TEXAS INSTRUMENTS

Application Report

SPRA793D – April 2003

Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System

Davor Magdic *Texas Instruments, Santa Barbara*
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 3 (RF3) is intended to enable designers to create applications that use multiple channels and algorithms while minimizing the memory requirements by using static configuration only. The key design goal for RF3 is ease-of-use. RF3 is architected to enable new DSP designers to create compact products with several algorithms and channels. RF3 provides more flexibility than lower Reference Framework levels. For example, it instantiates any XDAIS algorithm simply and efficiently. However, RF3 is not intended for use in large-scale DSP systems with 10s of algorithms and channels. Instead, RF3 targets medium-complexity systems and makes design decisions such as using static configuration to keep the memory footprint small.

This application note first provides an overview of Reference Frameworks. It then explains how to install, run, explore, and adapt Reference Framework Level 3.

Contents

Figures

Tables

1 Overview of Reference Frameworks for eXpressDSP Software

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

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

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

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

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

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

1.1 Reference Framework Target Levels

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

Yet, the classic, constrained, embedded DSP application is still out there—reminding us that DSP development will always value real-time performance, power, minimized code size, and cost optimization. To that end, several frameworks are required to meet such diverse needs. For example, the memory management scheme for a small, static system such as a digital hearing aid need not be as full-featured as a farm of DSPs in a telecommunications media server.

Several Reference Frameworks for eXpressDSP Software (RFs) will be produced. These frameworks will range in complexity from RF1, which is aimed at designers trying to produce extremely compact, consumer systems, to levels 5-10 with multiple algorithms, many channels, and different execution rates. The key for developers will be to pick the Reference Framework that best approximates a system and its future needs.

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

Table 1. Reference Framework Characteristics by Level

Design Parameter	RF1	RF3	RF5
Absolute minimum memory footprint	✓	✗	✗
Static configuration	✓	✓	✓
Dynamic object creation (e.g., DSP/BIOS objects)	✗	✗	✓ Supported, but static configuration preferred for simplicity and footprint
Static memory management	✓	✓	✓
Dynamic memory allocation	✗	✓	✓
Number of channels recommended	1-3+	1-10+	1-100+
Number of eXpressDSP-compliant algorithms recommended	1-3+	1-10+	1-100+
Uses DSP/BIOS real-time analysis	✓ In stop-mode only. RTDX not configured by default.	✓	✓
Uses DSP/BIOS scheduling kernel	✗ HWI and IDL only	✓ no TSKs	✓
Uses Chip Support Library	✓	✓	✓
Uses XDAIS algorithms	✓	✓	✓
Portable to other devices, ISAs, boards	✓	✓	✓
Supports multiple execution rates and priorities	✗	✓ Single rate per channel	✓
Supports thread blocking	✗	✗	✓
Implements control functionality	✗	✓	✓
Implements DSP-GPP functionality	✗	✗	✗ Will be key feature of RF6

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 manager in RF3 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.

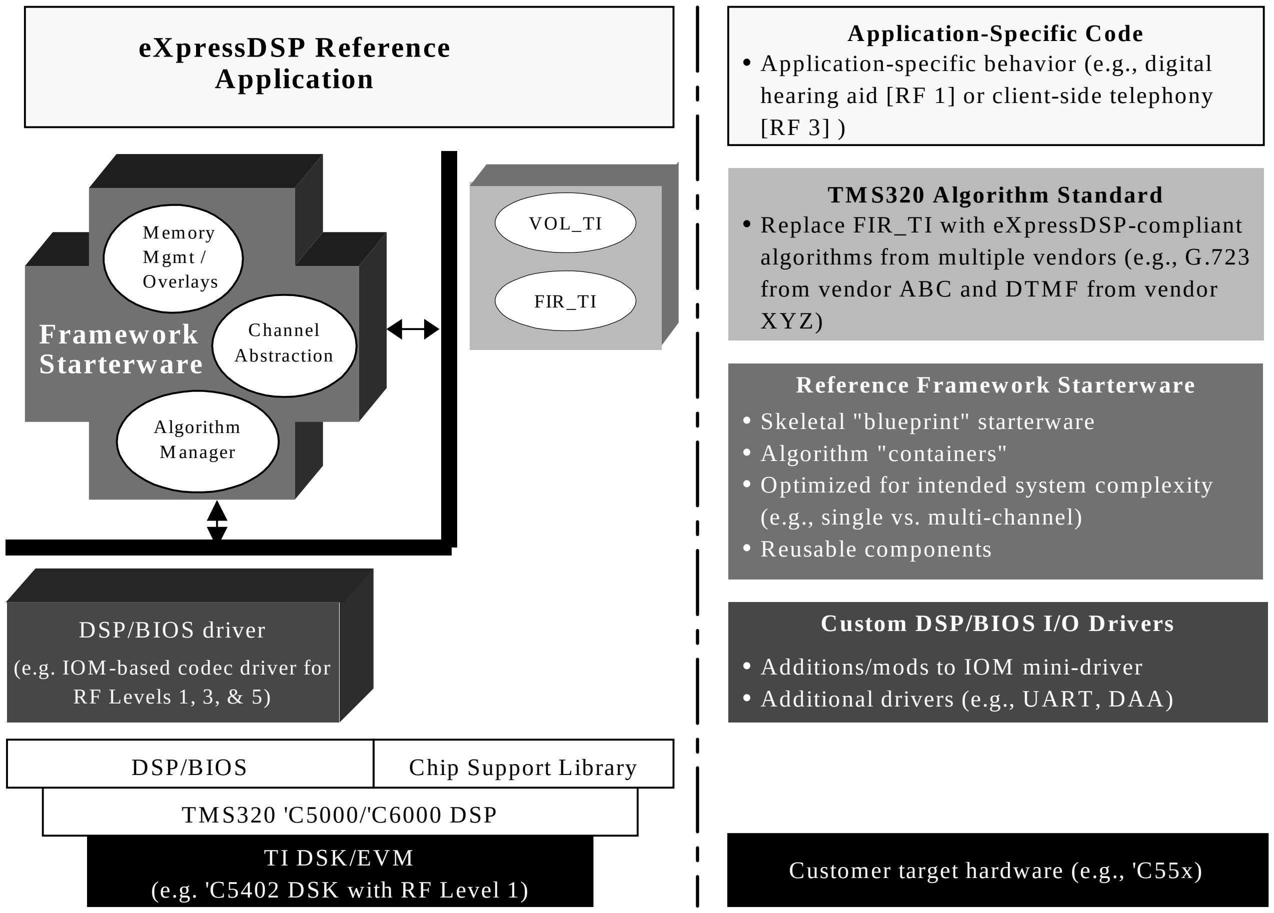

eXpressDSP Reference Application
Memory Mgmt / Overlays
Framework Starterware
Channel Abstraction
Algorithm Manager
VOL_TI
FIR_TI
DSP/BIOS driver
(e.g. IOM-based codec driver for RF Levels 1, 3, & 5)
DSP/BIOS
Chip Support Library
TMS320 'C5000/'C6000 DSP
TI DSK/EVM
(e.g. 'C5402 DSK with RF Level 1)
Application-Specific 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 board 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 dynamic, multi-channel, multi-algorithm frameworks of RF5 on a high-end 'C6711 or '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 very 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:

- **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 IOM model detailed in the *DSP/BIOS Driver Developer's Guide* (SPRU616). Standard conventions for hardware drivers makes integration and adaptation easier in much the same way that XDAIS simplifies algorithm integration.

1.4 Bill of Materials

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

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

- Production-worthy, reusable, eXpressDSP C language source code.
- A complete, "generic" application using the Reference Framework running on a Texas Instruments DSK.
- 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 add new algorithms, add more channels, and adapt the low-level DSP/BIOS driver(s).
- An API Reference Manual for new module libraries introduced in the Reference Frameworks.

2 Introduction to Reference Framework Level 3

Reference Framework Level 3 (RF3) is intended to enable designers to create compact products with several algorithms and channels. It is ideal for systems where memory footprint is an important concern, yet feature flexibility is required.

RF3 provides more flexibility than lower Reference Framework levels, which support absolute minimum memory footprints. However, RF3 does not use dynamic object creation or blocking threads, which require more memory resources. As a result, RF3 is not intended for use with applications that have complex interdependencies between threads or that dynamically create threads. RF3 retains the advantages of eXpressDSP Software Technology, including the ability to easily integrate eXpressDSP-compliant algorithms.

By default, RF3 converts an incoming audio signal to digital data at a given sampling rate. It splits this signal into two channels (by duplicating the signal) and processes both channels independently by applying a low-pass filter to the left channel and a high-pass filter to the right channel. It then applies a volume control to each channel. Finally, it sends the output to the output codec. The volume for each channel can be selected at run-time.

The default application behavior simply provides an example of the behavior of applications suited to using RF3. The application logic can easily be modified while leaving the data flow path in place.

2.1 Characteristics of RF3 Applications

Typical applications that are suited for use with RF3 require a relatively small memory footprint, but are not restricted to an absolute minimum memory footprint. Additional flexibility is required, either now or in the future, over that provided by RF1. However, the application objects and data path are static at run-time. That is, all processing threads are created when the program is initialized, and are never deleted.

The goal of RF3 is ease-of-use. A designer with applications expertise, but little exposure to eXpressDSP, should be able to use it to construct systems in less time than before.

RF3 includes foundation software, such as DSP/BIOS, the Chip Support Library (CSL), and XDAIS. It also calls on the services of various Reference Framework modules. For example, the UTL module is used for debugging and diagnostics. Developers can either build upon the current framework or port their applications to a higher-level framework with relative ease.

Table 2 shows the characteristics of RF3 in more detail than Table 1. It should be used as a guide to determine whether or not RF3 is suitable as the basis of your end-application.

Table 2. RF3 Application Characteristics

Design Parameter	RF3	Notes
Static configuration	✔	
Dynamic object creation (e.g., DSP/BIOS objects)	✖	
Static memory management	✔	
Dynamic memory allocation	✔	
1-10 channels	✔	

Design Parameter	RF3	Notes
11-100 channels	✖	There are no restrictions on the channel count. However, RF5 provides mechanisms intended for use in applications with high channel density.
1-10 eXpressDSP-compliant algorithms	✔	
11-100 eXpressDSP-compliant algorithms	✖	There are no restrictions on the number of algorithms. However, RF5 provides mechanisms intended for use in applications with 10s of algorithms.
Absolute minimum memory footprint	✖	
Uses DSP/BIOS real-time analysis	✔	
Uses DSP/BIOS scheduling kernel	✔	Uses only HWI and SWI threads. Does not use TSK threads, which are capable of blocking.
Uses Chip Support Library	✔	
Uses XDAIS algorithms	✔	
Portable to other devices, ISAs, boards	✔	
Supports multiple execution rates and priorities	✔	Supports a single rate per channel (data stream). Independent data streams can be processed at different single rates.
Supports thread blocking	✖	
Implements control functionality	✔	A separate thread or set of threads can modify processing parameters based on user actions.

If your application does not match the characteristics suited to RF3, you should use a different Reference Framework.

2.2 Key Features of RF3

RF3 provides the following key features:

- **Good compromise between overhead and flexibility.** RF3 provides lower memory and performance overhead than the extensive RF5 framework. At the same time, it provides more flexibility and scalability than the minimalist RF1 framework.
- **Maximizes ease-of-use.** A key goal for RF3 is to maximize ease-of-use by minimizing complexity. Tricks to minimize footprint such as those used in RF1 are not included in RF3. Similarly, the RF5 modules that support large numbers of algorithms and channels are not used in RF3.
- **SWI-based application.** In contrast, RF5 is TSK-based. The TSK module provides more scheduling flexibility, but carries with it more performance and memory overhead.
- **PIP-based I/O.** In contrast, RF5 is SIO-based. The PIP module is simpler and less flexible than SIO, but carries with it the advantage of reduced overhead.
- **Allows easy debugging.** The well-defined structure of RF3 allows for fast debugging. In addition, the UTL module allows debugging levels to be changed rapidly.
- **Easy replacement of I/O drivers.** The IOM model allows alternate mini-drivers to be connected to the PIO adapter used by RF3.
- **Wide variety of boards supported.** RF3 currently supports a wider variety of boards than other frameworks.

2.3 Applications Suited to RF3

All Reference Frameworks are application-agnostic. Each framework can be adapted for use in many applications, including telecommunication, audio, video, and more. The following figures provide examples of applications to which RF3 is suited:

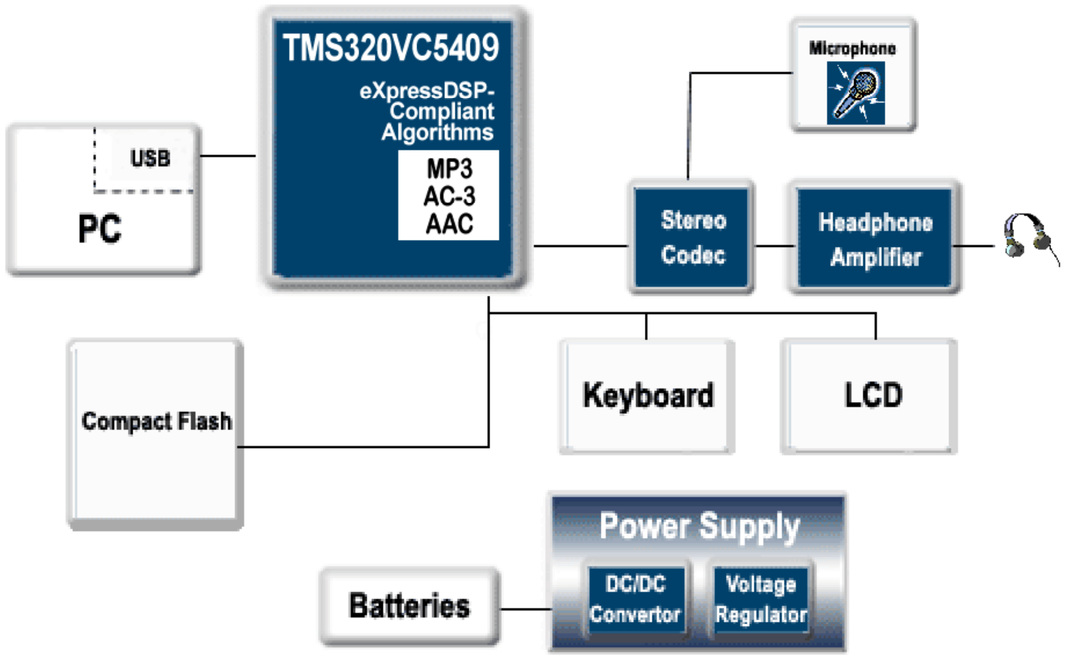


Figure 2. Internet Audio Player

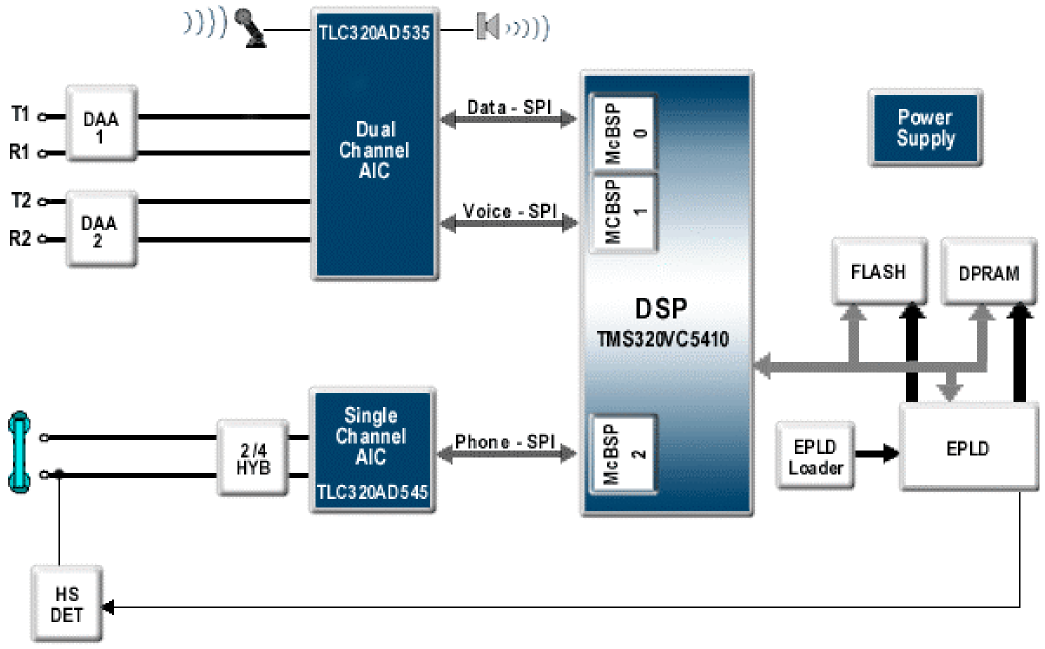


Figure 3. Web Phone

Other applications for which RF3 can be adapted include:

- Digital radios
- Client-side telephony solutions
- Digital scanners
- Hands-free voice kit
- Digital still camera
- Digital video camera
- Audio enhancement

3 Installing and Running the RF3 Application

This section describes how to build and run Reference Framework Level 3 (RF3) as it is. Later sections describe how the application works and how it can be adapted for your own application.

3.1 Preparing the Hardware

The board-specific portions of the RF3 application have been ported to several boards. The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93. Additional boards may be added in the future. If you want to run RF3 on a different target, you need to port hardware-dependent parts of the application.

The following steps provide an overview of how to connect the hardware to your host PC. For details and diagrams, see the documentation provided with your board. For additional board-specific information, see the DSP/BIOS Driver Developer's Kit documentation.

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

3.2 Preparing the Software

The following list outlines the software installation and setup steps required to run RF3. For details, see the appropriate Quick Start Guides, online help, or the readme.txt file.

1. If you have not already done so, install CCStudio 2.2 or a later version. It is recommended that you have the latest version of the CCStudio software, as it may contain important features or problem fixes.

2. Check the configuration of your parallel printer (LPT) port. Make sure the parallel port is in ECP or EPP mode and note the first address of the port. This is normally 0x378. For details on checking the parallel port configuration, see the Quick Start Guide provided with your board.

3. Use the Setup Code Composer Studio application to configure the software for your board. For details, see the documentation provided with your board.

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

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

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

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

3.3 Building and Running the RF3 Application

After installing the package, you are ready to build and run the RF3 application.

1. Within CCStudio, choose Project→Open and select the app.pjt project in *RF_DIR*\apps\rf3\target, where *target* matches your board. (For example, *RF_DIR*\apps\rf3\dsk5402.)

   The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93.

2. Choose Project→Build to build the RF3 application.

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

3. Choose File→Load Program and load the app.out file in the Debug subfolder.
4. Start your CD player or other audio input.
5. Choose Debug→Run (or F5). You should hear the FIR filtered audio output through the speakers connected to the target board.
6. Choose File→Load GEL and select the app.gel file from the project folder (above the Debug folder).

   The app.gel file is a GEL script file displays some GUI controls. When you move these controls, the script writes certain values to the target's memory. These values control which of the two channels is active, and the volume level for each channel.
7. Choose GEL→Application Control→Set Active Channel. A slider with two positions appears. The positions select the output channel. The down position selects channel 0, which uses a low-pass filter. The up position selects channel 1. Channel 1 uses a high-pass filter, which makes the music sound somewhat better.
8. Choose GEL→Application Control→Set_channel_0_gain. A slider appears with values for channel 0's volume ranging from 0 to 200. The default value is 100. Channel 0 must be the active channel for the slider to produce audible changes.
9. Similarly, to control the volume of channel 1, choose GEL→Application Control→Set_channel_1_gain, and use the slider to control channel 1's volume.

4 Exploring the RF3 Application

This section provides an introduction to the organization of the RF3 application.

4.1 Folder Structure Overview

The Reference Framework folder tree contains application sources and library modules. You can begin to explore RF3 by examining the folder tree that contains the application and associated files. We recommend that you retain the provided structure for your development.

Figure 4 shows the folders used by RF3 and highlights some important files they contain.

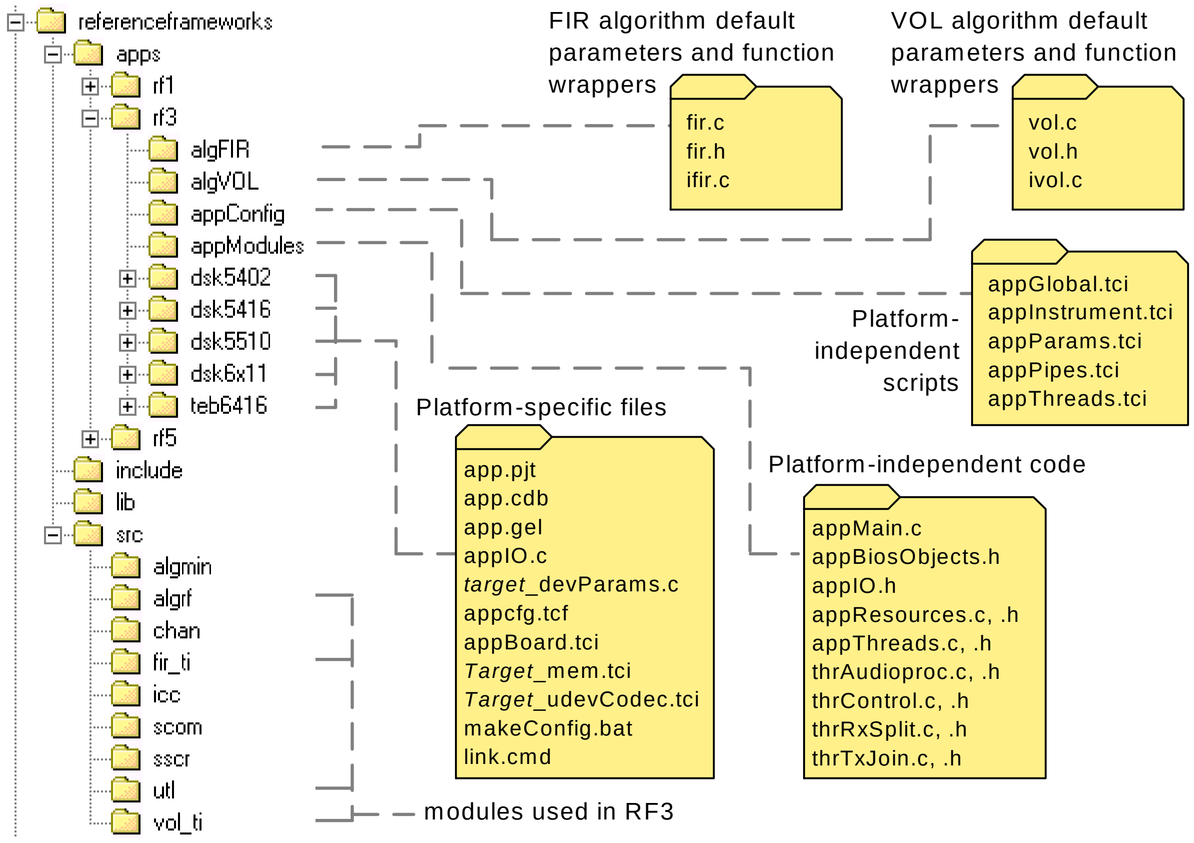


Figure 4. RF3 Folder Structure

Folders to notice include:

- **apps\rf3.** Contains the CCStudio projects that make up RF3. To modify RF3, make a copy of the rf3\ tree at the same folder level, and modify the copy.
  - **appConfig.** Contains the hardware-independent script files for the RF3 configuration.
  - **appModules.** Contains the hardware-independent files for the RF3 application. This includes the implementation of threads (thr*.* files) and the initialization code (app*.*).
  - ***target*.** Contains hardware-specific files for the RF3 application. This includes the configuration files, the project file, and the linker file. These files are placed in platform-

named folders so that RF3 can be provided for multiple platforms. The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93.

  - **algFIR.** Contains application-side wrappers for communicating with the FIR algorithm. Your algorithms can be wrapped similarly to match the RF3 structure. For example, you might create an algMP3 folder.
  - **algVOL.** Contains application-side wrappers for communicating with the VOL algorithm.

- **include.** Contains a number of public header files used by Reference Frameworks. RF3 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. RF3 uses some, but not all, of these libraries. Each library module has one library per DSP chip in this folder.

- **src.** Contains folders with source files for the modules in the include and lib folders. The readme.txt files in each of these folders provide information about the modules and their use. Library modules typically need little or no modification. The modules used by RF3 are:
  - **algrf.** Contains source files for the IALG implementation used by RF3 to instantiate eXpressDSP-compliant algorithms using the DSP/BIOS MEM module for dynamic memory allocation.
  - **fir_ti.** Contains files for the Finite Impulse Response (FIR) filter, an eXpressDSP-compliant algorithm provided by Texas Instruments and used by default in RF3.
  - **vol_ti.** Contains files for the Volume algorithm, an eXpressDSP-compliant algorithm provided by Texas Instruments and used by default in RF3.
  - **utl.** Contains source files for the UTL debugging and diagnostics module.

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

4.2 Application Modules

RF3 uses several collections of modules. Figure 5 shows the high-level framework architecture.

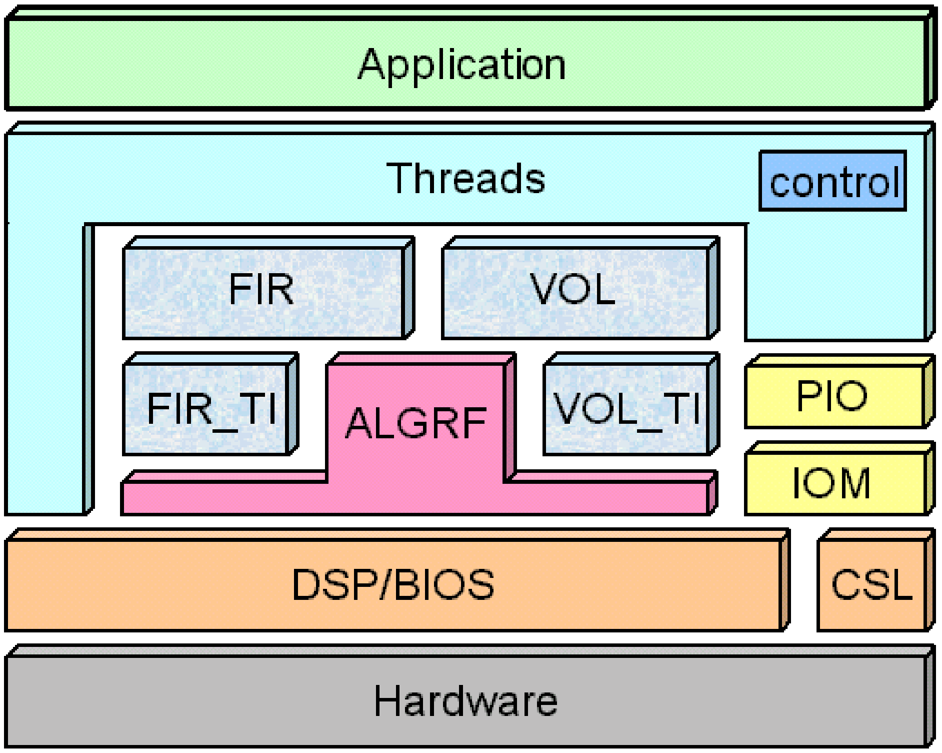


Figure 5. Topology of Modules in RF3

Table 3 describes the components of this architecture diagram.

Table 3. Architecture Components

Component	Description	Adaptation Required
Application	Reference Frameworks can also be adapted for use in many applications, including telecommunication, audio, video, and more. Such changes are not detailed in this application note, but are typically made through changes to the main source file.	Some 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. The four threads in RF3 are two channel data processing threads (named "Audioproc"), and the RxSplit and TxJoin threads that split/merge channel data. Threads are generally hardware-independent.	Yes, to add or remove channels
Control	This thread controls settings for other threads. It may have to read I/O registers and apply their values to data processing. It is possible to insulate the control thread from the hardware by having the ISR that posts the control thread read hardware values and store them in the control thread's data structures.	If desired
FIR	Framework interface to the FIR algorithm. A caller-friendly wrapper to the FIR algorithm.	Minimal, to switch algorithms
FIR_TI	TI's eXpressDSP-compliant implementation of the Finite Impulse Response (FIR) algorithm.	No

Component	Description	Adaptation Required
VOL	A caller-friendly wrapper to the VOL algorithm.	Minimal, to switch algorithms
VOL_TI	TI's eXpressDSP-compliant implementation of a volume (VOL) control algorithm.	No
ALGRF	IALG interface implementation. Implements functions such as ALGRF_create() for creating instances of XDAIS algorithms. The functions are similar to the functions in the CCStudio's ALG module, but more efficient and controllable. Along with ALGRF, the UTL module is provided to support debugging and diagnostics. Described in SPRA147.	No, but source code is provided
PIO	Pipe adapter for the IOM model. The upper-layer of the device driver. Allows application to read and write blocks of data from/to the codec using only pipes. Executes DSP/BIOS PIP calls to manage buffer transfers. This layer is hardware-agnostic—if a different codec or board is used, this layer need not know. Described in SPRU616.	No
IOM	I/O Mini-Driver Interface. Simple, low-level device driver interface between application threads and hardware devices. Defines a set of device-agnostic APIs to interface with. Described in SPRU616.	Yes, if you need to port mini-driver to your hardware
DSP/BIOS	The set of modules provided as a scalable real-time kernel. This includes modules for scheduling, instrumentation, I/O, and memory management.	No
CSL	Chip Support Library. Simplifies the job of developing device drivers and interacting with peripherals.	No
Hardware	The target board, including codecs and other peripherals.	--

The architecture can also be organized according to the amount of adaptation typically required to adapt RF3 to an application. Figure 6 groups architecture components in this manner.

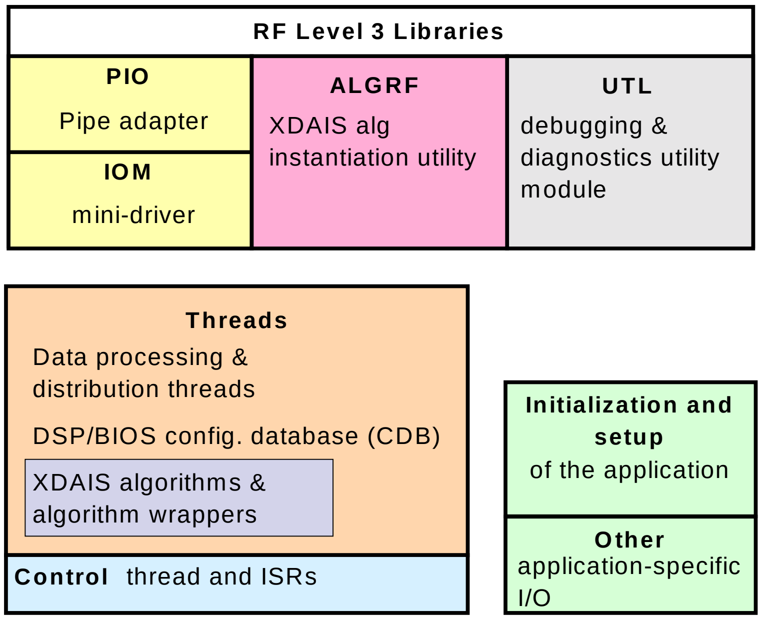

RF Level 3 Libraries
PIO
Pipe adapter
IOM
mini-driver
ALGRF
XDAIS alg instantiation utility
UTL
debugging & diagnostics utility module
Threads
Data processing & distribution threads
DSP/BIOS config. database (CDB)
XDAIS algorithms & algorithm wrappers
Control thread and ISRs
Initialization and setup of the application
Other application-specific I/O

Figure 6. RF3 Application Modules

- **Libraries.** These components typically need little or no modification (unless you need to port an IOM mini-driver to a different device).
- **Threads and algorithms.** These components can be modified by following the recipes provided in Section 8, *Adapting the RF3 Application*, page 62.
- **Initialization and setup.** This refers mainly to the application's main() function, which calls initialization functions for other modules. It may also invoke application-specific initialization procedures.
- **Other.** This refers to application-specific I/O functions, such as controlling a device's LCD display or LEDs and reading from and writing to flash memory.

5 Overview of the RF3 Foundation: DSP/BIOS and XDAIS

In this section we briefly examine two components on which the RF3 application is built:

- **DSP/BIOS**. A real-time, low-footprint, highly modularized operating system for TI 'C5000 and 'C6000 DSP processors. It manages hardware and software resources.
- **TMS320 DSP Algorithm Standard** (also known as XDAIS). Provides uniform, standardized ways of executing complex, off-the-shelf DSP algorithms.

Experienced DSP/BIOS and XDAIS users can safely skim or skip this section and move on to Section 6, *RF3 Design Approach*, page 28.

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). What follows is only a short introduction to both, from the viewpoint of RF3. An acquaintance with DSP/BIOS and XDAIS is needed for writing DSP applications at this level.

5.1 DSP/BIOS Overview

RF3 calls DSP/BIOS service routines. DSP/BIOS has a number of modules for different application needs. However, to reduce the memory footprint, RF3 includes only the modules that are needed by applications suited to using RF3. The most important application characteristic to consider when deciding which modules to exclude is the static structure of RF3 applications. In addition, the application retains several DSP/BIOS mechanisms for debugging, tracing, and real-time analysis.

5.1.1 DSP/BIOS Configuration

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

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

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

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

- **Easily port frameworks to new target boards and platforms.** The configuration scripts clearly separate application-specific settings from target-specific settings. This makes it easy to port applications to new target boards and platforms. Only those settings made for target-specific reasons require modification when porting to a new board.
- **Eliminate potential update issues.** The configuration database (CDB) file used with the graphical configuration tool must be updated with every new version of CCStudio. In some cases, this conversion can be problematic. Textual configuration uses scripts as source files, and eliminates this conversion altogether. These scripts are much smaller and easier to maintain than their CDB counterparts.

Configuration scripts are written in JavaScript, a powerful scripting language that has a C-like syntax, which helps to reduce the learning curve for a typical C programmer.

Reference Frameworks provide textual configuration by including a .tcf file that can be run to create the configuration database (.cdb). This .tcf file imports both application-specific scripts (for example, appInstrument.tci) and target-specific scripts (for example, appBoard.tci). These scripts contain the JavaScript code to create and configure static objects. Typically, a target-specific script first loads a particular platform and then imports sub-scripts to set up memory sections, drivers, and other target-specific objects.

More information about the configuration scripts provided for RF3 is provided in Section 8.2.1, Textual Configuration Scripts, page 64. Complete documentation on textual configuration can be found in the *DSP/BIOS TextConf User's Guide* (SPRU007). For information on specific properties of DSP/BIOS objects, please refer to the DSP/BIOS API manual for your DSP family.

5.1.2 DSP/BIOS Components

DSP/BIOS provides a number of optional modules. For example, an RF3 application uses the clock manager (CLK), but not the mailbox manager (MBX). If a module is defined in the configuration to be used by the application, or if you call a module's functions directly from your code, only then will the module be included in the final executable.

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

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

5.1.3 DSP/BIOS Program Execution Flow

DSP/BIOS takes over control when the application is loaded to the target, which may have been accomplished via, for example, a ROM loader, or a PC. Its first phase is the *initialization* phase, in which various hardware settings are established and DSP/BIOS objects are initialized. The second phase consists of calling the user-provided main() function, which performs application-specific initialization. After returning from main(), DSP/BIOS performs its *startup* phase, in which DSP/BIOS objects are enabled or started.

The course of execution from that point on depends on the programming model. In general, and in the RF3 application, DSP/BIOS falls into an *idle loop*, waiting for a hardware event to occur. For example, a hardware event may be a clock interrupt or a DMA transfer completion interrupt.

A basic concept in any real-time operating system is the threading model. In DSP/BIOS terminology, a *hardware interrupt* or *software interrupt thread* runs non-blocking code that executes from its entry point until its end, without (voluntarily) relinquishing CPU control. In contrast to a *task thread*, a hardware or software interrupt thread cannot wait for a semaphore or a message, or block in any other way. When a hardware or software interrupt thread starts executing, the data it will process must be ready—in fact, the thread most likely started executing because its data was ready. Note that such threads are created at system startup, and are never destroyed. Each thread is associated with a C procedure to be called again the next time the thread runs.

Hardware threads are referred to as HWIs; software threads can be either software interrupt routines (SWI), or periodic threads (PRD). A hardware thread runs, usually very quickly, as a result of an interrupt. A software thread runs when explicitly *posted*, either by a hardware thread, or by another software thread (or by itself). Posting a thread essentially means performing a DSP/BIOS operation that results in changing the state of the thread to *ready to run*. Periodic threads are scheduled to run automatically at regular intervals.

DSP/BIOS task threads (TSK) support more complex thread interactions because of their ability to block while waiting for a resource or allowing another thread to run. However, task threads require more memory resources than software interrupt threads. RF3 does not use task threads and is not intended for use with applications that have complex interdependencies between threads or that dynamically create threads. The remainder of this document, when referring to a thread, implies a hardware or software interrupt thread.

HWI and SWI threads are very efficient in terms of memory requirements, as they all use the same stack, and also in terms of context switching, for much the same reason. This efficiency comes at the price of having to design a system so that the availability of the data automatically results in triggering the code that has to process that data. However, this design fits the nature of RF3: Data is processed in streams, and those streams are fixed both in terms of their number and their sources. Typically, we use a thread for a semantically isolated processing unit—for example, if data arrives independently through two different codecs, we use one thread for each stream.

Threads have *priorities*. Higher-priority threads preempt lower-priority ones. Hardware interrupt threads have the highest priority. Software interrupt threads have lower priorities than hardware interrupt threads. The idle loop (*IDL*), which is technically not a thread, has the lowest priority of all. The idle loop runs if there are no hardware or software interrupt threads ready to run. In addition, if real-time analysis (RTA) is turned on, the idle loop communicates with the host and sends it analysis data.

In a common scenario, an HWI runs when a DMA transfer is complete, thus it runs only if another block of the data stream is ready to be processed. The HWI initiates transfer of the next input block and activates a SWI. The SWI, when started, processes the first input block to produce the output block, which results in initiating the transfer of the output block, and the cycle repeats for the next block of data. The remaining time is consumed by the idle loop, during which RTA data is sent to the host.

Figure 7 shows the flow for a system with three hardware threads, four software threads, and the idle loop.

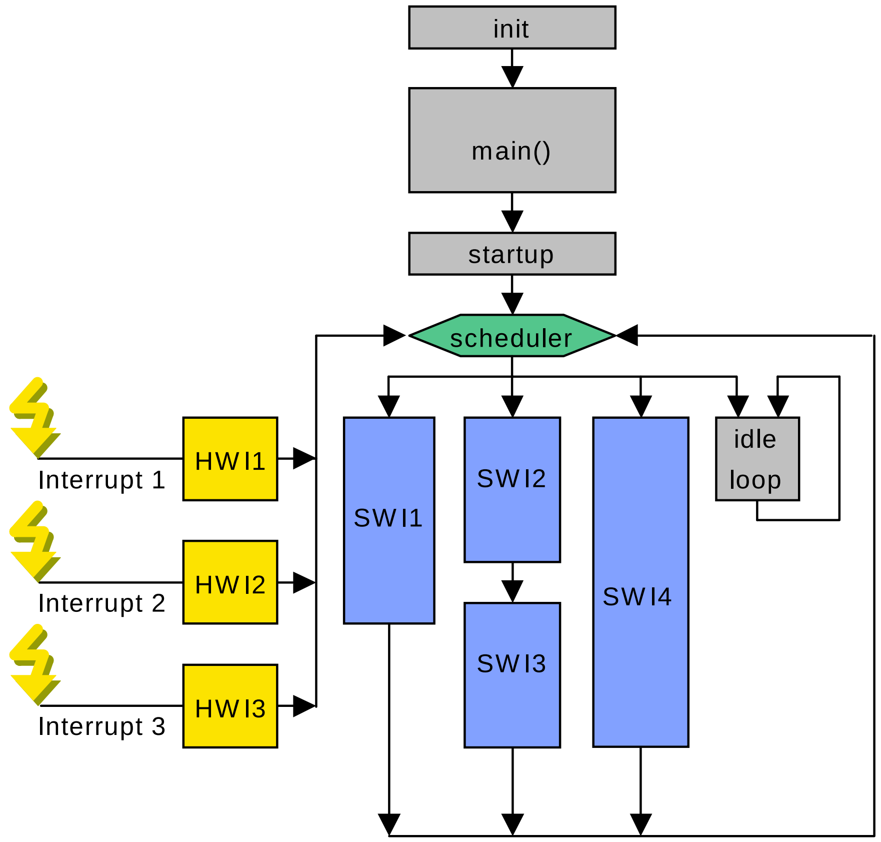


Figure 7. Execution Flow Example Diagram in a DSP/BIOS Application

In Figure 7, HWI1, HWI2, and HWI3 are hardware threads, and SWI1, SWI2, SWI3, and SWI4 are software interrupt threads. The priority of threads decreases from left to right. (SWI2 and SWI3 have the same priority.) A thread can be interrupted by a thread to its left, if that thread becomes ready to run and no higher priority threads are ready. Each software interrupt thread can be preempted by any hardware interrupt thread. A software interrupt thread can be preempted by a higher-priority software interrupt thread. This may occur if the lower-priority software thread itself posts a higher-priority thread.

How does a thread become ready to execute? If the thread is a hardware interrupt thread, the occurrence of its interrupt makes it ready. If the thread is a software interrupt thread, it has to be posted. Usually this happens when some other thread determines that the data for the thread in question is ready.

A software interrupt thread can be posted in several ways. One way is through direct posting—the user calls SWI_post() to make it ready to run. Another common way to post a SWI is to assign a *mailbox value* to the thread, and then have different parts of the program clear different bits of the mailbox value. When the mailbox becomes zero, the thread is posted and the mailbox is reset to its initial value. Typically, the initial mailbox value has as many non-zero bits as there are resources that the thread needs. When a resource becomes ready, the appropriate bit is cleared, until the mailbox becomes zero and the thread runs. This mechanism has the advantage that the thread's readiness is not dependent on the order of resource availability, nor on repeated availability of a single resource.

The idle loop, IDL, is always ready, but it runs only if the CPU has nothing else to do.

5.1.4 Data Communication with Pipes in DSP/BIOS

Several means of data communication are provided by DSP/BIOS. Of these, pipes are the most convenient choice for static applications such as RF3. The PIP module implements pipes.

A pipe is a fixed-size block data storage device with synchronization capabilities. It has a single reader and a single writer. A pipe can hold one or more frames of the same size. The number of frames and their size is statically defined in the configuration. The writer can inquire how many *free* frames are available; the reader can inquire how many *full* frames are available. Typically pipes have two frames: While the reader reads one frame, the writer writes to another and then they switch. This is commonly referred to as a *ping-pong* buffer.

If there is a free frame in the pipe, the writer can allocate it, learn its address and size, write data to it, and optionally set the new size. The writer then puts the frame to the pipe, which becomes available to the reader.

If there is full frame in the pipe, the reader can get it, learn its address and size, read data from it, and free it. The option to use a size other than the default is useful for variable-size data frames.

When a writer allocates and puts a frame in a pipe, or a reader gets and frees a pipe frame, the frame data itself does not physically move. These actions merely change the state of internal buffer descriptors, and activate some implicit synchronization mechanisms.

The synchronization aspect of pipes lies in their *notify* functions. In the configuration, one *notifyReader* function and one *notifyWriter* function can be specified for each pipe. A notifyReader or notifyWriter function can be any function with up to two arguments. A pipe's notifyReader function is called when a new full frame becomes available for reading; notifyWriter function is called when a new free frame becomes available for writing.

In a system where SWI threads communicate using pipes, the pipes' notify functions commonly determine when a SWI thread is posted. The most common case is where a SWI thread (S) reads from pipe A and writes to pipe B, as in Figure 8.

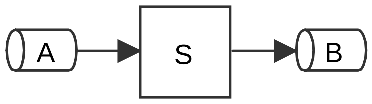


Figure 8. SWI Thread with Pipes

Thread S should be posted only when both pipe A has at least one frame for reading and pipe B has one frame for writing, since the thread cannot wait on resources. So thread S has a mailbox of 3 (binary 11), and the notify functions could act as follows:

```
pipeA: notifyReader = SWI_andnHook( S, 1 );  (clears bit 1 of the mailbox)
       notifyWriter = ...
pipeB: notifyWriter = SWI_andnHook( S, 2 );  (clears bit 2 (10 binary) of mailbox)
       notifyReader = ...
```

This posts thread S when both pipes become available, regardless of their availability order.

To illustrate access to pipe data, the following code segment copies the contents of a full frame from pipe A to a free frame in pipe B, if both frames are available.

```
/* first check if there is at least one frame for reading and one for writing */
if (PIP_getReaderNumFrames( &pipeA ) > 0 && PIP_getWriterNumFrames( &pipeB ) > 0) {
    Int *src, *dst, size;         /* size units for pipe frames are words */

    /* get the full frame for reading from pipe A */
    PIP_get( &pipeA );

    /* get the frame's address and size */
    src  = PIP_getReaderAddr( &pipeA );
    size = PIP_getReaderSize( &pipeA );

    /* allocate a free frame for writing in pipe B */
    PIP_alloc( &pipeB );

    /* get its address */
    dst  = PIP_getWriterAddr( &pipeB );

    /* copy the content of frame in A to frame in B */
    memcpy( dst, src, size );

    /* set the size of the output frame to be equal to that of the input frame */
    PIP_setWriterSize( &pipeB, size );

    /* free the input frame and put the output frame */
    PIP_free( &pipeA );
    PIP_put( &pipeB );
}
```

This code would not differ much for any other type of pipe frame processing. Instead of the memcpy() call there could be an XDAIS algorithm call or a custom procedure call. (We don't need the if() test for full and free frame availability if we make sure the thread is posted only if both are available.)

5.1.5 Real-time Analysis Tools in DSP/BIOS

The two DSP/BIOS real-time analysis modules used most in an RF3 application are the LOG module and the STS module.

The LOG module is often used with the LOG_printf() function, which has the following syntax:

```
LOG_printf( LOG_object, "format" [, arg1 [, arg2] ] )
```

Unlike the traditional printf(), LOG_printf simply stores three integers on the target: the address of the format string in the executable, the value of arg1, and the value of arg2. All formatting is performed on the host PC when it retrieves LOG data from the target.

Each LOG buffer has a fixed size, determined at configuration time. When the application runs a LOG function, a new triplet is added to the buffer. Whenever idle, the target sends LOG buffer data to the host, using RTDX (Real-Time Data eXchange). As long as the target uses less than 100% of the CPU time, the host can receive fresh target data. If the LOG buffer fills faster than it is read, some data is lost. Increasing the LOG buffer size can solve this problem.

The STS, or the Statistics module, implements STS objects, which store statistics information such as the minimum and maximum execution time and the number of executions of a procedure or accesses to a resource. Statistics objects execute similarly to LOG objects—the information is sent by the target when the idle loop runs and formatted on the host.

5.2 XDAIS Algorithm Standard Overview

This section briefly introduces TMS320 DSP Algorithm Standard concepts. Complete technical details can be found in *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) and the *TMS320 DSP Algorithm Standard API Reference* (SPRU360). See Section 11, *References*, page 82 for a list of related documentation and application notes.

A fundamental requirement of the Algorithm Standard is that all algorithms implement the IALG interface to define their memory requirements, enabling efficient use of on-chip data memories in client applications. The uniform memory management scheme enables multiple algorithms to co-exist without contention in a single application.

The IALG memory interface defines various types and constants with the key element being a global structure of type IALG_Fxns. It contains a set of function pointers, which is commonly called the v-table. Some of the functions are optional, while algAlloc(), algInit() and algFree() must always be implemented.

```
typedef struct IALG_Fxns {
  Void *implementationId;
  Void (*algActivate)(IALG_Handle);
  Int  (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
  Int  (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
  Void (*algDeactivate)(IALG_Handle);
  Int  (*algFree)(IALG_Handle, IALG_MemRec *);
  Int  (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
  Void (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
  Int  (*algNumAlloc)(Void);
} IALG_Fxns;
```

The algAlloc() function returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm.

Based on the information retrieved from the memTab[ ] descriptor structure, the application allocates the requested memory before calling the algInit() initialization function.

It is the application’s responsibility to initialize a pointer, usually of type IALG_Handle, to point to the v-table structure when creating an instance of the algorithm. This provides access to each of the algorithm methods through the function table, without necessarily exposing the vendor’s specific function names. The algorithm enables this operation by specifying an instance object containing all of the code’s state or context information. Its first field is always of type IALG_Obj, which, in turn, makes the IALG functions accessible to the client. The reserved field memTab[0] allows the application to point to the instance object thus implying reentrancy since all read/write algorithm data memory is encapsulated in a per-channel structure.

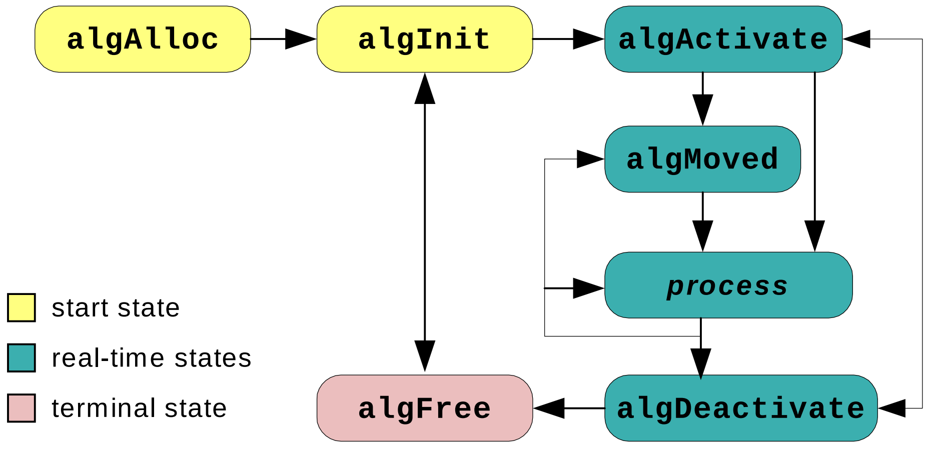


Figure 9. IALG Interface Function Call Order

algInit() performs all initialization necessary to complete the run-time creation of an algorithm’s instance object. After a successful return from algInit(), the object is ready to process data.

The algActivate() and algDeactivate() methods give the algorithm a chance to initialize and save scratch memory outside the main processing loop. Partitioning memory into two groups, scratch and persistent, allows data placement flexibility and footprint optimization. An algorithm can freely use scratch memory without regard to its prior contents. Persistent memory is any area of memory that can be safely written to, knowing that the contents will be unchanged between successive invocations by the application. This distinction enables the following optimization.

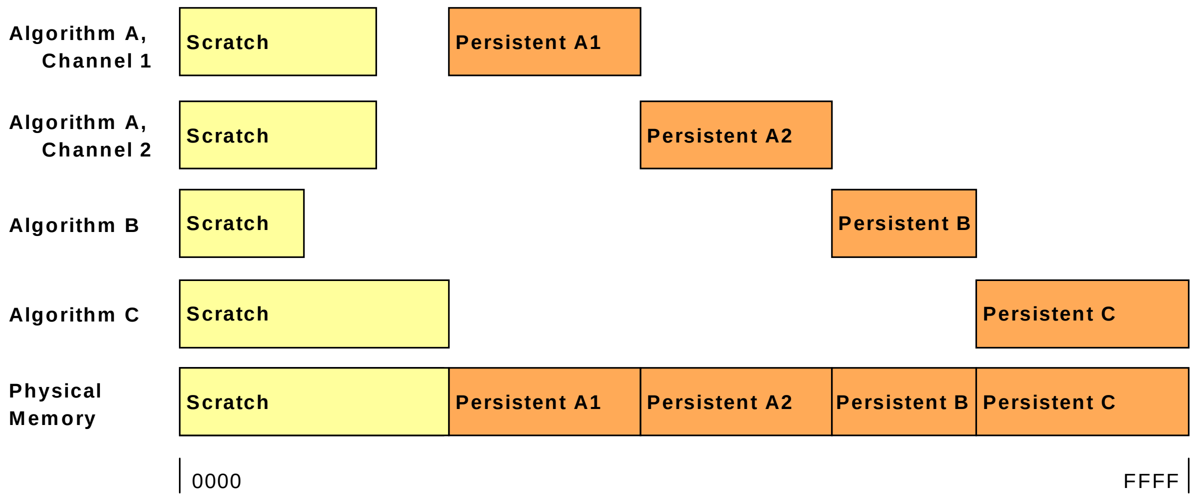


Figure 10. Scratch vs. Persistent Memory Allocation

Algorithm scratch memory can be “overlaid” on the same physical memory. In controlled circumstances, several algorithms and/or channels can reuse the same block.

While it is true that a stack offers much the same rewards, the absence of separate scratch memory would require a very large stack. In small, compact systems this facet may prohibit the stack from being placed on-chip, severely impacting performance.

algFree() is the last of the required functions. It is the algorithm’s responsibility to make the client aware of the current base addresses and size of each memory block previously requested in algAlloc(), so that the application can delete the instance object and all its buffers, without creating memory leaks.

5.2.1 *Finite Impulse Response (FIR) Algorithm*

This Reference Framework executes a FIR filter. It is packaged as a fully compliant XDAIS algorithm. It executes the mathematical equation shown in Figure 11.

$$y(k) = \sum_{n=0}^{N} w(n)x(k-n) \quad k = 0,1,\ldots$$

Figure 11. FIR Filter Equation

The advantages of using this algorithm are:

- Relatively simple to understand
- Highlights many of the XDAIS concepts discussed in the previous section.
- It is not intellectual property. It can therefore be freely distributed.

Its only disadvantage is that it may be too simple. The FIR filter may not make sense for codecs with higher sampling rates. However, since the intent of the Reference Frameworks is that the default filters will be replaced with useful algorithms, there is no need to use different dummy algorithms for difference codecs.

The function pointer for algAlloc() maps to FIR_TI_alloc(). It requests three data buffers, two of which must hold persistent state and one for a scratch, working buffer. The sizes for the persistent filter history and the work buffer both depend on the supplied input parameters:

```
/* Request memory for persistent filter history buffer */
memTab[HISTORY].size      = (params->filterLen - 1) * sizeof(Short);
memTab[HISTORY].alignment = 2;
memTab[HISTORY].space     = IALG_EXTERNAL;
memTab[HISTORY].attrs     = IALG_PERSIST;

/* Request memory for shared working buffer */
memTab[WORKBUF].size      = (params->filterLen - 1 + params->frameLen) * sizeof(Short);
memTab[WORKBUF].alignment = 2;
memTab[WORKBUF].space     = IALG_DARAM0;
memTab[WORKBUF].attrs     = IALG_SCRATCH;
```

FIR_TI_initObj() then assigns pointers within the persistent instance object to point to the memory newly allocated by the framework. FIR_TI_activate() then copies a small amount of filter history to the beginning of the larger scratch buffer, priming it for execution of the main processing method, FIR_TI_filter(). Its job is to produce filtered output as a function of the present input data and the static filter coefficients that dictate the nature of the filtering (for example, low pass or high pass).

Updates to the filter history data are saved back to the small, external, persistent buffer in FIR_TI_deactivate(). This preserves the last results since the scratch buffer may be reused by another algorithm or FIR channel thereafter.

All global identifiers comply with Rule 8 of the Algorithm Standard. A prefix of FIR_TI_ satisfies the requirement that all external definitions be either API identifiers (for example, HWI_disable) or MODULE and VENDOR prefixed. Uniqueness of the vendor tag, _TI_ in this case, avoids namespace collisions if the algorithm is used in multi-vendor, multi-algorithm applications.

6 RF3 Design Approach

Here, we walk through the process of designing the RF3 application.

6.1 Application Requirements for the Base Application

In this section, we more accurately describe the behavior of the base RF3 application. The basic goal of Reference Frameworks is to provide a template-like real application that can be morphed into most RF3 applications without substantial changes. For that reason, some of the requirements specified—even if a bit unnatural considering the platform—are there to make the base application as close to an "average RF3 application" as possible.

The application specifications for the base RF3 application are as follows:

The application takes a incoming audio signal and converts it to digital data at a given sampling rate. The application splits the samples to process the channels independently. Each block of audio signal is represented by two signed 16-bit samples, one for the left and one for the right channel. (For a stereo codec, the signal is split. For a mono codec, two channels are emulated by duplicating each mono sample to the left and right channels.)

The application processes each channel by applying a filter to the data. The left (first) channel data is applied to a low-pass filter, and the right (second) channel data is applied to a high-pass filter. A volume control is then applied independently to each channel, both channels having possibly different amplification/attenuation factors. Finally, the two channels are merged into one, before being sent to the output. (For mono codecs, one channel is used and the other is discarded. This selection is user-controllable.) The resulting data is sent to the output codec.

The number of channels is a modifiable constant, which allows the application to be easily extended to more or fewer channels should the need arise.

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

In addition, the application has a control function, so that the user can, from the host, modify the volume for each channel and select which channel should actually be sent to the codec. Users can change the parameters via GEL, CCStudio's scripting language for accessing target memory, with simulated knobs and switches whose values are written to a designated area in memory that the target periodically reads.

Based on this specification, we can make the following crude block-diagram of the system:

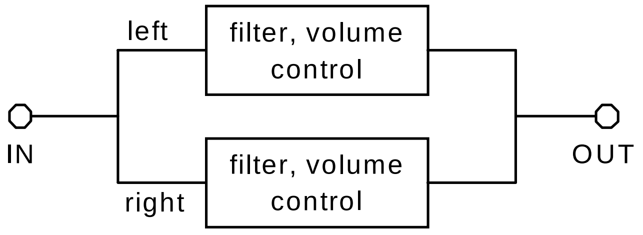


Figure 12. General Block Diagram of the Base Application

The algorithms we use process contiguous blocks of single-channel data. In a stereo system, there are two channels in the incoming data stream. The input codec delivers left and right samples interspersed. A single mono codec can be used to simulate an N-codec system or a single codec with N channels—for example, time-division multiplexed (TDM) data.

6.2 Base Application Data Path

Our first step is to turn the system block-diagram from the previous section into a data path that can be made of DSP/BIOS components.

In our system, a *data path* is a flow graph of streaming audio data through the system, from the input codec to the output codec. In that system, as is the case with most RF3 systems, the incoming synchronous data is, after being sampled, grouped into *blocks* or *frames* and processed as such. In that light, our data path can be presented as shown in Figure 13.

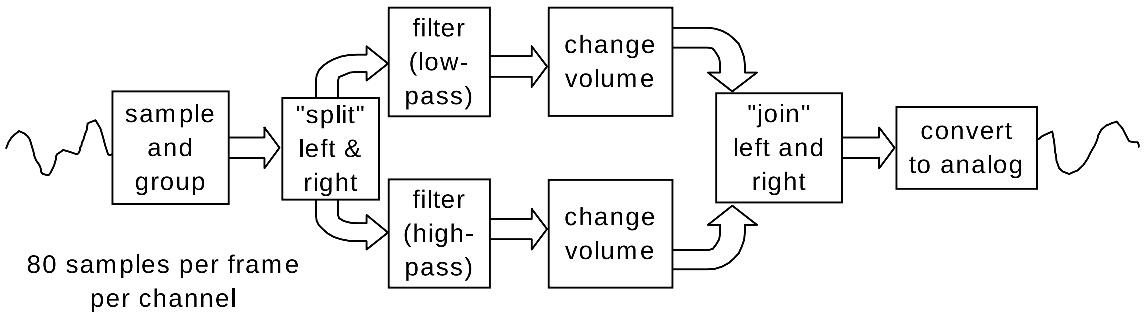


Figure 13. Data Path with Data Blocks

Boxes in the data path indicate processing functions—an XDAIS algorithm, or a custom function, or any combination of those. In the picture, filter and change volume are XDAIS algorithms, whereas "split" and "join" are not. A box is akin to a function that takes zero or more input buffers and values and produces zero or more output buffers and values.

An RF3 application could have more than one stream. Also, some processing boxes might produce not a stream of data, but a single value. For instance, touch-tone key detection (DTMF) takes a block of telephone audio samples and detects any touch-tone sound in it.

6.3 Determining Rates and Sizes of Data in the Data Path

The data stream is processed "on the fly" and output at the same rate as it is sampled. Sampling rates differ according to the board on which the RF3 application is running. RF3 works fine running at a variety of rates; the only difference is that if the data is processed more frequently, the CPU load is higher.

Arrows in the data path shown in Figure 13 indicate the path of blocks of 80 16-bit samples of data. We somewhat arbitrarily decided to use blocks—or frames—of size 80. In an actual system, this decision would be based on such things as algorithm requirements, system requirements, and available CPU time.

The default RF3 application has the same frame length on all boards, because system builders creating the same system on multiple boards want processing consistency. However, the frame length can easily be changed. This is because XDAIS algorithms typically have fixed requirements for frame lengths. For example, G.723 algorithms have a frame length of 240 samples, and G.729 algorithms have a frame length of 80 samples.

To change the frame length to match the requirements for your algorithms, modify the PIP object framesize properties in the configuration and change the line that sets the FRAMELEN constant in appResources.h.

```
#define FRAMELEN    80
```

When we use DMA, the CPU does little work to receive and transmit the frames. Therefore, the larger the frame size, the less time the CPU spends on I/O, and the more time remains for processing. Larger buffer sizes, however, result in larger latencies, and consume more memory, so a compromise must be found.

You can see how often data buffers are processed at run-time by looking at the values shown for stsTime0 in the DSP/BIOS Statistics View with the units set to milliseconds. This object measures the time between two executions of the "split" function. The value should be approximately equal to 1000/sampling rate * FRAMELEN. Naturally, the timing is influenced by other CPU activity such as ISR processing and periodic threads. Driver writers may use this calculation as a quick check to confirm the sample frequency.

The following subsections show rate and size details for some specific targets.

6.3.1 'C5402 DSK Rates and Sizes

Figure 14 shows the data path for the 'C5402 DSK, which has a mono codec (AD50) with a sampling frequency of 8 kHz. Each sample is 16 bits and PIP framesizes are measured in 16-bit units. The calculation for how often data is processed is as follows:

1 second / frequency (Hz) * FRAMELEN =
1/8000 * 80 =
0.01 s = 10 ms

For a mono codec, the input frame is simply duplicated to the left and right instead of being split. The "join" function discards output from either the high-pass or low-pass filter as selected by the user via the control thread and the GEL Set Active Channel slider.

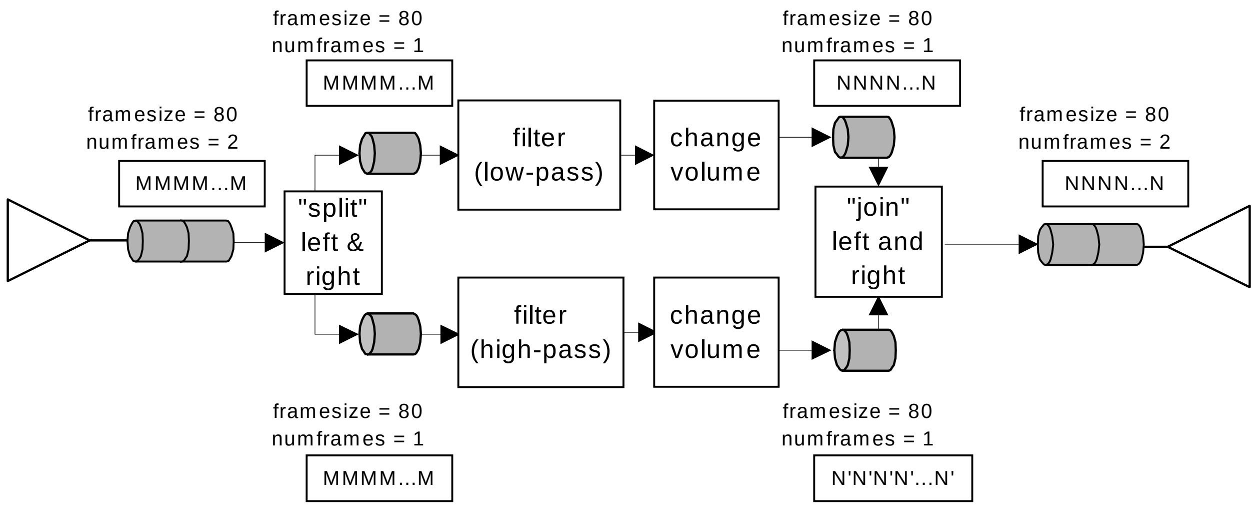


Figure 14. 'C5402 DSK Buffers Sizes and Rates

6.3.2 'C5510 DSK Rates and Sizes

Figure 15 shows the data path for the 'C5510 DSK, which has a stereo codec (TLV320AIC23) with a default sampling frequency of 44.1 kHz, which is the principal sampling rate for MP3 algorithms. Each sample is 16 bits and PIP framesizes are measured in 16-bit units. The calculation for how often data is processed is as follows:

1 second / frequency (Hz) * FRAMELEN =
1/44100 * 80 =
0.00181 s = 1.81 ms

For stereo codecs, the first and the last transfers are twice as large—their size is 160 16-bit samples, 80 for each channel. The Set Active Channel GEL slider has no effect.

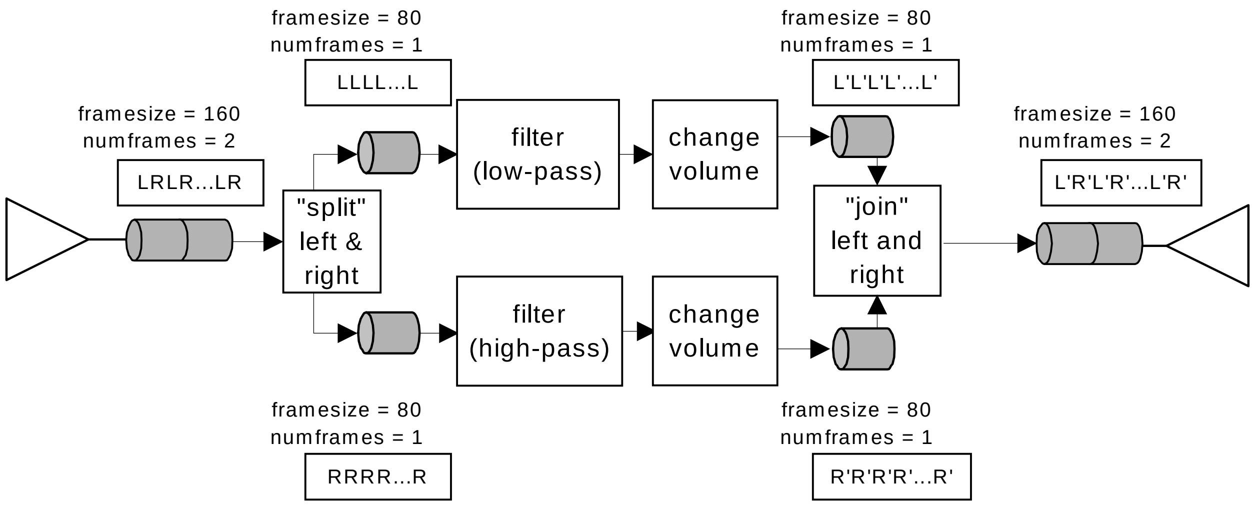


Figure 15. 'C5510 DSK Buffers Sizes and Rates

6.3.3 'C6x11 DSK Rates and Sizes

Figure 16 shows the data path for the 'C6x11 DSK, which has a mono codec (AD535) with a sampling frequency of 8 kHz. Each sample is 16 bits and PIP framesizes are measured in 32-bit units. The calculation for how often data is processed is as follows:

1 second / frequency (Hz) * FRAMELEN =
1/8000 * 80 =
0.01 s = 10 ms

For a mono codec, the input frame is simply duplicated to the left and right instead of being split. The "join" function discards output from either the high-pass or low-pass filter as selected by the user via the control thread and the GEL Set Active Channel slider.

On the 'C6000, the PIP object framesizes are 40, but the application FRAMELEN is still 80. This is because on the 'C6000, PIP object framesizes are measured in 32-bit units. Therefore, a PIP object that contains 40 32-bit locations is the same size as a PIP object that contains 80 16-bit locations.

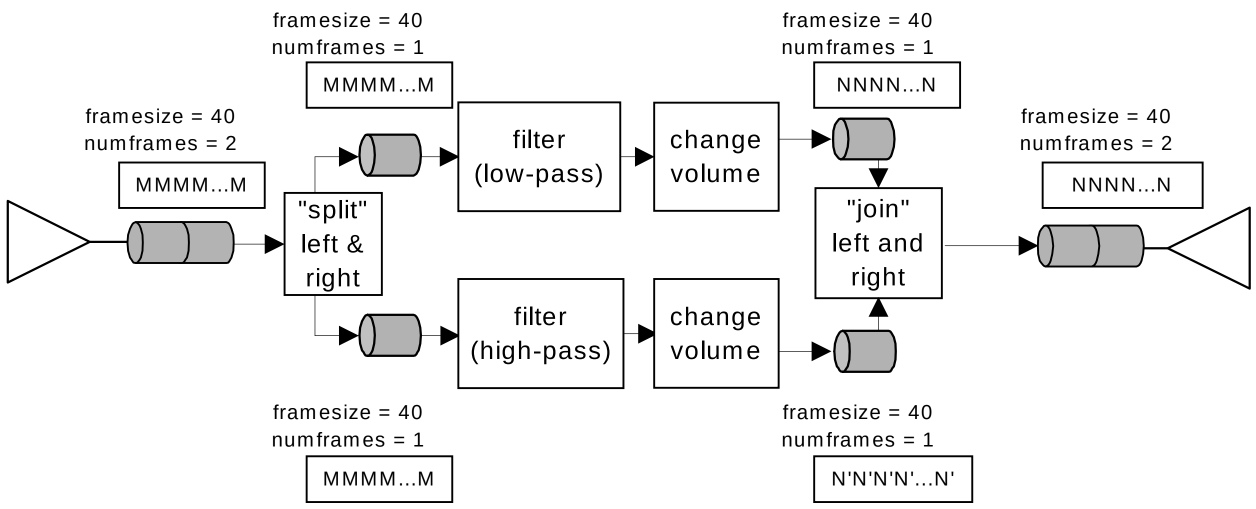


Figure 16. 'C6x11 DSK Buffers Sizes and Rates

Input and output buffers are aligned on a 32-word boundary (128 bytes) to avoid potential cache coherency issues if these buffers are placed in external memory. However, it is best to leave these pipes in internal memory to avoid this issue altogether.

6.3.4 'C6416 TEB Rates and Sizes

Figure 17 shows the data path for the 'C6416 TEB, which has a stereo codec (PCM3002) with a sampling frequency of (up to) 48 kHz. Each sample is 16 bits and PIP framesizes are measured in 32-bit units. The calculation for how often data is processed is as follows:

1 second / frequency (Hz) * FRAMELEN =
1/48000 * 80 =
0.01 s = 1.66 ms

For stereo codecs, the first and the last transfers are twice as large—their size is 80 32-bit samples, 40 for each channel. The Set Active Channel GEL slider has no effect.

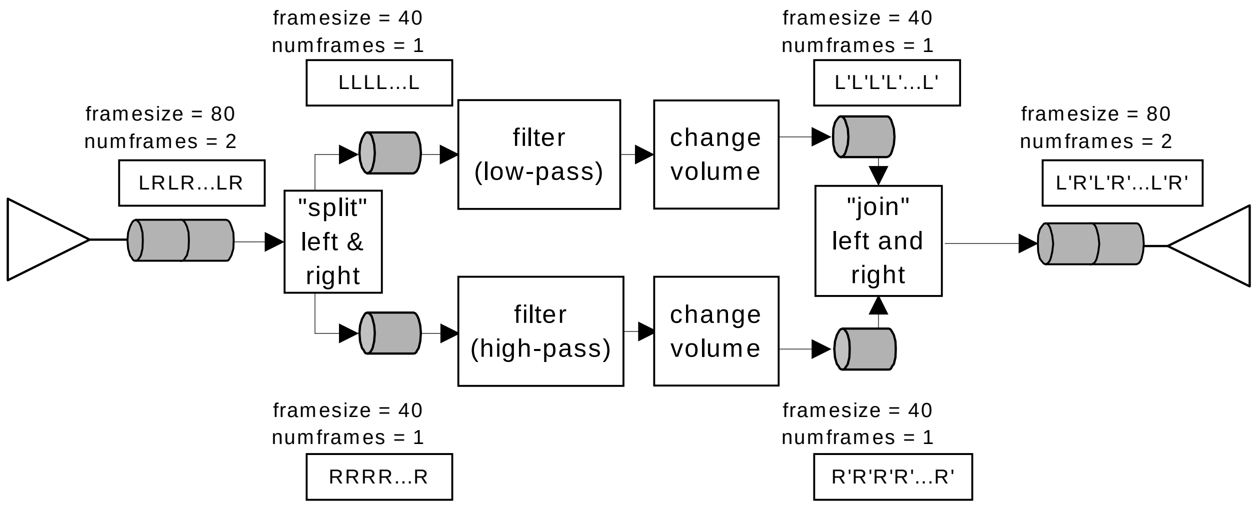


Figure 17. 'C6416 TEB Buffers Sizes and Rates

On the 'C6000, the PIP object framesizes are 40, but the application FRAMELEN is still 80. This is because on the 'C6000, PIP framesizes are measured in 32-bit units. Therefore, a PIP object containing 40 32-bit locations is the same size as a PIP object that contains 80 16-bit locations.

6.3.5 Rate and Size Modifications in Other Applications

In a general case, buffer rates or sizes may be different. For example, with a stereo codec, the first and the last buffer would be 160 16-bit samples large (or 80 32-bit samples large, depending on the codec) while the rest would be 80 samples large. All the buffers are still processed once every 10 ms. In a data encoding application, uncompressed data may arrive, for instance, every 10 ms, but be accumulated in a buffer ten times larger (800 samples), for compression. In that case, the large buffer is accessed (for reading) at a ten times lower rate (once every 100 ms).

Data may also be arriving asynchronously and leaving synchronously, as in the case of terminal characters from the host sent by the modem to the telephone line. In those cases, instead of a fixed-size buffer, you may need a different data structure, such as a circular buffer, for holding the data.

6.4 Identifying Processing Threads on the Data Path

After determining buffer sizes and their rates, we can approach the most important design issue—grouping functions into threads. A thread consists of one or more functional blocks in the data path. Note that our system is *static*—the structure of the data path does not change in time.

Here are the intuitive rules for doing such a split:

- If two functional units operate in parallel, they must be placed in different threads.
- If two functional units operate at different rates, they must be placed in different threads.

This is consistent with a general notion that a thread represents code that may run *in parallel* with other threads. This parallelism is not real, of course, when using a single-core DSP. But this loose understanding of parallelism is adequate for our design purposes.

We may go one or more steps further and split the data path into more threads than absolutely necessary, if we want to. In fact, the upper limit for the number of threads is the number of functional units, but placing each such unit (box) into a separate thread may be unnecessary. In the figures that follow, dashed boxes indicate threads, and all functional units within a dashed box belong to a single thread.

In our case, the "sample and group" box, which coordinates signal sampling and grouping by the DMA, and its output counterpart, "convert to analog," which controls the DMA to send samples from the output frame to the codec, are implemented as hardware threads. All the remaining blocks in the system operate at the same rate, so they could all be grouped into one thread as shown in Figure 18.

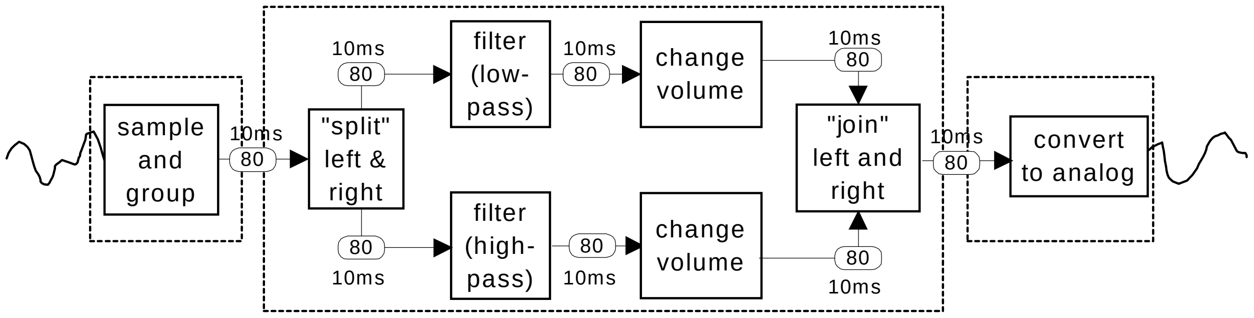


Figure 18. Data Path with Three Threads

Such grouping is valid since, in a system with a stereo codec, neither channel can begin processing without having the incoming frame with both left and right samples, and the outgoing frame cannot be formed before both channels finish processing.

However, for the base application, we decided to split left and right channels into separate threads to illustrate techniques for using multiple channels. Multiple channels are useful in applications with different multiplexed contents or multiple sources. For applications that do not need multiple channels, collapsing all functions into a single thread is easy, and we do that as an exercise in Section 8.4, *Removing the Second Channel*, page 68.

The decision to use two threads for the two channels gives us the following design:

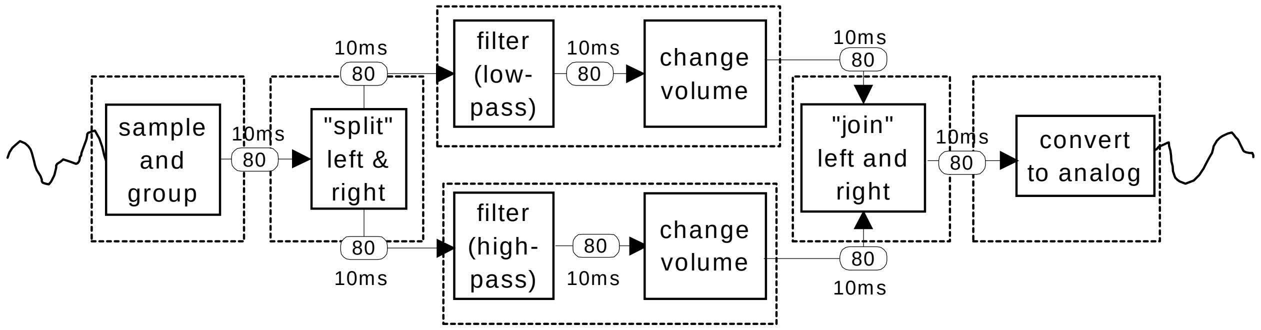


Figure 19. Data Path with Six Threads

6.5 Implementing the Data Path with DSP/BIOS Modules

After grouping functional units into threads, we introduce DSP/BIOS modules into the design. In Figure 19 we identify the following distinct elements:

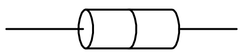
- Input/output threads (threads in charge of transferring data to and from physical devices)
- Data processing threads (which do not communicate with hardware devices)
- Data buffers between threads
- Data buffers within threads

Next, we find an appropriate module for each of these elements.

6.5.1 Data Buffers Between Threads: The PIP Module

While we could use simple static arrays for inter-thread data communication, we would have to implement a synchronization mechanism as well. A better solution is to use a DSP/BIOS module that provides both—block data storage and implicit synchronization: a DSP/BIOS pipe (PIP).

We set our pipes to have two frames, implementing the double buffering, or "ping-pong," scheme, between those threads that really must be distinct threads—in our case between the input thread ("sample and group") and the data splitting thread, and between the data joining thread and the output thread ("convert to analog"). Elsewhere, a single frame is sufficient.



For a pipe, we use the following symbol: On the left side of the pipe is its writer, and on the right side is the reader. The pipe in the picture is double-buffered (two frames of fixed size). A single-buffered pipe has only one cylinder in the symbol.

6.5.2 Data Buffers Within Threads: Simple Static Arrays

As expected, the buffer used within a thread is a simple array. The processing function that calls the filtering algorithm stores results in that array, which is then used by the volume algorithm.

If your system is memory-constrained and two or more threads need such a buffer but can never preempt one another, you can make the buffer a global variable and *share* it amongst threads.

6.5.3 Input/Output Threads: The IOM Driver Model

The box that collects samples and groups them into blocks, as well as its output counterpart, uses the PIO adapter and an IOM mini-driver. These are described in the *DSP/BIOS Driver Developer's Guide* (SPRU616) manual.

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

- **Class driver.** The class driver typically provides serialization and synchronization of multi-threaded I/O requests. Class drivers are device-independent. They interface with the application objects used to perform I/O. In the case of RF3, the class driver interfaces with PIP objects.

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

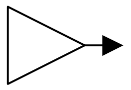
The class driver used by RF3 is a combination of the PIP module and the PIO adapter. See Section 7.1.2, *The PIO Module*, page 42 for more information about PIO.

Several mini-driver libraries are provided with Reference Frameworks. The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93. For information about porting mini-drivers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616) manual.

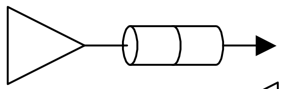
The IOM driver model transfers data to and from given blocks of memory. However, we use pipes as the main communication device between threads, so to simplify matters further we use a "PIP adapter to an IOM mini-driver," or PIO. This module uses a single pipe to transfer data from the codec to the client, if the module is configured for input, or to transfer data from the client to the codec, if the PIO object is configured for output. After initialization, the application need only get the data from the input pipe(s), and store the data to output pipe(s), without having to worry about hardware. This provides for uniform implementation of all data processing threads as SWIs surrounded by pipes wherever block transfer is involved.

The PIO module defines a "class," so there can be as many instances of PIO objects as necessary. In a system such as ours, where there is one codec, there is one input PIO object and one output PIO object.

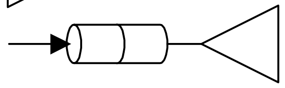
The symbol we use for an input IOM mini-driver coupled with a PIO object is a triangle followed by an arrow to indicate the direction of flow.



Coupled with a double-buffered pipe, the symbol becomes:



Similarly, an output PIO object with a double-buffered pipe is depicted as:



6.5.4 Full-Specified Data Path

After identifying all elements on the data path in terms of DSP/BIOS modules, we can draw the diagram shown in Figure 20. The picture shows that at any given time, the input part, PIO, transfers data from the input device to one frame of the input pipe. At the same time the previously transferred frame flows through the system, and still at the same time, the output part transfers the previously processed frame to the output device:

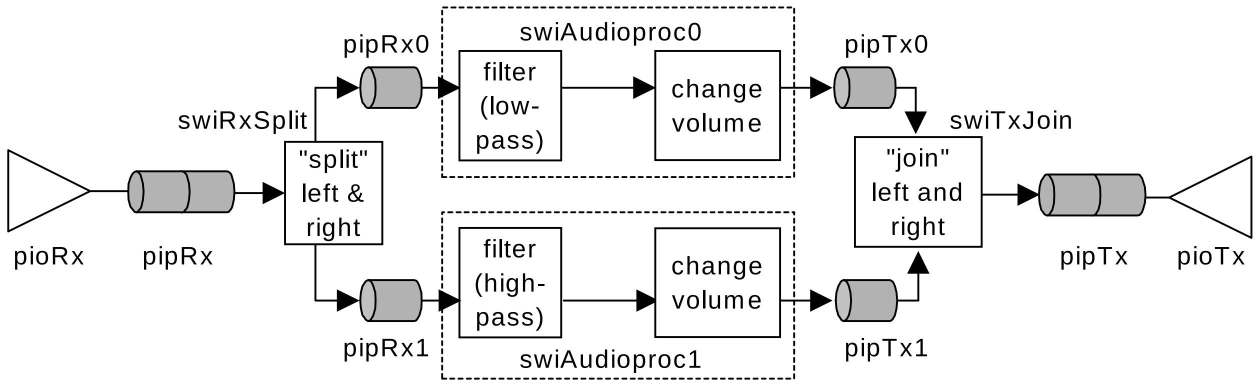


Figure 20. Data Path with DSP/BIOS Objects Identified

The left and right channels are labeled channel #0 and channel #1, respectively. An index is more convenient should there be more than two channels. The names of the objects are based on their type, their function, the side they belong to (Rx = receive, Tx = transmit), and the channel number, if applicable.

The following is a summary of the data flow in terms of DSP/BIOS objects in the data path:

- At initialization time, pioRx is activated. This function starts codec, McBSP, and the DMA, and begins transferring the first input block immediately. The data is placed in the first frame of the pipRx pipe.
- After 10 ms, exactly 80 samples have arrived and a DMA interrupt occurs. It calls a pioRx function that puts the pipRx frame with the data and allocates the other frame, already initiating transfer of the next block.

  As noted in Section 6.3, *Determining Rates and Sizes of Data in the Data Path*, page 30, the number of milliseconds is based on the board-dependent sampling rate. This summary uses the codec sampling rate of 8 kHz used on the 'C5402 DSK.
- As the first frame for reading is available, the swiRxSplit thread is posted. When run, it copies its input, the data frame in pipRx, to pipRx0 and pipRx1.
- When pipRx0 and pipRx1 are written to, both the swiAudioproc0 and swiAudioproc1 threads are posted when their other mailbox bits are cleared. When swiRxSplit finishes, swiAudioproc0 runs first while swiAudioproc1 waits.
- The swiAudioproc0 thread passes its input frame through the filter using the FIR algorithm, stores the result in the intermediate buffer, performs the volume changing function on it using the VOL algorithm, and stores the result in pipTx0 pipe.
- The swiAudioproc1 thread, running next, performs similar functions, using the pipRx1 and pipTx1 pipes.
- The swiTxJoin thread is posted and runs—none of the others are ready yet—and picks one of its input pipes to copy its contents to pipTx, discarding the other. When pipTx is written to (and put), pioTx is activated. It initiates transfer of data from the pipTx frame to the codec.
- The output frame is transferred after exactly 10 ms (80 samples at 8 kHz) and the other DMA interrupt is raised. It calls a pioTx function that checks to see if the next output frame is available (if it is not, we have missed the real-time deadline!) and, if so, initiates its transfer to the codec.
- Exactly 10 ms after the first input frame has arrived, the second one arrives and the cycle begins again.

As we can see, the heartbeat of the system comes from the codec/McBSP/DMA, and with our parameters it is set to 10 ms.

6.6 Adding the Control Thread

One remaining item we have to include to meet the specification of our application is to implement control—to provide the user with a means of changing the volume parameter for each channel and selecting which of the two channels is heard.

Let us assume that our RF3 application runs on a physical device that has two volume sliders and one channel selection switch. Changing the position of those represents a hardware event to which we want to respond.

We can assume that these hardware parts have values readable as registers in the I/O space. To learn the value of a slider or the switch, the application need only read those I/O registers at convenient times and apply them to the processing threads.

What is a "convenient time" for reading a value of an external knob or button? That depends on whether the hardware part can generate an interrupt to the DSP chip when the user changes its value. If it can, we can read the register for the hardware part in the ISR for the interrupt—that is, only when something happens.

If a change in the value/position of the hardware piece does not raise an interrupt, we have to resort to *polling*—periodic checking of the value. So we need a periodic thread that runs, for example, every 20 ms, reads the I/O registers, and updates processing parameters if necessary. Of course, the choice of the period depends on the desired reaction time we need and the overhead we can allow. The more frequently we poll the registers, the more cycles we spend and the faster we can respond.

Since most default boards do not have any knobs to poll, RF3 simulates them using CCStudio's GEL language. A simple GEL script displays two sliders and one switch. Whenever either one changes, the script temporarily halts the target and writes the changed value into an area in the target's memory. The location used is "deviceControlsIOArea[]" integer array. The application pretends this location is in the I/O area.

If we want to perform a periodic task, such as polling these "registers" every 20 ms in DSP/BIOS, we can use either a PRD object or a CLK object. A PRD object is a software thread like a SWI, and is set to call a user-specified procedure every user-specified number of period ticks. A clock tick in our application is set to occur every 1 ms. If we use a CLK object, it calls a user-specified function every clock tick from within the timer interrupt service routine. Unlike PRD functions, which can be lengthy calculations, CLK functions must be short since they are called from an interrupt.

To illustrate the use of both, we used a CLK object in a way that mimics the PRD approach. Our CLK function is called every tick—every 1 ms—from the timer interrupt routine. Since we want to poll the "registers" every 20 ms, it returns immediately 19 times out of 20. Every 20 ms it reads the control "registers" and stores their value in a data structure for a SWI thread, "swiControl," which it then posts. That thread applies the values to the processing threads.

This approach is useful when we get an interrupt on a hardware event and need to *read* the hardware values immediately as they change, but can *respond* to them with a delay. This delay can be quite small though. In cases where such accuracy is not necessary, a PRD object alone would be sufficient.

Thus, we split the control procedure in two parts: *reading* the values and *applying* them. In our example, they are applied when the control thread, swiControl, runs. This thread modifies parameters for the two VOL algorithm instances we use for volume control, and one value in the TxJoin thread that determines which channel to let through and which to reject.

When that thread runs depends on its priority. Its priority should never be higher than that of threads whose parameters the control thread modifies. Otherwise, a processing thread may be preempted in the midst of its calculations by the control thread, and when the processing thread resumes, it may find some of its values changed and continue the rest of its calculation with the new values, which is rarely what we want. That situation is shown in Figure 21.

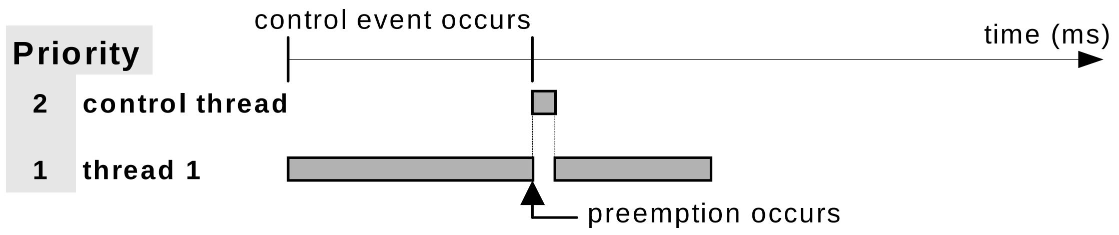


Figure 21. Incorrect Results Due to Higher Priority of Control Thread

The control thread in Figure 21 has a higher priority than the data processing thread it controls. When the user changes a control, the control thread is posted shortly after and runs because it has a higher priority. It preempts thread 1 and changes it parameters. When thread 1 resumes, it continues processing with new parameters and likely produces incorrect results.

If we give the control thread a lower priority than the threads whose parameters it has to change, there are no consistency problems, but the control thread may *starve* as shown in Figure 22, if the other threads use 100% of CPU time.

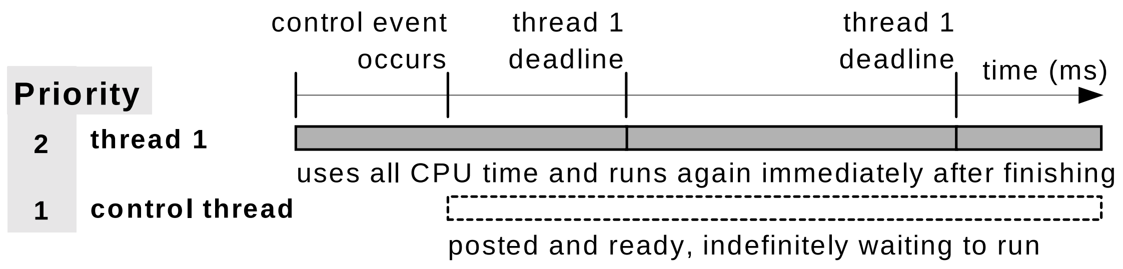

control event occurs
thread 1 deadline
thread 1 deadline
time (ms)
Priority
2
thread 1
uses all CPU time and runs again immediately after finishing
1
control thread
posted and ready, indefinitely waiting to run

Figure 22. Unresponsiveness Due to Lower Priority of Control Thread

A better solution is to make the control thread have *the same* priority as the threads whose parameters it changes. Then neither consistency problems nor starvation can happen. If the processing thread happens to be running when the control thread is posted to run, the control thread waits until the processing thread finishes. If the processing thread is not running at the time the control thread is posted, the changes are applied immediately:

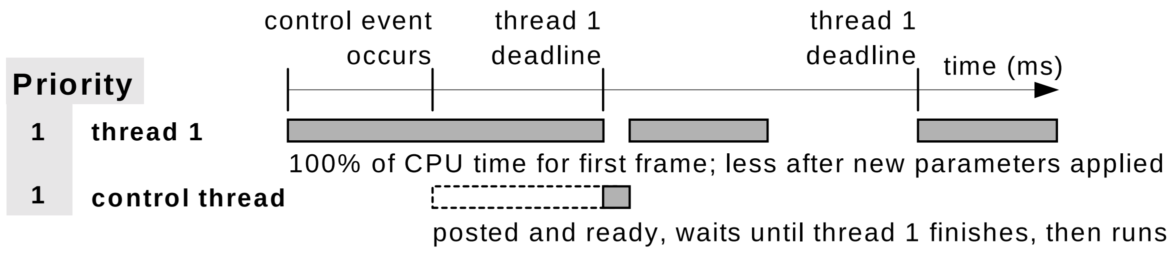


Figure 23. Correct Action with Control and Processing Threads at Equal Priority

Making changes to a thread's parameters is usually done by calling an XDAIS algorithm's control() function, but can be anything that affects data processing.

Adding the control thread to our DSP/BIOS-based data path gives us the final diagram shown in Figure 24.

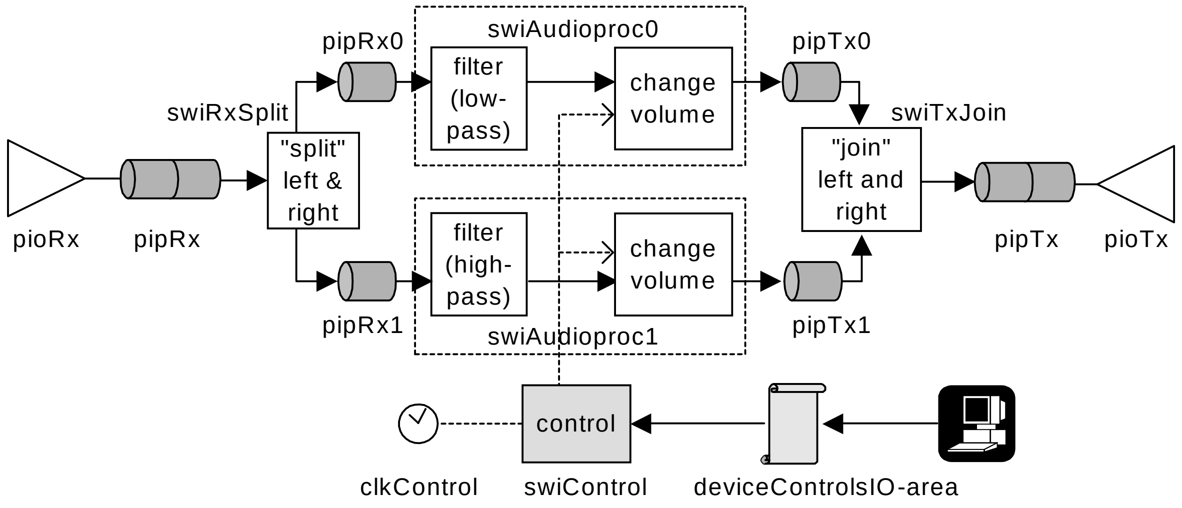


Figure 24. Final RF3 Data Path

The control thread we added is easy to remove for applications that do not need it. Section 8.5.2, *Removing the Control Thread*, page 74 shows how to remove the control thread.

7 RF3 Application Structure

In this section we expose the implementation details of RF3 based on the design we arrived at in the previous section. We discuss the division of the application into modules, data structures and procedures in each module, and the flow of information when compiling and linking the application.

While this section uses the base RF3 application as an example, most of the material presented is applicable to systems you develop using RF3 as a starting point.

7.1 Libraries

This section provides an overview of the library modules used in the RF3 application.

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

Hint: In CCStudio, using Options→Customize→Directories menu option, you can specify which folders CCStudio should search to locate the source file. If you specify source code folders for the modules used in RF3, CCStudio opens windows with their source code automatically as you step into a library module procedure.

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

7.1.1 Rebuilding Libraries

Libraries are built with debugging enabled (-g) and no optimization. For performance reasons you may wish to rebuild the libraries using optimization switches for post-development versions of your applications.

If you rebuild a library and then rebuild the Reference Framework application, either delete the executable file (app.out) or use Project→Rebuild All in order to build with the new library. CCStudio does not check for dependencies on rebuilt libraries.

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

7.1.2 The PIO Module

The PIO module and the underlying IOM model provide convenient and platform-independent access to streaming data. PIO is a pipe adapter to IOM mini-drivers. An IOM mini-driver implements basic I/O functions: opening the device, closing the device, controlling the device, taking a buffer for a transfer to or from the device, and performing a callback from the driver when the transfer is complete.

The mini-driver transfers blocks of streaming data, using the codec and CSL modules such as McBSP, DMA, or EDMA. The codec code is isolated in the mini-driver to make porting easier. Several mini-drivers are provided with Reference Frameworks. Mini-driver names are formed by combining the names of the board, CSL module used, and the codec. For example, the 'C5402 DSK mini-driver uses DMA, and is called DSK5402_DMA_AD50. The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93.

The RF3 source code never calls any of the mini-driver functions directly, except for the mini-driver's initialization function. That is the job of the PIO adapter. It lets the application read blocks of data from the codec by simply reading a pipe, and sending blocks of data to the codec by writing a pipe.

The application initializes instances of *PIO objects* (structures of type PIO_Obj). A PIO object encapsulates all information for an input or output stream: the function table for the IOM mini-driver and the name of the pipe where the PIO object should store input data or read output data from.

Using a PIO object consists of declaring it somewhere in the source, initializing it with PIO_new() call, and passing the name of the pipe object and the user-defined device object as parameters, along with the object type (input or output). In addition, the application's configuration must specify the appropriate PIO module functions for the pipe notify functions. If the pipe is used for input, its notifyWriter function must be PIO_rxPrime(). If the pipe is used for output, its notifyReader function must be PIO_txPrime().

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

7.1.2.1 Data Streaming with PIO/IOM

Recall the front and the back of the data path we arrived at in Section 6.6, *Adding the Control Thread*, page 38:

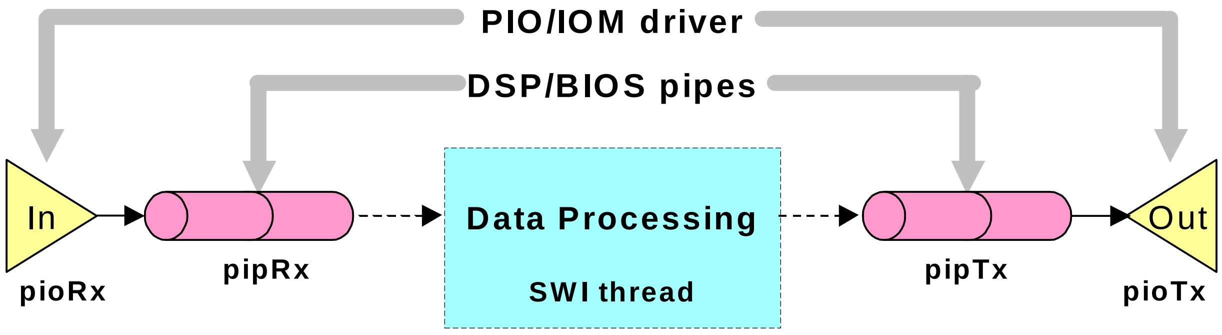


Figure 25. Ends of Data Path with PIO and PIP Objects

Input frames are delivered from the codec to pipe pipRx using an instance of the PIO driver whose properties are encapsulated in the pioRx object. Similarly, the frames we store into pipTx pipe are taken by the PIO driver, via its pioTx instance, and sent to the codec.

In Figure 26, numbers indicate the sequence of events when receiving a frame. The sequence of events is symmetric on the transmission side.

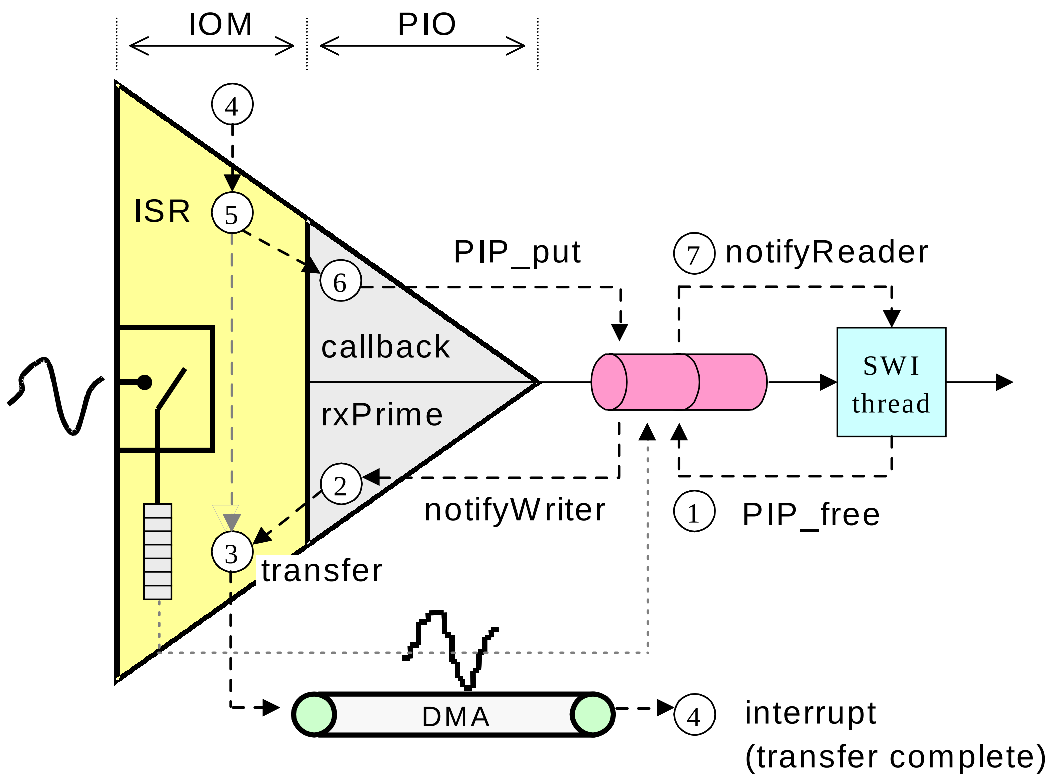


Figure 26. Data Connections and Execution Flow in PIO/IOM Driver

1. When the input pipe receives a frame, it calls PIP_free for the pipRx pipe object to free the frame.
2. The pipRx pipe object runs its notifyWriter function to notify the writer that a frame is available. It does this by calling PIO_rxPrime for the pioRx object. (During system startup, DSP/BIOS automatically runs the notifyWriter functions when initializing pipes.)
3. The call to PIO_rxPrime for pioRx initiates a DMA transfer from the codec to the free frame in the pipe.
4. After a full frame has been transferred, a DMA completion event is signaled.
5. The completion event causes the mini-driver to run its ISR function with the transfer mode—IOM_INPUT—as the argument. (Mini-driver ISRs are configured programmatically by the mini-driver's initialization function.) The ISR uses the mode as an index to retrieve the address of the callback function for that state. For IOM_INPUT, the callback function is rxCallback. In addition, the ISR checks to see if another job is pending. If a job is pending, as is usually the case, the ISR reprograms the DMA. (This reduces round-trip latency between the ISR and a DMA transfer. This latency reduction enables the driver to operate at high frequencies without missing samples.)
6. The driver ISR calls rxCallback for pioRx. This function indicates that the transfer to the frame is complete, so pioRx calls PIP_put for the pipRx pipe. The data is already in the frame, as the frame's address was used when the transfer was initiated.

7. Calling PIP_put causes pipRx to run its notifyReader function, which is a SWI_andnHook() call. This clears one of the SWI object's flags. The other flag is cleared when the output pipe contains an empty frame. After data is transferred to the output pipe (pipTx), that pipe calls PIP_free and the cycle begins again.

Note that since the pipRx pipe is double-buffered, the transfer of the *next* input frame begins immediately after the previous frame is completed so we never lose any samples.

Figure 27 provides more details on the data streaming path in RF3.

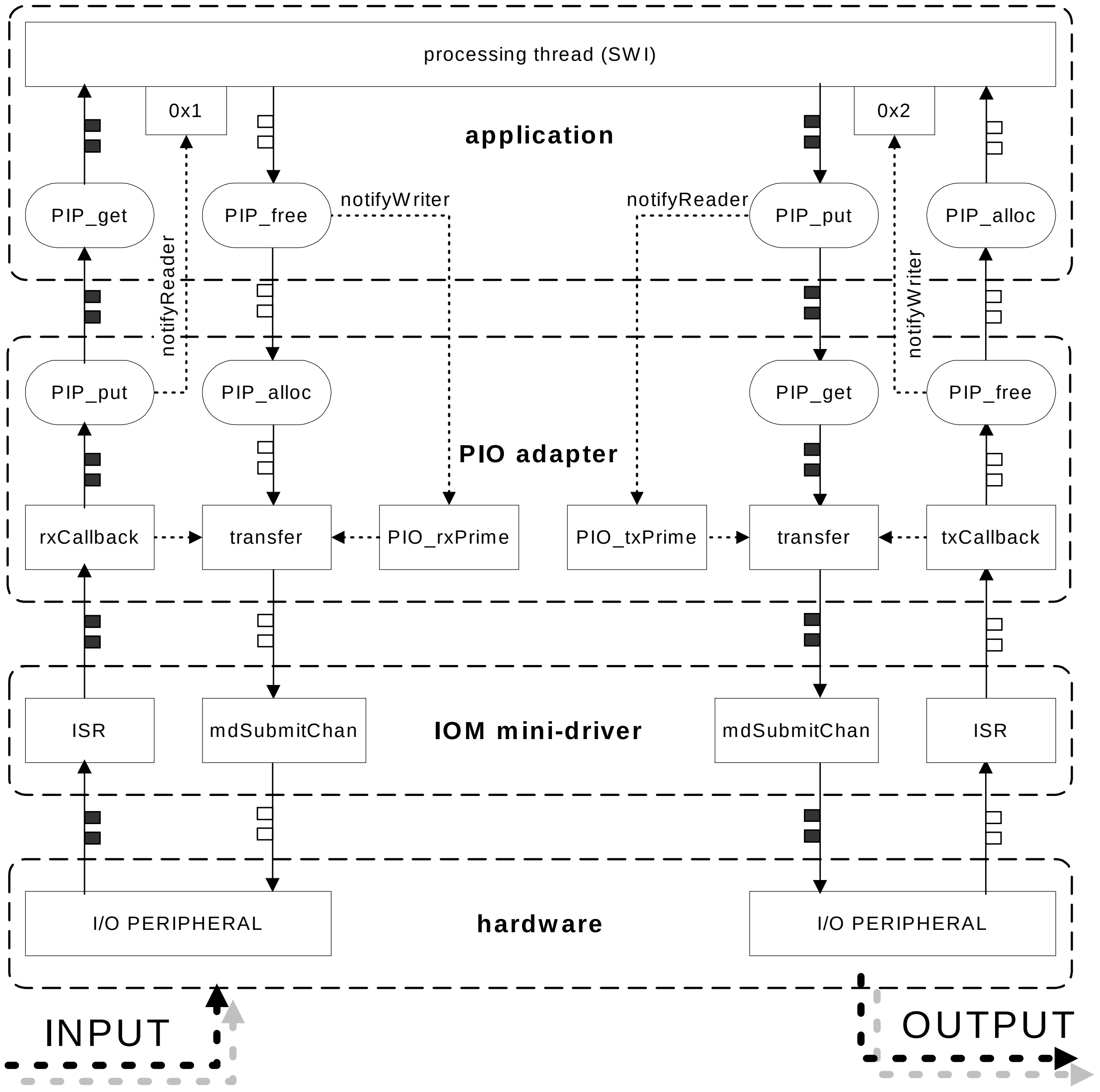


Figure 27. Data Streaming Details

7.1.3 The ALGRF Module

The ALGRF module exists to create and delete XDAIS algorithms by using the DSP/BIOS MEM memory manager. It is a Reference Framework service that simplifies the use of XDAIS components in end-applications. Any module that implements IALG (hence any XDAIS-compliant algorithm) can be used with ALGRF.

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

Higher-level Reference Frameworks typically use the ALGRF module to create, configure, and delete instances of XDAIS algorithms. Another implementation exists in the form of the ALG module supplied with CCStudio. ALG is for general-purpose use, while ALGRF uses the DSP/BIOS MEM module for memory allocation. ALGRF is also smaller and more controllable than ALG. ALGMIN (used by RF1) is the smallest implementation of the three.

Table 4 compares the three IALG implementations.

Table 4. IALG Implementation Characteristics

	ALGMIN	**ALGRF**	**ALG**
Provided with	RF1	RF3	CCStudio
Best used with	Compact, static systems	RF3 and higher	General purpose use
Memory allocation	Static	Dynamic; uses DSP/BIOS MEM module	Dynamic; supports both malloc() and MEM allocation.
Key points	• Super-compact. • ALGMIN_new is key instantiation function. • ALGMIN_new calls algInit but not algAlloc since buffers are pre-generated.	• Uses DSP/BIOS MEM module for dynamic allocation. • Smaller footprint than ALG. • Supports sharing of scratch memory.	• Supports both malloc() and DSP/BIOS MEM dynamic allocation. • Not tuned for footprint.

Naturally these modules are mutually exclusive. Only one should be used in an end-application. ALGRF fits the needs of RF3 and, very likely, other RF levels. It is not appropriate however for extremely compact, low-end systems such as RF1.

ALGRF has the following advantages over ALG:

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

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

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

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

RF3 does not use this API. It merely calls ALGRF_create() without sharing scratch data memory. This design choice was made since many unknown application factors exist which may influence the scratch re-use policy. For example, applications using a single priority have few scratch considerations compared to applications with many priority levels.

The door to scratch re-use is left open however, with the ALGRF module library.

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

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

The ALG module will remain in CCStudio to support legacy content and enable non-DSP/BIOS, or "generic" applications. ALGRF sits side-by-side as an alternative XDAIS instantiation module.

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

7.1.4 The UTL Module

Throughout RF3 modules you find UTL module calls—macros beginning with the UTL_ prefix. These are macros that can either be expanded to the code that performs the desired debugging function, or to nothing.

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

Syntax for all UTL macros is provided in the *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147) application note.

You define the desired amount of debugging features by setting the UTL_DBGLEVEL flag to a desired level in your application's Build Options. A debugging level includes the class for its level and all the classes from previous levels. You can also individually include or exclude classes regardless of the specified level. Figure 28 shows the debug levels and the features they enable:

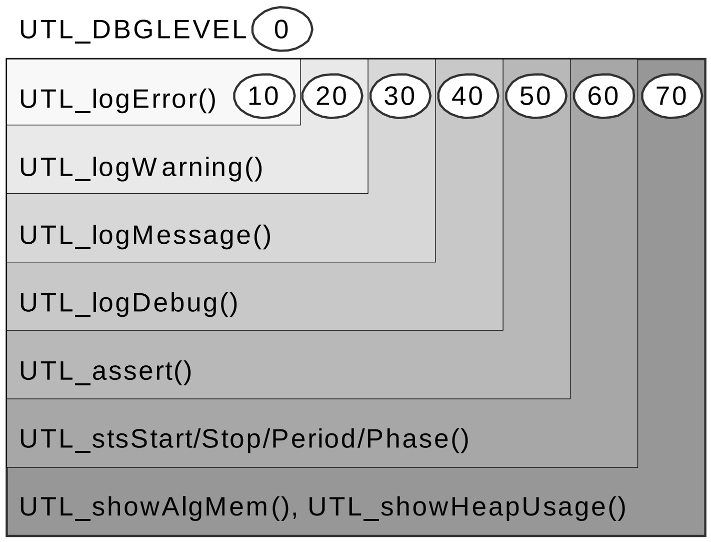


Figure 28. UTL Debugging Levels

By default, the RF3 projects are built with UTL_DBGLEVEL set to 70.

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

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

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

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

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

The UTL module implements the following classes of debugging features:

- **Message severity macros.** Errors, warnings, diagnostic messages, and debugging messages can be sent through macros that expand to LOG_printf()-style calls. The macro names are UTL_logError(), UTL_logWarning(), UTL_logMessage(), and UTL_logDebug().
- **Assertion macro.** Stops execution for a failed assertion and brings the offending source code line to focus in the debugging window. The macro syntax is UTL_assert(<condition>). This function is particularly useful in a system where all initialization operations (such as algorithm instantiation) must succeed. Of course, some may fail as you debug applications. So, rather than returning various success/failure codes from initialization functions, we halt the target if a crucial operation fails.

- **Time measurement macros.** These can report execution time, time between two periodic executions of a point in the program, and phase between two periodic points. The statistics are shown in STS objects in CCStudio. The macros are UTL_stsStart() and UTL_stsStop() for execution times, UTL_stsPeriod() for measuring time between periodic executions of a point in the program, and UTL_stsPhase() for measuring phase between two periodic points. The configuration for RF3 contains ten STS objects that can be used for time/period measurements with UTL_sts* functions. The thrRxSplit.c and thrTxJoin.c files use the UTL_stsPeriod() and UTL_stsPhase() macros.

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

Figure 29 shows some example calls to UTL functions and some sample UTL output messages.

■ Display memory usage

```
UTL_showAlgMem(thrAudioproc[i].algFIR);
UTL_showHeapUsage( INTERNALHEAP );
```

■ Avoid success/error codes at initialization

```
thrAudioproc[i].algFIR =
  FIR_create( &FIR_IFIR, &firParams );
UTL_assert(thrAudioproc[i].algFIR != NULL);
```

■ Check “must be true” conditions at runtime

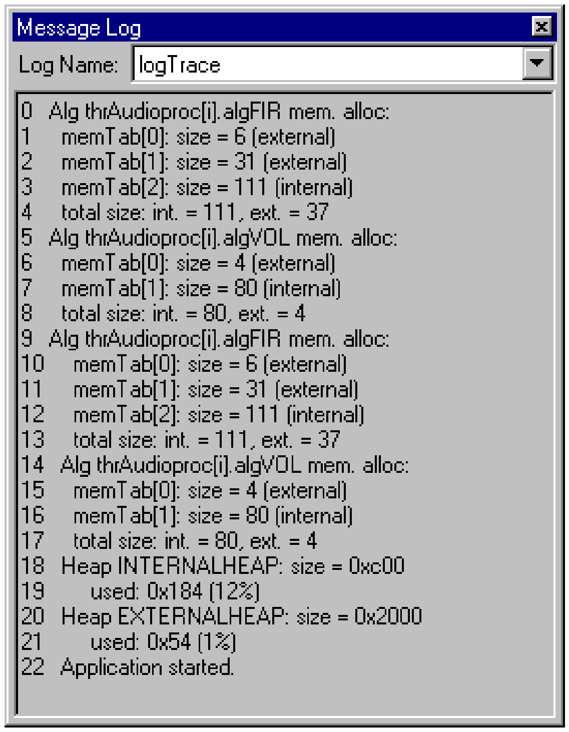
```
UTL_assert( bufPtr != NULL );
```

■ Measure intervals between function calls

```
Void thrProcessRun() {
  /* process new frame of data ... */
  UTL_stsPeriod( stsProcess );
}
```

■ Display diagnostic messages

```
UTL_logDebug( "Application started." );
```



```
0  Alg thrAudioproc[i].algFIR mem. alloc:
1    memTab[0]: size = 6 (external)
2    memTab[1]: size = 31 (external)
3    memTab[2]: size = 111 (internal)
4    total size: int. = 111, ext. = 37
5  Alg thrAudioproc[i].algVOL mem. alloc:
6    memTab[0]: size = 4 (external)
7    memTab[1]: size = 80 (internal)
8    total size: int. = 80, ext. = 4
9  Alg thrAudioproc[i].algFIR mem. alloc:
10    memTab[0]: size = 6 (external)
11    memTab[1]: size = 31 (external)
12    memTab[2]: size = 111 (internal)
13    total size: int. = 111, ext. = 37
14  Alg thrAudioproc[i].algVOL mem. alloc:
15    memTab[0]: size = 4 (external)
16    memTab[1]: size = 80 (internal)
17    total size: int. = 80, ext. = 4
18  Heap INTERNALHEAP: size = 0xc00
19      used: 0x184 (12%)
20  Heap EXTERNALHEAP: size = 0x2000
21      used: 0x54 (1%)
22  Application started.
```

Figure 29. UTL Message Example

7.2 RF3 Source Code

This section shows the application hierarchy and program flow and then discusses the contents of individual C files.

7.2.1 *The Project File*

The application project file, app.pjt, includes all the .c and .h files in the rf3\ folder, as well as app.cdb file and the link.cmd linker-command file.

All debugging options are on, and the UTL_DBGLEVEL variable, which determines the amount of debugging support pulled in from the UTL module, is set to the maximum.

7.2.2 *Compilation Information Flow*

The following diagram shows the flow of header files. An arrow in the diagram indicates that the file at which the arrow ends includes the file from which the arrow starts.

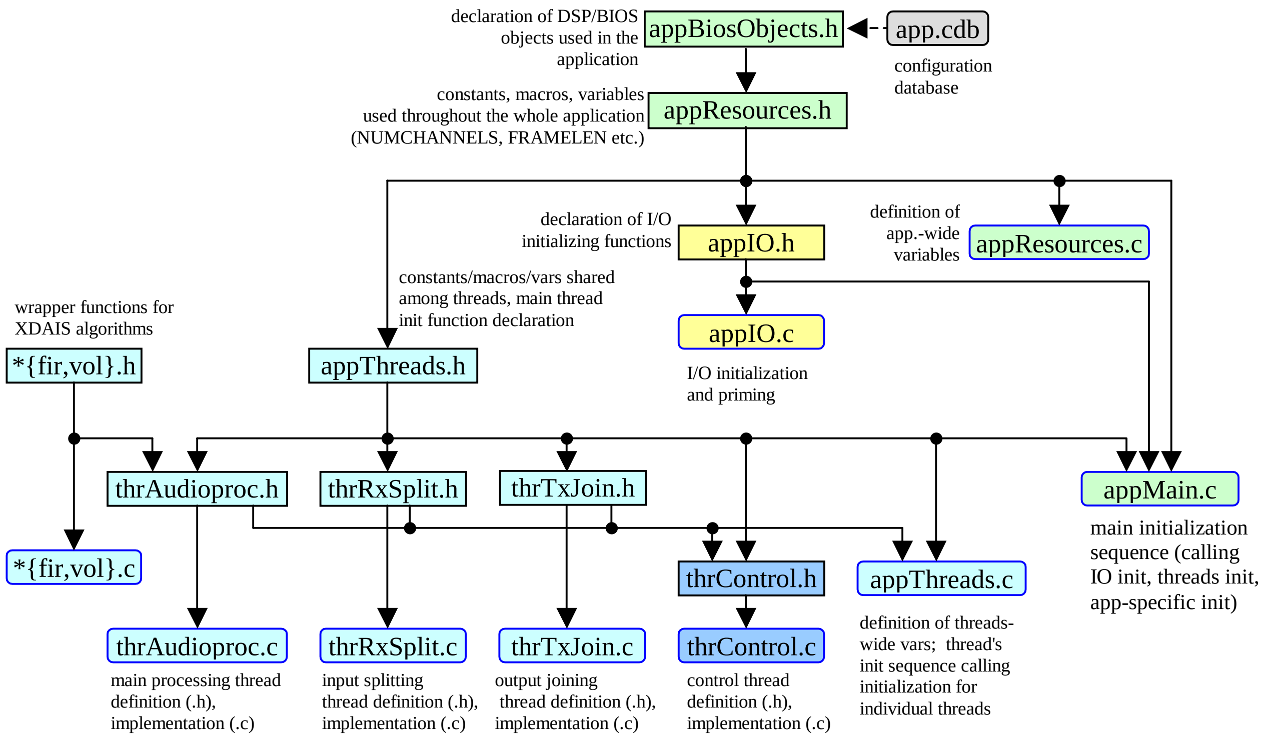

declaration of DSP/BIOS objects used in the application
appBiosObjects.h
app.cdb
configuration database
constants, macros, variables used throughout the whole application (NUMCHANNELS, FRAMELEN etc.)
appResources.h
declaration of I/O initializing functions
appIO.h
definition of app.-wide variables
appResources.c
constants/macros/vars shared among threads, main thread init function declaration
appIO.c
I/O initialization and priming
wrapper functions for XDAIS algorithms
*{fir,vol}.h
appThreads.h
thrAudioproc.h
thrRxSplit.h
thrTxJoin.h
appMain.c
main initialization sequence (calling IO init, threads init, app-specific init)
*{fir,vol}.c
thrControl.h
appThreads.c
thrAudioproc.c
thrRxSplit.c
thrTxJoin.c
thrControl.c
definition of threads-wide vars; thread's init sequence calling initialization for individual threads
main processing thread definition (.h), implementation (.c)
input splitting thread definition (.h), implementation (.c)
output joining thread definition (.h), implementation (.c)
control thread definition (.h), implementation (.c)

Figure 30. Flow of Header File Information

Our intention was to implement modules in the system as *objects*, not unlike objects in C++. For example, the TxJoin thread's data object is thrTxJoin, an instance of type ThrTxJoin. The type is defined in the thread's header file, thrTxJoin.h, and its "public" functions—its interface—are declared in the same header file. The .c file defines the thrTxJoin instance with its initial values, and implements the thread's public functions. A well-defined data-processing thread should be usable anywhere in the data path with only its static initialization code modified. This approach improves readability and maintainability of code without incurring performance penalties.

One apparent exception to this rule is the control thread, which reads and writes "private" members of data processing threads' data object. This, however, is similar to the "friend" concept in object-oriented programming, and was necessary to avoid performance costs.

In DSP/BIOS, modules follow the "object" approach. For a module MOD, an object is defined as a variable of type MOD_Obj, and MOD_Handle is a pointer to the object. All the module functions take the form MOD_functionName(MOD_Handle handle, …).

The naming approach in the top-level application is somewhat different since we do not build a module but a whole application. For that reason we do not prefix our functions and data types with MOD_, but use a (simplified) Hungarian notation, in which the lowercase prefix of the variable name determines its type. We use this notation for global variables only.

7.2.3 Global Data Objects and Functions

To aid understanding of the program execution flow, the following sections list all non-DSP/BIOS global objects and functions in the RF3 application.

7.2.3.1 Data

To implement threads, we make a thread-specific data structure that encapsulates the thread's private state information for each thread. Some of the information in the structures is determined statically, at load time, some is determined in initialization time, and some, of course, is used and modified throughout the life of the thread.

Let us look first at the Audioproc threads' data structure:

```
typedef struct ThrAudioproc {
    /* algorithm(s) */
    FIR_Handle  algFIR;     /* an instance of the FIR algorithm */
    VOL_Handle  algVOL;     /* an instance of the VOL algorithm */

    /* input pipe(s) */
    PIP_Handle      pipIn;

    /* output pipe(s) */
    PIP_Handle      pipOut;

    /* intermediate buffer(s) */
    Sample         *bufInterm;

    /* everything else that is private for a thread comes here */

} ThrAudioproc;

extern ThrAudioproc thrAudioproc[ NUMCHANNELS ];
```

As we can see, an Audioproc thread is described with the following elements: two algorithm handles, one each for the FIR and VOL algorithms; one pointer to the thread's input pipe, one pointer to the thread's output pipe; and one pointer to the intermediate buffer the thread uses.

The principle is that we do not want to "hardwire" the thread to DSP/BIOS and other global objects it uses. Instead, we use handles for all the objects and then "connect" handles to the actual objects in the initialization phase. In the Audioproc0 example, we statically bind the pipIn handle to the pipRx0 pipe, and the thread's run() function never refers to pipRx0 but always to pipIn when reading input. If we were to put another thread before the Audioproc0 thread in the data path that would pre-preprocess the block Audioproc0 receives, we would only modify the connection in the static initialization without having to change the thread's run-time code.

The same applies for the output pipe and the address of the intermediate buffer the thread uses.

We use the principle of grouping thread information for threads that do not have multiple instances as well. The RxSplit thread is one such thread. While it is true that the RxSplit would not have to have any information enclosed in a thread-specific structure, we choose to do so nevertheless, for the reasons of uniformity and easiness of debugging—if an important thread variable is static it is not visible outside the thread's module. Grouping thread's data into named structures incurs no extra cost in access time or memory requirements.

This is the private structure for the RxSplit thread:

```
typedef struct ThrRxSplit {
    PIP_Handle pipIn;                  /* input pipe with joined channel data */
    PIP_Handle pipOut[ NUMCHANNELS ];  /* outpipes with individ. channel data */
} ThrRxSplit;

extern ThrRxSplit thrRxSplit;
```

Handle pipIn is initialized to point to pipRx, and the pipOut[] array handle is initialized to point to [pipRx0, pipRx1]. (NUMCHANNELS is a global macro indicating the number of channels, and is set to 2 in our application.)

We statically initialize the thrRxSplit object as follows:

```
ThrRxSplit thrRxSplit = {
    &pipRx,                 /* pipIn */
    { &pipRx0, &pipRx1 }    /* pipOut[ NUMCHANNELS ] */
};
```

This thread works wherever we plug it in the data path, if its static initialization is correct for its position in the data path.

The pioRx and pioTx global objects are two PIO objects that capture physical-device-to-pipe and pipe-to-physical-device information. The deviceControlsIOArea integer array is a simple array to which the GEL script writes the values of the GUI sliders.

7.2.3.2 Functions

The following diagram shows the function call hierarchy.

All the *run() functions are threads' run functions and are called by the corresponding SWI objects. The thrControlIsr() is called by the clkControl object, and main() is called by DSP/BIOS after initialization and before startup.

```
- main()
    +---- appIOInit()
    |       `-------- PIO_new() x 2
    |
    +---- thrInit()
    |       +-------- ALGRF_setup()
    |       +-------- thrRxSplitInit()
    |       +---------thrAudioprocInit()
    |       |           `---------------- [FIR_create(), VOL_create()] x 2
    |       |
    |       +-------- thrTxJoinInit()
    |       `-------- thrControlInit()
    |
    `---- appIOPrime()

- thrRxSplitRun()

- thrAudioprocRun()
    `---- FIR_apply(), VOL_apply()

- thrTxJoinRun()

- thrControlIsr()

- thrControlRun()
    +---- thrTxJoinSetOutputChannel()
    `---- thrAudioprocSetOutputVolume()
           `---- VOL_control() x 2
```

7.2.4 Source Code Files

This section briefly explains the purpose and content of important source files in the application. For detailed information, see the actual files, which are well commented.

While the discussion focuses on the source code in the context of the RF3 application, comments are given for situations where you want to modify the RF3 application or to build your own application from it.

7.2.4.1 appBiosObjects.h

This file "exports" all the DSP/BIOS objects to which the application code may refer, including PIP objects and LOG objects. Since this file is at the root of the inclusion tree, those objects are visible to every module.

Whenever you change names or add new objects to the configuration, you should update this file manually. In the future this file may be generated automatically.

7.2.4.2 appResources.h

This file contains application-wide macros, data types, and function/data declarations that may be used throughout the application. It is similar to appBiosObjects.h files, except that appBiosObjects.h contains information about DSP/BIOS objects only.

In RF3, this file defines the number of channels, NUMCHANNELS, size of one frame, FRAMELEN, and the data type that holds one data sample—that is, type Sample.

Type Sample is defined as Short since Short is 16 bits on all 'C5000 and 'C6000 devices. We needed a new type to distinguish words sizes. Pipes count their size in *words*. On a 'C5000 a word is 16 bits, so it holds one 16-bit sample. But on a 'C6000 a word is 32 bits, so it holds two 16-bit samples. For that reason, this file provides two macros, sizeInSamples() and sizeInWords(), to perform conversion and avoid confusion.

7.2.4.3 appResources.c

This file should *define* (that is, allocate space and initialize, if needed) all the global variables declared in appResources.h. RF3 doesn't have any such variables (except for the UTL_sts objects, which are used for debugging), so this file is essentially empty.

7.2.4.4 appIO.h

The appIO.h file declares two I/O functions, appIOInit() and appIOPrime(), for initializing I/O devices and for *priming* I/O pipes. Priming refers to placing initial zero frames in the pipes. Priming must be a separate operation from initializing, since I/O devices are usually initialized first and primed as the last operation in the main() calling sequence.

This file also declares the two PIO objects, pioRx and pioTx, for the input and output side, respectively. While these objects are not used outside appIO.c, conceivably you may need to access them elsewhere in your application so they are made globally visible.

This is the place where you should define the interface to any other application-specific functions your application needs, if you have them.

7.2.4.5 appThreads.h

Analogous to appResources.h, which defines information needed throughout the entire application, appThreads.h defines macros, constants and declares variables and functions that are needed by the threads in the system.

In RF3 this file declares the intermediate buffer used by both Audioproc threads, and declares the thrInit() function, which initializes all threads. This function is called from main().

7.2.4.6 appMain.c

This file contains the application's main() function. It includes the global resource header file, appResources.h, and the header files defining the interface to I/O and the threads, appIO.h and appThreads.h. The main() function calls appIOInit(), appThreadInit(), and appIOPrime(). It then exits and gives control to DSP/BIOS.

In your application, this file should perform any other necessary initialization.

7.2.4.7 appIO.c

The appIO.c file is the core location for board-specific code. A different version is provided in the folder for each board. This file encapsulates calls that differ due to varying codecs and sample rates. It implements the I/O initialization and priming, and it defines the two PIO objects.

To initialize the I/O, this file calls the PIO_new() function for each PIO object. A call to PIO_new() consists of the address of the PIO object, the address of the associated pipe, the name of the IOM mini-driver (configured as a user-defined device), the operation mode (input or output), and the address of the driver's function table.

The priming function places two empty frames in the output pipe, pipTx. It does so to ensure continuous output and avoid initial pops and clicks that might be heard otherwise—which brings us to the subject of priming.

To explain priming, let us look at an example where priming is not performed. Assume that frames are 80 16-bit samples long at 8 kHz sampling rate. That is, a new input frame arrives every 10 ms. Obviously, a new output frame must be sent every 10 ms. In other words, as soon as the transfer of one output frame is complete, the next processed frame must be ready and waiting. If not, there is silence on the output until the next output frame is ready, which is not what we want, since we need continuous output.

Assume that the first and second input frames take 4 ms to be processed, and the third frame for some reason needs 7 ms to be processed. Assume also that both input and output pipes are double-buffered. Let's look at the time diagram of the events:

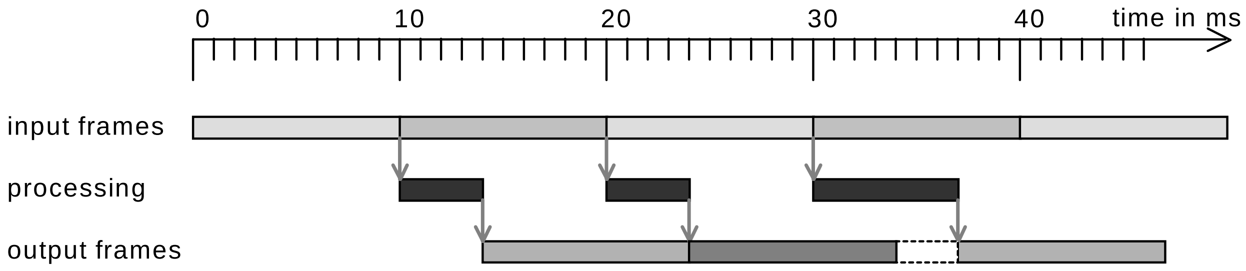


Figure 31. Missed Real-Time Deadline with Output Pipe Not Primed

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, we can begin to transfer the third output frame. We do not want the 3 ms gap, which can occur any time the processing time is larger than all previous processing times. So even if processing always completes 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 perform initial processing. If we prime the *output* pipe with two zero frames of size 80 (full size), we get the following diagram:

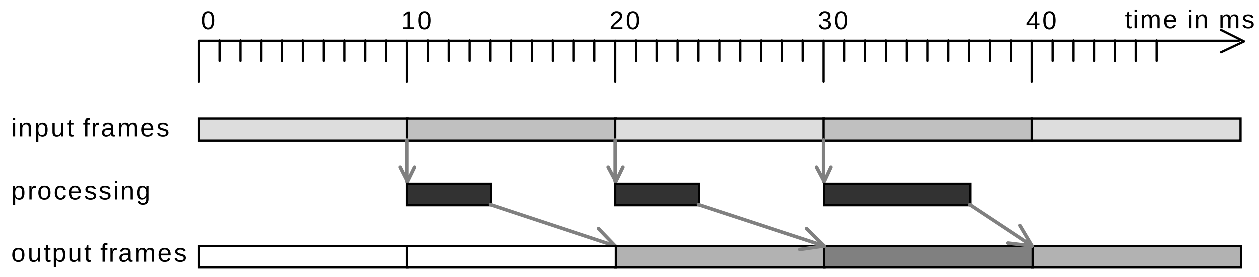


Figure 32. Continuous Output with Primed Output Pipe

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

7.2.4.8 thrRxSplit.h

This header file declares the RxSplit thread's data object, thrRxSplit. Since the RxSplit thread is simple, its structure needs only the handles to its input pipe and two output pipes.

This file also declares the RxSplit thread's two "public" functions, thrRxSplitInit() and thrRxSplitRun(). The former is called from main(). The latter is called from within the SWI object, and not directly by the application code. However, it is declared here for completeness.

7.2.4.9 thrRxSplit.c

This file implements the RxSplit thread. As with other threads, it has three parts:

- **Static initialization of the thread's data object.** Static initialization consists of connecting the thread's pipe handles to addresses of actual pipes, which is a matter of topology—the data path. Again, if we had to put this thread elsewhere in the data path, we would only need to change this connection.
- **Run-time initialization of the thread.** Runtime initialization, performed by the thrRxSplitInit() function, is empty, since the RxSplit thread is simple, but the function is included for symmetry.
- **The run() function called by the SWI object.** The thrRxSplitRun() function is called by the swiRxSplit object. Recall that swiRxSplit is automatically posted when its mailbox drops to binary 000 from binary 111. Each of the three pipes used by the thread clears one bit in the thread's mailbox via the SWI_andnHook() function in the pipes' notify function. So when the thread is posted and ready, it "knows" that its input pipe has a frame to read, and its two output pipes each have a free frame to be written to.

  This function gets the input pipe—again, using the pipIn handle and not the pipRx name—via PIP_get() function, determines the address and the size of the input frame with PIP_getReaderAddr() and PIP_getReaderSize(). Then, in a loop, it allocates each output pipe and stores the address of its free frame into an array (of size NUMCHANNELS).

The center activity in the thrRxSplitRun() function is copying the source frame to each of the destination frames. For a stereo codec, we copy every other source sample to a destination. For a mono codec, we copy all the samples.

Finally, for each output pipe, we inform the pipe about the size of the frame we just copied, and we put the pipe. Finally, we free the input pipe, and the function exits.

7.2.4.10 thrTxJoin.h

The file is similar to thrRxSplit.h file, with a small difference that the ThrTxJoin structure has the "Int outputChannel" member, which holds the number of the channel whose frames should be sent to the output. The control thread can change this information based on the user's action.

7.2.4.11 thrTxJoin.c

This file is also similar to its RxSplit counterpart, with the difference that it copies one input to the only output, as determined by thrTxJoin.outputChannel value, and discards the other.

Note, however, that even though we do not use one channel, we have to get and free the input pipe for that channel (pipTx0 or pipTx1) for our system to work. Otherwise, the pipe would never call its notifyWriter function, the corresponding swiAudioproc(0 or 1) thread would not be posted as its mailbox would not be cleared, so it in turn would not read its input pipe (pipRx0 or pipRx1), that pipe would not call its notifyWriter function, the swiRxSplit thread would not be posted, and the whole system would freeze after only two frames.

There is one additional operation this thread performs. The supported mono codecs require that the lowest bit of output samples be cleared. For that reason, the thrTxJoinRun() procedure copies the selected input frame to the output frame differently depending on whether APPMONOCODEC is defined in CCStudio in the project Build Options. For mono codecs, the code clears the last bit in each sample from the source buffer as it copies it to the destination buffer.

```
/* copy its samples to appropriate places in the destination frame */
   #ifdef APPMONOCODEC     /* simulation of stereo/N-ch. codec with mono */
   if (ch == thrTxJoin.outputChannel) {
      /* do the copy only if we're processing the active output channel */
      for (i = 0; i < FRAMELEN; i++) {
          /* AD50 and AD535 codecs must have LSB set to 0 */
          dstBuf[i] = srcBuf[i] & 0xfffe;
      }
   }
   #else /* real stereo or N-channel codec */
   for (i = 0; i < FRAMELEN; i++) {
      dstBuf[ i * NUMCHANNELS + ch ] = srcBuf[i];
   }
   #endif
```

Note: The AD50 and AD535 codecs require that you clear the lowest bit in each sample that is output to the codec. For these codecs, if you remove the TxJoin thread from the data path, make sure the LSB is cleared before delivering the data to the pipe that the driver reads.

7.2.4.12 fir.h, fir.c, vol.h, vol.c

These four files provide function wrappers for the two dummy XDAIS algorithms provided. They let the application call algorithm functions in a "friendly" way. For example, the application can call the FIR_apply(<params>) instead of using the firHandle->fxns->filter(<params>) syntax. These function wrappers increase the readability of the application code.

Most wrappers are static inline functions implemented in the .h file. Wrappers that must be implemented in code are in the .c file. Typically, where *alg* is the algorithm name, they implement *alg*_create(), *alg*_delete(), *alg*_activate(), *alg*_deactivate(), *alg*_init(), *alg*_exit(), *alg*_control(), and algorithm-specific functions such as FIR_apply for the FIR algorithm.

Typically algorithm vendors provide wrapper files. In order to use the wrapper files with RF3, they need to be modified slightly to use the ALGRF module for XDAIS instantiation instead of the ALG module used by most default wrappers. See Section 8.5.4, *Creating Algorithm Wrappers*, page 75 for the steps to perform to modify provided wrapper files or to create new wrapper files.

7.2.4.13 ifir.c, ivol.c

Files ifir.c, ivol.c contain default algorithm instance parameters and are provided by the vendor. We simply copied these files into their local folders and included them in the application. It is recommended not to modify these files, but to overload the parameter values at initialization time.

7.2.4.14 thrAudioproc.h

The thrAudioproc.h header file defines the Audioproc thread's data structure and external interface. Let us examine the structure again:

```
typedef struct ThrAudioproc {
    /* algorithm(s) */
    FIR_Handle  algFIR;     /* an instance of the FIR algorithm */
    VOL_Handle  algVOL;     /* an instance of the VOL algorithm */

    /* input pipe(s) */
    PIP_Handle      pipIn;

    /* output pipe(s) */
    PIP_Handle      pipOut;

    /* intermediate buffer(s) */
    Sample         *bufInterm;

    /* everything else that is private for a thread comes here */

} ThrAudioproc;

extern ThrAudioproc thrAudioproc[ NUMCHANNELS ];
```

An Audioproc thread has a FIR handle for a FIR instance, a VOL handle for a VOL instance, a PIP handle for an input pipe, a PIP handle for an output pipe, and a pointer to an intermediate buffer. Anything else that encapsulates the state of the thread would be placed here as well. By grouping private data this way, we can create arrays of threads, which we indeed do, one

Audioproc thread for each channel. When a thread runs, it knows its index—channel number—which it uses to access its data structure.

Other than the standard declaration of the thread's init() and run() functions, this thread has the following declarations as well:

```
extern IFIR_Fxns FIR_IFIR;      /* FIR algorithm */
extern IVOL_Fxns VOL_IVOL;      /* VOL algorithm */
```

We know that an XDAIS algorithm is identified by its function table. For the FIR algorithm, the function table is of type IFIR_Fxns and is named FIR_TI_IFIR. At the linking stage, we define the vendor-independent FIR_IFIR symbol to be equal to FIR_TI_IFIR. This enables us to use a different vendor's FIR algorithm without recompiling the application but just by relinking it.

7.2.4.15 thrAudioproc.c

This file is the core of the application, as it implements the thread that processes the data streams by calling XDAIS algorithms. As with other threads, it has three parts:

- **Static initialization of the thread's data object.** Static initialization is similar to that of other threads. We declare the thread object(s) and "connect" the pointers to actual objects in the data path. For Audioproc it is one input pipe, one output pipe, and one intermediate buffer. Since NUMCHANNELS is 2, we initialize that many Audioproc threads in an array. We try to initialize statically as much as possible. However, with the ALGRF module, the XDAIS algorithm instances must be created dynamically, so we use NULL for their initial values.

- **Run-time initialization of the thread.** The next phase is dynamic initialization. The code first prepares static parameters that are used in the dynamic initialization. In the default case, we have two sets of FIR filter coefficients, one for a low-pass filer and one for a high-pass filter. Your algorithms may require other static data that you can initialize this way.

  The thrAudioprocInit() function must initialize *all* instances of the Audioproc threads, in our case thrAudioproc[0] and thrAudioproc[1]. For each thread, in a loop, it creates and initializes all XDAIS algorithm instances the thread uses.

  For each thread, we first create one FIR instance and one VOL instance, thus initializing the algFIR and algVOL fields in the thread's data structure. We create a FIR instance by assigning the default parameters structure to a local parameters variable and then changing the fields that differ from the defaults. We do this similarly for the VOL instances.

- **The run() function called by the SWI object.** The thrAudioprocRun() function is fairly straightforward. The function is called by the swiAudioproc0 and swiAudioproc1 objects, which pass 0 and 1 as the argument, respectively. The "chan" argument is the channel number, which is used to access the correct thread data structure.

  The function follows the regular pattern of getting the input and allocating the output pipe, determines the addresses and sizes of both frames, processes the input frame, and stores the result in the output frame after calling FIR and VOL. The input for FIR is the input frame and the output for FIR is the intermediate buffer. The intermediate buffer is the input for VOL, and its output is the output frame. Finally it frees the input pipe and puts the output pipe before it exits.

The following code for the thrAudioprocRun() function has been shortened by omitting UTL calls and some of the comments.

```
Void thrAudioprocRun( Arg aChan )
{
    Sample *src, *dst;
    Int     size;       /* in samples */
    Int     chan;

    /* cast 'Arg' types to 'Int'. This is required for the 'C55x large data model. */
    chan = ArgToInt( aChan );

    /* get the full buffer from the input pipe */
    PIP_get( thrAudioproc[chan].pipIn );
    src = PIP_getReaderAddr( thrAudioproc[chan].pipIn );
    /* get the size in samples (the function below returns it in words) */
    size = sizeInSamples( PIP_getReaderSize( thrAudioproc[chan].pipIn ) );

    /* get the empty buffer from the out-pipe */
    PIP_alloc( thrAudioproc[chan].pipOut );
    dst = PIP_getWriterAddr( thrAudioproc[chan].pipOut );

    /* apply filter and store result in intermediate buffer */
    FIR_apply( thrAudioproc[chan].algFIR,
               src, thrAudioproc[chan].bufInterm );

    /* amplify the signal in the interm. buffer and store result in dst */
    VOL_apply( thrAudioproc[chan].algVOL,
               thrAudioproc[chan].bufInterm, dst );

    /* Record the amount of actual data being sent */
    PIP_setWriterSize( thrAudioproc[chan].pipOut, sizeInWords( size ) );

    /* Free the receive buffer, put the transmit buffer */
    PIP_free( thrAudioproc[chan].pipIn  );
    PIP_put ( thrAudioproc[chan].pipOut );
}
```

This file also contains the thrAudioprocSetOutputVolume() function, which is called by the control thread to set the output volume. It gets the handle for a VOL instance from the thrAudioproc[] structure and invokes the control() function to change the volume.

Note that we change the state of each VOL instance by first reading the current status, then changing a single field in the returned structure, and passing the new structure to the algorithm's control() function. This ensures that we do not pass corrupt, uninitialized values in the parameters structure we created locally.

7.2.4.16 thrControl.h, thrControl.c

The control thread is a PRD thread. Part of the control process is the thrControlIsr() function as well, which is called by the clkControl CLK object every tick. The latter simulates a regular interrupt routine that is called when an external control-related hardware event occurs. This ISR reads the pretend-I/O registers containing the current value of two sliders (volume for each channel) and one switch (selection of active channel). These "I/O registers" in the "deviceControlsIOArea" integer array are really written by the GEL script or the debugger in the watch window.

Every twentieth tick, thrControlIsr() does not return immediately but reads the "registers," interprets them, and stores the result—the channel volume and active channel—in the Control thread's data structure, thrControl. Then it posts the thread.

The control thread runs when appropriate, since it has the same priority as the threads whose values it modifies. The swiControl object calls thrControlRun(), which disables hardware interrupts while getting local copies of the output channel and volume variables from the thrControl structure. It then re-enables interrupts and sets the "outputChannel" value in the TxJoin thread's data structure, thrTxJoin. Finally, it calls the thrAudioprocSetOutputVolume() function (in thrAudioproc.c) once for each channel.

7.2.4.17 appThreads.c

The purpose of appThreads.c is to define all the global variables used across all threads—the intermediate buffer Sample bufAudioproc[FRAMELEN] in our case—and to implement the thrInit() function that initializes all threads.

The thrInit() function first calls ALGRF_setup() to designate internal and external heaps. When XDAIS algorithms ask for memory, they can ask for different types of memories, which in practice means either internal or external. Our internal heap is created in an internal memory segment and has an identifier of INTERNALHEAP. The external heap has an identifier of EXTERNALHEAP. If we had only internal memory in our system, we would call ALGRF_setup(INTERNALHEAP, INTERNALHEAP), thus forcing the "external" sections into internal memory as well.

The procedure then simply calls the init() function for all the threads, and returns. Note that we do not return information about whether initialization was successful—it *has* to be successful in a system like ours. But we are informed about any failures via the UTL_assert() call, part of the debugging module, which halts the application if the initialization fails.

7.2.4.18 link.cmd

The linker command file link.cmd governs the linking process. As its first step, it includes the automatically generated appcfg.cmd file, which places different sections of the executable in their appropriate memory regions, as defined by the MEM object in the DSP/BIOS configuration. The appcfg.cmd file also generates various other DSP/BIOS linking information.

In our application, the link.cmd file includes all the libraries we use: the libraries for the IOM mini-driver and for the PIO, UTL, ALGRF, FIR_TI, and VOL_TI modules and algorithms. It also assigns vendor-dependent algorithm function table symbols to generic function table symbols.

7.2.4.19 app.gel

This GEL source file is very simple. It displays three sliders, one that ranges only from 0 to 1 with step 1 and is used as a selector between the two channels, and two volume sliders, one for each channel. Every time the user moves a slider, appropriate GEL procedures are called and they write the new values in the target's deviceControlsIOArea array. When writing, CCStudio temporarily halts the target and stores the value, and then resumes. This pause causes slight disruption in the output sound.

Using GEL scripts to simulate control is a simple and effective way to test the application in the early stages of development.

8 Adapting the RF3 Application

In this section we first discuss some general considerations when building your own application on top of RF3. Much of the "theoretical" background for such design has been laid out in previous sections, so we focus mainly on situations that are not covered by the RF3 code, but which can be supported by our design approach.

After that we present two step-by-step examples of modifying the RF3 application. In the first example we remove one channel while retaining FIR and VOL algorithms. In the second, we replace those algorithms with G.726 encoder and decoder algorithms that come with CCStudio.

Note that these adaptations are exercises—they do not apply to all systems. For example, if you are using a board with a stereo codec, you typically would not remove the second channel.

8.1 General Considerations

Reference Frameworks are designed so that, as much as possible, you do not need to modify the main source and linker files to change algorithms and channels. Most such changes are made to supporting files and command files included by the main linker command file.

Reference Frameworks can also be adapted for use in many applications, including telecommunication, audio, video, and more. Such changes are not detailed in this section, but are typically made through changes to the main source file.

To create a custom application from RF3, you may need a new I/O driver for your hardware, your data path may be different, as may the application control. Since both the hardware and the algorithms used differ from those in RF3, some application- and hardware-specific I/O will need to be incorporated into your data processing threads.

8.1.1 I/O Driver

Since, in general, your application is concerned with processing data *streams*, the IOM driver model is suitable for if the data is synchronous or isochronous (defined later) and can be processed in blocks.

The IOM driver model assumes that it always gets a full buffer of data to transmit to the output, or that it always transmits a full buffer of data from the input. The PIO module is built on the same assumption, since it delivers data to the IOM mini-driver via pipes, which hold data frames of fixed size.

As we mentioned earlier, to port the driver to a different platform you need to write a different initialization routine for different codecs, and update the McBSP/DMA code to match your specific DSP. For details on writing IOM mini-drivers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616) application note for a full driver discussion and porting guide.

Regarding data synchronicity, in the simple case of synchronous data as with the RF3 application, a double-buffered pipe connected to the PIO object is sufficient. In the case of isochronous data, which does not arrive synchronously but a certain amount of data is guaranteed to arrive within a certain time, you only need to make the PIO-connected pipe multi-buffered, or multi-framed (the number of frames depending on the specified average arrival rate), without having to introduce changes in the design.

8.1.2 Data Path

Section 6, *RF3 Design Approach*, page 28 provided general principles for designing data paths in terms of DSP/BIOS objects. These principles apply to any RF3 application.

As far as thread activity goes, the mechanism of SWI mailboxes with bits cleared by pipes' notifyReader() and/or notifyWriter() functions should be sufficient in most cases. However, you always have the option of implementing more complex logic by making notify() functions execute your own code to decide when to post a certain SWI thread.

The mailbox mechanism works well even in the cases where the flow of data is *output-driven*. In the RF3 application the flow of data is input-driven: The codec pumps incoming data at the fixed rate, and as long as the real-time deadlines are met, the DMA delivers a new frame every exactly 10 ms, which causes the rest of the system to engage in work.

There are applications in which this is not the case. Consider an example of a simple portable MP3 player, which reads compressed data from a FLASH memory (possibly compressed at a variable rate), decompresses it, applies some equalization and volume control to it, and sends it to the output:

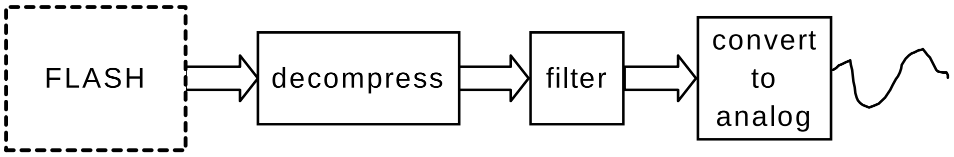


Figure 33. MP3 Player Data Path

The "convert to analog" block can clearly be realized through the PIO module, since the output must be produced at a fixed rate. But the "FLASH" is a storage, passive unit, so it cannot drive the input. The decompression and filtering blocks conceivably could be posted at a fixed rate by a CLK module and somehow synchronized with the PIO module, but there is a simpler way: Let the pipes' notify() functions post the threads as the demand for a new frame flows from the back of the path to the front of the path:

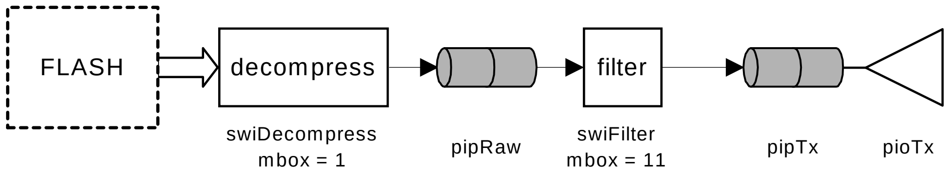


Figure 34. MP3 Player Data Path with DSP/BIOS Objects

The threads have mailboxes with as many non-zero lower bits as they have pipes, and the pipes clear the corresponding mailbox bits of their reader/writer. When pioTx reads and frees one frame from pipTx, the pipe demands a new frame. Its notifyWriter() function runs and clears one bit from swiFilter's mailbox. If a new decompressed frame is waiting in pipRaw, swiFilter is posted. When it runs, it reads and frees a frame from pipRaw, and the pipe demands a new frame. Its notifyWriter() function clears the only bit of the swiDecompress thread's mailbox, and it is posted. So the demand for a new frame travels from the pioTx module to the decompression thread.

8.1.3 Application Debugging

While every application calls for different and unpredictable debugging adventures during its development, there is some generally applicable advice when building this kind of application:

- Most important is to use the UTL_assert(<condition>) macro to test for all conditions that could possibly fail. Use of other UTL functions is encouraged as well.
- If instantiation of an XDAIS algorithm fails, you can step into ALGRF_create() and see exactly why that happens—whether there is insufficient memory to fulfill algorithm requests or the algorithm itself fails. By observing algorithm's memTab[] records within ALGRF_create() you can see if memory allocation is a problem.
- If the application produces no output, it is best to set breakpoints at all application procedures that can be called by DSP/BIOS: SWI/PRD/CLK threads' functions, pipes' notify functions (if they do not simply call SWI_*() functions), and the ISRs. It is especially useful to set breakpoints at the mini-driver's ISR function(s) and the PIO's prime functions (PIO_rxPrime, PIO_txPrime). By observing execution flow, you can verify that the sequence of events in the application is correct.
- Subtle real-time deadline misses can be detected using the UTL_sts* functions.

8.2 Porting the Configuration

As described in Section 5.1.1, DSP/BIOS Configuration, page 19, there are now two methods of configuring DSP/BIOS applications. The traditional method, graphical configuration requires that you create a new configuration file when porting to a different target. The newer method, textual configuration allows you to edit and run scripts that define the configuration. Information about porting via both of these configuration methods is provided in the subsections that follow.

8.2.1 Textual Configuration Scripts

Figure 35 shows which folders contain the configuration scripts for RF3.

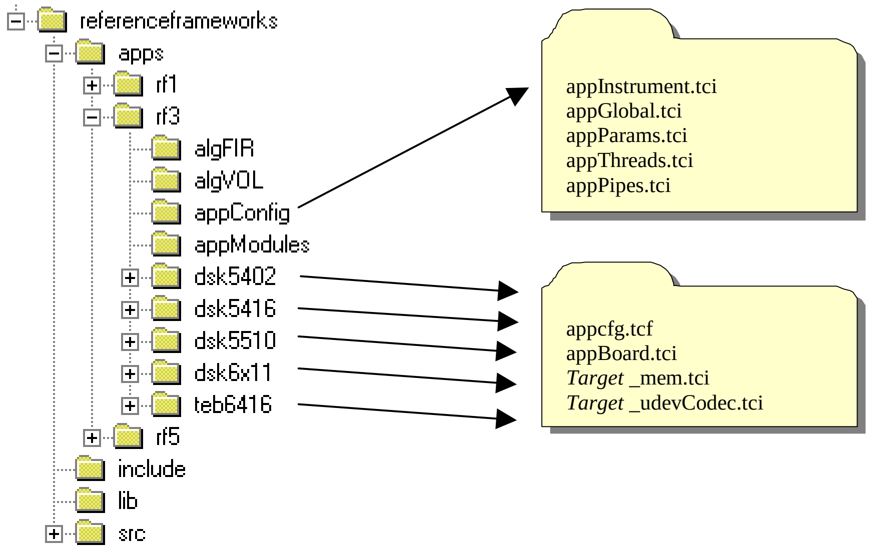


Figure 35. Folders Containing Configuration Scripts

- ***RF_DIR*\apps\rf3\appConfig.** Contains application-specific sub-scripts for use with any target. These files statically create and configure instrumentation objects, global properties, application-specific parameters, application thread object(s), and PIP objects. Typically, these scripts need no changes when porting to a new board. The following sample shows the sub-script that creates and configures DSP/BIOS LOG objects:

```
/* ======== appInstrument.tci ======== */

/* Log objects */

/* create a LOG object logTrace of size 128 */
logTrace          = LOG.create( "logTrace" );
logTrace.comment  = "default LOG object for application messages";
logTrace.bufLen   = 128;

/* change the size of the LOG_system object to a value large enough */
LOG_system.bufLen = 128;
```

- ***RF_DIR*\apps\rf3\target.** Contains target-specific scripts. The appcfg.tcf file is the top-level script file; it imports target-specific settings in appBoard.tci and the sub-scripts in *RF_DIR*\apps\rf3\appConfig. Other .tci files in this folder create memory sections, create UDEV driver objects, and set hardware-specific parameters such as the data word size and memory model. The following example shows the first part of appBoard.tci, which loads the appropriate platform and selects the memory model:

```
/* ======== appBoard.tci ========
 *  Platform-specific portion of the configuration database script
 */

/* Load the Platform file for the appropriate board */
utils.loadPlatform("Dsk5510");

/*  This file makes all the default DSP/BIOS objects known,
 *  plus it defines shortcut javascript variable objects for easy
 *  access to those objects; for example, instead of
 *  prog.module("LOG").instance("LOG_system").buflen = <some value>;
 *  we can simply say
 *  LOG_system.buflen = <value>;
 */
utils.getProgObjs(prog);

/* Call model: use "large" instead of "small" */
GBL.MEMORYMODEL = "LARGE";
```

To port the configuration to a different target, first make any necessary modifications to the .tci files in your *RF_DIR*\apps\rf3\target folder. Then, you can generate a new configuration database file (app.cdb) that matches all your edits by running the makeConfig.bat file. This file is provided for each target in the *RF_DIR*\apps\rf3\target folder. In order to run this file, you must first run the dosrun.bat file in the CCStudio root directory to make the necessary environment settings. The commands to run these files are as follows:

```
cd c:\referenceframeworks\apps\rf3\target
c:\ti\dosrun
makeconfig
```

8.2.2 Graphical Configuration

Except for board-specific settings such as memory section names and placement, the following list shows all additions and changes made to the configuration template. To port RF3 to a different platform, apply the same changes to the configuration template for that target. You may want to make additional target-specific changes to the configuration.

The tables show which fields were changed. If a field is not mentioned, use the default value.

- **SWI (additions).** Software threads used in the application:

SWI Object Name	Function and Arguments	Mailbox (binary)	Priority
swiAudioproc0	thrAudioprocRun(0)	0011	1
swiAudioproc1	thrAudioprocRun(1)	0011	1
swiRxSplit	thrRxSplitRun()	0111	1
swiTxJoin	thrTxJoinRun()	0111	1
swiControl	thrControlRun()	0	1

- **PIP (additions).** Pipes used for transporting blocks of streaming data:

PIP Object	notifyWriter Function and Args	notifyReader Function and Args	Framesize (words)	numFrames
pipRx	PIO_rxPrime(pioRx)	SWI_andnHook(swiRxSplit, 1)	80 ('C5000 mono) 160 (C5000 stereo) 40 ('C6000 mono) 80 ('C6000 stereo)	2
pipRx0	SWI_andnHook(swiRxSplit, 2)	SWI_andnHook(swiAudioproc0, 1)	80 ('C5000) 40 ('C6000)	1
pipRx1	SWI_andnHook(swiRxSplit, 4)	SWI_andnHook(swiAudioproc1, 1)	80 ('C5000) 40 ('C6000)	1
pipTx0	SWI_andnHook(swiAudioproc0, 2)	SWI_andnHook(swiTxJoin , 2)	80 ('C5000) 40 ('C6000)	1
pipTx1	SWI_andnHook(swiAudioproc1, 2)	SWI_andnHook(swiTxJoin, 4)	80 ('C5000) 40 ('C6000)	1
pipTx	SWI_andnHook(swiTxJoin, 1)	PIO_txPrime(pioTx)	80 ('C5000 mono) 160 (C5000 stereo) 40 ('C6000 mono) 80 ('C6000 stereo)	2

Framesizes for 'C6000 are half that of framesizes for 'C5000 because PIP frame sizes are expressed in words (Int), which are 16 bits on 'C5000 and 32 bits on 'C6000. See the comments in appResources.h for details and conversion macros.

- **CLK (addition).** Time-based activation of the control ISR and the control thread:

CLK Object Name	Function and Arguments
clkControl	thrControlIsr()

- **Memory layout (modifications).**

MEM Object	Heap Identifier
Internal Memory (name varies by target)	INTERNALHEAP
External Memory (name varies by target)	EXTERNALHEAP

By default, heaps are of size 0x400. You may want to increase these sizes. Create any other MEM sections and decide about section placement as necessary for your application. You can pick the optimal value for the heap by using the UTL_showHeapUsage() macros for all segments when the necessary objects are created. Then you can reduce heap sizes to match the reported values.

- **System (modifications).**

Module Name	Modification
TSK	use task manager = false
SYS	putc function = FXN_F_nop

- **LOG (modifications).** Objects used for printing user and system messages:

CLK Object Name	Size
logTrace	128
LOG_system	128

- **STS (additions).** Time measurement objects:

STS Object Name	Type
stsTime0	"High resolution time based"
stsTime1	"High resolution time based"
…	…
stsTime9	"High resolution time based"

Remember that symbol names for functions and variables entered in DSP/BIOS Configuration Tool fields must be preceded by an underscore (because they are C procedures called from assembly, and C global symbols when translated have underscores prefixing their names) . For example, for the pipRx pipe, the notifyReader function, shown in the previous list as SWI_andnHook(swiRxSplit, 1) must be entered as:

- notifyReader: _SWI_andnHook
- nwarg0: _swiRxSplit
- nwarg1: 1

Finally, to disable the task manager, simply uncheck the "Enable TSK Manager" field in the "properties" dialog for the TSK module.

8.3 Porting Build Options

If you are porting to a different board, don't forget to adjust the Board Options in CCStudio to identify whether the codec is mono or stereo.

If the codec is mono, APPMONOCODEC should be defined in the Compiler Settings tab as -d"APPMONOCODEC". You may need to scroll to the right to see this definition.

If the codec is stereo, remove any definition of APPMONOCODEC from the Compiler Settings.

8.4 Removing the Second Channel

In this exercise, we completely remove the second channel (channel #1, or "right" channel) and the two splitting/joining threads, leaving the design with one thread that reads the single input pipe, processes it (applies FIR and VOL), and writes the result in the single output pipe.

NOTE: This adaptation is provided as an exercise—it does not apply to all systems. For example, if you are using a board with a stereo codec, you typically would not want to remove the second channel.

If, instead of removing a channel, you want to add an additional channel, you would add additional copies of the objects being removed, increase the number of bits in the SWI object mailboxes, and similarly reverse the effects of the steps listed in this section.

Let us look at the simplified data path:

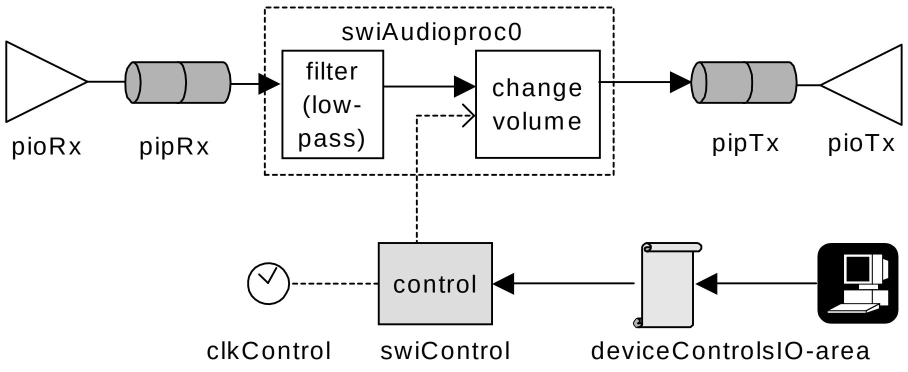


Figure 36. Data Path with Single Channel Adaptation

The phases of this adaptation are as follows:

- Creating the Application Folder (Section 8.4.1, page 68)
- Modifying the Configuration File (Section 8.4.2, page 69)
- Modifying the Code (Section 8.4.3, page 70)

8.4.1 Creating the Application Folder

We recommend that you copy the application folder so the unmodified application remains available for reference.

1. In the apps\ folder, duplicate the entire rf3\ folder tree and name the copy "rf3_single".

   In file paths that follow, *RF_DIR* is your "referenceframeworks" folder at the top of the Reference Frameworks folder tree. Other file paths are relative to the location of the folder you create in this step (for example, *RF_DIR*\apps\rf3_single).

2. In CCStudio, open the project file, *RF_DIR*\apps\rf3_single\target\app.pjt (for example, *RF_DIR*\apps\rf3_single\dsk5402\app.pjt).

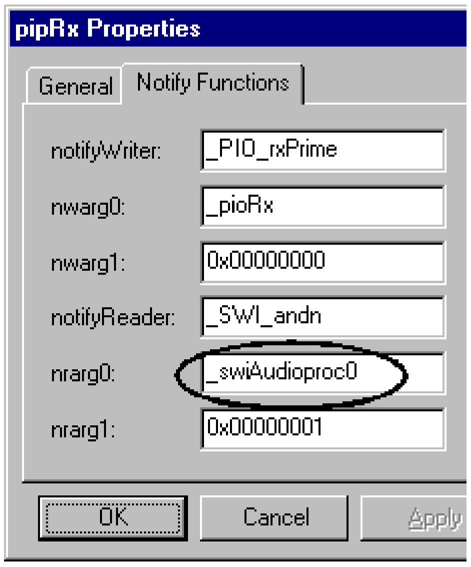
   The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93.

8.4.2 Modifying the Configuration File

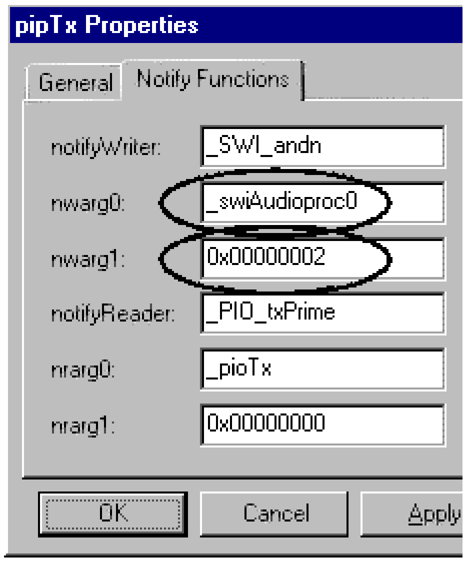
Follow these steps to modify the configuration to get the data path shown in Figure 36:

1. Open the app.cdb configuration file in the "DSP/BIOS Config" folder of the Project View.
2. In the Scheduling→SWI folder, delete the **swiAudioproc1**, **swiRxSplit**, and **swiTxJoin** objects. These objects represent the second channel, the input splitting thread and the output joining thread that we no longer have.
3. In the Input/Output→PIP folder, delete the **pipRx0**, **pipRx1**, **pipTx0**, and **pipTx1** pipes.

   The pipRx0,1 pipes were used by the RxSplit thread to deliver data from pipe pipRx to the two Audioproc threads. The pipTx0,1 pipes were used by the TxJoin threads to read data from the two Audioproc threads and store it to the pipTx thread. Now Audioproc0, the only data processing thread, reads and writes data directly to/from the pipRx and pipTx objects that the PIO objects use.
4. Right-click on pipe pipRx and choose Properties. Change the notifyReader function's argument nrarg0 to **_swiAudioproc0** (see Figure 37a). This causes the notifyReader function to clear the first bit from swiAudioproc0's mailbox by calling SWI_andnHook(swiAudioproc, 1).

a.


notifyWriter: _PIO_rxPrime
nwarg0: _pioRx
nwarg1: 0x00000000
notifyReader: _SWI_andn
nrarg0: _swiAudioproc0
nrarg1: 0x00000001

b.


notifyWriter: _SWI_andn
nwarg0: _swiAudioproc0
nwarg1: 0x00000002
notifyReader: _PIO_txPrime
nrarg0: _pioTx
nrarg1: 0x00000000

Figure 37. Notification Properties for pipRx and pipTx

5. Right-click on pipe pipTx and choose Properties. Change the notifyWriter function's argument nwarg0 to **_swiAudioproc0** (see Figure 37b). Also change the second

argument, nwarg1, to **2** (binary 010) to clear the second bit when this pipe is ready. Now its notifyWriter function calls SWI_andnHook(swiAudioproc, 2).

The pipRx and pipTx pipes should retain the same notifyWriter and notifyReader functions respectively. Their communication with the PIO module is unchanged.

6. Save the modified app.cdb file.

8.4.3 Modifying the Code

To modify the code related to the second channel, follow these steps:

1. Remove the thrRxSplit.c and thrTxJoin.c files from the project. We no longer need them, and we do not want dead code in our executable.

2. Open the appModules\appBiosObjects.h file, and remove the external declarations of the PIP objects you removed:

```
~~extern PIP_Obj pipRx0;          /* receive pipe  - channel 0 */~~
~~extern PIP_Obj pipTx0;          /* transmit pipe - channel 0 */~~
~~extern PIP_Obj pipRx1;          /* receive pipe  - channel 1 */~~
~~extern PIP_Obj pipTx1;          /* transmit pipe - channel 1 */~~
```

3. Open the appModules\appResources.h file, and change the number of channels to **1** by changing the definition for NUMCHANNELS macro. Save the changes.

4. Open thrAudioproc.c file for editing.

   – In the channel #0 portion of the thrAudioproc[NUMCHANNELS] array declaration, replace &pipRx0 with **&pipRx** and &pipTx0 with **&pipTx** . These are now the addresses of the input and output pipes.

```
/* input pipe(s) */
&pipRx~~0~~,                /* pipIn */

/* output pipe(s) */
&pipTx~~0~~,                /* pipOut */
```

   – Remove the entire channel #1 array element from the thrAudioproc[NUMCHANNELS] array declaration, so that the compiler does not complain that we are trying to initialize more elements than we have in the array.

```
~~{ /* channel #1 */~~
~~   ...~~
~~   ...~~
~~   ...~~
~~}, /* end channel # 1 */~~
```

   – Remove one of the two sets of filter coefficients enclosed in the curly braces from the filterCoefficients[NUMCHANNELS][32] array initialization. You can remove either set. For a high-pass filter, remove the first set of coefficients.

– Since we removed the TxJoin thread, which cleared the last bit of each output sample for mono codecs, we have to clear those bits in thrAudioprocRun(). Therefore, add the following line to the declarations at the beginning of the thrAudioprocRun() function:

```
Int i;
```

And, add the following statements after the call to VOL_apply():

```
/* For mono codecs, clear last bit in each sample from destination buffer */
#ifdef APPMONOCODEC
for (i = 0; i < size; i++) {
    dst[i] = dst[i] & 0xfffe;
}
#endif
```

Note that, unlike the corresponding statement in thrTxJoin.c, this code does not move the sample from the source buffer to the destination buffer; the sample has already been moved to the destination buffer by the VOL_apply() function.

The remainder of the file is OK; it uses the NUMCHANNELS constant to instantiate algorithms, and the channel index (always zero) to access the thread data structure.

5. Open the thrControl.c file for editing.

– Remove the line that includes thrTxJoin.h.

~~#include "thrTxJoin.h"   /* definition of thrTxJoin thread data */~~

– Edit the initialization array for "deviceControlsIOArea". Remove the second "100" default volume value as we now have only one channel.

```
Int deviceControlsIOArea[] = {
    FALSE,   /* initially, no user action */
    0,       /* default active channel */
    100,     /* default volume for channel #0 */
~~    100,     /* default volume for channel #1 */~~
};
```

– In the thrControlRun() function, remove the statements that declare and set the outputChannel variable. It is not needed since there is only one output channel.

```
/* a local copy of all thrControlIsr-writable variables: */
~~Int outputChannel;~~
Int outputVolume[ NUMCHANNELS ];

/*
 *  Disable the control ISR, make a local copy of all the variables
 *  that the conrol ISR can write to, and restore interrupts. After
 *  that we will use local copies only; that is to prevent the ISR
 *  partially overwriting our current set of values (i.e. to prevent
 *  race conditions).
 */
interruptState = HWI_disable();
~~outputChannel = thrControl.outputChannel;~~
for (i = 0; i < NUMCHANNELS; i++) {
    outputVolume[i] = thrControl.outputVolume[i];
}
```

- In the thrControlRun() function, remove the statement that tells the thrTxJoin thread which channel to output, since we no longer have the TxJoin thread.

```
~~/* Instruct thrTxJoin thread which channel to output */~~
~~thrTxJoinSetOutputChannel( outputChannel );~~
```

6. Open the appThreads.c file for editing:
   - Remove the lines that include thrRxSplit.h and thrTxJoin.h files.

```
~~#include "thrRxSplit.h"      /* definition of the multiplexing thread  */~~
~~#include "thrTxJoin.h"       /* definition of the demultiplexing thread */~~
```

   - Remove the lines that call the thrRxSplitInit() and thrTxJoinInit() functions from the thrInit() procedure.

```
~~thrRxSplitInit();       /* RxSplit thread   */~~
thrAudioprocInit();     /* Audioproc thread */
~~thrTxJoinInit();        /* TxJoin thread    */~~
thrControlInit();       /* Control thread   */
```

7. Save all the source files you edited. Also, save the project.
8. Rebuild the application.
9. Load the executable and run it to verify the changes were correct. Load the app.gel file and verify that you can change the volume of channel #0 only. Switching the channel selection control should have no effect on the volume or filtering.
10. Optionally, you may remove thrRxsplit.* and thrTxJoin.* files from the appModules folder.

If you have multiple applications with only one channel, you may want to save a backup copy of this design, as you can use it as the base for other applications.

In this section, you adapted RF3 by removing one of the channels. If, instead of removing a channel, you want to add an additional channel, you would add additional copies of the objects being removed, increase the number of bits in the SWI object mailboxes, and similarly reverse the effects of the steps listed in this section.

8.5 Switching to Other Algorithms

In this exercise we use the single-channel application created in the previous exercise, and replace the two algorithms supplied in RF3 with two voice decoding/encoding XDAIS algorithms. The hardware platform remains the same and so does the I/O driver. Similar steps apply to any other XDAIS algorithms you use in your applications.

NOTE: The G.726 encoder and decoder used in this example *are not optimized algorithms and should not be used in an actual product application*. These "performance-detuned" algorithms are provided in CCStudio as examples for TMS320 DSP Algorithm Standard (XDAIS) developers and users.

The G.726 algorithms used in this example are provided only for certain boards. If they are not available for your platform, adapt the steps as necessary to integrate the eXpressDSP-compliant algorithms you plan to use in your application. For example, for the 'C6x11 DSK platform, G.723 algorithms are provided instead of G.726 algorithms.

Our new data path is shown in Figure 38.

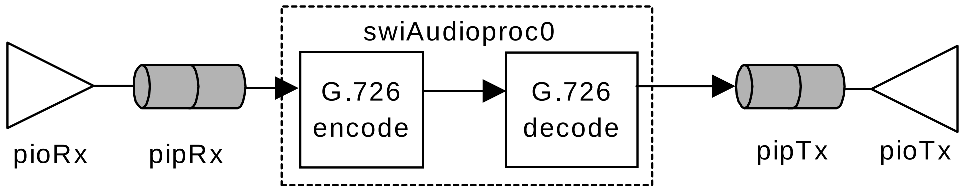


Figure 38. Data Path with Single Channel, No Control Thread, and G.726 Algorithms

Since we simply encode and immediately decode the audio data, the application makes little business sense, but it illustrates the steps to follow to plug in a new XDAIS algorithm. (It can also show how voice quality is degraded at high compression ratios.) Although it is not shown in the picture, the thread still uses an intermediate buffer to store the results of data encoding before using it as input for the decoding block.

We chose to remove the control thread from the design for simplicity, otherwise we would have to adapt it to modify G.726 encoder/decoder parameters. All the other parameters of the application remain—hardware platform, driver, section placement etc. We also assume that we built rf3_single application correctly in the previous exercise.

The phases of this adaptation are as follows:

- Creating the Application Folder (Section 8.5.1, page 74)
- Removing the Control Thread (Section 8.5.2, page 74)
- Placing Libraries and Header Files in the Folders (Section 8.5.3, page 75)
- Creating Algorithm Wrappers (Section 8.5.4, page 75)
- Modifying Code to Use the New Algorithms (Section 8.5.5, page 77)
- Modifying the Project (Section 8.5.6, page 79)

8.5.1 Creating the Application Folder

We recommend that you copy the application folder so the unmodified application remains available for reference.

1. In the apps\ folder, duplicate the entire rf3_single\ folder tree and name the copy "rf3_vocode".

   In file paths that follow, *TI_DIR* is c:\ti if you installed CCStudio in the default location, and *RF_DIR* is your "referenceframeworks" folder at the top of the Reference Frameworks folder tree. Other file paths are relative to the location of the folder you create in this step (for example, *RF_DIR*\apps\rf3_vocode).

2. In CCStudio, open the project file, *RF_DIR*\apps\rf3_vocode\target\app.pjt (for example, *RF_DIR*\apps\rf3_vocode\dsk5402\app.pjt).

   The targets supported as of the publication date are listed in *Appendix E: Reference Framework Board Ports*, page 93.

NOTE: The instructions provided for this adaptation assume that you are starting with the single-channel RF3 application created in the previous section.

8.5.2 Removing the Control Thread

In this exercise, we remove the control thread from the design for simplicity. If you wish to keep the control thread, adapt it to modify the parameters for your algorithms. To remove the objects and code related to the control thread, follow these steps:

1. Open the app.cdb configuration file in the "DSP/BIOS Config" folder of the Project View.
2. In the Scheduling→SWI folder, delete the SWI object **swiControl**.
3. In the Scheduling→CLK folder, delete the CLK object **clkControl**.
4. Save the modified app.cdb file.
5. Remove the thrControl.c file from the project. You may optionally remove the thrControl.* files from the appModules\ folder as well.
6. Open the appModules\appThreads.c file for editing:
   – Remove the #include "thrControl.h" line.

```
#include "appResources.h"   /* application-wide common info */
#include "appThreads.h"     /* thread-wide common info */
#include "thrAudioproc.h"   /* definition of thread Audioproc */
~~#include "thrControl.h"    /* definition of the control thread */~~
```

   – Remove the line that calls the thrControlInit() function from the thrInit() procedure.

```
thrAudioprocInit();     /* Audioproc thread */
~~thrControlInit();       /* Control thread   */~~
```

7. Open the appModules\appBiosObjects.h file, and remove the external declaration of the SWI object you removed:

```
~~/* SWI objects: declare those that are or might be posted manually */~~
~~extern SWI_Obj swiControl;      /* control thread */~~
```

8.5.3 Placing Libraries and Header Files in the Folders

As with any XDAIS algorithm, there are four parts to the G.726 encoder and decoder: libraries, header files, function wrappers, and default parameters. To simplify the paths to these files, we place the libraries in the common library folder, the header files in the common include folder, and the wrappers and parameters in local folders.

In the following steps, *TI_DIR* is the CCStudio install folder (normally c:\ti).

The files for the G.726 encoder and decoder example algorithms are located in the *TI_DIR*\examples*target*\xdais\demo\lib and *TI_DIR*\c*XXXX*\xdais\src\vocoders folders. If you are adapting RF3 to use third-party algorithms, see the documentation provided by your vendor(s) for file locations.

1. Copy the library files (where "*XX*" is the code for your target).

   Copy from: *TI_DIR*\examples*target*\xdais\demo\lib
   Copy to: *RF_DIR*\lib

g726enc_ti.l*XX*	(for example, g726enc_ti.l54f)
g726dec_ti.l*XX*	(for example, g726dec_ti.l54f)

2. Copy the generic G.726 interface header files. These files are independent of the actual vendor's G.726 algorithm implementation.

   Copy from: *TI_DIR*\c*XXXX*\xdais\src\vocoders
   Copy to: *RF_DIR*\include

ig726.h
ig726enc.h
ig726dec.h

See Figure 39 to check the locations of the files you copied.

8.5.4 Creating Algorithm Wrappers

Algorithm vendors typically provide wrapper files. Algorithms wrappers let the application call algorithm functions in a "friendly" way. Most wrappers are implemented as static inline functions in the .h file. Wrappers implemented in code are in the .c file.

To use provided wrapper files with RF3, most need to be modified to use ALGRF for XDAIS instantiation instead of the ALG module used by most default wrappers. If wrapper files have been provided, follow these steps. (Modify the filenames to match your algorithms.)

1. In your apps\rf3_vocode\ folder, create the algG726ENC\ and algG726DEC\ folders. This is where you will place the function wrappers and default parameter files.

2. For the encoder, copy the following source and header files:

   Copy from: *TI_DIR*\c*XXXX*\xdais\src\vocoders
   Copy to: *RF_DIR*\apps\rf3_vocode\algG726ENC

ig726enc.c	(default parameters)
g726enc.h	(header file with function wrappers)
g726enc.c	(.c file with function wrappers)

3. For the decoder, copy the following source and header files:

   Copy from: *TI_DIR*\c*XXXX*\xdais\src\vocoders
   Copy to: *RF_DIR*\apps\rf3_vocode\algG726DEC

ig726dec.c	(default parameters)
g726dec.h	(header file with function wrappers)
g726dec.c	(.c file with function wrappers)

The files you copied to various folders should be as shown in Figure 39:

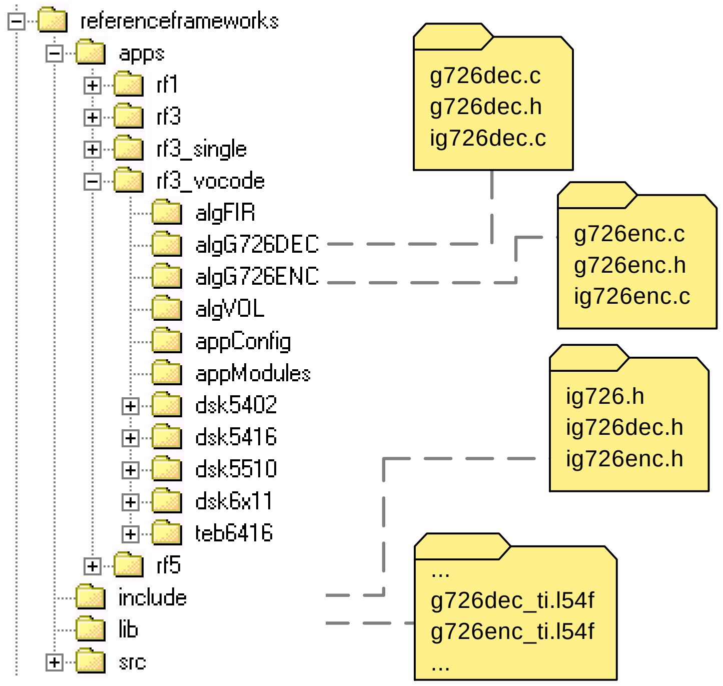


Figure 39. Folder Locations and Contents for Algorithm Files

4. Edit the following wrapper files that you copied:

apps\rf3_vocode\algG726ENC\g726enc.h
apps\rf3_vocode\algG726ENC\g726enc.c
apps\rf3_vocode\algG726DEC\g726dec.h
apps\rf3_vocode\algG726DEC\g726dec.c

In each file, make the following case-sensitive global find-and-replace changes. To perform case-sensitive replaces in CCStudio, choose Edit→Find/Replace and click Properties and put a checkmark in the Match case box.

Find	Replace	Notes
<alg.h>	<algrf.h>	
ALG_create	ALGRF_create	
ALG_delete	ALGRF_delete	
ALG_activate	ALGRF_activate	
ALG_deactivate	ALGRF_deactivate	
ALG_Handle	ALGRF_Handle	Be careful not to replace **IALG_Handle**, only **ALG_Handle**.

The wrapper functions are made for use with the ALG module. We are adjusting them for use with the ALGRF module.

If you do not have wrapper files, do the following (where *myalg* is the lowercase name of your algorithm and *MYALG* is the uppercase name):

1. In your apps\rf3_vocode\ folder, create a folder called alg*MYALG*\.
2. Copy algVOL\vol.h and algVOL\vol.c to alg*MYALG*\ and rename them *myalg*.h and *myalg*.c.
3. Edit *myalg*.h, *myalg*.c and replace every instance of "vol" with "*myalg*" and every instance of "VOL" with "*MYALG*".

8.5.5 Modifying Code to Use the New Algorithms

All references in the application code to the FIR and VOL algorithms are located in the Audioproc thread's files, so the only files that need to be changed are thrAudioproc.h and thrAudioproc.c.

NOTE: In this example, you replace references to FIR with G726ENC and references to VOL with G726DEC. However, there is no relation between FIR and the encoder nor between VOL and the decoder. The replacement is such only because FIR is first and VOL is second.

1. Open thrAudioproc.h for editing. Make the following case-sensitive global changes:

Find	Replace	Notes
FIR	G726ENC	
fir	g726enc	
VOL	G726DEC	
vol	g726dec	Be careful not to replace **Int volume;**

These changes include modifications to the #include statements, the algorithm handle declarations, and the algorithm function structure declarations.

2. Remove the thrAudioprocSetOutputVolume() function declaration from thrAudioproc.h.

```
~~/* parameter change function, called by the control thread */~~
~~extern Void thrAudioprocSetOutputVolume( Int chan, Int volume );~~
```

3. Open thrAudioproc.c for editing. Make the following case-sensitive global changes:

Find	Replace	Notes
FIR	G726ENC	
fir	g726enc	
VOL	G726DEC	
vol	g726dec	Be careful not to replace instances of the **volume** variable.

These changes include modifications to a number of statements and declarations.

4. In thrAudioproc.c, remove the static Sample filterCoefficients array declaration and initialization completely (it is used only for the FIR filter).

```
~~static Sample filterCoefficients[ NUMCHANNELS ][ 32 ] = {~~
~~    /* 8kHz, 32 Taps, 800 Hz,1300Hz, pass ripple 0.897dB, stop atten 51 dB */~~
~~    {~~
~~        0x0015,0x0043,0x0085,0x00AD,0x006C,0xFF69,0xFD7D,0xFAEC,~~
~~        0xF88C,0xF79B,0xF957,0xFE63,0x0654,0x0F94,0x17C7,0x1C9A,~~
~~        0x1C9A,0x17C7,0x0F94,0x0654,0xFE63,0xF957,0xF79B,0xF88C,~~
~~        0xFAEC,0xFD7D,0xFF69,0x006C,0x00AD,0x0085,0x0043,0x0015,~~
~~    },~~
~~};~~
```

5. In thrAudioproc.c, change the default parameters for the encoder. Remove the lines that are crossed out below. And, add the line shown in bold.

NOTE: Don't remove the `g726encParams = G726ENC_PARAMS;` statement; it causes the algorithm to use default values for the parameters not specified here. (IG726_ALAW is not supported by the simple G.726 encoder/decoder in this case.)

```
/* Set the parameters structure to the default, i.e.
 *  the one used in i<alg>.c, and modify fields that are different.
 */
g726encParams = G726ENC_PARAMS;              /* default parameters */
~~g726encParams.coeffPtr    =                  /* filter coefficients */~~
~~  (Short *)filterCoefficients[i];~~
~~g726encParams.filterLen   =                  /* filter size */~~
~~  sizeof( filterCoefficients[i] ) / sizeof( Sample );~~
g726encParams.frameLen    = FRAMELEN;        /* frame size in samples */
**g726encParams.mode        = IG726_LINEAR;**
```

6. In thrAudioproc.c, change the default parameters for the decoder: Remove the lines that are crossed out below. And, add the lines shown in bold.

NOTE: Don't remove the `g726decParams = G726DEC_PARAMS;` statement.

```
/* do the same for the G726DEC algorithm: create parameters structure */
g726decParams = G726DEC_PARAMS;              /* default parameters */
~~g726decParams.frameSize   = FRAMELEN;        /* size in samples */~~
~~g726decParams.gainPercentage = 100;          /* default gain */~~
**g726decParams.frameLen = FRAMELEN;**
**g726decParams.mode     = IG726_LINEAR;**
```

7. In the G726ENC_apply() call, cast (XDAS_Int16 *) before the second parameter (src), and cast (XDAS_Int8 *) before the third parameter. We could have done the cast in the wrapper functions instead to make the thread's run() function look neater.

```
G726ENC_apply( thrAudioproc[chan].algG726ENC,
               (XDAS_Int16 *)src, (XDAS_Int8 *)thrAudioproc[chan].bufInterm );
```

8. In the G726DEC_apply() call, cast (XDAS_Int8 *) before the second parameter, and cast (XDAS_Int16 *) before the third parameter (dst).

```
G726DEC_apply( thrAudioproc[chan].algG726DEC,
           (XDAS_Int8 *)thrAudioproc[chan].bufInterm, (XDAS_Int16 *)dst );
```

9. Remove the entire thrAudioprocSetOutputVolume() function from thrAudioproc.c.
10. Modify the comments in the code as needed to describe your algorithms.
11. Save all the source files you edited.

8.5.6 Modifying the Project

1. Remove fir.c, ifir.c, vol.c, and ivol.c from the project.
2. Add the default parameter files and the algorithm wrapper files to the project:

algG726ENC\ig726enc.c
algG726ENC\g726enc.c
algG726DEC\ig726dec.c
algG726DEC\g726dec.c

3. In the Project View, open the link.cmd file for editing. Make the following changes:

Find	Replace
-lfir_ti.l*XX* (where *XX* is the two-digit platform)	-lg726enc_ti.l*XX*
_FIR_IFIR = _FIR_TI_IFIR;	_G726ENC_IG726ENC = _G726ENC_TI_IG726ENC;
-lvol_ti.l*XX*	-lg726dec_ti.l*XX*
_VOL_IVOL = _VOL_TI_IVOL;	_G726DEC_IG726DEC = _G726DEC_TI_IG726DEC;

4. In the Build Options for the app.pjt project, select the Preprocessor category. In the Include Search Path field, make the following changes:

Find	Replace
..\algFIR	..\algG726ENC
..\algVOL	..\algG726DEC

5. Save all the files you edited and the project. Then rebuild the application. Verify that it is correct by loading it and running it on the target.

You should hear audio with slight degradation compared to the simple FIR filter and VOL processing. This degradation is normal given the compression rate of the vocoder. If you do not hear audio at all, check the g726encParams and g726decParams structures in thrAudioproc.c. Missed steps in setting parameters may produce problems with the audio.

In this section, you adapted RF3 to run a G.726 encoder and decoder. You can use this procedure as a template for populating the framework with other algorithms.

9 Performance and Footprint

The performance and footprint for each framework are provided 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.

9.1 Performance Characteristics

The performance characteristics of RF3 are shown in Table 5. The majority of the time is spent processing the algorithm, which is as it should be. The framework imposes very 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 5. RF3 CPU Usage Statistics

Configuration	CPU Load
RF 3 as supplied	11.7%
RF 3 two-channel, minus FIR and VOL algorithm processing	4.7%
RF 3 single-channel, minus FIR and VOL algorithm processing	2.3%

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

The CPU load percentages were calculated using instruction cycle measurements from STS objects. This is a more accurate method than the CPU Load graph available in CCStudio.

Measurements were made under the following conditions:

- Platform: 'C5402 DSK running at 100 MHz
- Sampling rate: 8 kHz
- Samples per frame: 80
- Optimization flags: none
- Debug flags: -g
- All critical code and data in internal memory

9.2 Framework Footprint

RF3 enables the creation of complete applications that fit in a memory footprint of 16 K words. This size allows applications to fit on a TMS320C5402. The actual RF3 footprint sizes are shown in Table 6.

Table 6. RF3 Memory Footprint

	TMS320VC5402	**TMS320VC5510**
RF3 as supplied (program + data)	15.3 K words	31.2 K words
RF3 minus algorithms and application-specific components	11.4 K words	11.8 K words

For example, since a 'C5402 has 16 K words and the RF3 framework uses 11.4 K words, the XDAIS-compliant algorithms and application-specific code and data can consume up to 4.6 K words on that target.

In this calculation, the "application-specific components" include algorithms, buffers and heaps used by the algorithms, and unused stack and heap space. For details about the size of the RF3 footprint, see *Appendix A: RF3 Memory* Footprint, page 83.

10 Conclusion

Reference Framework Level 3 (RF3) is intended to enable designers to create applications that use multiple channels and algorithms while minimizing the memory requirements by using static configuration only. The key design goal for RF3 is ease-of-use. RF3 is designed to enable new DSP designers to create compact products with several algorithms and channels. RF3 provides more flexibility than lower Reference Framework levels. For example, it instantiates any XDAIS algorithm simply and efficiently. However, RF3 is not intended for use in large-scale DSP systems with 10s of algorithms and channels. Instead, RF3 targets medium-complexity systems and makes design decisions such as using static configuration to keep the memory footprint small.

The framework is supplied as highly reusable C language source code to enable switching between 'C5000 and 'C6000 ISAs. It is packaged as a complete application running on several TI DSP platforms.

This application note includes framework selection criteria, a list of typical applications, explanations of how the framework functions, and working examples of key framework adaptations. This enables designers to quickly determine if RF3 fits their needs, and also to rapidly prototype for end-equipment.

RF3 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.

11 References

For additional information, see the following sources:

Product Documentation

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

Application Notes

- *Reference Frameworks for eXpressDSP Software: A White Paper* (SPRA094)
- *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147)
- *Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System* (SPRA791)
- *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795)
- *The TMS320 DSP Algorithm Standard - A White Paper* (SPRA581)
- *Zero Overhead with the TMS320 DSP Algorithm Standard IALG Interface* (SPRA716)
- *DSP/BIOS II Sizing Guidelines for the TMS320C54x DSP* (SPRA692)
- *DSP/BIOS II Timing Benchmarks on the TMS320C54x DSP* (SPRA663)
- *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (SPRA783A)

Web Resources

- www.dspvillage.com

Appendix A: RF3 Memory Footprint

The overall footprint sizes for RF3 are listed in Section 9.2, *Framework Footprint*, page 81. This appendix provides details on those numbers. This breakdown assists the system designer in planning an overall DSP solution.

The numbers in Table 7 were obtained on a 'C5510 DSK using the supplied version of this Reference Framework application and its libraries with no additional optimization. The total size is shown in 16-bit words so that it can easily be compared with the totals shown for other Reference Frameworks in their respective application notes.

Table 7. RF3 Application Footprint

Category	Size in 16-Bit Words (hex and decimal)	Run-time Code Size (for RF modules)
.DARAM$heap	0xc00 (3072)	
.SARAM$heap	0x2000 (8192)	
.sysstack	0x400 (1024)	
.stack	0x400 (1024)	
rts	0x1a4 (420)	
ALGRF	0x161 (353)	0x49 (73)
hwiVec	0x80 (128)	
FIR	0x133 (307)	0xdf (223)
VOL	0xe7 (231)	0x98 (152)
application	0x60e (1550)	
PIO	0x26b (619)	0x146(326)
driver	0x56a (1386)	0x2ca(714)
CSL	0xc14 (3092)	
UTL	0x26e (622)	0x18d(397)
LOG	0x16d (365)	
STS	0xcb (203)	
RTDX	0x806 (2054)	
DSP/BIOS kernel	0x1c8b (7307)	
Total	**0x7ccd (31949)**	

However, components such as the FIR algorithm are not included in your end-application. Therefore, to determine whether your XDAIS algorithms can fit in a certain DSP's on-chip memory, we first need to subtract the size of components to be removed. Table 8 shows how the total size of the basic framework code and data is calculated by subtracting various application-specific portions:

Table 8. Calculating RF3 Basic Framework Size

Total Size	**31,949 16-bit words**
Unused internal heap	minus 2680
Unused external heap	minus 8084
Application buffers	minus 640
Heaps used by algorithms	minus 365
VOL algorithm	minus 231
FIR algorithm	minus 307
FIR coefficient table	minus 64
Unused stack and SysStack	minus 1712
LOG object for debugging	minus 365
STS object for debugging	minus 203
UTL debugging module	minus 622
RTDX module	minus 2054
All cinit records	minus 2528
Resulting Basic Framework Size	**12,094 16-bit words**

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

The result is a total of 12,094 words on the 'C5510, which is the basic footprint for the supplied Reference Framework Level 3.

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

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

This data is provided for the VOL algorithm in *Appendix C: VOL_TI Algorithm Characterization*, page 88.

Appendix B: Comparing RF3 and RF5 Performance

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

Performance Comparison Setup

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

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

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

As part of porting RF3 and RF5 to the 'C6711 DSK with the AIC23 EVM, the following changes were made to both applications to give more realistic benchmark numbers:

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

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

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

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

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

Performance Comparison Test Results

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

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

Table 9. CPU Load Percentages at Various Frame Sizes and Sample Rates

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

With data processing removed, the lines in Figure 40 overlap at different frequencies for RF3 (and similarly for RF5). So running at 44.1 kHz with a frame size of 32 (1378 frames/second) is basically the same as running at 96 kHz with a frame size of 64 (1500 frames/second) from a framework-only perspective.

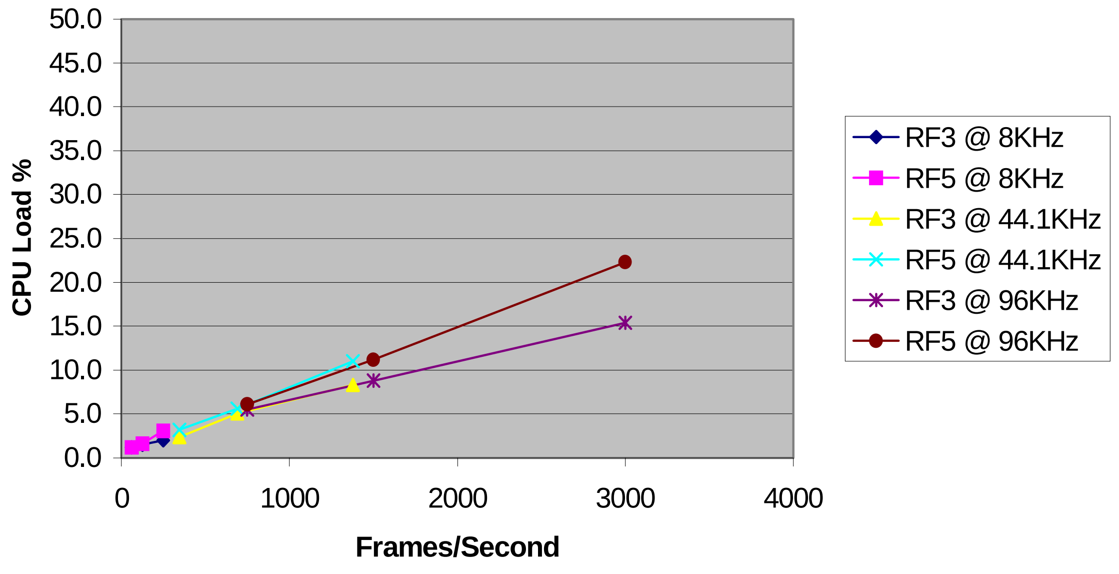


Figure 40. Graph of CPU Load Percentage for Various Settings

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

Performance Comparison Conclusions

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

- RF3 and RF5 frameworks are frame-size independent. The CPU loads are approximately the same when the frames/second are the similar (for example, 44.1 kHz at 64 samples per frame and 96 kHz at 128 samples per frame).
- At lower frequencies (for example, 8 kHz on the C6711 DSK) there is not a significant difference between RF3 and RF5 from a performance standpoint.
- At higher frequencies (for example, 96 kHz on the C6711 DSK), attempts should be made to minimize the number of SWIs and TSKs to avoid context switches and inter-thread communication.
- At higher frequencies (for example, >96 kHz on the C6711 DSK), moving some amount of data processing (e.g. splitting and joining) to DMA is advised.
- The SIO/TSK modules have more overhead than the PIP/SWI modules.
- When collecting CPU Load values, you should use “Disable All” on the RTA Control Panel. This will give a more accurate CPU Load Graph value. On RF5 (framework-only, with a single thread) running at 96 kHz with a frame size of 32, the value was 17.6% with everything enabled on the RTA Control Panel, and 14.3% when everything was disabled.
- A separate test involved placing critical code sections explicitly in internal memory instead of relying on the cache. There was minimal improvement for RF3 and a very slight improvement for RF5. The conclusion is that for these applications, there was little L2 cache thrashing.

Appendix C: VOL_TI Algorithm Characterization

Module	Vendor	Variant	Arch	Mem Model	Version	Library Name
VOL	TI		54	far (f)		vol_ti.l54f

ROMable (Rule 5)

Yes	No
X	

Note: The unit for size is (8–bit) bytes and the unit for align is Minimum Addressable Units (MAUs).

FACTOR8BITBYTES = 2 for 'C54x

FACTOR8BITBYTES = 1 for 'C6x

Persistent / Scratch Data Memory (Rule 19)

memTab	Attribute	Size (bytes)	Align (MAUs)	Space
0	Persist	8 * FACTOR8BITBYTES	0	External
1	Persist	(frameSize) * FACTOR8BITBYTES	0	DARAM0

Stack Space Memory (Rule 20)

	Size (bytes)	Align (MAUs)
Worst Case	32 * FACTOR8BITBYTES	0

Static Data Memory (Rule 21)

.cinit					.bss				
Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch	Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch
vol_ti_ivolvt	0x18 * FACTOR8BITBYTES	0	R	No	vol_ti_ivolvt	0x16 * FACTOR8BITBYTES	0	R	No

Program Memory (Rule 22)

Code Sections	Code	
	Size (bytes)	Align (MAUs)
.text	0x8c * FACTOR8BITBYTES	0
.text:algAlloc	0x38 * FACTOR8BITBYTES	0
.text:algInit	0x28 * FACTOR8BITBYTES	0
.text:algFree	0x26 * FACTOR8BITBYTES	0
.text:algMoved	0x10 * FACTOR8BITBYTES	0
.text:init	0x03 * FACTOR8BITBYTES	0
.text:exit	0x03 * FACTOR8BITBYTES	0

Interrupt Latency (Rule 23)		
Operation	**Typical Call Frequency (microsecond)**	**Worst-case (Instruction Cycles)**
amplify()	On Demand	0
control()	On Demand	0

Period / Execution Time (Rule 24)				
Operation	**Typical Call Frequency (microsecond)**	**Worst-case Cycles/Period**	**Worst-case Cycles/Period**	**Worst-case Cycles/Period**
amplify()	On Demand	5261 + (frameSize * 2)	No periodic execution	No periodic execution
control()	On Demand	92	No periodic execution	No periodic execution

Additional Notes for TMS320 DSP Algorithm Standard Compliance

- Rule 1: This algorithm follows the run-time conventions imposed by TI's implementation of the C programming language.
- Rule 2: This algorithm is re-entrant within a preemptive environment (including time-sliced preemption).
- Rule 3: All algorithm data references are fully relocatable.
- Rule 4: All algorithm code is fully relocatable.
- Rule 6: This algorithm does not directly access any peripheral device.
- Rule 10: This algorithm follows the naming conventions of the DSP/BIOS for all external declarations.
- Rule 25: This algorithm was compiled in little endian mode (c6x only).
- Rule 26: This algorithm accesses all static and global data as far data (c6x only).
- Rule 27: This algorithm operates properly with program memory operated in cache mode (c6x only).
- Rule 28: All core run-time support functions are accessed as far functions (c54x far version only).
- Rule 29: All algorithm functions must be declared as far functions (c54x far version only).
- Rule 30: The size of any object file does not exceed the code space available on a page when overlays are enabled (c54x only).
- Rule 32: This algorithm accesses all static and global data as far data, and is instantiable in a large memory model. All OBJECT files were compiled with [–ml] option (c55x only).
- Rule 33: This algorithm operates properly with program memory operated in cache mode (c55x only).

Appendix D: Converting RF3 from LIO to IOM Drivers

In this release of Reference Frameworks, RF3 has been modified to use the IOM-based mini-driver model described in the *DSP/BIOS Driver Developer's Guide* (SPRU616) manual. Previously, RF3 used the LIO driver model described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note, which is now obsolete.

If you created an RF3-based application using an earlier release of Reference Frameworks, you are encouraged to convert to IOM-based mini-drivers. To perform such a conversion, follow these steps.

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

2. Modify *RF_DIR*\apps\rf3\appModules\appIO.h as follows:

Find	Replace With
plio	pio
PLIO	PIO
LIO	IOM

3. Modify *RF_DIR*\apps\rf3\target\appIO.c as follows:

   – Make the following changes using global search-and-replace:

Find	Replace With
plio	pio
PLIO	PIO
lio	iom
LIO	IOM

   – Remove the line from *RF_DIR*\apps\rf3\target\appIO.c that #includes the driver. For example, remove the following for the C5402 DSK.

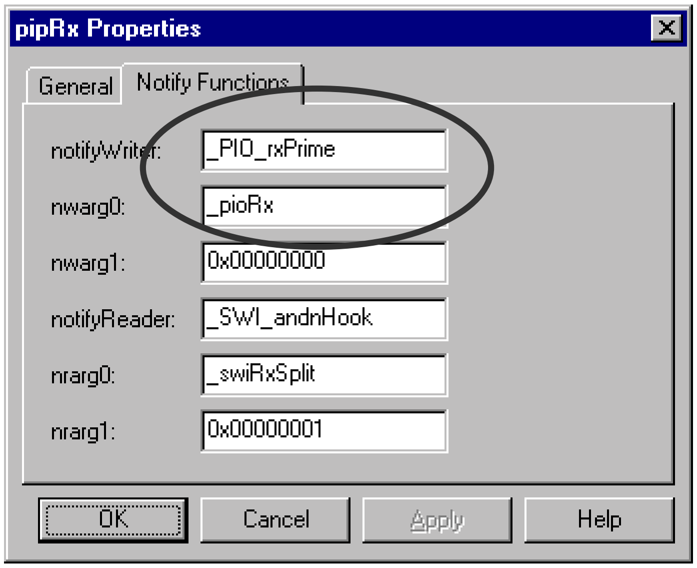
```
#include <dsk5402_dma_ad50.h>
```

   – In the appIOInit() function, remove the calls to the driver initialization (*_init) and setup (*_setup) functions. You will configure a user-defined device object to call the appropriate initialization function.

   – In the appIOInit() function, modify the calls to PLIO_new (renamed to PIO_new) to use the following syntax:

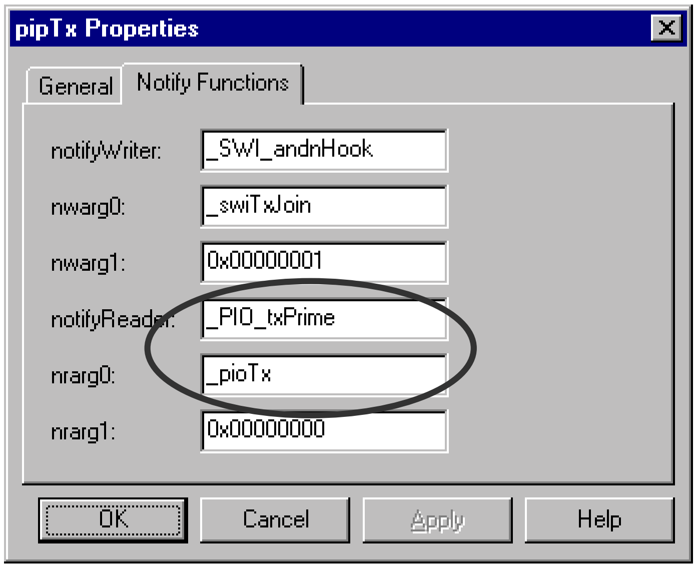
```
PIO_new( &pioRx, &pipRx, "/udevCodec", IOM_INPUT, NULL );
PIO_new( &pioTx, &pipTx, "/udevCodec", IOM_OUTPUT, NULL );
```

4. Open the configuration file (app.cdb) for your project and make the following changes:
   - In the Input/Output folder, modify the notifyWriter and nwarg0 properties of the pipRx PIP object using the values circled below:



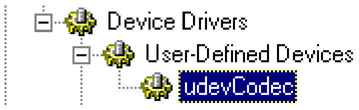
notifyWriter:	_PIO_rxPrime
nwarg0:	_pioRx
nwarg1:	0x00000000
notifyReader:	_SWI_andnHook
nrarg0:	_swiRxSplit
nrarg1:	0x00000001

   - In the Input/Output folder, modify the notifyReader and nrarg0 properties of the pipTx PIP object using the values circled below:



notifyWriter:	_SWI_andnHook
nwarg0:	_swiTxJoin
nwarg1:	0x00000001
notifyReader:	_PIO_txPrime
nrarg0:	_pioTx
nrarg1:	0x00000000

   - In the Input/Output folder, create a User-Defined Device under the Device Drivers category. Rename the new object "udevCodec".



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

▪ init function:	_DSK5402_DMA_AD50_init
▪ function table ptr:	_DSK5402_DMA_AD50_FXNS
▪ function table type:	IOM_Fxns
▪ device id:	0x0
▪ device params ptr:	_DSK5402_DEVPARAMS
▪ device global data ptr:	0x0

The mini-driver's initialization function (for example, DSK5402_DMA_AD50_init) configures the mini-driver's ISR functions programmatically by calling HWI_dispatchPlug.

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

- In the General tab, set the function property to HWI_unused.
- In the Dispatcher tab, remove the checkmark from the Use Dispatcher property.

– Save your changes to the configuration.

5. Copy the *RF_DIR*\apps\rf3\*target**target*_devParams.c file to the corresponding folder of your project. Add this source file to your project in CCStudio.

6. Modify your linker command file (*RF_DIR*\apps\rf3\target\link.cmd) or project to include the libraries for the IOM driver and PIO adapter specified for your target in the new Reference Framework distribution. For example, you might need to include pio.l54 rather than plio.l54.

Appendix E: Reference Framework Board Ports

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

Table 10. Boards Supported by Various Reference Frameworks

Board	Codec	RF1	RF3	RF5	Notes
C5402 DSK	TLC320AD50 mono codec	✓	✓		Large installed base.
C5416 DSK	PCM3002 stereo codec	✓	✓		
C5510 DSK	TLV320AIC23 stereo codec	✓	✓	✓	All three frameworks support DSK 5510.
C6x11 DSK	TLC320AD535 mono codec		✓		Large installed base.
C6416 TEB	PCM3002 stereo codec		✓	✓	

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

For details about how to implement mini-drivers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616). Additional mini-drivers may be added in the future. 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).

The Reference Framework configurations refer directly to the mini-driver's initialization function. It is the job of the PIO adapter to use IOM mini-driver functions directly. This lets the application read blocks of data from the codec by simply reading a pipe, and sending blocks of data to the codec by writing a pipe.

To port RF3 to a new target, other changes are required in addition to creating or using a different mini-driver. For example, a configuration must be created for the target with appropriate objects and properties. And, code in the target-specific folder for the framework requires some changes. See Section 8.2, *Porting the Configuration*, page 64 for details about configuration settings.