can specify which folders Code Composer Studio should search to locate the source file. If you specify source code folders for the modules used in RF6, Code Composer Studio 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.

NOTE: 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.

Source code for most IOM device drivers is included in the DSP/BIOS Driver Developer's Kit (DDK). The Reference Frameworks distribution includes source code only for device drivers that are not included in the 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).

Source code for DSP/BIOS Link's IOM driver is not included with Reference Frameworks. Potential versioning issues are minimized by encouraging users to refer to the latest version of DSP/BIOS Link for the source code.

5 RF6 Technical Overview

This section gives an overview of several aspects of RF6. Included are an overview of the application behavior, the module hierarchy, and the directory hierarchy.

5.1 Application Behavior Overview

RF6 extends the behavior of RF5 by adding data and control communication channels between the DSP and GPP.

To summarize, the default RF6 application processes an incoming stereo audio signal on the DSP, then sends the data to the GPP, which has the option to process the data before sending it back to the DSP for further processing. Finally, it sends the signal to the output codec. A control thread on the GPP sets DSP algorithm parameters such as volume and high-/low-pass filtering.

Figure 11 shows the overall processing flow in the default RF6 application.

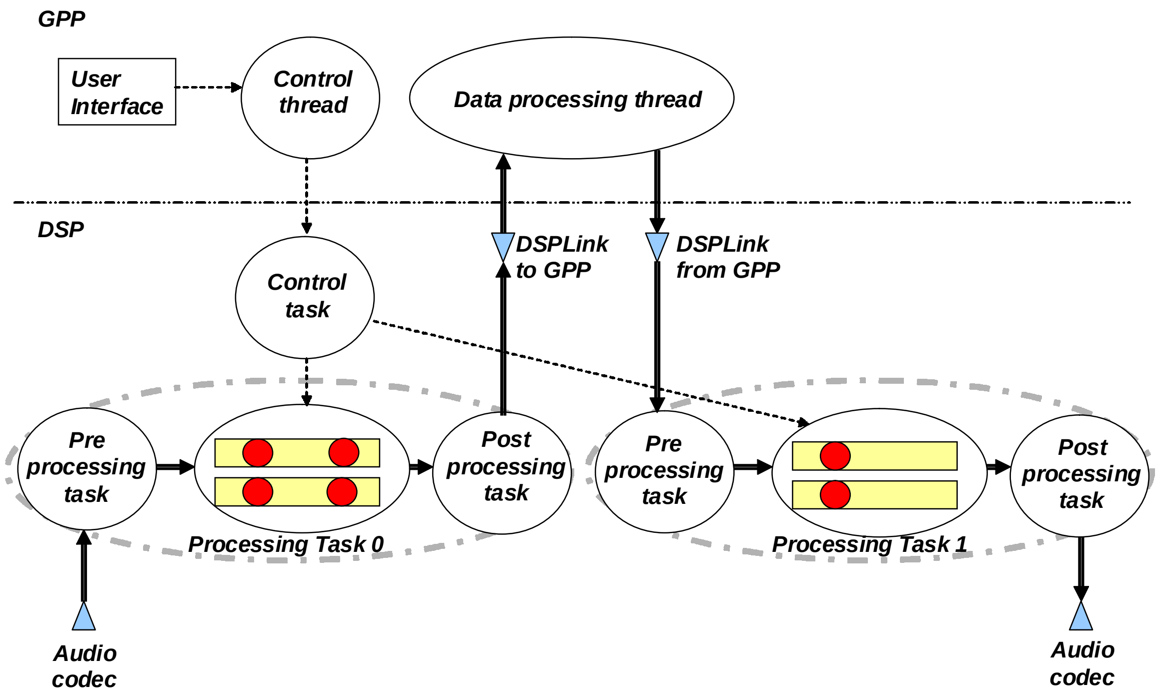


Figure 11. RF6 Application Processing Flow

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 blocks into frames of a given size before processing them.

For processing task 0, 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 in processing task 0. 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) are set for this task via the GPP-side application. Volume control messages are ignored by this task in the default application.

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 GPP via DSP/BIOS Link. The default GPP-side application performs a simple memcpy() and sends the frame back to the DSP via DSP/BIOS Link. The memcpy() is a place-holder for adaptation (such as algorithm processing) on the GPP side.

Processing task 1 is similar to processing task 0, however it uses only the VOL algorithm since applying the FIR algorithm twice would degrade the signal. Volume control values for this task are set via the GPP-side application.

The pre-processing, processing, and post-processing tasks form a triplet that isolates the execution of a particular data path. Tasks are organized this way for easier adaptation. For example, Section 8.1, *Adapting RF6 for an MP3 Decoder: One Directional Data Flow* described an adaptation that chops out the task triplet on the left in Figure 11. This leaves only the second triplet, whose purpose is to run the Internet Audio decoder algorithm(s).

As with all Reference Framework levels, the pre- and post-processing can be optimized in either of the following ways:

- Pre- and post-processing may be folded into the main processing task, thus reducing the total number of TSKs in the system
- Pre- and post-processing may be executed in the IOM codec device driver directly. For example, the C55x DMA controller could be programmed to sort left and right channel data directly.

The subsections that follow describe portions of the RF6 application in further detail.

5.1.1 DSP Data Processing Elements

The four basic DSP data processing elements in RF6 are *tasks*, *channels*, *cells*, and *XDAIS algorithms*.

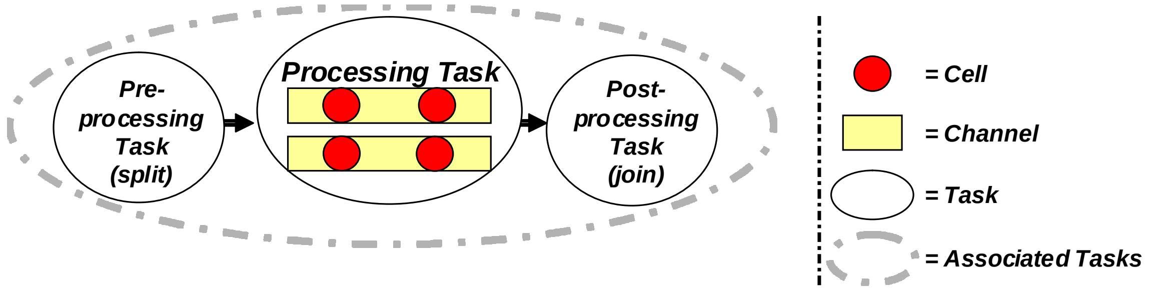


Figure 12. Processing Elements in RF6

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*.

A *XDAIS algorithm* is an off-the shelf, reusable data processing component, that implements a certain interface (IALG). Typically it implements 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 used in RF6.

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 interface. The function signatures of algorithm processing functions can vary. In fact, an algorithm may have multiple processing functions. A cell provides a standard interface between the algorithm and the outside world. 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.

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 require any additional code. Typically several channels contain sets of cell *instances* that perform identical functions, possibly with different parameters.

A *task* can consist of a collection of channels, which are also executed in series. The purpose of a 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 have task-specific code, which the user writes. This code usually just sends and receives data to and from the outside world and executes channels. A task can 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, as is the case with the pre-processing and post-processing tasks.

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. The supplied RF6 application also uses hardware interrupts (HWIs) for high-priority event processing triggered by peripherals.

Details about cells, channels, and tasks are provided in the application note for RF5 (SPRA795). The processing triplet shown in Figure 12 is essentially identical to the processing performed in the default RF5 application.

5.1.2 DSP Processing Threads

The DSP side of the RF6 application is divided into the following threads. This list reflects roughly the sequence in which threads are performed. (The control threads and processing threads may be interspersed.)

- **tskControl.** The control thread on the DSP. This task runs the thrControlRun() function, which checks for control messages sent from the GPP using the MSGQ module. If there are any control messages, it sends them to the appropriate message queues, which are in turn checked by processing tasks.
- **tskPreProcessCodec.** The pre-processing task for tskProcess0. This task runs the thrPreProcessRun() function with the thrPreProcessCodec argument. In the default application, this task gets data from the codec (using SIO APIs) and splits the stereo signal into two channels.
- **tskProcess0.** This task executes the cells defined for each of its channels. It runs the thrProcess0Run() function. In the default application, this task contains two channels, which each contain cells for the FIR and VOL algorithms. Note that the VOL algorithm instances in tskProcess0 act as dummy placeholders for other algorithms in adaptations, and are not controlled by the user interface to vary the audio volume.

- **tskPostProcessLink.** The post-processing task for tskProcess0. This task runs the thrPostProcessRun() function with the thrPostProcessLink argument. In the default application, this task joins the two channels and sends data to the GPP (using SIO APIs).
- **tskPreProcessLink.** The pre-processing task for tskProcess1. This task runs the thrPreProcessRun() function with the thrPreProcessLink argument. In the default application, this task gets data from the GPP (using SIO APIs) and splits the stereo signal into two channels.
- **tskProcess1.** This task executes the cells defined for each of its channels. It runs the thrProcess1Run() function. In the default application, this task contains two channels, which each contain a cell for the VOL algorithm.
- **tskPostProcessCodec.** The post-processing task for tskProcess1. This task runs the thrPostProcessRun() function with the thrPostProcessCodec argument. In the default application, this task joins the two channels and sends data to the output codec (using SIO APIs).

5.1.3 GPP Data Processing Threads

The Reference Framework content on the GPP-side is not as rich as the DSP-side. This is intentional for the following reasons:

- **Operating system diversity.** There are many GPP operating systems. The more code in the RF6 GPP-side application, the more difficult it is for users to port it to their chosen OS.
- **Operating system expertise.** Many system integrators have strong GPP backgrounds and bring with them their own large GPP codebase.

The GPP side of the RF6 application is divided into the following threads:

- **rf6_gpp.** The main RF6 GPP process. This process presents the user interface to allow commands to by sent. This is a Linux process. When the user sets a processing value (for the FIR or VOL algorithm), that value is sent to the DSP via the MSGQ module. This process also calls the DSP/BIOS Link PROC module APIs to bootload the DSP with the app.out executable.
- **streamThread.** Performs a memcpy to copy data coming from the DSP via DSP/BIOS Link to a location where it will be sent back to the DSP via DSP/BIOS Link. It also performs codec initialization by writing commands to the I2C peripheral, which is connected to the AIC23 codec on the GPP-side of the OMAP591x processor.

  The streamThread is a Linux pThread. POSIX threads (pThreads) are popular in the GPP space, and consequently most GPP OSs support an implementation of POSIX threads. They allow you to spawn a new concurrent process flow. Threads require less overhead than “forking” or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. The streamThread shares the same address space as its parent rf6_gpp process. We use POSIX threads in the RF6 GPP-side application because they are efficient and easy to code.

5.1.4 GPP-DSP Communication

The RF6 application uses bi-directional data communication between the GPP and DSP. In addition, a control channel passes messages from the GPP to the DSP. You can modify this behavior as needed when creating your own application.

Control communication (that is, messaging) between the GPP and DSP is performed via the MSGQ module, which has similar APIs on both sides. Data communication (that is, streaming) between the GPP and the DSP is performed via the CHNL module on the GPP and the SIO module on the DSP.

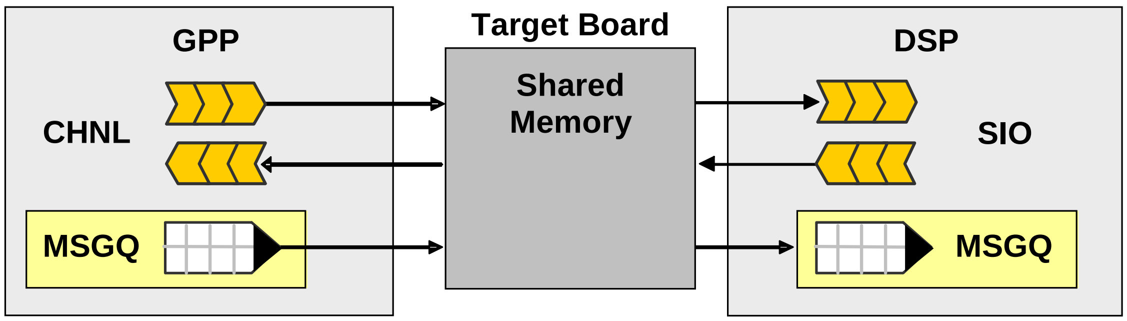


Figure 13. GPP-DSP Communication Path

For more details, see Section 6.1, *MSGQ: Variable Length DSP<->GPP Messaging*.

5.1.5 Application Control

The default GPP application uses a message queue to send control messages to the tskControl thread on the DSP. The tskControl thread checks for messages. If it finds a message, it uses message queues to place the message in a place where it will be read by the appropriate processing task.

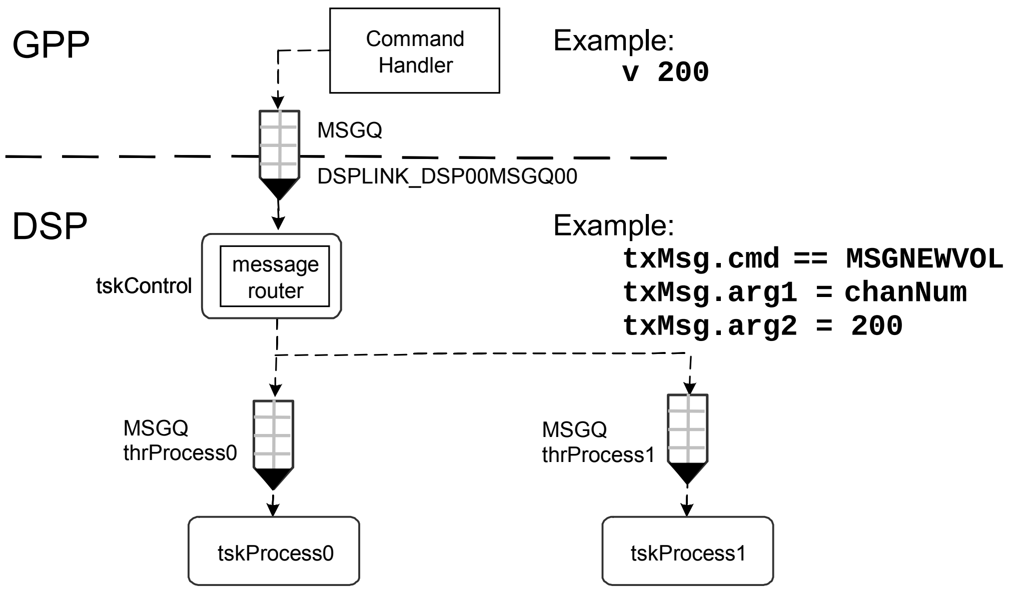


Figure 14. GPP-DSP Control Communication

5.1.6 *Data Flow to Input and Output Codecs*

RF6 uses the IOM driver methodology described in SPRU616. With this methodology, it uses the DIO stream device adapter to connect an IOM mini-driver for the AIC23 codec on the target board to SIO objects used by DSP/BIOS.

As shown in Figure 15, the SIO and DIO layers are hardware-independent. Only the mini-driver is specific to the hardware.

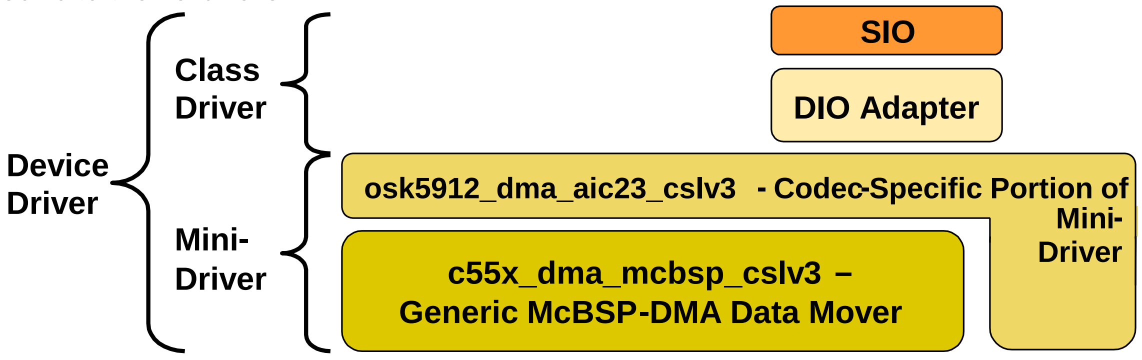

Device Driver
Class Driver
Mini-Driver
SIO
DIO Adapter
osk5912_dma_aic23_cslv3 - Codec-Specific Portion of Mini-Driver
c55x_dma_mcbsp_cslv3 – Generic McBSP-DMA Data Mover

Figure 15. IOM Codec Driver

5.2 DSP-Side Module Hierarchy

The DSP-side RF6 application uses several collections of modules. Figure 16 shows the high-level framework architecture.

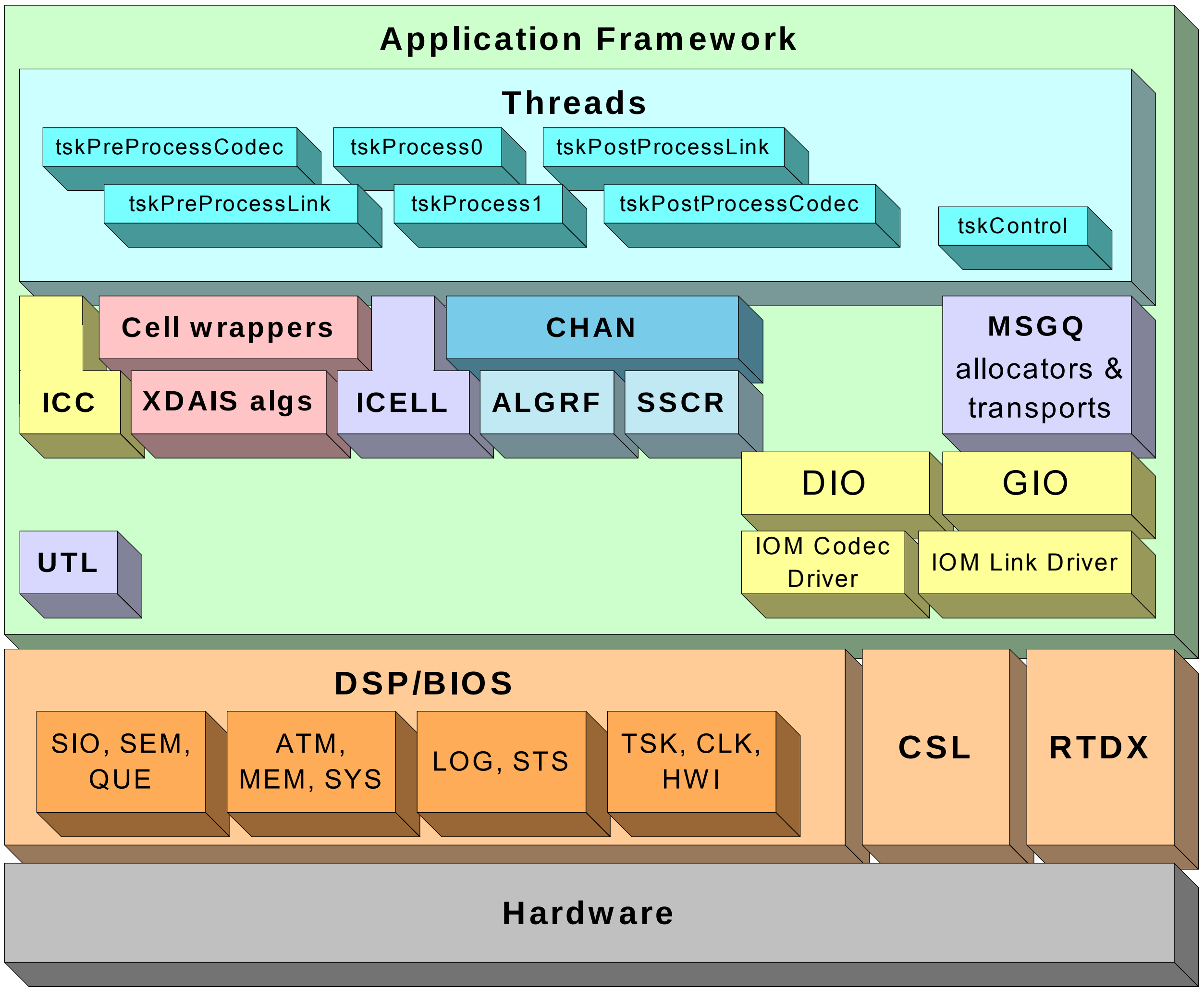


Figure 16. Topology of DSP-Side Modules in RF6

Table 6 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 6. DSP Architecture Components

Component	Description	Adaptation Required
Application Framework	RF6 can be adapted for use in many applications, including telecommunication, audio, video, and more. Such changes are not detailed in this application note.	Modification likely
Threads	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.	Yes, to customize for your application.
Cell wrappers	Application-side cell implementation code. Contains simple "glue code" to match the XDAIS algorithm interface to the framework APIs.	Typically minimal, to switch algorithms
XDAIS algorithms	TI's eXpressDSP-compliant implementation of the simple algorithms used by default. Algorithms may be written by you or provided by a vendor.	No
ALGRF	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 Code Composer Studio's ALG module.	No, but source code is provided
CHAN	Manages the serial execution of XDAIS algorithms contained in cells.	No, but source code is provided
ICELL	Interface specification for encapsulating XDAIS algorithm instances in cells.	Interface must be implemented for each algorithm used.
ICC	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.	No, but source code is provided
SSCR	Manages the overlaying of on-chip scratch data memory requested by XDAIS algorithms.	No, but source code is provided
MSGQ	Supports the DSP-side of DSP-GPP messaging. This module will be integrated as a DSP/BIOS module. It provides a similar interface on the GPP and DSP. MSGQ replaces the MSGLINK module, which had limited functionality.	No
GIO	Generic device adapter. Used in RF6 because the DSP/BIOS Link implementation of MSGQ transports uses it for the class adapter layer.	No
DIO	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-independent. Described in SPRU616.	No
IOM Link driver	Simple, low-level device driver interface between application threads and DSP/BIOS Link.	Maybe, to port DSP/BIOS Link to your hardware
IOM codec driver	Simple, low-level device driver interface between application threads and hardware devices. For example, "osk5912_dma_aic23_cslv3". Defines a set of device-independent APIs to interface with.	Yes, to port controller to your hardware
UTL	Provided to support debugging and diagnostics.	No, but source code is provided
DSP/BIOS	The set of modules provided as a scalable real-time kernel. This includes modules for scheduling, memory management, instrumentation, and I/O.	No
CSL	Chip Support Library. Simplifies the job of developing device drivers and interacting with peripherals. Note that the OMAP targets use a newer CSL software architecture (CSL v3.x) than the existing DSP-only devices.	No
Hardware	The target board, including codecs and other peripherals.	—

5.3 GPP-Side Module Hierarchy

The GPP-side RF6 application uses several collections of modules. Figure 17 shows the high-level framework architecture.

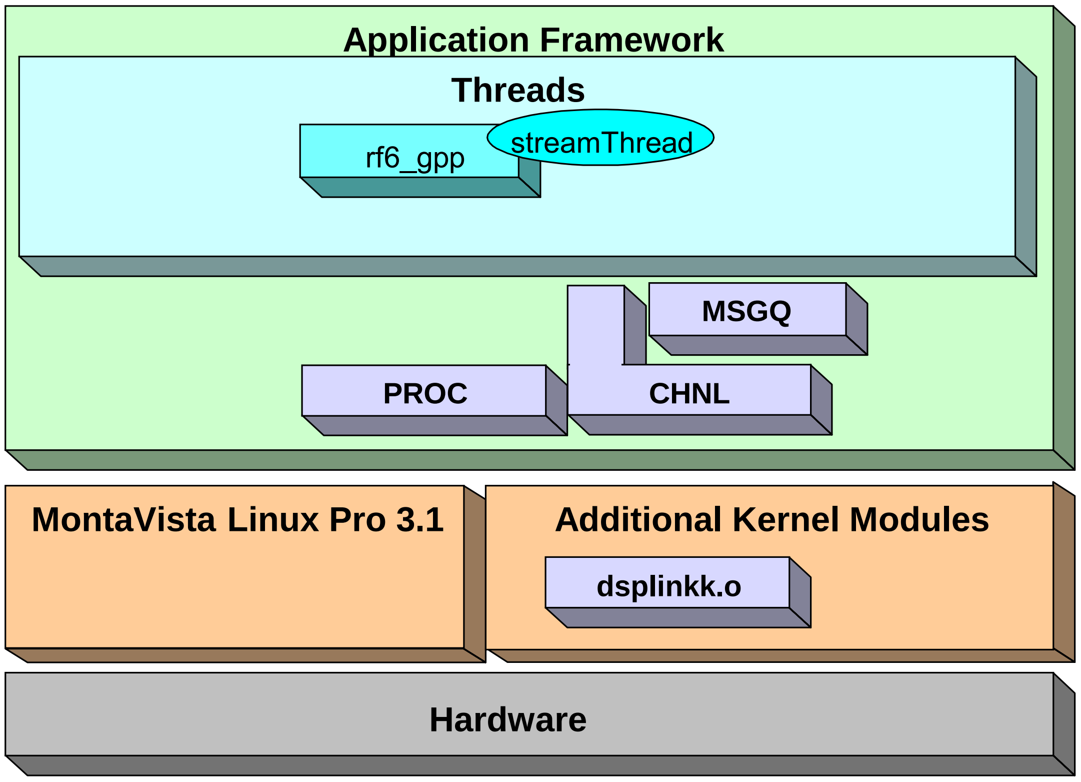


Figure 17. Topology of GPP-Side Modules in RF6

Table 7 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 7. GPP Architecture Components

Component	Description	Adaptation Required
rf6_app	This Linux process instantiates DSP/BIOS Link, loads the DSP application, and issues control commands	Yes
streamThread	This Linux POSIX thread performs codec initialization and data streaming.	Yes
PROC	DSP/BIOS Link module to load and unload a DSP executable.	No
CHNL	DSP/BIOS Link module for data streaming.	No
MSGQ	Supports the GPP-side of DSP-GPP messaging. This module provides a similar interface on the GPP and DSP. MSGQ replaces the MSGLINK module, which had limited functionality.	No
dsplinkk.o	CHNL and PROC make ioctl calls into this kernel module to do the actual work.	No

5.4 Detailed Data Flow

A combined view of the data flow in the default RF6 application looks as shown in Figure 18.

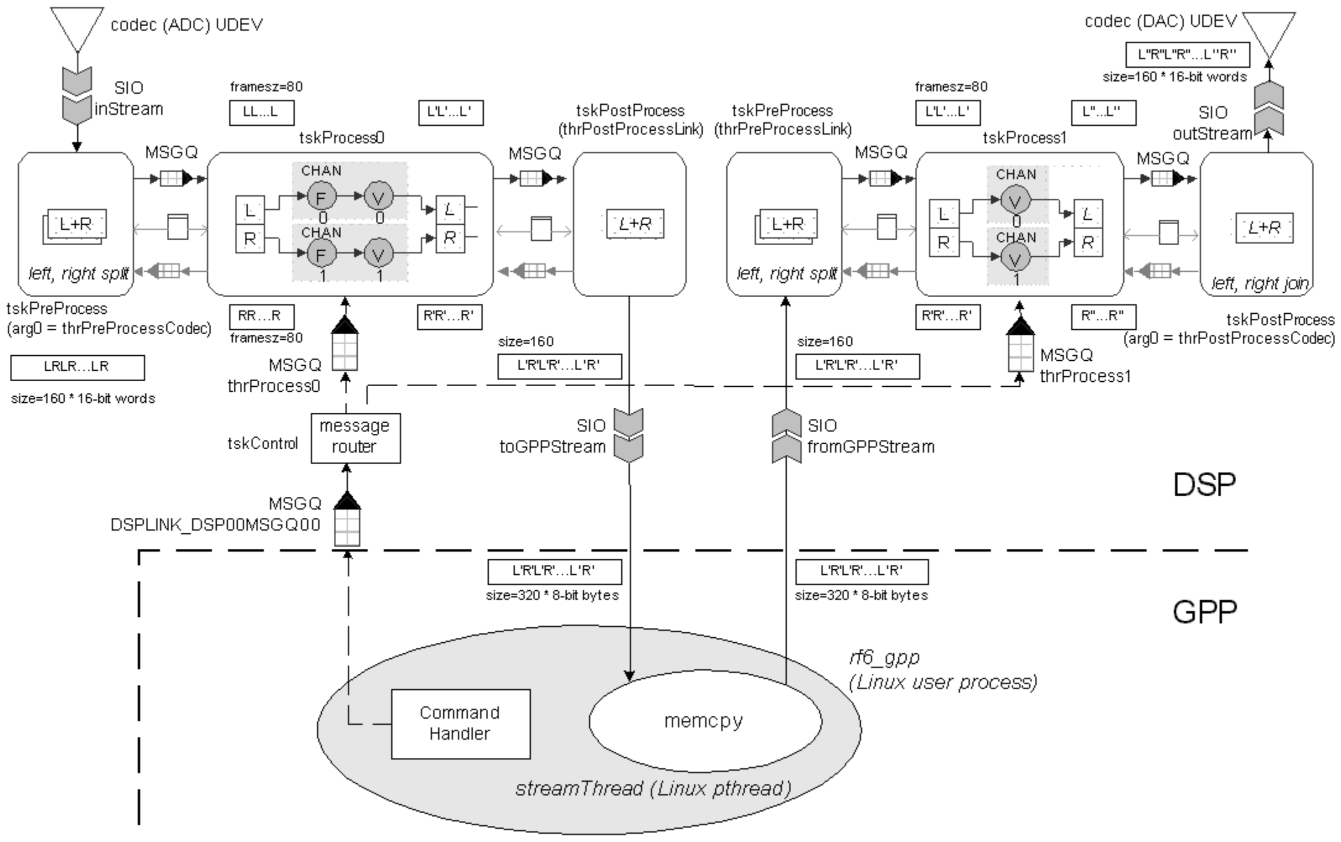


Figure 18. Detailed RF6 Data Flow Diagram

5.5 Folder Hierarchy

You can explore RF6 by examining the folder trees that contain the RF6 applications and associated files. We recommend that you retain the provided structure for your development.

5.5.1 Linux Workstation

Figure 19 shows the GPP-side folders for RF6 and highlights some important files they contain.

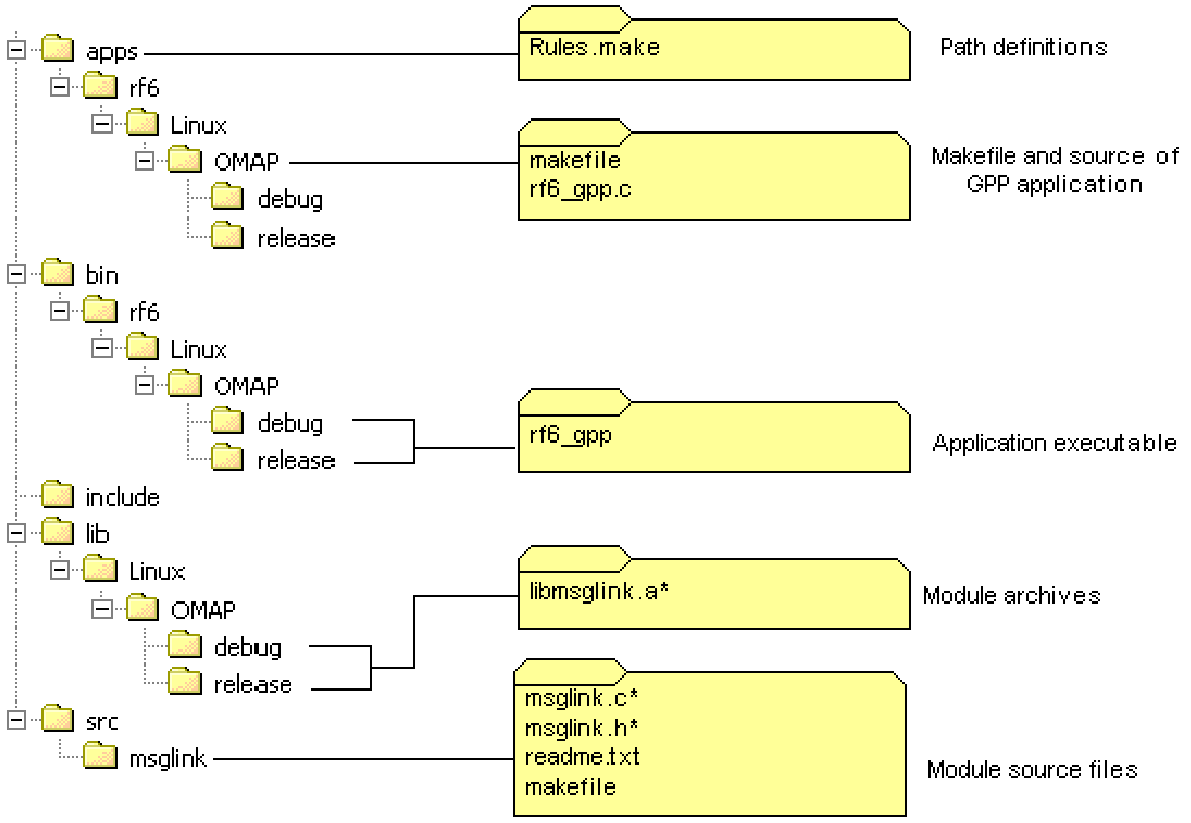


Figure 19. Linux Folder Hierarchy

* The MSGLINK module is deprecated in favor of the MSGQ module.

Folders to notice include:

- **/rf_gpp.** Contains the top-level Rules.make file used to define all the paths necessary for the RF GPP-side build.
- **/include.** Contains OS-independent module header files.
- **/src.** Contains source code for GPP-side modules. Currently it contains only the MSGLINK module. This module has been deprecated. It is provided only for compatibility with previous RF6 applications. Users should move to MSGQ, which is fully supported.
- **/apps/rf6/Linux/OMAP.** Contains source files and the makefile that builds both debug and release versions of the rf6_gpp GPP-side application
- **/bin/rf6/Linux/OMAP.** Contains the rf6_gpp GPP-side application executable, in both debug and release versions.
- **/lib/Linux/OMAP.** Contains the pre-built RF6 libraries

5.5.2 Windows Workstation

The Reference Framework DSP-side folder tree contains application sources and library modules. All the code that RF6-based applications could possibly share has been pushed into libraries for reusability (although you can also find source code for these libraries in the tree). Application-specific files, such as task code and cell wrappers, are in the \apps sub-tree.

Figure 20 shows the DSP-side folders for RF6 and highlights some important files they contain.

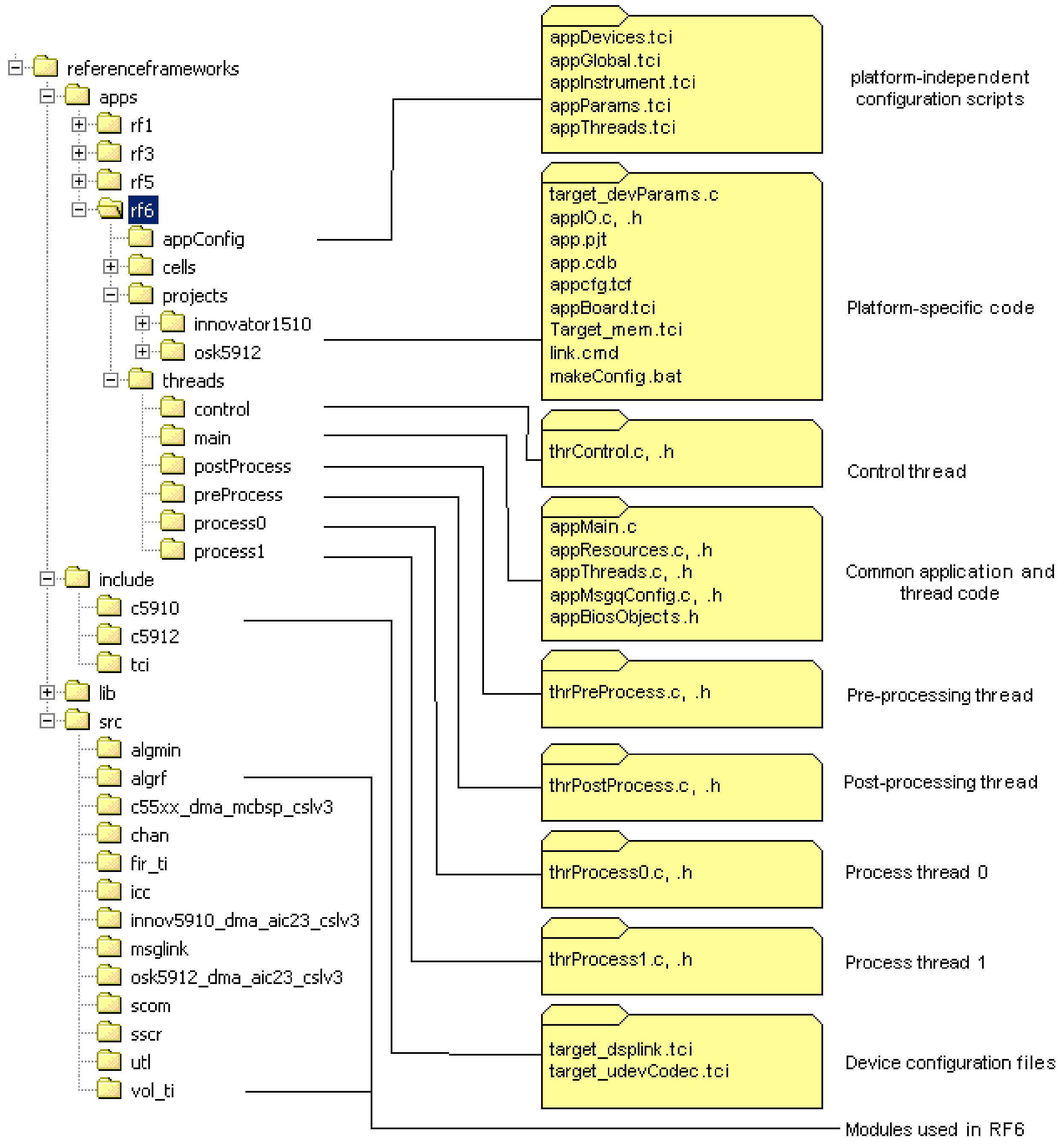


Figure 20. Windows Folder Hierarchy

Folders to notice include:

- **apps\rf6.** The root folder for the RF6 DSP application. To modify it, make a copy of the rf6\ 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 platform-specific files for the RF6 DSP starter-application. This includes board specific configuration files, project files and linker command files. These files are placed in platform-named folders so that RF6 can be provided for multiple platforms. Targets supported as of the publication date are listed in *Appendix C: Reference Framework 6 Board* Ports, page 73.
  - **threads.** Contains hardware-independent source files for the threads.
- **include.** Contains a number of public header files used by the Reference Frameworks. RF6 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. The DSP/BIOS Link libraries are also located here. RF6 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 RF modules. It includes .pjt files to rebuild each library and a readme.txt for each module. The readme.txt files provide information about the modules and their use. Library modules typically need little or no modification.

6 Modules Introduced in RF6

This section describes only those modules that are used in RF6, but in no other Reference Frameworks.

RF6 shares a number of modules with RF5. These modules—ALGRF, CHAN, ICELL, ICC, SSCR, and UTL—are described in *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795) and detailed in *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147).

6.1 MSGQ: Variable Length DSP<->GPP Messaging

RF6 uses the MSGQ module to send control messages from the GPP to the DSP. See the *MSGQ Module: DSP/BIOS Support for Variable Length Messaging* (SPRA987) application note for details about the MSGQ module on the DSP. See *DSP/BIOS Link Messaging Component* (LNK_031_DES) for details about the MSGQ module on the GPP. This section provides an overview of the module and describes how it is used in RF6.

Messages are sent and received via a message queue. A reader is a thread that gets (reads) messages from a message queue. A writer is a thread that puts (writes) a message to a message queue. Each message queue has one reader and can have many writers. A thread may read from or write to multiple message queues.

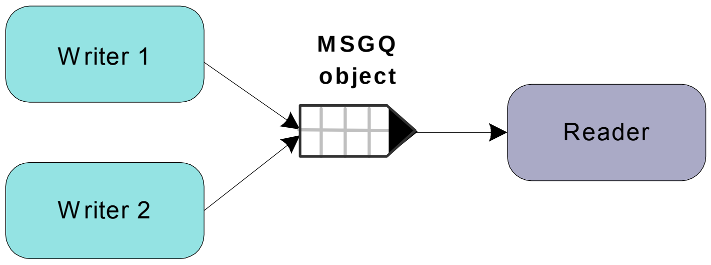


Figure 21. Writers and Reader of a Message Queue

Conceptually, the reader thread owns a message queue. The reader thread creates a MSGQ object. Writer threads locate existing MSGQ objects to get access to them.

The MSGQ module has the following components:

- **Allocators.** Messages sent via MSGQ must be allocated by an allocator. The allocator determines where and how the memory for the message is allocated.
- **Transports.** Transports are responsible for moving messages to the proper destination. That destination can be either on the local processor or a remote processor.
- **MSGQ API.** Applications call the MSGQ functions to create and use a message queue object to send and receive messages.

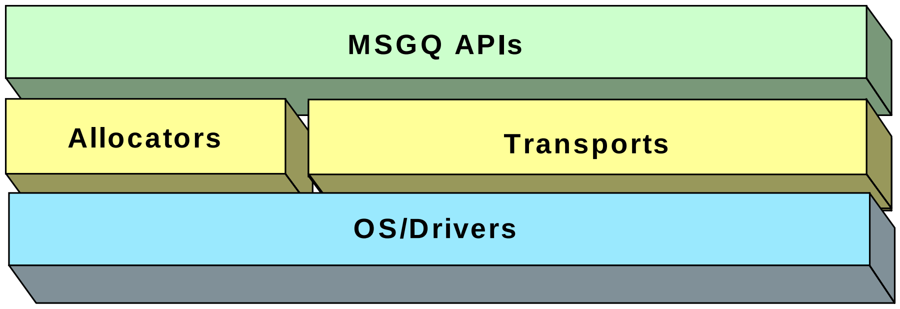


Figure 22. Components of the MSGQ Architecture

The MSGQ module describes standard interfaces that allocators and transports must provide. The allocator and transport interface functions are called by the MSGQ functions and not by the user application.

6.1.1 Allocator

DSP/BIOS Link ships a BUF-based allocator for both the GPP and DSP. The BUF module is part of DSP/BIOS (see DSP/BIOS API Reference for your DSP platform). The DSP/BIOS BUF module has been ported to the GPP with the same functionality.

The BUF-based allocator can manage multiple pools of buffers. The number of pools, the number of messages in each pool, and the size of the messages in each pool are configured at initialization time on both the DSP (via MSGQ_init() and the MSGQ_Config structure) and on the GPP (via MSGQ_AllocatorOpen()). These configuration parameters must be the same on both the GPP and DSP.

The allocator allocates memory based on an exact fit. Attempts to allocate a message of a size different from the ones specified will fail. For example, if the application sets up the allocator to manage three pools having 32, 64, and 128 MADU buffers, only MSGQ_alloc() calls with one of these sizes will work.

On the GPP, RF6 uses one allocator instance. Table 8 shows the two pools in the GPP allocator.

Table 8. GPP Allocator Message Pools

Pool Description	Size	Number of Messages in Pool
GPP->DSP control messages	sizeof(CtrlMsg)	NUMCTRLMSGS
Messages used internally by the remote transport	RMQT_CTRLMSG_SIZE	NUMMQTCTRLMSGS

On the DSP, RF6 uses two allocator instances. The first is used for DSP<->GPP messages. This allocator instance mirrors the GPP allocator instance to allow easy removal of messaging between the DSP and GPP. It also provides for an easier visual verification that the sizes on both sides are the same (as required by this allocator implementation).

Table 9. First DSP Allocator Instance's Message Pools

Pool Description	Size	Number of Messages in Pool
GPP->DSP control messages	sizeof(CtrlMsg)	NUMCTRLMSGS
Messages used internally by the remote transport	MQTDSPLINK_CTRLMSG_SIZE	NUMMQTCTRLMSGS

The other allocator instance on the DSP manages messages passed between different threads on the DSP.

Table 10. Second DSP Allocator Instance's Message Pools

Pool Description	Size	Number of Messages in Pool
Messages used to pass data between DSP processing threads	sizeof (LocalDataMsg)	NUMDATAMSGS

6.1.2 *Transports*

DSP/BIOS Link provides two transports on the GPP side—one local and one remote. DSP/BIOS Link provides one transport for the DSP side—the matching remote transport. RF6 uses these transports along with the local transport (for the DSP) that is shipped with MSGQ.

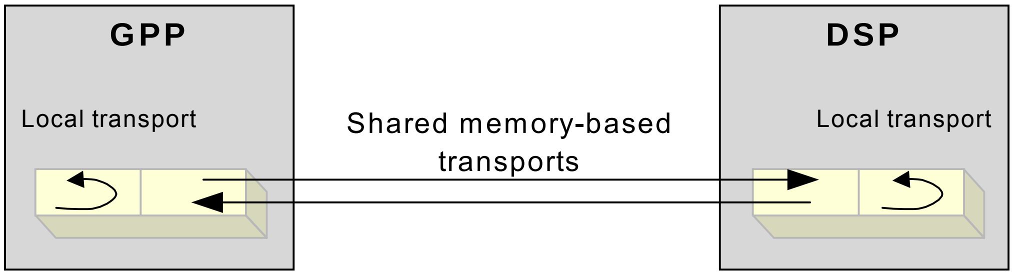


Figure 23. Transports in RF6

6.1.3 *MSGQ API*

The MSGQ APIs are not the same on the GPP and DSP. On the DSP, the DSP/BIOS coding conventions were followed. On the GPP, the DSP/BIOS Link coding conventions were followed.

6.1.3.1 MSGQ API on the DSP

MSGQ is currently distributed separately from DSP/BIOS. In the future, it will be integrated with DSP/BIOS. Reference information for the MSGQ module is provided in the *MSGQ Module: DSP/BIOS Support for Variable Length Messaging* (SPRA987) application note. This section describes how it functions and how it is used in RF6.

Figure 24 shows the call sequence of the main MSGQ functions on the DSP:

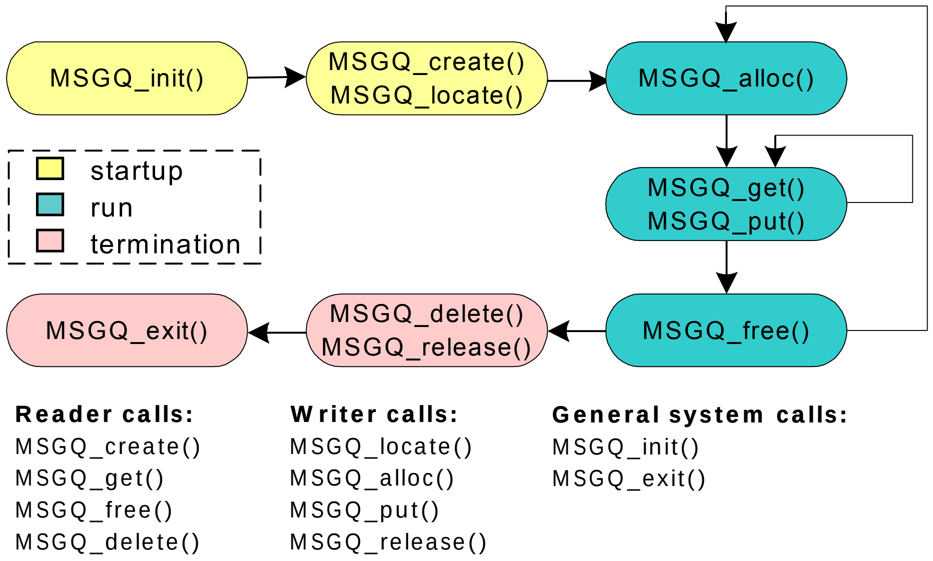
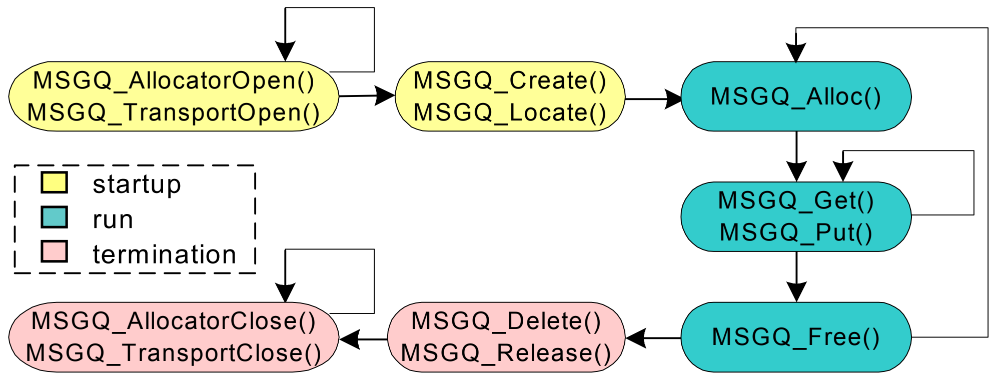


Figure 24. MSGQ Function Calling Sequence on the DSP

6.1.3.2 MSGQ API on the GPP

MSGQ on the GPP is currently distributed with DSP/BIOS Link. Refer to the *DSP/BIOS Link Messaging Component* (LNK_031_DES) document for more details.

Figure 25 shows the call sequence of the main MSGQ functions on the GPP.



Reader calls:	**Writer calls:**	**General system calls:**
MSGQ_Create()	MSGQ_Locate()	MSGQ_AllocatorOpen()
MSGQ_Get()	MSGQ_Alloc()	MSGQ_AllocatorClose()
MSGQ_Free()	MSGQ_Put()	MSGQ_TransportOpen()
MSGQ_Delete()	MSGQ_Release()	MSGQ_TransportClose()

Figure 25. MSGQ Function Calling Sequence on the GPP

6.1.4 MSGQ in RF6

The MSGQ module is used in several ways throughout RF6. The following sections describe how it is used and why certain design decisions were made.

6.1.4.1 GPP—>DSP Control Messages

The MSGQ module is used to send control messages from the GPP to the DSP control thread. The control thread then forwards the control requests to the appropriate destination (for example, tskProcess0 or tskProcess1).

The control thread is not actually required. The GPP could send messages directly to the final destination without the help of a control thread. However, using a separate control thread provides for easy modification of the priority of control message handling. If control messages must be processed quickly, simply raise the priority of the control thread above the level of the processing threads. If control messages should not interfere with the data flow, set the priority of the control thread lower than that of the processing threads. The control thread could be a SWI to reduce footprint, but then the control thread would always have a higher priority than the processing threads, which might not be acceptable for some applications.

6.1.4.2 DSP Inter-Thread Communication

RF6 uses MSGQ to move data between the processing threads. Each processing task has its own message queue. Preprocessing tasks use their message queues to receive empty data buffers. Postprocessing tasks use their message queues to receive full processed data. The processing tasks (tskProcess0 and tskProcess1) use their message queue for several purposes: receiving full data buffers ready to be processed, and receiving empty data buffers ready to be filled with processed data and control messages. Figure 26 shows the threads that write to the processing tasks' message queues.

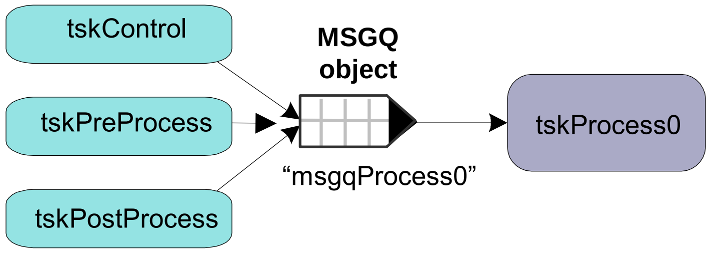


Figure 26. Writers of Processing Task Message Queues

The processing tasks use the MSGQ_getMsgId() API to determine which type of message they have received. Having one message queue per task allows a task to block at one place; thus removing the need to perform any type of polling.

6.1.4.3 Allocators

As mentioned in Section 6.1.1, there are two allocator instances on the DSP. This allows for easy removal of the control messages from the DSP. (You would simply remove the first allocator from the configuration.)

The remote transport is a copy-based transport and the allocator implementation is an exact fit model. Thus the allocator configuration of the control message sizes must match exactly on both the DSP and GPP. Having a dedicated allocator on the DSP for the GPP<->DSP messages makes it easier to check for a match with the GPP configuration.

6.1.4.4 Endianism

The MSGQ module does not perform any type of endian or word-size conversion on the user portion of the message. It does perform conversions on the required header portion of the message (MSGQ_MsgHeader on the DSP and MsgqMsgHeader on the GPP). For more details refer to Section 9.2, Endianism and External Memory on OMAP591x.

6.1.4.5 MSGQ Configuration on the DSP

To minimize footprint, MSGQ requires that the application provide the configuration of the module via the MSGQ variable. RF6 isolates the DSP configuration code in the appMsgqConfig.c and appMsgqConfig.h files.

The following code shows how the MSGQ variable is declared.

```
static const MSGQ_Config msgqConfig = {
        (MSGQ_Obj *)msgQueues,
        (MSGQ_AllocatorObj *)allocators,
        (MSGQ_TransportObj *)transport,
        NUMMSGQUEUES, NUMALLOCATORS, NUMTRANSPORTS,
        MQTLOCALBIOSQUEID};                         /* local transport */
MSGQ_Config *MSGQ = (MSGQ_Config *)&msgqConfig;
```

RF6 has 7 messages (NUMMSGQUEUES) on the DSP. These are the actual MSGQ objects that are returned by MSGQ_create(). If the control thread or any processing threads are removed, this number can be made smaller to save footprint.

RF6 uses two allocators. The following code sets up the allocators array. The first part of the code snippet is the allocator-specific configuration.

```
static const Uint16 localMsgSizes[NUMLOCALMSGPOOLS] =
    {sizeof(LocalDataMsg)};                 /* Size of local data messages */

static const Uint16 remoteMsgSizes[NUMREMOTEMSGPOOLS] =
    {sizeof(CtrlMsg),                       /* Size of control messages */
     MQTDSPLINK_CTRLMSG_SIZE};              /* Required by dsplink transport */

static const Uint16 numLocalMsgs[NUMLOCALMSGPOOLS]  = {NUMDATAMSGS};
static const Uint16 numRemoteMsgs[NUMREMOTEMSGPOOLS]= {NUMCTRLMSGS,
                                                       NUMMQTCTRLMSGS};

static const MQABUF_Params localMqaBufParams =
    {NUMLOCALMSGPOOLS,                      /* Number of buffer pools       */
     (Uint16 *)localMsgSizes,               /* Msg sizes for each pool      */
     (Uint16 *)numLocalMsgs,                /* Number of msgs for each pool */
     0};                                    /* Segment for DSP/BIOS objects */
static const MQABUF_Params remoteMqaBufParams =
    {NUMREMOTEMSGPOOLS,                     /* Number of buffer pools       */
     (Uint16 *)remoteMsgSizes,              /* Msg sizes for each pool      */
     (Uint16 *)numRemoteMsgs,               /* Number of msgs for each pool */
     0};                                    /* Segment for DSP/BIOS objects */

static const MSGQ_AllocatorObj allocators[NUMALLOCATORS] =
    {{"MQABUFREMOTE",                       /* Name of the allocator    */
      (MSGQ_AllocatorFxns *)&MQABUF_FXNS,   /* Allocator interface fxns*/
      (MQABUF_Params *)&remoteMqaBufParams, /* Allocator configuration */
      NULL,                                 /* Filled in by allocator   */
      REMOTEMQABUFID},                      /* Allocator Id             */
      {"MQABUFLOCAL",                       /* Name of the allocator    */
     (MSGQ_AllocatorFxns *)&MQABUF_FXNS,    /* Allocator interface fxns*/
      (MQABUF_Params *)&localMqaBufParams,  /* Allocator configuration */
      NULL,                                 /* Filled in by allocator   */
      LOCALMQABUFID}                        /* Allocator Id             */
    };
```

The following snippet shows the configuration of both the local and remote transports. Note that the local transport has no configuration options.

```
static const MQTDSPLINK_Params mqtDspLinkParams =
    {MAXNUMREMOTEMSGQS,                          /* max remote MSGQs */
     MQTDSPLINK_CTRLMSG_SIZE,                    /* max message size */
     REMOTEMQABUFID};                            /* default MQA ID   */

/*
 *  Transports static Configuration
 */
static const MSGQ_TransportObj transport[NUMTRANSPORTS] =
 {{"MQTLOCALBIOSQUE",                            /* Name of the transport    */
  (MSGQ_TransportFxns *)&MQTBIOSQUE_FXNS,        /* Transport interface fxns */
     NULL,                                       /* Transport config.        */
     NULL,                                       /* Filled in by transport   */
     MQTLOCALBIOSQUEID},                         /* Transport Id             */
     {"REMOTEMQT",                               /* Name of the transport    */
     (MSGQ_TransportFxns *)&MQTDSPLINK_FXNS,     /* Transport interface fxns */
     (MQTDSPLINK_Params *)&mqtDspLinkParams,     /* Transport configuration  */
     NULL,                                       /* Filled in by transport   */
     REMOTEMQTID},                               /* Transport Id             */
     };
```

7 Limitations of the RF6 Application

Because RF6 consists of two parts—the GPP and the DSP applications—there are constants and data structures that need to match in both parts. This section describes the entities that must match. When you are modifying RF6 for your use, refer to this section for an explanation of what needs to be changed in both the GPP and DSP applications. This will help you avoid difficult to detect bugs.

7.1 Control Message Data Structure

To exchange messages between the GPP and the DSP, the data structure of the control messages should match. For the base RF6 application, the DSP side defines the following data structure:

```
typedef struct CtrlMsg {
    MSGQ_MsgHeader header;
    Uint16 cmd;                          // Message code
    Uint16 arg1;                         // First message argument
    Uint16 arg2;                         // Second message argument
} CtrlMsg;
```

On the GPP side, a matching data structure is defined:

```
typedef struct CtrlMsg {
    MsgqMsgHeader header;
    uint16_t cmd;    // Message code
    uint16_t arg1;   // First message argument
    uint16_t arg2;   // Second message argument
} CtrlMsg;
```

Because the Uint16 type on the DSP is of the same size (16 bits) as an unsigned short int (uint16_t) on the GPP, the two data structures match and the application can interpret the control messages correctly on both the DSP and GPP sides. If you increase the size of the fields (for example, to 32-bit fields), the application must perform endianism swapping on OMAP targets in order to interpret the messages correctly.

7.2 Data Channels and Buffers

DSP/BIOS Link allows you to set up many data channels between the GPP and DSP. RF6 uses two DSP/BIOS channels for data streaming. The channel numbers or identifiers must match on the DSP and the GPP sides to ensure correct data transfers.

A change in channel numbers can occur with the data streaming channels. On the GPP side, the application makes the following declarations:

```
#define CHNL_ID_OUTPUT 0
#define CHNL_ID_INPUT 1
```

The DSP side sets the following matching stream names in appThreads.c:

```
thrPreProcessConfig.inStreamName = "/dio_dsplink0";
   ...
thrPostProcessConfig.outStreamName = "/dio_dsplink1";
```

This sets channel 0 to act as the output from the GPP and the input to the DSP, and channel 1 to work in the reverse direction. Should you want to add data channels, it is important to remember to set up the data channels in consistent fashion.

Note that the default RF6 application has no endianism issues on OMAP because the audio sample size is 16-bit. However, if you change the sample data type size, the application must perform endianism swapping to get correct data on both sides.

7.3 Link Buffer Sizes

Another quantity that needs to match on the GPP and DSP sides is the size of the buffers exchanged through DSP/BIOS Link. The sizes of control messages are guaranteed to match if both sides use equivalent data structures for control messages, as described in Section 7.1. On the other hand, data buffers exchanged during data streaming are defined as follows:

Table 11. Size of Audio Buffers

	Symbolic value	Actual default value
DSP side	FRAMELEN * NUMCODECCHANS	80 * 2 = 160 words (16-bit)
GPP side	BUFFERSIZE	320 bytes (8-bit)

These sizes must match in order for data to be copied correctly by DSP/BIOS Link. Naturally any buffer size can be used (within the limits of the available DSP memory). You simply need to ensure that the size matches up on both sides.

8 Adapting the RF6 Application

Adapting RF6 to a custom application can range from changing its XDAIS algorithms to modifying the data flow between the GPP and DSP. When beginning the adaptation process, make a copy of the RF_DIR\apps\rf6 folder tree and call it RF_DIR\apps\my_rf6 for example.

This section describes the following adaptations that affect communication between the GPP and DSP:

- Adapting RF6 for an MP3 Decoder, Section 8.1
- Adding Status Threads, Section 8.2

The application-specific source code that glues the RF6 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.

Adaptations that affect only the DSP side of the application are described in SPRA795, which discusses RF5. That application note discusses the following adaptations:

- Adding cells and XDAIS algorithms
- Adding channels and threads
- Porting the DSP configuration
- Changing the audio device driver
- Providing communication between DSP threads running at different rates.

8.1 Adapting RF6 for an MP3 Decoder: One Directional Data Flow

RF6 is a suitable starting point for portable multimedia products such as MP3 players. Internet audio systems often decode a wide variety of music formats such as MP3 and WMA. However encoding is often not required in such systems. RF6 is easy to adapt to applications that decode only. Figure 27 shows a block diagram for the hardware in a typical Internet Audio system:

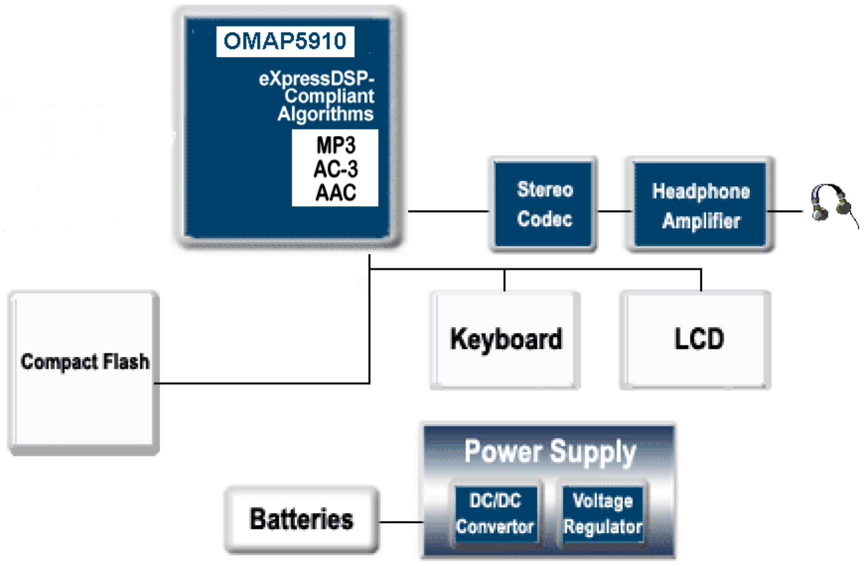


Figure 27. Internet Audio Decoder Hardware

How does this translate to the threads, streams, and XDAIS algorithms in RF6? Figure 28 shows a potential mapping.

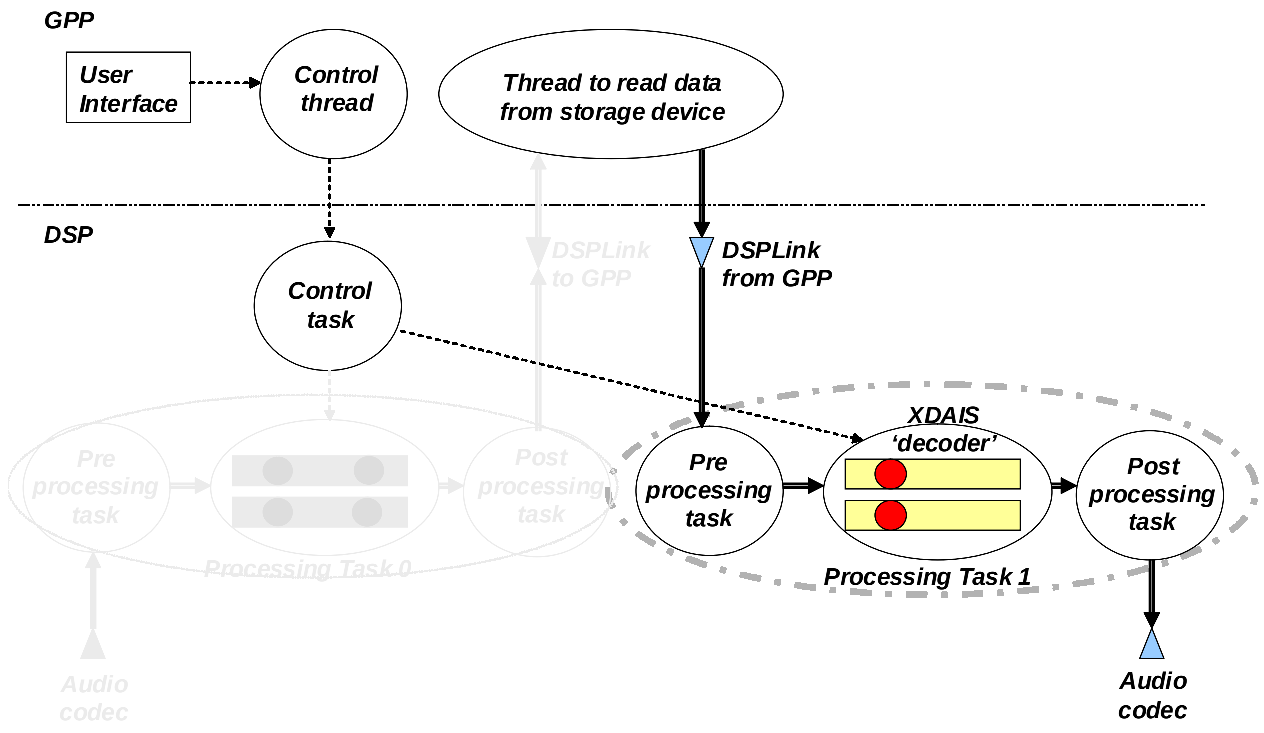


Figure 28. RF6 Data Flow for Internet Audio Decoder

In Figure 28, portions of the base RF6 application that are removed for this adaptation are grayed out. Notice that the data flow is directly taken from the default RF6 application. We simply remove one of the "task triplets" from the DSP side. In a decoder-only system there is no input from an audio codec. Instead the source is a storage device such as Compact Flash or a hard disk.

MP3 encoded data is retrieved from the GPP-side filesystem and sent via DSP/BIOS Link to the DSP in chunks. Some pre- and post-processing may or may not be required, but the key element is the processing task. It would be reconfigured to run an MP3 decoder XDAIS algorithm instead of the dummy VOL algorithm. Although the algorithm is significantly more complex, its integration to RF6 is not.

The decoded data is then streamed out to the D/A converter. The data flow in Figure 28 assumes it is a stereo codec. There are two instances of the MP3 decoder cell—one for the left channel, one for the right. Admittedly, MP3 algorithms often accept stereo input data, hence only a single channel may be required.

Control commands such as buttons pressed to play, stop, and change volume fit nicely into the DSP control task and its partner control thread on the GPP.

This adaptation shows that it is straightforward to flip the inputs and outputs. Such revectoring of the inputs and outputs is made possible by the fact that DSP/BIOS Link is just another IOM driver on the DSP. We can plug any logical IOM driver into RF6 and expect it to work, since all drivers must adhere to the same IOM_Fxns interface.

Another common adaptation might be to cut DSP/BIOS Link data transfers out of the equation completely and use DSP/BIOS Link only for boot-loading the DSP executable. In this case, you would modify the TCONF scripts and associated SIO_create code to plug the audio input to the preprocessing task of the second task triplet. You would eliminate the first task triplet as in this decoder-only solution, but in addition, you would remove the data streaming path from the GPP to the DSP.

8.2 Adding Status Threads to RF6

A “heartbeat” signal is a common feature in multi-processor systems. Typically, the DSP periodically sends a signal to inform the host GPP that it is still running. Since DSPs typically don't have protected modes, they often rely on the host for recovery when they stop sending the “heartbeat” signal. As Figure 29 shows, it is straightforward to add such a DSP status thread to RF6.

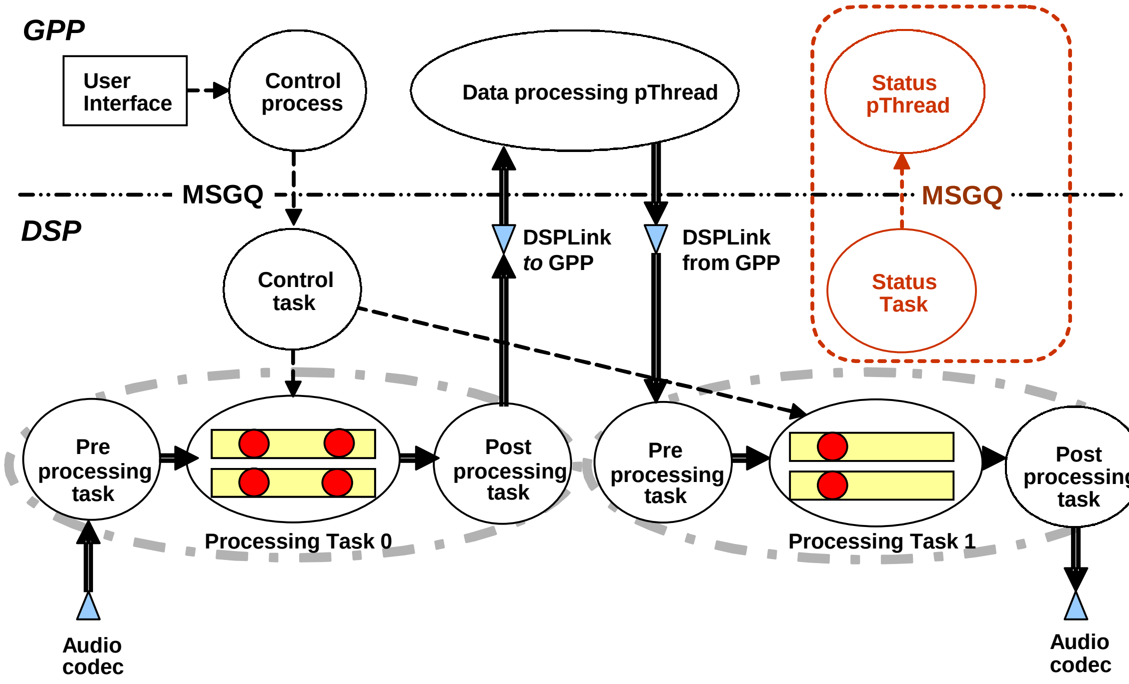


Figure 29. Adding Status Thread(s) to RF6

The DSP status task does the opposite of the control task. That is, it calls MSGQ_put() to deliver data to the GPP via DSP/BIOS Link. Its partner on the GPP might be a Linux pThread receiving status messages with MSGQ_get().

In the GPP status pThread initialization, it must create a message queue on which it will receive the status message from the DSP. The DSP status thread must then locate this message queue (via MSGQ_locate) as part of its initialization.

Rules for control messages also apply to status messages. That is, the message sizes must match on both processors, you should keep messages short, and you should try to use 16-bit fields to avoid OMAP endianism issues.

The DSP status task can have any priority, but it is best to give it a lower priority than the core data streaming and control tasks. Under high load conditions, the status task may be temporarily starved, but the integrity of the system would hold up. This is the same principle used by the DSP/BIOS RTA tools running in the IDL background loop, yielding to more important processing threads.

Naturally more than just a heartbeat “keep-alive” can be sent by the DSP. The following example also periodically sends an approximation of the current DSP CPU load.

```
typedef struct StatusMsg {
    MSGQ_MsgHeader header;
    Uint16 cmd;                         // message code
    Uint16 arg1;                        // first message argument
} StatusMsg;
...

Void thrStatusRun()
{   StatusMsg * msg;
    MSGQ_Status status;
    MSGQ_Handle gppQueue;
    Uns flag = 0;
    ...

    while ( TRUE ) {    // Main loop
        msg = (StatusMsg *)MSGQ_alloc(REMOTEMQABUFID, sizeof(StatusMsg));
        UTL_assert( msg != NULL );
        if (msg != NULL) {
            switch (flag) {
                case MSGSTATUSDSPKEEPALIVE:        // actions for DSP CPU load msg
                    msg->cmd = MSGSTATUSDSPKEEPALIVE;
                    msg->arg1 = ALIVEMSGARG;
                    break;
                case MSGSTATUSDSPCPULOAD:          // actions for ‘keep-alive’ msg
                    msg->cmd = MSGSTATUSDSPCPULOAD;
                    msg->arg1 = (MdUns)LOAD_getcpuload();
                    break;
                default:
                    UTL_logDebug("flag is invalid\n");
                    break;
            }

            // alternate between message types
            flag++;
            flag %= NUM_THRSTATUSENUM_ELEMENTS;

            // send the message via Link to GPP
            status = MSGQ_put(gppQueue,        /* destination message queue */
                (MSGQ_Msg)msg,                 /* the message               */
                STATUSMSGID,                   /* message ID                */
                NULL);                         /* reply message queue       */
            UTL_assert(status == MSGQ_SUCCESS);
        }

        // suspend self for X ticks, and then poll again
        TSK_sleep( NUMSLEEPCLOCKTICKS );
    }
}
```

The status POSIX thread on the GPP blocks to wait for a message to arrive, interprets the message, and in this simple case prints the result to the serial terminal.

```
while( flag == 1 ) {          // body of status pThread

    // Block-wait on a msg from DSP
    status = MSGQ_Get( GPPMSGQID, WAIT_FOREVER, (MsgqMsg *) &msg );
    if (!DSP_SUCCEEDED (status)) {
        fprintf(stderr, "Error receiving status message\n");
    }

    switch (msg->cmd) {
    case MSGSTATUSDSPCPULOAD:
        if (printStatusEnabled) {
            printf("Approx DSP Cpu load = %d %%\n", msg->arg1);
        }
        break;
   case MSGSTATUSDSPKEEPALIVE:
        if (printStatusEnabled) {
            printf(".\n");
        }
        break;
    default:
        printf("I'm in the default\n");
        break;
    }

    status = MSGQ_Free((MsgqMsg) msg);
    if (!DSP_SUCCEEDED(status) ) {
        fprintf(stderr, "Error freeing message\n");
    }
}
```

This adaptation can be used as an automated regression and stress test. CPU load data can be sent from the DSP to the host for a range of codec sampling frequencies and buffer sizes.

9 Advanced Topics

This section describes topics you should explore after you have learned the basics of how to run and adapt RF6. Depending on the modifications you make to the base RF6 application, you may or may not need to understand these topics.

9.1 Buffer Priming

Priming of buffers prevents gaps in the signal output. It does this by increasing the latency slightly so that there is time for variation in the amount of time required to process buffers. In RF6, both the output codec and the GPP–DSP data communication channels are primed.

9.1.1 Priming the Codec

To explain how the codec is primed, let us look at an example where priming is not performed. Assume that frames are 80 16-bit samples long at an 8 kHz sampling rate. That is, a new input frame arrives every 10 ms. Therefore, an output frame must be sent every 10 ms. As soon as the transfer of one output frame is complete, the next processed frame must be ready. If not, the output is silent until the next output frame is ready, which is not what we want.

Assume that the first and second input frames take 4 ms to process, and the third frame for some reason takes 7 ms to process. Assume also that both input and output pipes are double-buffered. Figure 30 shows a time diagram of the events:

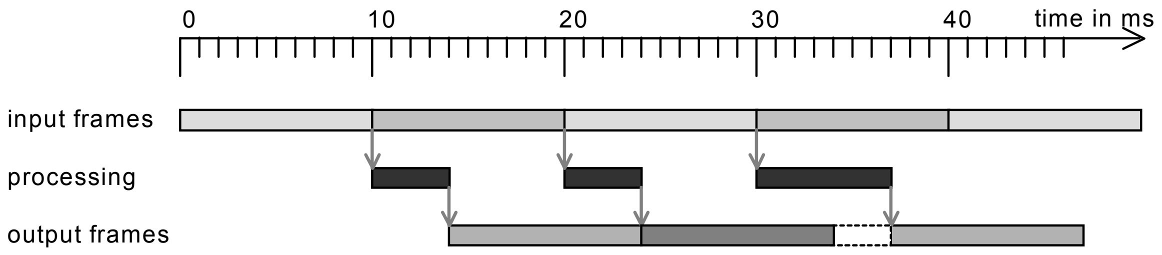


Figure 30. Missed Real-Time Deadline with Output Pipe Not Primed

The first output frame is sent at t=14 ms, which is 4 ms after the first input frame arrived. Processing the second input frame completes at t=24 ms, exactly the same time the transfer of the first output frame completes. However, at t=34 ms the DMA is ready to transfer the third output frame, which is still being processed. When processing completes 3 ms later, the third output frame can be transferred. However, we do not want the 3 ms gap, which can occur whenever the processing time is larger than all previous processing times. Even if processing is always complete within 10 ms, at times we miss the real-time deadline and produce incorrect output.

This problem is easily solved by priming the output—by giving the output device some zeroes to transfer while we process the initial frame. If we prime the output pipe with two silent frames of 80 samples (full size), we get the diagram shown in Figure 31:

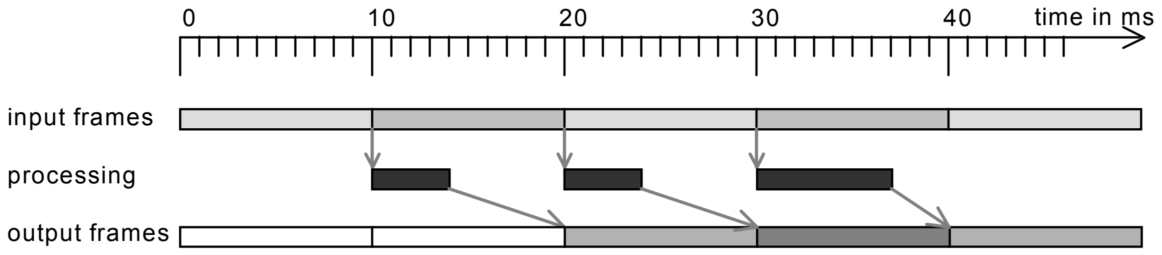


Figure 31. Continuous Output with Primed Output Pipe

The two initial output frames are full of zeroes. As a result, there is continuous output irrespective of processing time variance. If the third frame takes 7 ms to process, the real-time constraints are met because the output frames are time-shifted. Variance in processing time (up to 10 ms) does not affect the output, since output frames are transferred every 10 ms regardless of how much earlier they are ready. As a result, we have correct output with no gaps.

9.1.2 Priming the GPP–DSP Data Channels

The GPP can be thought of as a black-box device that processes data at a variable rate, depending on its operating system (for example, a soft real-time OS) and its CPU load. To maintain a fixed data rate to the output audio codec on the DSP, the GPP needs to be primed with an adequate amount of buffering. Then, if the GPP is busy servicing higher-priority processes, the DSP can continue to consume buffered data.

In addition to the traditional codec priming described in the previous section, RF6 also primes the data channels going out to the GPP and coming in from the GPP. Assume that we prime DSP/BIOS Link channels in RF6 using w, x, y and z number of buffers as shown in Figure 32.

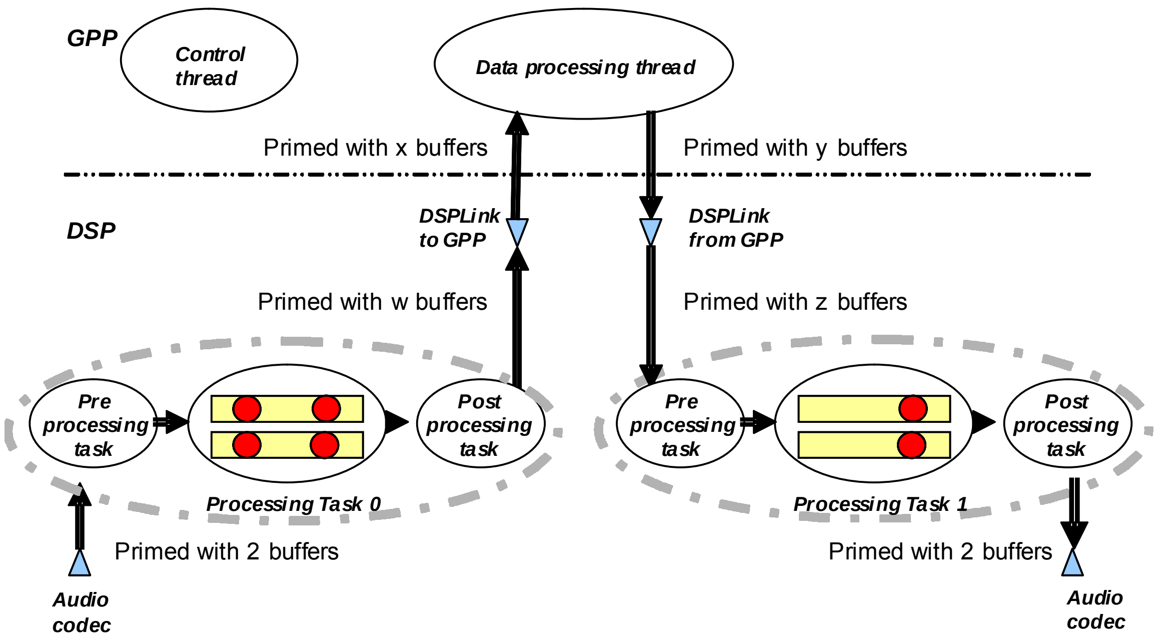


Figure 32. Data Channel Priming in RF6

The output priming amounts—w and y—add to the latency of the system and represent actual buffers of silences to be played by the audio codec. The input priming amounts—x and z—are used by the DSP/BIOS Link driver to queue up data from the DSP or from the GPP.

The default RF6 application sets all four priming amounts to 2 to enable double buffering on both sides of both channels. This is sufficient if the GPP application does little additional processing. However, if a higher-priority thread wakes up on the GPP and performs a significant amount of processing, the GPP might starve the DSP of data for some period of time. For example, the control thread might suddenly need to perform a large amount of standard I/O using printf statements. This would cause the DSP application to miss real-time deadlines, throwing the DSP out of sync with respect to the codec priming.

To allow the GPP more time, you can increase z and either w or y to supply more silent buffers during initialization. For example, if you increase z and w to 3, the GPP can be busy for up to 3 buffer delays, after which it would need to catch up on data processing to refuel the buffers.

An example of this would be when RF6 is run in Debug mode using the Debug version of DSP/BIOS Link where the GPP CPU load is quite high. If you send frequent control messages to the DSP, the GPP can "get behind" and starve the DSP of audio data despite the double buffering.

The total system latency is determined by the number of buffers used to prime the output to the GPP (w), to the DSP (y), and to the output side of the codec driver. In Figure 32, these correspond to w + y + 2 = 6 buffers. For example, a system with an 8 kHz sampling rate and a buffer size of 80 samples would have its latency calculated as follows:

```
(80 samples/buffer * 6 buffers)/(8000 samples/sec) = 60 milliseconds
```

This value increases if you add more buffering to counteract variable real-time behavior of the OS on the GPP. In cases where the CPU load is low on the GPP, you may be able to reduce w and y to 1 to decrease the latency to 4 sample buffers.

One way to lower the CPU load on the GPP might be to increase the buffer size, but a larger buffer size also increases the latency. You should find the best compromise between latency and meeting real-time deadline through experimentation. Note that the number of buffers used to prime the output codec (see Section 9.1.1) should not be less than 2 to make sure the DSP can meet real-time requirements.

9.2 Endianism and External Memory on OMAP591x

The endianism of a bus or a memory device determines the way data is stored on the bus or in the memory. There are two types of endianism: big endian and little endian. On the OMAP591x, the ARM core uses little endian, while the C55x DSP core is big endian. Because of this difference, sharing external memory between the ARM and the DSP has to be done with care.

But you might ask why we care about external memory? To keep production costs down, internal memory is very limited on the DSP. If you use DSP/BIOS Link for 16-bit transfers only, and the application code is small enough to be put in internal memory, you can ignore this section.

Here are two situations in which the ARM and DSP access the same memory. These situations require endianism conversion:

- The DSP accesses external memories via the Traffic Controller. Endianism conversion is performed at the DSP Memory Management Unit (MMU) interface.
- The ARM accesses DSP resources such as DSP internal memory or DSP public peripherals. Endianism conversion is performed at the microprocessor unit (MPU) interface.

In this section, we are concerned with the first situation. In this case, external memory is made available to the DSP through DSP/BIOS Link's programming of the DSP MMU.

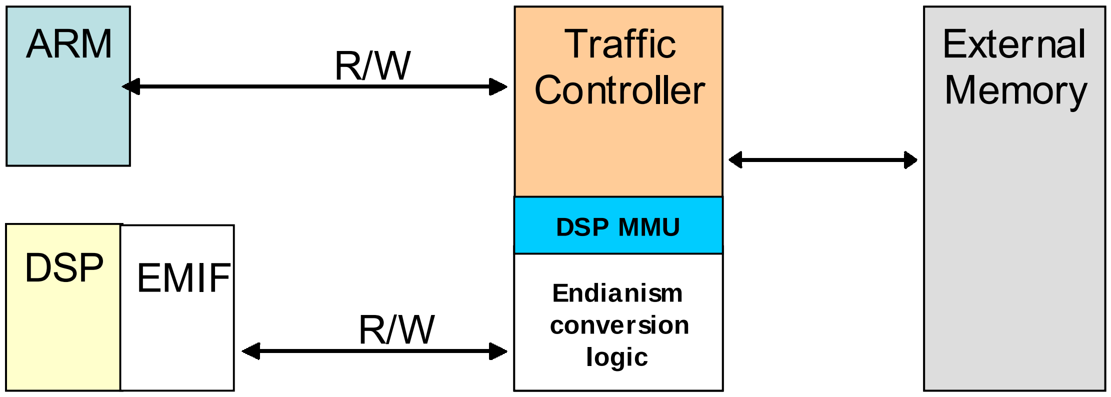


Figure 33. External Memory Sharing Between ARM and DSP

Endianism conversion can be controlled through an ENDIANISM register on the OMAP chip, available on the ARM side. The following three modes are available:

- **NOSWAP.** Register value = 0x0 (default) implies endianism conversion is disabled and there is no swapping.
- **WORDSWAP.** Register value = 0x3 implies endianism conversion is enabled and word (16-bit) swapping is on.
- **BYTESWAP.** Register value = 0x1 implies endianism conversion is enabled and byte (8-bit) swapping is on.

By default, DSP/BIOS Link and RF6 both support the NOSWAP mode. To change the mode, you would need to write your own ARM9 code in Linux kernel mode to change this register, and then modify the code in DSP/BIOS Link to support the new mode.

The rest of this section concentrates on the NOSWAP mode. In this mode, write accesses to external memory from both the ARM and the DSP store data in little endian format. For example, suppose the DSP writes a 32-bit value to external memory as follows:

```
long a;   // 32-bit global variable in .bss, placed in external memory on C55x core

void main() {
    a = 0x12345678;
    ...
}
```

Suppose this value is written to addresses 0x28046 and 0x28047. From a big-endian perspective (the DSPs normal mode), the expected and actual values in these addresses are as follows:

Table 12. Direct DSP Reads and Writes

Variable in External Memory	Address	Expected (Big-Endian)	Actual (Little-Endian)
a	0x28046	0x1234	0x5678
	0x28047	0x5678	0x1234

A 32-bit read from 0x28046 by either the DSP or ARM yields 0x12345678 as expected. However, individual 16-bit reads from the 0x28046 and 0x28047 addresses result in values of 0x5678 and 0x1234 respectively. This is contrary to what a DSP programmer would expect from a big-endian processor.

As a result of this, the simplest advice for system integrators is to avoid placing data sections (such as .const and .bss) in external memory. Any 32-bit value written to external memory by the ARM or DSP that is read back by the DSP as separate 16-bit entities causes unexpected results on the DSP side. Likewise, any value written as separate 16-bit entities and read back as a 32-bit value on the DSP would lead to similar problems.

Since it is difficult to guarantee that 32-bit quantities declared on the heap will not be accessed as individual 16-bit quantities in optimized code (and vice-versa), RF6 configures the “external” memory heap in internal SARAM memory. The following Tconf configuration script is from the Osk5912_mem.tci file in the *RF_DIR*\apps\rf6\projects\osk5912 folder.

```
/* Allocate heap named "EXTERNALHEAP" of size 0x1000 in internal memory.
 * Note that we use internal memory here to avoid endianism issues on the OMAP board.
 */
tibios.SARAM.base            = 0x80ff;
tibios.SARAM.len             = 0xbf01;
tibios.SARAM.createHeap      = true;
tibios.SARAM.heapSize        = 0x1000;   /* 4K  */
tibios.SARAM.enableHeapLabel = true;
tibios.SARAM.heapLabel       = prog.extern( "EXTERNALHEAP" );
```

Code sections, on the other hand, can freely be placed in external memory. When DSP/BIOS Link reads from the COFF file, it automatically stores the code in external memory in little-endian format, using 32-bit writes. Then code is fetched by the C55x DSP into the instruction buffer queue as 32-bit blocks. Therefore the code does not get corrupted. The only reason to leave DSP code sections in internal memory is performance, since the instruction cache has not been enabled on the DSP. (The ICACHE module, which allows control of the instruction cache, will soon be supported by CSL for OMAP.)

For data streaming and control message passing, DSP/BIOS Link does 16-bit reads/writes of data on the DSP side. To avoid the need for endianism swapping, RF6 uses 16-bit fields for the control messages exchanged by MSGQ and a 16-bit sample size. If you want to use larger control message fields or sample sizes, it is important that your programs perform endianism swapping in order to interpret data correctly.

For more details, see the *OMAP5910 Dual-Core MPU Subsystems Reference Guide* (SPRU671) and the *OMAP5912 Multimedia Processor OMAP3.2 Subsystem Reference Guide* (SPRU749).

10 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.

10.1 DSP-Side Performance

The DSP performance characteristics of the RF6 application are shown in Table 13. The framework imposes relatively 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 13. RF6 DSP CPU Usage Statistics

Configuration	DSP CPU Load
RF6 as supplied	20%
RF6, minus FIR and VOL algorithm processing	16%

Load values for the application minus the FIR and VOL algorithm processing were obtained by removing the algorithms. Note that the FIR and VOL algorithms are compiled using the –o2 optimization flag. Measurements were made under the following conditions:

- Platform: OSK running at 192 MHz
- Sampling rate: 44.1 kHz
- Samples per frame: 80 16-bit words
- Optimization flags: -o2 on the RF6 application, Reference Framework modules, and drivers
- Debug flags: -g
- UTL_DBGLEVEL: 0
- Version of DSP/BIOS Link: Release
- All critical code and data in internal memory

10.2 GPP-Side Performance

The GPP performance characteristics of RF6 are shown in Table 2.

Table 14. RF6 GPP CPU Usage Statistics

Configuration	User Time CPU Load	System Time CPU Load
RF6 as supplied	3%	13%

Measurements were made under the following conditions:

- Platform: OSK running at 192 MHz
- Sampling rate: 44.1 kHz
- Samples per frame: 160 8-bit bytes
- Optimization flags: none
- Debug flags: none
- Version of DSP/BIOS Link: Release
- All critical code and data in internal memory

10.3 DSP-Side Footprint

Although RF6 does not aim to create a minimum-footprint application, footprint is always a concern. The actual RF6 DSP footprint (program and data) and the space available for algorithms are shown in Table 15.

Table 15. RF6 DSP Memory Footprint

	OSK (program + data)
RF6 as supplied	48,687 16-bit words
RF6 minus algorithms and application-specific components	25,586 16-bit words

In these calculations:

- “RF6 as supplied” includes the entire application—the application code, algorithms, drivers, CSL, DSP/BIOS, etc.
- 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: RF6 Detailed Memory Footprint, page 65.

10.4 GPP-Side Footprint

Table 16. RF6 GPP Memory Footprint

	OSK (program + data)
RF6 as supplied (debug version)	18,943 8-bit bytes
RF6 as supplied (release version)	13,993 8-bit bytes

In this calculation, the value represents the RF6 GPP application. It does not include Linux or any of the Linux kernel objects (for example, DSP/BIOS Link).

For more details, see Appendix A: RF6 Detailed Memory Footprint, page 65.

11 Conclusion

Reference Framework Level 6 (RF6) is intended to enable designers to create applications that make use of a General Purpose Processor (GPP) in addition to the DSP. Communication between the two processors can include both data streaming and control/status channels.

RF6 also provides the same support as RF5 for many channels and algorithms while minimizing the memory requirements by employing static configuration techniques. Like RF5, RF6 is a solid starting point for extensive, high-density applications.

The framework is supplied as highly reusable C language source code to enable porting to other platforms. The supplied application currently runs on the Innovator5910 EVM and the OSK5912, which contain both an ARM9 processor and a TMS320C55x DSP.

This application note includes framework selection criteria, a list of typical applications, explanations of how the framework functions, and references to related documents.

RF6 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.

12 References

An acquaintance with DSP/BIOS, XDAIS, and DSP/BIOS Link is needed for writing DSP applications at this level.

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). Source code and documentation for DSP/BIOS Link can be found on the DSP/BIOS Link web page: http://www.ti.com/bioslink.

For additional information, see the following sources:

Table 17. Product Documentation

Document	Reference Number	Where to Find It
TMS320 DSP/BIOS User's Guide	SPRU423	TI DSPVillage
TMS320C5000 DSP/BIOS API Reference Guide	SPRU404	TI DSPVillage
DSP/BIOS TextConf User's Guide	SPRU007	TI DSPVillage
DSP/BIOS Driver Developer's Guide	SPRU616	TI DSPVillage
TMS320 DSP Algorithm Standard Rules and Guidelines	SPRU352	TI DSPVillage
TMS320 DSP Algorithm Standard API Reference	SPRU360	TI DSPVillage
MontaVista® Linux® Professional Edition for the Texas Instruments Innovator™ Development Kit for OMAP™ 1510 and OMAP™ 5910	n/a	MontaVista
OMAP5910 Dual-Core Processor Data Manual	SPRS197	TI DSPVillage
Innovator™ Development Kit for the Texas Instruments OMAP™ Platform Deluxe Model User's Guide	n/a	Part of the Innovator Development Kit documentation
DSP/BIOS Link for MontaVista Linux (MVL) User's Guide	LNK_022_USR	Part of DSP/BIOS Link installed documentation
OMAP5910 Dual-Core MPU Subsystems Reference Guide	SPRU671	TI DSPVillage
OMAP Starter Kit for the OMAP5912 (OSK5912) User Guide		Spectrum Digital
OMAP5912 Applications Processor Data Manual	OMAP5912.pdf	TI DSPVillage
OMAP5912 Multimedia Processor OMAP3.2 Subsystem Reference Guide	SPRU749A	TI DSPVillage
DSP/BIOS Link Messaging Component	LNK_031_DES	Part of DSP/BIOS Link installed documentation

Table 18. Application Notes

Document	Reference Number	Where to Find It
Reference Frameworks for eXpressDSP Software: API Reference	SPRA147	TI DSPVillage
MSGQ Module: DSP/BIOS Support for Variable Length Messaging	SPRA987	TI DSPVillage
The TMS320 DSP Algorithm Standard – A White Paper	SPRA581	TI DSPVillage
Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System	SPRA791	TI DSPVillage
Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System	SPRA793	TI DSPVillage
Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System	SPRA795	TI DSPVillage
PlayWave Application Using DSP/BIOS Link	n/a	http://omap.spectrumdigital.com
How to Create, Build, and Debug a Linux Application on OMAP Innovator	SPRA395	TI DSPVillage
Building and Installing Open Source Linux for the OMAP Innovator	n/a	http://omap.spectrumdigital.com
Using IDMA2-Based XDAIS Algorithms in eXpressDSP RF5	SPRA842	TI DSPVillage
A Multi-Channel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation	SPRA904	TI DSPVillage

Web Resources

- TI DSPVillage: http://www.dspvillage.com
- OMAP5912 website: http://omap.spectrumdigital.com
- MontaVista Zone: http://support.mvista.com
- DSP/BIOS Link web page: http://www.ti.com/bioslink

Appendix A: RF6 Detailed Memory Footprint

The overall footprint sizes for RF6 for both GPP and DSP are listed in Section 10, *Performance and Footprint*. This appendix provides details on those numbers. This breakdown assists the system designer in planning an overall DSP-GPP solution.

DSP Memory Footprint

The numbers in Table 19 were obtained using the supplied version of the RF6 DSP-side 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.

DSP-side footprint numbers shown in this appendix were obtained under the following conditions:

- Platform: OSK5912
- Samples per frame (framesize): 80. Each sample is 16-bits.
- Flags: -g –ml –v5510:2
- UTL_DBGLEVEL: 70
- Flavor of DSP/BIOS Link v1.10.01: Release

Note that a framesize of N denotes the framesize an algorithm operates on. Since we are using a stereo codec, the actual size of data buffers handled by the codec and DSP/BIOS Link is 2N. See Figure 18 for a diagram that includes these size issues.

For modules, Table 19 also shows module sizes and the names of sections that can be placed in external memory with little or no impact on performance. Most of these sections are used for initialization, create, startup, or run-once code.

Table 19. RF6 DSP-Side Footprint

Category	Total Size, 16-bit hex words (decimal)	Size of External Memory Placeable code (16-bit words)	Section Name for External Memory Placement
DARAM heap	0x1000 (4096)		
SARAM heap	0x1000 (4096)		
Stack	0x180 (384)		
SysStack	0x100 (256)		
UTL	0x296 (662)		
MSGQ	0x392 (914)		
BIOSQUE MSGQ Transport	0xbe (190)		
DSP/BIOS Link MSGQ Transport (Release)	0x5ac (1452)		
DSP/BIOS Link MSGQ Allocator (Release)	0x190 (400)		
DSP/BIOS LINK (Release)	0x589 (1417)		

Category	Total Size, 16-bit hex words (decimal)	Size of External Memory Placeable code (16-bit words)	Section Name for External Memory Placement
SSCR	0x1de (478)	0x1c8 (456)	.text:init .test:setup .text:SSCR_createBuf .text:SSCR_prime
Real Time System Library (RTS)	0x30e (782)		
ALGRF	0x1db (475)	0x184 (388)	.text:init .text:create .text:setup
Hardware Vector Table	0x80 (128)		
CHAN	0x1a3 (419)	0x14f (335)	.text:init .text:setup .text:CHAN_open .text:CHAN_regCell
FIR Algo	0x116 (278)	0x61 (97)	.text:algInit .text:algAlloc .text:algFree
VOL algo	0xca (202)	0x3e (62)	.text:algInit .text:algAlloc .text:algFree
CELL	0xc8 (200)		
ICC	0x21 (33)	0x21 (33)	.text:init .text:ICC_linearCreate
Application	0x1948 (6472)		
IOM driver	0x5c7 (1479)	0x106 (262)	.text:init
Task Stacks	0x1f00 (7936)		
CSL	0xbc7 (3015)	0x763 (1891)	.text:csl_section:init .text:csl_section:intc .text:csl_section:mcbsp*
LOG	0x26d (621)		
STS	0xbb (187)		
RTDX	0x6a2 (1698)	0x393 (915)	.rtdx_text
BIOS	0x28b1 (10417)		
Total	**0xc717 (48687)**	**0x1157 (4439)**	

* Placing the entire .text:csl_section:mcbsp section in external memory may affect system performance, since calls to CSL_mcbspHwControl function occur frequently in the IOM driver.

However, components such as the FIR and VOL 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 20 shows how the total size of the basic framework code and data is calculated by subtracting various application-specific portions. 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 total size listed in Table 20 is the basic footprint for the supplied Reference Framework Level 6 on the OSK5912.

Table 20. RF6 DSP Basic Framework Size

Total	**48,687 16-bit words**
Unused internal heap	minus 2344
Unused external heap	minus 3972
Application buffers	minus 1920
Heaps used by algorithms	minus 567
VOL algorithm	minus 202
FIR algorithm	minus 278
FIR coefficient table	minus 96
Cell algorithm code	minus 200
Unused stack and SysStack	minus 343
Unused task stack	minus 7020
LOG object for debugging	minus 621
STS object for debugging	minus 187
UTL debugging module	minus 662
RTDX module	minus 1698
All cinit records	minus 2991
Total	**25,586 16-bit words**

When calculating whether algorithms can fit on a particular DSP, use 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.

GPP Memory Footprint

The numbers in Table 21 were obtained using the supplied version of the RF6 GPP application and its libraries. This table provides data for both debug and release versions of the appropriate modules. Sizes were obtained using the ARM tools size utility. All sizes are shown in 8-bit bytes.

GPP-side measurements shown in this section were made under the following conditions:

- Platform: OSK5912
- Samples per frame (framesize): 80 stereo 16-bit samples. This equates to 320 8-bit bytes on the GPP side.
- Compiler flags: Debug version: –g
  Release version: none
- Flavor of DSP/BIOS Link v1.10.01: both Debug and Release (dsplink.lib)

Table 21. RF6 GPP-side Footprint in 8-Bit Bytes

Category	Text	Data	Bss	Total (decimal)	Total (hex)
Total RF6 GPP-Side Application (Debug version)	18,147	376	420	18,943	0x49ff
Total RF6 GPP-Side Application (Release version)	13,197	376	420	13,993	0x36a9

Appendix B: RF6 Performance Analysis

The overall performance numbers for RF6 are listed in Section 10, *Performance and Footprint*. This appendix provides details on those numbers. This appendix also investigates the effect on performance of different buffer sizes and different sampling frequencies.

DSP Measurement Techniques

The status adaptation described in Section 8.2, *Adding Status Threads to RF6*, was used to obtain the CPU usage percentage on the DSP.

The DSP performance was tested in two ways: out of the box and framework-only. The framework-only tests were performed to show the overhead of the RF6 framework. The framework-only tests include data moving, DSP/BIOS, drivers, application code, and more. The only things that were removed were the actual FIR and VOL algorithms.

DSP-side measurements shown in this appendix were made under the following conditions:

- Platform: OSK running at 192 MHz
- Sampling rate: 22.05 kHz and 44.1 kHz
- Samples per frame (framesize): 32, 64, 128, and 256. Each sample is 16-bits.
- Debug flags: -g
- Flavor of DSP/BIOS Link v1.10.01: Release

The following build options were changed:

- Optimization flags: -o2 on the RF6 application, Reference Framework modules, and drivers
- UTL_DBGLEVEL: 0

Note that a framesize of N denotes the framesize an algorithm operates on. Since we are using a stereo codec, the actual size of data buffers handled by the codec and DSP/BIOS Link is 2N. See Figure 18 for a diagram that includes these size issues.

GPP Measurement Techniques

The amount of RF6 application code on the GPP is very small. The bulk of the user time for the application is spent receiving, copying, and sending the audio data. No actual processing (that is, algorithms) is performed in the generic RF6 application.

For GPP performance measurement, we used the Linux vmstat utility, which provides a running report on virtual memory statistics. This includes information about processes, memory, swap space, I/O, system, and CPU usage.

Another useful command is “top”, which provides similar information. The data is reported on a per-process basis.

Vmstat is easier to use to obtain these benchmarks, because it can run in the background. This allows us to interact with the RF6 adaptation using the standard interface. You could avoid this problem when running top by using telnet to connect to the target platform's IP address (via DHCP). However, the additional telnet session would consume CPU resources on the GPP, thus defeating the purpose.

The advantage of using standard Linux utilities to measure the performance is that it is unobtrusive—no changes are required in the application to get GPP-side performance data. The disadvantage is that your measurements may vary depending on which MontaVista Linux edition you are using and the kernel options you selected in your configuration.

The following command runs vmstat:

```
[>] vmstat 5 &
```

This yields output like the following sample:

```
   procs                      memory    swap          io     system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy  id
 1  0  0      0  18772      0   6356   0   0     0     0  633   521   2   5  93
```

The key fields for this exercise are those that are circled:

- us. 2% percent of CPU time is consumed by user processes.
- sy. 5% percent of CPU time is consumed by system processes.
- id. 93% percent of CPU time is spent idle.

To kill the vmstat process, type ps on the command line, read the process ID for vmstat, and kill vmstat with the kill utility.

GPP-side measurements were made under the following conditions:

- Platform: OSK running at 192 MHz
- Kernel: MontaVista Linux Professional v3.1
- Sampling rate: 22.05 kHz and 44.1 kHz
- Samples per frame: 32, 64, 128 and 256 stereo 16-bit samples. This equates to 128, 256, 512, and 1024 8-bit bytes on the GPP side.
- Version of DSP/BIOS Link: Release (both dsplinkk.o and dsplink.lib)
- Version of RF6 GPP application: Release (no optimization or –g options)
- "vmstat 10" run as the measurement averaging period (10 seconds), with each dataset un for at least 40 seconds.

DSP Results

Table 22 shows the DSP CPU load results. The Framework Only column includes the CPU load for data movement, DSP/BIOS, drivers, application code, and more. The only things that were removed were the actual FIR and VOL algorithms.

Table 22. RF6 DSP CPU Load Percentages at Various Frame Sizes and Sample Rates

kHz	Samples/Frame	Frames/Sec	Framework + Algorithms %CPU Load	Framework only* %CPU Load
22.05	256	86	6	4
22.05	128	172	7	5
22.05	64	345	9	7
22.05	32	689	13	11
44.1	256	172	14	10
44.1	128	345	18	12
44.1	64	689	22	18
44.1	32	1378	32	27

Figure 34 shows a graphical representation of the data in Table 22.

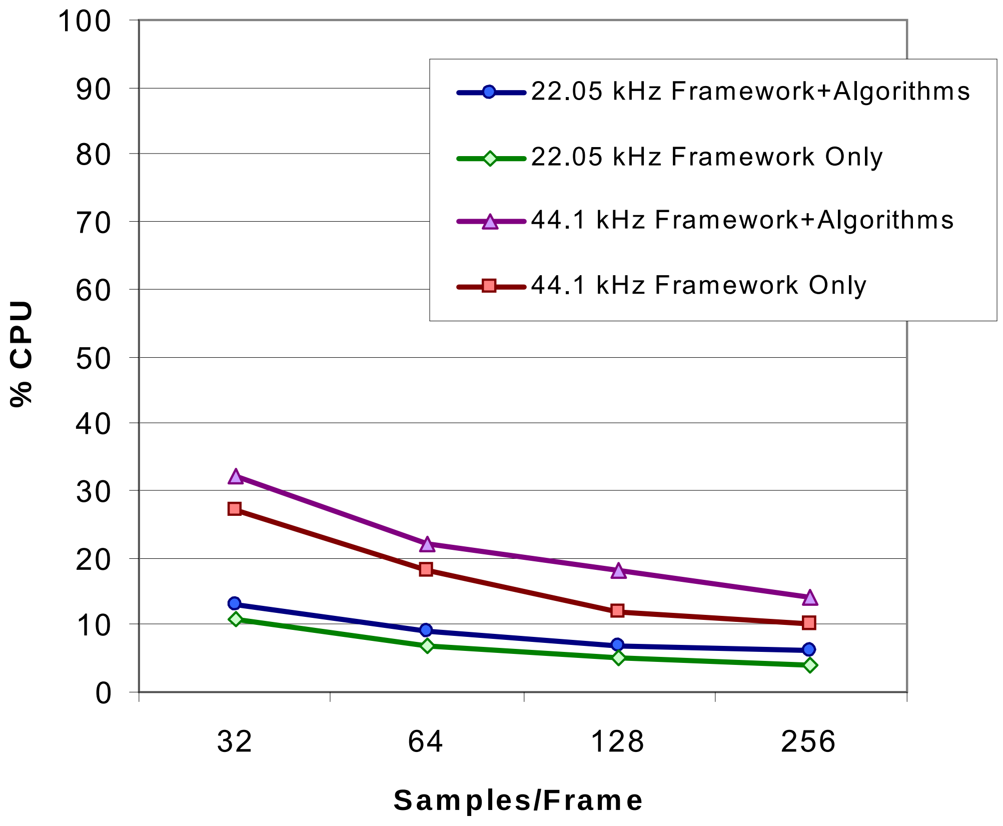


Figure 34. Graph of DSP CPU Load Percentage for Various Settings

GPP Results

Table 23 shows the GPP CPU load results.

Table 23. RF6 GPP CPU Load Percentages at Various Frame Sizes and Sample Rates

kHz	Samples/Frame	Frames/Sec	%CPU User Time	%CPU System Time
22.05	256	86	1	3
22.05	128	172	1	5
22.05	64	345	1	10
22.05	32	689	1	15
44.1	256	172	2	5
44.1	128	345	2	10
44.1	64	689	3	15
44.1	32	1378	4	30

Figure 35 shows a graphical representation of the above data.

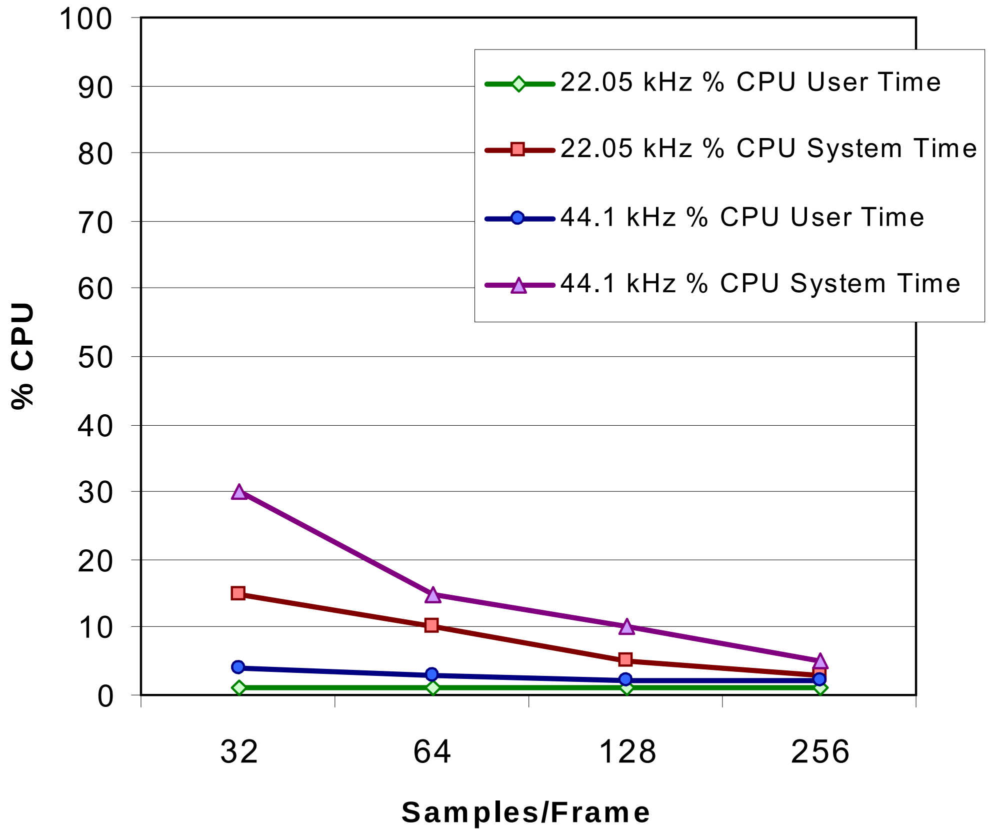


Figure 35. Graph of GPP CPU Load Percentage for Various Settings

Performance Conclusions

While the test results cited here are for the generic RF6 audio application on the OSK, several generalizations can be made about RF6:

- At higher frequencies (for example, ≥44.1 kHz), attempt to minimize the number of SWIs and TSKs on the DSP to avoid context switches and inter-thread communication.
- At higher frequencies (for example, ≥44.1 kHz on the OSK), moving some amount of data processing (for example, splitting and joining) to DMA is advised.
- Better performance is obtained with larger samples/frames. The trade-off is latency.

Caveats

This measurement technique and data has the following known caveats:

- If other intensive ARM-side CPU processes run on your system, it may be best to run “top” from a separate telnet session, instead of vmstat which reports only the total CPU load.
- vmstat and top are Linux/UNIX commands. Other GPP operating systems may not provide these utilities.

Appendix C: Reference Framework 6 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 24. Boards Supported by Various Reference Frameworks

Board	Codec	RF1	RF3	RF5	RF6
C5402 DSK	TLC320AD50 mono codec	✓	✓		
C5416 DSK	PCM3002 stereo codec	✓	✓		
C5510 DSK	TLV320AIC23 stereo codec	✓	✓	✓	
C6x11 DSK	TLC320AD535 mono codec		✓		
C6416 TEB	PCM3002 stereo codec		✓	✓	
C6416 DSK	TLV320AIC23 stereo codec		✓	✓	
C6713 DSK	TLV320AIC23 stereo codec		✓	✓	
Innovator 1510/5910	TLV320AIC23 stereo codec				✓
OSK5912	TLV320AIC23 stereo codec				✓

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).