Reference Frameworks for eXpressDSP Software:
RF1, A Compact Static System

Alan Campbell
Texas Instruments, Santa Barbara

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 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 1 (RF1) is intended to enable designers to create extremely
compact, consumer products that run on DSPs with restricted memory resources while retaining
the ability to easily integrate eXpressDSP-compliant algorithms. The goal of RF1 is to enable the
creation of complete applications that fit in a memory footprint of less than 8 K words. This size
allows applications to fit on a ‘C5401, TI’s cheapest DSP.

This application note first provides an overview of Reference Frameworks. It then explains
how to install, run, explore, and adapt Reference Framework Level 1.

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

Contents

1 Overview of Reference Frameworks for eXpressDSP Software
1.1 Reference Framework Target Levels
1.2 Reference Framework Architecture
1.3 Adapting the Reference Frameworks
1.4 Bill of Materials
2 Introduction to Reference Framework Level 1
2.1 Characteristics of RF1 Applications
2.2 Applications Suited to RF1
2.3 Suitable Target DSPs for RF1
8 Conclusion.......................................................................................................................... 65
9 References.......................................................................................................................... 66
Appendix A: RF1 Memory Footprint ...................................................................................... 67
Appendix B: FIR Algorithm Characterization ........................................................................ 69
Additional Notes for TMS320 DSP Algorithm Standard Compliance ........................................ 70
Appendix C: Issues Reference .............................................................................................. 72
Appendix D: Reference Framework Board Ports ..................................................................... 73

Figures
Figure 1. Reference Frameworks for eXpressDSP Software and Entry Points ......................... 6
Figure 2. Digital Hearing Aid System ....................................................................................... 11
Figure 3. Low-Cost Internet Audio Player Running a Single Audio Decoder (MP3) ................. 11
Figure 4. LOG Properties for Genbufs Application ................................................................. 15
Figure 5. RF1 Folder Structure ............................................................................................... 17
Figure 6. IALG Interface Function Call Order ......................................................................... 19
Figure 7. Scratch vs. Persistent Memory Allocation .................................................................. 20
Figure 8. FIR Filter Equation .................................................................................................. 21
Figure 9. UTL Debugging Levels ............................................................................................ 23
Figure 10. genbufs Produces a C Source File for the RF1 End-Application ......................... 25
Figure 11. Resulting Generated File ......................................................................................... 26
Figure 12. Construction of XDAIS Buffer Names .................................................................. 28
Figure 13. Topology of Modules in RF1 .................................................................................. 30
Figure 14. Simplified RF1 Data Streaming Diagram ............................................................... 32
Figure 15. Data Connections and Execution Flow in PLIO/LIO Driver ................................. 32
Figure 16. RF1 Data Streaming Details .................................................................................... 34
Figure 17. RF1 Threading Diagram ......................................................................................... 35
Figure 18. Thread Execution Priorities .................................................................................... 35
Figure 19. FIR Module Architecture ....................................................................................... 36
Figure 20. XDAIS Program Code Linked “Out” of RF1 Executable ....................................... 41
Figure 21. Effect of UNION Directive in Achieving Data Memory Overlays ....................... 42
Figure 22. Overlaid Program and Data at the Same Physical Memory Addresses ............... 44
Figure 23. RF1 Adapted as G.726 Encoder/Decoder ............................................................... 50
Figure 24. Folder Locations and Contents for Algorithm Files ............................................. 52
Figure 25. Build Options for G.726 RF1 Adaptation ............................................................... 55

Tables
Table 1. Reference Framework Characteristics by Level .......................................................... 5
Table 2. RF1 Application Characteristics .................................................................................. 10
Table 3. TMS320VC5402 Specifications ............................................................................... 12
Table 4. IALG Implementation Characteristics ...................................................................... 21
Table 5. Persistent / Scratch Data Memory (Rule 19) ............................................................... 24
Table 6. Size Comparison of printf() and LOG_printf() ...................................................... 26
Table 7. DSP/BIOS modules used in RF1 ............................................................................. 30
Table 8. Chip Support Library modules in RF1 ..................................................................... 31
Table 9. DSP/BIOS I/O Driver Model Components in RF1 .................................................... 31
Table 10. Algorithm Used in RF1 ............................................................................................ 31
Table 11. RF1 Interface to XDAIS Algorithms ..................................................................... 31
1 Overview of Reference Frameworks for eXpressDSP Software

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>RF1</th>
<th>RF3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static configuration</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Dynamic object creation</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Static memory management</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>1-10 channels</td>
<td>✔️ 1-3 recommended</td>
<td>✔️</td>
</tr>
<tr>
<td>11-100 channels</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>1-10 eXpressDSP-compliant algorithms</td>
<td>✔️ 1-3 recommended</td>
<td>✔️</td>
</tr>
<tr>
<td>11-100 eXpressDSP-compliant algorithms</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Absolute minimum memory footprint</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Uses DSP/BIOS real-time analysis</td>
<td>✔️ In stop-mode only. RTDX is not configured by default.</td>
<td>✔️</td>
</tr>
<tr>
<td>Uses DSP/BIOS scheduling kernel</td>
<td>✗ HWI and IDL only</td>
<td>✔️ HWI, SWI, and IDL only</td>
</tr>
<tr>
<td>Uses Chip Support Library</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Uses XDAIS algorithms</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Portable to other devices, ISAs, boards</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Supports multiple execution rates and priorities</td>
<td>✗</td>
<td>✔️ Single rate per channel</td>
</tr>
<tr>
<td>Supports thread blocking</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Implements control functionality</td>
<td>✗</td>
<td>✔️</td>
</tr>
</tbody>
</table>

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

In effect, a Reference Framework is an application blueprint. Memory management policies, thread models, and channel encapsulations are common framework elements developers construct today. By relegating these elements to the blueprint, developers can focus on their system’s needs. Developers starting new designs can build on top of the framework, confident that the underlying pieces are robust and fit the characteristics of the target application.

Reference Frameworks are not simply demonstration tools. Instead, they contain production-worthy, reusable, eXpressDSP C language source code for TMS320 ‘C5000 and ‘C6000 DSPs. Some framework elements should be treated as binary libraries, although source code is provided for all levels. For example, the memory 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.

![Diagram of Reference Frameworks for eXpressDSP Software and Entry Points]

---

**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 Evaluation Modules (EVM). 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 new DSP/BIOS I/O Driver Model detailed in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note. 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 or EVM.
- 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).
2 Introduction to Reference Framework Level 1

Reference Framework Level 1 (RF1) is intended to enable designers to create extremely compact, consumer products that run on DSPs with restricted memory resources while retaining the advantages of eXpressDSP Software Technology, including the ability to easily integrate eXpressDSP-compliant algorithms.

By default, RF1 streams a single channel of data through a single algorithm. The algorithm used is the Finite Impulse Response (FIR) filter, an eXpressDSP-compliant algorithm provided by Texas Instruments. Section 6, Adapting the RF1 Application, page 49 describes how to adapt RF1 by changing or adding algorithms, adding channels, and making other changes to meet the needs of your application.

2.1 Characteristics of RF1 Applications

Typical applications that are suited for use with RF1 have a number of cost-based limitations. Low-priced DSPs are typically selected, external memory is often not used, and a large budget for high-priced starterware is not in the plans.

The goal of RF1 is to enable the creation of complete applications that fit in a memory footprint of less than 8 K words. This size allows applications to fit on a TMS320VC5401, TI's cheapest DSP.

In such applications, the entire application must often reside in internal memory. That is, no external, volatile memory is available to the DSP. The minimal application includes foundation software, DSP/BIOS, the Chip Support Library (CSL), XDAIS, as well as the framework glue.

With such stringent size constraints, why should developers be concerned with structure? Is it not better to develop a home-brew minimal framework and drivers to leave as much room as possible for algorithm processing?

In some cases, the answer may be yes. However, typical systems evolve over time. A single-channel MP3 decoder may evolve to support multiple standards such as Advanced Audio Coding (AAC) or Windows Media Format (WMF). A digital hearing aid may then be combined to tune the listening experience to a user’s hearing parameters.

When the system evolves, the benefits of structured programming become obvious. Developers can either build upon the current framework or port their applications to a higher framework level with relative ease. In any case, the goal for the RF1 footprint is to match or better that of the homebrew framework.
Table 2 shows the characteristics of RF1 in more detail than Table 1. It should be used as a guide to determine whether or not RF1 is suitable as the basis of your end-application. Full details of the implementation of design choices are provided in Section 4, Exploring the RF1 Application, page 17.

Table 2. RF1 Application Characteristics

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>RF1</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static configuration</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Dynamic object creation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static memory management</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Dynamic object creation (e.g., DSP/BIOS objects)</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>1-10 channels</td>
<td>✔</td>
<td>1-3 channels recommended. This is a guideline, not a requirement. However, mechanisms for high channel density will be addressed in another Reference Framework level.</td>
</tr>
<tr>
<td>11-100 channels</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>1-10 eXpressDSP-compliant algorithms</td>
<td>✔</td>
<td>1-3 algorithms recommended. This is a guideline, not a requirement. However, mechanisms for 10s of algorithms will be addressed in another Reference Framework level.</td>
</tr>
<tr>
<td>11-100 eXpressDSP-compliant algorithms</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>Absolute minimum memory footprint</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Uses DSP/BIOS real-time analysis</td>
<td>✔</td>
<td>In stop-mode only. That is, RTDX is not configured by default in order to minimize the memory footprint.</td>
</tr>
<tr>
<td>Uses DSP/BIOS scheduling kernel</td>
<td>❌</td>
<td>Uses only HWI dispatcher and IDL threads. Does not use full-featured preemptive scheduling kernel.</td>
</tr>
<tr>
<td>Uses Chip Support Library</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Uses XDAIS algorithms</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Portable to other devices, ISAs, boards</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Supports multiple execution rates and priorities</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>Supports thread blocking</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>Implements control functionality</td>
<td>❌</td>
<td></td>
</tr>
</tbody>
</table>

If your application does not match the characteristics suited to RF1, you should use a higher-level Reference Framework.
2.2 Applications Suited to RF1

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 RF1 is suited:

Figure 2. Digital Hearing Aid System

Figure 3. Low-Cost Internet Audio Player Running a Single Audio Decoder (MP3)
Other applications for which RF1 can be adapted include:

- Smart toys
- Wireless baby monitor
- Digital scanner
- Single-channel proprietary image processing algorithm
- Low-power pay telephone
- Handheld patient monitoring system
- Acoustic echo cancellation in a speaker-phone

2.3 Suitable Target DSPs for RF1

The TMS320VC5402 is a popular DSP for such applications. Its key specifications are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>TMS320VC5402 Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>100</td>
</tr>
<tr>
<td>MIPS</td>
<td>100</td>
</tr>
<tr>
<td>On-chip RAM (K words)</td>
<td>16</td>
</tr>
<tr>
<td>Data Memory reach (K words)</td>
<td>64</td>
</tr>
<tr>
<td>Program Memory reach (K words)</td>
<td>1024</td>
</tr>
<tr>
<td>Core Supply (volts)</td>
<td>1.8</td>
</tr>
<tr>
<td>I/O Supply (volts)</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Many products that use the ‘C5402 are available on the market today. These include:

- IP conference phones
- Digital hearing aids
- Client-side telephony solutions
- Digital scanners

The TMS320VC5401, introduced in Dec 2000, is also a popular DSP for extremely compact, consumer systems suited to RF1. The ‘C5401 is a pin-compatible sub-set of the ‘C5402. Key features are: 50MIPS, 8Kw of on-chip RAM, and the same peripherals and package options as the ‘C5402. This device expands the ‘C5000 line, and makes it possible for designers of even the most cost-sensitive applications to utilize the versatility of programmable DSPs. When ordered in high-volume, the price of a TMS320VC5401 is less than $4.
3 Installing and Running the RF1 Software

This section describes how to use RF1 without making modifications. Section 4, Exploring the RF1 Application, page 17 describes how the application works, and Section 6, Adapting the RF1 Application, page 49 describes how to adapt RF1 to meet the needs of your own application.

3.1 Preparing the Hardware

Currently, the board-specific portions of RF1 are provided for the 'C5402 DSK board, which is available as part of the 'C5402 DSP Starter Kit. If you want to run RF1 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 readme.txt file in the RF_DIR\src\driver folder for your board (for example, RF_DIR\src\dsk5402ad50).

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 RF1. For details, see the appropriate Quick Start Guides, online help, or readme file.

1. If you have not already done so, install CCStudio 2.1 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. For details, see Appendix C: Issues Reference, page 72.

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.

**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.

5. Set the `TERRUPT_DIR` environment variable to point to the folder where you installed CCStudio. This variable is used in the RF1 project files to point to various files.

   - **Windows 98:** Place a line similar to the following in your autoexec.bat file. (Substitute the actual location if it is not `c:\ti`.) Then, either run your autoexec.bat file or reboot your PC.

     ```
     set TERRUPT_DIR=c:\ti
     ```

   - **Windows NT and Windows 2000:** Use the System icon in the Control Panel to set `TERRUPT_DIR` as a user environment variable. Alternately, you can place the definition in the autoexec.bat file.

3.3 Running the genbufs Application

In addition to the actual application, RF1 contains a simple generator application that is run first to generate a source file used in the end-application. The `genbufs.out` target executable is small, however there is never any need to load it on the target along with the actual end-application. Instead, it is run first to determine the needs of the algorithms. It is then replaced on the target by the actual end-application. The reason for this extra step is explained in Section 4.4, The XDAIS Buffer Pre-Generator—`genbufs`, page 24.

Because the genbufs project includes the XDAIS algorithm used by the end-application, it can use the IALG interface to determine the buffer configuration requested by the algorithm. It then generates a file that statically configures the buffers for the algorithm. (All eXpressDSP-compliant algorithms must implement the IALG interface.)

1. Within CCStudio, choose Project→Open and open the `genbufs.pjt` project in `RF_DIR|apps\rf1\genbufs\target`, where `target` matches your board. (For example, `RF_DIR|apps\rf1\genbufs\dsk5402`.)

   The genbufs application generates a source file called `mod_staticcfg.c`—for example, `fir_staticcfg.c`. This file will be used directly in the end-application to declare the FIR algorithm's data buffer requirements.

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

3. Choose File→Load Program and load the `genbufs.out` file in the Debug subfolder onto the target.

4. Choose DSP/BIOS→Message Log to display the MOD_xdasBufSrcTrace log.
5. Right-click on the Message Log area and select Property Page. Change the settings to match those in Figure 4. The log file should be `RF_DIR/apps/rf1/genbufs/fir_staticcfg.c`. It is important that you choose to overwrite the existing file and remove the checkmark from the Sequence Numbers in Output box.

![Message Log Properties](Image)

Figure 4. LOG Properties for Genbufs Application

6. Choose Debug→Run. You will see the generated file in the Message Log window.

7. Open the `fir_staticcfg.c` file and examine its contents. The statements should look similar to the following:

```c
#include <std.h>

#pragma DATA_SECTION(firChanBufId00, ".bss:firChanBufId00")
#pragma DATA_ALIGN(firChanBufId00, 4)

#pragma DATA_SECTION(firChanBufId01, ".bss:firChanBufId01")
#pragma DATA_ALIGN(firChanBufId01, 2)

#pragma DATA_SECTION(firChanBufId02Scr, "bss:firChanBufId02Scr")
#pragma DATA_ALIGN(firChanBufId02Scr, 2)

Char firChanBufId00[6];
Char firChanBufId01[31];
Char firChanBufId02Scr[287];
```

This source file is compiled with the rf1.pjt project in the next section. It statically creates the buffers requested by the algorithm with the requested size and alignment. It also assigns buffers to individual, relocatable data sections, enabling optimal placement and buffer reuse.

In effect, this source file satisfies the algorithm’s buffer needs statically. This eliminates the need for run-time heap memory allocation, which can be expensive in terms of footprint.
NOTE: The #pragma DATA_ALIGN is not yet supported on the ’C54x. These statements generate warnings when you compile rf1.out. These warnings are expected. For details, see item 1 in Appendix C: Issues Reference, page 69.

3.4 Building and Running the RF1 Application

After running the genbufs program to generate the RF_DIR\apps\rf1\genbufs\fir_staticcfg.c source file, you can build and run the RF1 application.

1. Within CCStudio, choose Project→Open and open the rf1.pjt project in RF_DIR\apps\rf1\target, where target matches your board. (For example, RF_DIR\apps\rf1\dsk5402.)

2. Notice that the fir_staticcfg.c file you regenerated is listed in the Source folder of the Project View.

3. Choose Project→Build to build the RF1 application. The warnings about unrecognized #pragmas are expected. For details, see item 1 in Appendix C: Issues Reference, page 69.

4. Choose File→Load Program and load the rf1.out file in the Debug subfolder onto the target.

5. Start your CD player or other audio input.

6. Choose Debug→Run (or F5). You should hear the FIR filtered audio output through the speakers connected to the target board.
4 Exploring the RF1 Application

RF1 actually consists of two separate CCStudio projects, genbufs.pjt and rf1.pjt. The genbufs project is run first to generate a source file used in the rf1.pjt project. The rf1.pjt project is the one you use as the basis for your own application.

4.1 Folder Structure Overview

You can begin to explore RF1 by examining the folder tree that contains the application and associated files. Figure 5 shows the folders used by RF1 and highlights some important files they contain.

Folders to notice include:

- **apps\rf1**. Contains the CCStudio projects that make up RF1. To modify RF1, make a copy of the rf1\ tree at the same folder level, and modify the copy.
  - **dsk5402**. Contains the CCStudio rf1 project for the 'C5402 DSK board used by default.
  - **firapp**. Contains application-side wrappers for communicating with the FIR algorithm. Your algorithms can be wrapped similarly to match the RF1 structure. For example, you might create an mp3app folder.
  - **genbufs**. Contains an application that is run first to generate a source file used in the RF1 application.

- **include**. Contains a number of public header files used by Reference Frameworks. RF1 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.

- **lib.** Contains a number of library files linked in with Reference Framework applications. RF1 uses some, but not all, of these libraries.

- **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 RF1 are:
  - **algmin.** Contains source files for the IALG implementation used by RF1 to instantiate eXpressDSP-compliant algorithms within a minimal memory footprint.
  - **dsk5402ad50.** Contains source files for the device controller used by default in RF1. Other device controllers are provided in similarly named folders.
  - **fir_ti.** Contains files for the Finite Impulse Response (FIR) filter, an eXpressDSP-compliant algorithm provided by Texas Instruments and used by default in RF1.
  - **plio.** Contains source files for the PLIO module, a set of functions that provide an interface between an LIO controller object and a PIP object.
  - **utl.** Contains source files for the UTL debugging and diagnostics module.

### 4.2 TMS320 DSP Algorithm Standard Concepts and Algorithms

This section briefly introduces several TMS320 DSP Algorithm Standard (also known as XDAIS) 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 9, References, page 66 for a list of related documentation and application notes.

#### 4.2.1 The IALG Interface

A fundamental requirement of the 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.

algInit() performs all initialization necessary to complete the run-time creation of an algorithm instance. After a successful return from algInit(), the object is ready to be used 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 can be safely written to, knowing the contents will be unchanged between successive invocations by the application. This distinction enables allocation optimization through scratch buffer overlays.

Figure 6. IALG Interface Function Call Order
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 required function. The algorithm must 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.

### 4.2.2 The ALGMIN Module

RF1 provides an implementation of the IALG interface called ALGMIN. In keeping with the aim of RF1, it is a "small but effective" module. Totaling no more than 100 lines of source code, it serves to statically instantiate any XDAIS algorithm.

ALGMIN requires use of the genbufs application or some other method of generating a source file defining the data buffers required by the algorithm. No dynamic memory is allocated from a heap when instantiating an algorithm. Instead, purely static techniques are used—thus reducing footprint size.

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

Naturally these modules are mutually exclusive. Only one should be used in an end-application. ALGMIN is the correct module for use with RF1.
### Table 4. IALG Implementation Characteristics

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

Why bother with a separate module for the small amount of code in ALGMIN? The answer is that the ALGMIN module makes RF1 easier to adapt and reuse in other applications. In fact ALGMIN can be used independently of RF1 if required.

The application requirements are simply to encapsulate the individual pre-generated buffers as an array of buffer pointers. This is done as follows for the FIR algorithm:

```c
Char *firChanBufs[] = { firChanBufId00, firChanBufId01, firChanBufId02Scr };
```

In this case, firChanBufs is then passed as a parameter to ALGMIN_new. The function loops through the buffers assigning XDAIS memTab[].base entries to each. The resulting memory descriptor table is passed to the algorithm's algInit function thus enabling it to use these data buffers for processing. The cycle is repeated for each new algorithm or channel.

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

### 4.2.3 Finite Impulse Response (FIR) Algorithm

The FIR filter library component is a fully compliant XDAIS algorithm. It executes the mathematical equation shown in Figure 8:

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

**Figure 8. 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 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:

```c
/* 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.

Full performance characterization for the FIR algorithm is provided in Appendix B: FIR Algorithm Characterization, page 69.

### 4.3 The UTL Module

RF1 contains a number of UTL module calls in appMain.c. These are macros that can either be expanded to code that performs a debugging function, or to nothing.

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 9 shows the debug levels and the features they enable:
By default, the RF1 projects are built with UTL_DBGLEVEL set to 60.

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

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

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

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

The UTL_logDebugHandle variable is a LOG_Handle that points to your default LOG object, which is initialized at startup via UTL_setLogs(). If the UTL_DBGLEVEL macro variable is set to 39 or less, the previous UTL_logDebug() macro expands to nothing. See Section 5.1, Disable All DSP/BIOS Instrumentation, page 46 for instructions that completely disable the UTL module.

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.

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 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 file for RF1 contains an STS object calls audioProcessExecTime that can be used for time/period measurements with UTL_sts* functions. The appMain.c file uses the UTL_stsStop() and UTL_stsStart() 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().

### 4.4 The XDAIS Buffer Pre-Generator—genbufs

As you saw in Section 3.3, *Running the genbufs Application*, page 14, running the genbufs application generates a source file that is, in turn, used by the RF1 project.

XDAIS allows significant memory management flexibility. Individual blocks with different sizes, alignment, attributes and location can be requested. At first, it appears that the only way to handle such diversity is dynamically at run-time. Each algorithm’s memory descriptor table can be queried and the DSP/BIOS MEM module can allocate buffers from the requested heap(s) with the correct alignment.

Reference Framework Level 3 employs a Memory Manager module named ALGRF, which builds on top of MEM. A key feature of this component is that all algorithms are instantiated in a common fashion. Developers need not know each algorithm’s memory requirements—they can therefore focus on the core application.

However, this approach comes at a price for the compact system designer. The MEM module is at least 650 words of code and data. ALGRF adds another few hundred words to the image. Justifying MEM and ALGRF in RF1 would be analogous to a car salesman trying to sell a racing car spare tire to a family car driver. It may do the job very well but it is overkill and ultimately the customer is not willing to pay the price.

A static method is the desired solution—one that does not incur the dynamic heap management overhead, yet remains comparatively simple.

One answer is to read the algorithm vendor’s performance documentation and physically declare buffers with the correct size, alignment, and memory space. For example, *Appendix B: FIR Algorithm Characterization*, page 69 contains the following information:

<table>
<thead>
<tr>
<th>memTab</th>
<th>Attribute</th>
<th>Size (bytes)</th>
<th>Align (MAUs)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Persist</td>
<td>6 * FACTOR8BITBYTES</td>
<td>4</td>
<td>External</td>
</tr>
<tr>
<td>1</td>
<td>Persist</td>
<td>(filterLen – 1) * FACTOR8BITBYTES</td>
<td>2</td>
<td>External</td>
</tr>
<tr>
<td>2</td>
<td>Scratch</td>
<td>(filterLen – 1 + frameLen) * FACTOR8BITBYTES</td>
<td>2</td>
<td>DARAM0</td>
</tr>
</tbody>
</table>
There are several disadvantages to this approach:

- Despite the high quality of TI’s third-party algorithm vendors, relying solely on documentation is a risk.
- It is error-prone. If sizes depend upon parameters, as above, then some arithmetic computation must be done.
- New calculations must be done if the algorithm undergoes a revision.

The genbufs application is a more robust, repeatable solution. It acts like a tool—generating a file that defines the required buffer outputs as a function of configurable inputs. It is also surprisingly simple, with no more than two pages of source code. The method is illustrated in Figure 10.

![Diagram](image)

**Figure 10. genbufs Produces a C Source File for the RF1 End-Application**

The pre-generator, genbufs, must be run prior to running the end-application. Step-by-step instructions can be found in Section 3.3, Running the genbufs Application, page 14. The pre-generator needs to be recompiled, built, and run for each XDAIS algorithm in the end-system. It is capable of handling buffer generation for up to 15 channels of any algorithm. It will run on any 'C5000 or 'C6000 simulator, DSK or EVM.

Section 6, *Adapting the RF1 Application*, page 49 describes the process of adapting the genbufs output to meet system needs. The algorithm module name, number of channels, and parameter configuration must all be entered. genbufs reads the input, runs, and then outputs the required data buffer configuration to a C source file.
How do we create the file? The obvious method would be to use the C Standard I/O functions such as fscanf() or fprintf(). However, as Table 6 shows, the DSP/BIOS LOG_printf() function, has a much smaller code size than the standard I/O functions.

Table 6. Size Comparison of printf() and LOG_printf()

<table>
<thead>
<tr>
<th>API</th>
<th>'C54x Size (16-bit words)</th>
<th>'C6000 Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf()</td>
<td>7924</td>
<td>27,648</td>
</tr>
<tr>
<td>LOG_printf()</td>
<td>29</td>
<td>132</td>
</tr>
</tbody>
</table>

Recall that the goal of RF1 is to meet the needs of extremely compact, consumer systems (for example, using the 'C5401). Typically development is done on the same DSP platform as the end-product. On a 'C5401-based platform, a pre-generator that called printf() could not run, thus preventing the designer from even starting RF1 development.

Instead, the highly optimized LOG_printf() DSP/BIOS function is used to generate the C source file. (See item 2 in Appendix C: Issues Reference, page 69.)

The resulting file, shown in Figure 11, can be saved and included directly in RF1. genbufs need not be run again unless a new number of channels or parameter reconfiguration is required.

```c
#include <std.h>
#pragma DATA_SECTION(firChanBufId00, ".bss:firChanBufId00")
#pragma DATA_ALIGN(firChanBufId00, 4)
#pragma DATA_SECTION(firChanBufId01, ".bss:firChanBufId01")
#pragma DATA_ALIGN(firChanBufId01, 2)
#pragma DATA_SECTION(firChanBufId02Scr, ".bss:firChanBufId02Scr")
#pragma DATA_ALIGN(firChanBufId02Scr, 2)

Char firChanBufId00[6];
Char firChanBufId01[31];
Char firChanBufId02Scr[287];
```

Figure 11. Resulting Generated File

The memory management overhead has therefore been eliminated from RF1 saving the system designer almost 1K of memory on the 'C5000.

4.4.1 Preprocessor #pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. Two pragmas are used in genbufs giving the user more flexibility in the configuration of XDAIS algorithm buffers.

- **DATA_SECTION.** Relocates data objects into named sections instead of a location such as the default .bss section
- **DATA_ALIGN.** Aligns a symbol on an alignment boundary
The DATA_ALIGN pragma is particularly useful on the 'C5000 architecture. On 'C54x devices, execution is faster if buffers are aligned on a 128-word boundary using Data-Page Referenced direct addressing. TI's optimized G.726 vocoder operates with 97 words of data memory per instance out of a maximum 128 per page. By forcing the alignment to 128, the CPU instructions need only encode 7 bits since the upper 9-bit data page is constant.

Each of the algorithm's data buffers is assigned a unique subsection from the .bss section. This enables the designer to make intelligent placement decisions. For example, memTab[0] may be referenced less frequently than other memTab[ ] buffers, and could therefore be deferred to external memory.

**NOTE:** The DATA_ALIGN pragma is not supported on 'C54x. It is supported on 'C55x and 'C6000. For details, see item 1 in Appendix C: Issues Reference, page 69.

### 4.4.2 XDAIS Parameter Considerations

Creation parameters often dictate the shape of buffers requested by the algorithm. Also, different parameters may be required for individual instances. For example, a digital hearing aid might use the same code with different parameters for left and right ears.

The main criteria to meet, however, is that the same mechanism be used in genbufs to produce the buffers, as in RF1 itself when initializing the algorithm. It is therefore essential to produce a foolproof method whereby changes to parameters in genbufs are automatically reflected in the end-application.

To that end, parameters are set in fir_setParams.c, which is included directly in both the pre-generator and the end-application.

```c
IFIR_Params FIR_STATIC_PARAMS[] = {
    sizeof(IFIR_Params),                /* sizeof base params structure */
    (short *)filterCoeffs,              /* coefficient array */
    COEFFSIZE,                          /* filter length */
    FILTERFRAMESIZE,                    /* frame length */
},
/* +ENTER NEW CHANNEL PARAMETER SETS BELOW+ */

/* end of FIR_STATIC_PARAMS[] */
```

An additional array of pointers to parameter sets, FIR_chanParamPtrs[ ] gives designers some flexibility in parameter configuration using multiple channels. For example, if three FIR channels were required, two of which used the same parameters, the structure might be defined as:

```c
IFIR_Params *FIR_chanParamPtrs[] = {
    &FIR_STATIC_PARAMS[0],
    &FIR_STATIC_PARAMS[0],
    &FIR_STATIC_PARAMS[1],
};
```
4.4.3 The Channel and Memory Descriptors Mask

The XDAIS buffer variable names are constructed in piecemeal fashion. Figure 12 shows the constituents making up the unique buffer names.

![Diagram showing the construction of XDAIS Buffer Names]

- **MODULENAME**: set in `mod_configure.h`
- **Identification mask**
- **firChanBufId01<Scr>**
  - Constant string defined in genbufs pre-generator
  - Scratch buffer tag (if attrs = IALG_SCRATCH)

**Figure 12. Construction of XDAIS Buffer Names**

The identification mask holds information on both the channel number (first digit) and the memory descriptor index, `memTab[]` (second digit). It seems peculiar at first. Why not separate the tags as in “Chan0Buf1”? The format used is due to the fact that DSP/BIOS `LOG_printf()` accepts no more than two arguments and a format string. One argument is reserved for the queried XDAIS buffer information (for example, `memTab[i].size`). This leaves one argument. Therefore, the channel number and `memTab[]` index are combined into a single mask variable.

If a system requires more channels and buffers, the generated file grows to accommodate this. For example, if an MP3 decoding system uses 4 channels and the algorithm requests 5 data buffers, a total of 20 buffers will be constructed ranging from `mp3decChanBufId00` to `mp3decChanBufId34<Scr>` for the final buffer.

4.4.4 Accounting for Sharable Scratch Memory

The advantages of scratch over persistent data memory were illustrated in Figure 7. Given that all RF1 processing functions execute at the same priority, there is ample opportunity for scratch sharing. When one algorithm completes it can give up the buffer to the next channel or algorithm in line. In this way data memory can be reused, minimizing overall data requirements in the end-application.

One question is how can we determine if a buffer is scratch or persistent in the application itself? If `algAlloc()` must be called again, it defeats the purpose of genbufs in eliminating the end-application’s buffer management overhead.

The solution is quite simple. Run `algAlloc()` in genbufs as normal and mark any buffers requesting `IALG_SCRATCH` with a tag ‘Scr’. In itself, this does not solve the data overlay problem. It does however give a sign to the designer that this buffer can safely be overlaid with others similarly marked.

The technique for achieving overlays is discussed in Section 4.8.2, *Data Memory Savings*, page 42.
4.4.5 **Scope and Limitations of genbufs**

The scope of the genbufs XDAIS buffer generator is as follows:

- Edit the entry points, build, execute, and save the generated file for one XDAIS algorithm at a time. 1 file per algorithm.
- Supplied on 'C5402 DSK and 'C6x11 DSK platforms. However, it can easily be reconfigured to run on any 'C5000 or 'C6000 simulator, DSK, or EVM.
- May be used as a standalone tool outside RF1.
- genbufs.c is algorithm-agnostic and need not be modified.

Its limitations are:

- A maximum of 15 channels per algorithm are supported.
- If the number of memTab[ ] entries is unusually high, the length of the fixed LOG buffer may place a limit on the number of channels that can be generated. For example, if a 'C54x algorithm requests 16 separate data buffers, the limit is 10 channels. Note however that:
  - Algorithms very rarely request 16 buffers. One to five is more typical.
  - The LOG buffer size, MOD_xdasBufSrcTrace, can be increased if memory permits.
- The DATA_ALIGN pragma is not supported on 'C54x. It is supported on 'C55x and 'C6000. For details, see item 1 in Appendix C: Issues Reference, page 69.

4.5 **Reference Framework Level 1 Architecture**

DSP/BIOS teaches mechanics, not methodology. That is, it provides a suite of low-level services without dictating a policy on how to use them. A Reference Framework is the opposite. It dictates which DSP/BIOS, Chip Support Library, and device driver modules are applicable. It is the “glue” that holds together the application, XDAIS algorithms, and the underlying infrastructure.

The value of the framework over a homegrown, tailored solution is:

- It is based on well-established low-level components and concepts.
- It can easily be reused for the next project.

RF1 teaches a methodology for building compact, static systems. The key to building a good RF1 framework and application lies in choosing the right low-level services and applying them in such a way as to achieve the minimum memory footprint possible. Figure 13 illustrates the framework architecture, detailing the services used in the RF1 implementation.
To better understand the framework architecture, each of the modules shown in Figure 13 is briefly described in the following tables.

### Table 7. DSP/BIOS modules used in RF1

<table>
<thead>
<tr>
<th>Name</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWI</td>
<td>Hardware Interrupt Manager</td>
<td>Critical Interrupt Service Routines. RF1 contains two statically configured HWI objects that run the device driver. These objects use the HWI dispatcher to enable a C language interface.</td>
</tr>
<tr>
<td>IDL</td>
<td>Idle Function Manager</td>
<td>Lowest priority thread. Runs continuously except when preempted by higher priority threads. RF1 contains a single IDL function that checks the status of input and output flags to determine when to process data.</td>
</tr>
<tr>
<td>PIP</td>
<td>Buffered Pipe Manager</td>
<td>Manage block-based I/O with synchronization. RF1 statically configures 2 PIP objects, pipRx and pipTx. A full pipRx buffer to process and an empty pipTx buffer to fill is the synchronization event for running the audioProcess() function.</td>
</tr>
<tr>
<td>ATM</td>
<td>Atomic Function Manager</td>
<td>Provides uninterruptible functions for basic operations such as AND and OR.</td>
</tr>
<tr>
<td>CLK</td>
<td>Clock Manager</td>
<td>Used in conjunction with the STS module for reporting close to single-cycle resolution.</td>
</tr>
<tr>
<td>STS</td>
<td>Statistics Object Manager</td>
<td>Stores key statistics while a program runs. RF1 collects the ready-to-completion time of the main processing function. The ‘Max’ field, for example will report if a deadline was ever missed.</td>
</tr>
<tr>
<td>LOG</td>
<td>Event Log Manager</td>
<td>Ever-popular printf-style debugging yet with exceptionally low footprint and fast execution speed. Widely used in RF1 for data flow debugging.</td>
</tr>
</tbody>
</table>
### Table 8. Chip Support Library modules in RF1

<table>
<thead>
<tr>
<th>Name</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP</td>
<td>Multi-channel Buffered Serial Port Manager</td>
<td>Enables configuration, setup and data transfer across the McBSPs. RF1 DSP/BIOS I/O device driver uses MCBSP_open() and MCBSP_config() to open and configure a McBSP port / channel.</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access Module</td>
<td>Enables configuration, setup and then initiating of data transfers in the background of CPU operation. Both input and output streams are DMAed in RF1—this is transparent to the framework. The I/O driver handles it.</td>
</tr>
<tr>
<td>TIMER</td>
<td>Timer Module</td>
<td>Uses 'C54x general-purpose on-chip timer in conjunction with CLK module to report high-resolution time.</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt Controller Module</td>
<td>Interface for managing peripheral interrupts to the CPU. IRQ_map() and IRQ_enable() are used in RF1’s driver to map and enable the DMA receive and transmit hardware interrupts.</td>
</tr>
</tbody>
</table>

### Table 9. DSP/BIOS I/O Driver Model Components in RF1

<table>
<thead>
<tr>
<th>Name</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIO</td>
<td>Low-Level I/O Device Driver Interface</td>
<td>Low-Level I/O Device Driver Interface. Simple, low-level device driver interface between application threads and block-oriented hardware devices. Defines a set of device-agnostic APIs to interface with. Described in SPRA802.</td>
</tr>
<tr>
<td>PLIO</td>
<td>PIP Adapter</td>
<td>The upper-layer of the device driver. 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. The main function creates two PLIO objects, plioRx and plioTx.</td>
</tr>
</tbody>
</table>

### Table 10. Algorithm Used in RF1

<table>
<thead>
<tr>
<th>Name</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR_TI</td>
<td>TI's FIR XDAIS algorithm</td>
<td>TI's eXpressDSP-compliant implementation of the FIR algorithm. A simple component, yet one that illustrates the XDAIS concepts taken advantage of in the RF1 framework.</td>
</tr>
</tbody>
</table>

### Table 11. RF1 Interface to XDAIS Algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGMIN</td>
<td>Minimal IALG interface implementation</td>
<td>This interface can statically instantiate any XDAIS algorithm. It is designed for use by compact, static systems. It requires pre-generated buffer declarations; it allocates no dynamic memory when instantiating algorithms.</td>
</tr>
<tr>
<td>FIR</td>
<td>Framework interface to FIR algorithm</td>
<td>RF1 framework implementation of a caller-friendly wrapper to the ALGMIN interface and the FIR algorithm. It is specific to compact, static systems. It minimizes the footprint while maintaining a structured, modular interface from the application end.</td>
</tr>
</tbody>
</table>
4.5.1 Data Streaming with PLIO/LIO

When the RF1 application starts, the main function initializes modules and creates objects that will be used by the application. The objects used for data streaming include DSP/BIOS pipe objects (pipRx and pipTx) and two PLIO objects (pioRx and plioTx). Figure 14 summarizes the data streaming path in RF1.

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

In Figure 15, numbers indicate the sequence of events when receiving a frame. Steps 3, 4, and 5 are specific to the DSK5402AD50 driver, which uses DMA to transfer samples to and from the McBSP port to which the codec is connected. Other steps are hardware-independent. The sequence is symmetric on the transmission side.
1. When the input pipe receives a frame, it calls PIP_free for the pipRx pipe object to free the frame. This causes the pipRx pipe to run its notifyWriter function. (After the main function runs, the two PIP objects call their notifyWriter functions automatically.)

2. The notifyWriter function of the writer pipe (pipTx) simply sets a flag to indicate that the output buffer is ready. The notifyWriter function of the reader pipe (pipRx) calls PLIO_rxPrime for the plioRx object.

3. The call to PLIO_rxPrime for plioRx allocates a PIP buffer and 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 driver to run its ISR function with the transfer mode—LIO_INPUT—as the argument. The ISR uses the mode as an index to retrieve the address of the callback function for that state. For LIO_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 plioRx. This function indicates that the transfer to the frame is complete, so plioRx 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, sets a flag to indicate the input buffer is ready. At that point, both input and output buffers are ready. After data is transferred to the output pipe (pipTx), that pipe calls PIP_free and the cycle begins again.

8. The IDL thread runs its functions in a loop when no other processing needs to occur. In RF1, the only IDL function is checkFlags(). If one or both flags are unset, the checkFlags() function returns. If both flags are set, the checkFlags() function starts a statistics timer, resets the flags, runs the audioProcess function, and returns. The audioProcess function gets the input buffer from pipRx and allocates the output buffer from pipTx. It then runs FIR_apply to run the FIR algorithm on the source buffer and put the results in the destination buffer. After the algorithm runs, the function puts the full buffer into the pipTx and frees the used buffer from the pipRx. Finally, it stops the statistics timer. Section 4.7, *Minimizing the DSP/BIOS Memory Footprint*, page 38 explains why an IDL function is used instead of a SWI function in this framework.
Figure 16 provides more details on the data streaming path in RF1.
4.5.2 Thread Scheduling

The input pipe (pipRx) sets a flag to indicate when the input buffer is ready. The output pipe (pipTx) sets a flag to indicate when the output buffer is ready. The checkFlags IDL function runs at low-priority and checks the status of input and output flags to determine when to process data.

![RF1 Threading Diagram](image)

**Figure 17. RF1 Threading Diagram**

Since RF1 does not use the SWI or TSK modules, the number of thread priorities is minimized. Hardware interrupts (HWI) run at high priority, and the idle thread (IDL) runs at low priority. Figure 18 shows the execution order and priorities of the threads in RF1.

![Thread Execution Priorities](image)

**Figure 18. Thread Execution Priorities**
4.5.3 Framework Interface to the FIR Algorithm

The FIR module presents a caller-friendly interface to the underlying algorithm. It is, however, optimized for the static, compact needs of RF1.

Why abstract the functionality into a separate module? The reason is that it frees the system integrator (who may be a different engineer than the framework designer) of the algorithm instantiation and execution protocol details. It also allows rapid prototyping of module wrappers to new algorithms enabling faster adaptation of RF1 to the desired end-equipment.

Figure 19 shows the FIR module architecture

![FIR Module Architecture Diagram](image)

Some existing frameworks use `MODULE_create()` functions as part of the algorithm setup. The naming convention used in the eXpressDSP Reference Frameworks, is outlined below to avoid confusion:

- `MODULE_init()`—Module-wide initialization. For example, initial assignments to global data. Often an empty function, but existence still mandatory to enable framework to implement a well-defined startup sequence.

- `MODULE_new()`—The remaining steps required to create a fully-qualified instance of the module. Memory has already been assigned prior to invoking this function. Typically only pointer assignment match-ups must be made in this function.

- `MODULE_create()`—Dynamic run-time creation of an object. One or more heap(s) are used to allocate the requested amount of memory to fully instantiate the instance. `MODULE_new()` equivalent pointer assignments are also encapsulated in this function.

RF1 does not use a `MODULE_create()` function. This is a key optimization in minimizing system footprint. The pre-generator, genbufs, is run prior to the application, creating a source file that specifies each channel’s buffers with the correct size, alignment and placement qualifiers.
The remaining work for the FIR_new function is simply to ensure that the buffer descriptions reach the FIR algorithm. As a result, FIR_new() (in firapp.h) simply returns a handle provided by a call to ALGMIN_new:

```c
/*
 * ======== FIR_new =======
 * Statically create a FIR instance ie no heaps etc
 */
static inline FIR_Handle FIR_new(const IFIR_Fxns *fxns,
                                 FIR_Params *params, Char *firChanBufs[], SmUns firNumChanBufs)
{
    return ( (FIR_Handle)ALGMIN_new((IALG_Fxns *)fxns,
                                     (IALG_Params *)params, firChanBufs, firNumChanBufs) );
}
```

This use of the firChanBufs array allows the number of buffers to be determined within the ALGMIN_new function. The ALGMIN implementation simplifies the process of adapting the RF1 application to other algorithms. Instead of modifying a list of buffers in the FIR_new function, only the declaration of the firChanBufs array in appMain.c needs to be modified.

The filter algorithm is performed by FIR_apply(). Existence of the scratch handling functions, algActivate() and algDeactivate() is first tested before calling them. TI’s implementation of the FIR algorithm makes use of these data-memory saving services, hence the framework is obliged to call them.

### 4.6 DSP/BIOS Device Driver Model

The value of having a solid driver model is immeasurable. Without one, elements of the driver may creep into the territory of the framework and/or application. At that point it becomes impossible to separate the concerns of device driver from the framework implementation. Adapting the framework (for example, for a new hardware codec), is a perilous task with the system integrator never sure whether faults are in the driver, hardware or framework.

These concerns were the primary motivation for adopting the new DSP/BIOS LIO Device Driver Model in all Reference Frameworks. The LIO driver model is described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note. Standard conventions for hardware drivers makes integration and adaptation easier in much the same way that XDAIS simplifies algorithm integration.

Low-Level I/O (LIO) is the basis of this model. It has been vastly simplified compared to a previous implementation. However, it retains core driver concepts, presented to the designer in an XDAIS-like function table. The five APIs that must be programmed are listed in Table 12.

<table>
<thead>
<tr>
<th>LIO Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Cancel all pending input / output requests. For example, the DSK5402 DMA codec driver uses the Chip Support Library function DMA_stop() to halt both receive and transmit DMA channels.</td>
</tr>
<tr>
<td>close</td>
<td>Close controller channel. For example, the DSK5402 DMA codec driver uses IRQ_disable() to physically disable DMA channels.</td>
</tr>
<tr>
<td>ctrl</td>
<td>Control/status function. Typically used for such things as modifying codec sample rate or gain in real-time.</td>
</tr>
<tr>
<td>LIO Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>open</td>
<td>Open controller channel. Initialize input or output channel data structure.</td>
</tr>
<tr>
<td>submit</td>
<td>Submit/service input or output requests. This is the real-time core of the driver. For example, the DSK5402 DMA codec driver updates the channel object and starts the DMA.</td>
</tr>
</tbody>
</table>

Initialization and setup hooks are also used to prepare the serial port, DMA, and codec. Unlike the APIs in Table 12, these functions are called directly by the application code, since they perform one-time setup rather than per channel or per instance work.

To examine the driver source code provided with the Reference Frameworks, see the files in the RF_DIR/src folders (for example, RF_DIR/src/dsk5402ad50). Full technical details on the device driver model are presented in the Writing DSP/BIOS Device Drivers for Block I/O (SPRA802) application note. See this application note for full technical details and information on how to port the driver to meet your needs.

One assuring testament came from a TI Internal Group, which ported a Reference Framework to their hardware platform within a week. The system had a TMS320AIC23 stereo codec with quite different characteristics from the 'C5402 DSK's on-board AD50 mono codec.

The distinct layering (shown previously in Figure 16), even inside the driver itself, was of great value in this case. Although the low-level LIO controller required some modifications (for example, new codec read-write sequencing, sample rate settings), the upper PIP adapter (PLIO) remained the same. Its primary function was simply to obtain synchronized buffers from the application and present them to the controller for consumption. The particular codec or general block-based device beneath was of no concern to this layer.

Since the goal of separating concerns of driver modification from framework adaptation has been met, the reader is again referred to the Writing DSP/BIOS Device Drivers for Block I/O (SPRA802) application note for a full driver discussion and porting guide.

4.7 Minimizing the DSP/BIOS Memory Footprint

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 is a waste of 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 Configuration Tool enables 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.

To further reduce footprint DSP/BIOS provides fine granularity in selection of services. The "dead code" problem is exceptionally rare in DSP/BIOS—only the APIs used are linked in to the end-executable.

Finally, DSP/BIOS kernel services are ROMable to provide further options for reduced RAM usage.

RF1 takes advantage of the first two aspects. All configuration is done statically via the DSP/BIOS Configuration Tool. There are no XXX_create() calls in the application.
In addition, the following actions were performed to eliminate unused OS services. The ease with which modules were removed or reconfigured demonstrates the scalability of DSP/BIOS.

- **Removal of Tasks (TSK).** To disable the TSK manager, the Enable TSK manager checkbox in the TSK Manager Properties dialog is cleared.

- **Use IDL objects instead of SWIs.** Software Interrupts (SWIs) are arguably the cornerstone of DSP/BIOS. They are lightweight, preemptible threads that share a common program stack and do not yield. That is, all of their input and output buffers must be ready prior to triggering a SWI, since it cannot block during execution.

  SWIs are patterned after HWIs but offer many more services. HWI functions post software interrupts to perform lower priority processing. SWIs are priority-based, supporting 14 levels of priority.

  However, in a very compact system, even SWIs may provide more services than required. If all channels and algorithms run at the same priority, the full power of the preemptive kernel need not be applied.

  At some point, applications become simple enough that SWIs have more costs than benefits. Instead, a simpler DSP/BIOS mechanism is called for. RF1 is appropriate for applications with timing and threading requirements so simple that SWIs are not beneficial.

  The simpler DSP/BIOS mechanism used is the IDL module. RF1 contains a single IDL function, checkFlags(), which continuously checks the status of input and output threads in the background. When both flags are set, the main processing sequence is triggered. SWIs on the other hand are often triggered via its mailbox mechanism—when all input and output bits have been cleared the mailbox reaches zero and the software interrupt is automatically posted.

  Scheduling times for both methods are similar. Degradation in IDL response time only becomes noticeable if additional modules creep into the system and the concept of job priorities reappears.

  The savings in program memory outweighed the potential disadvantages. SWI was replaced with the IDL method saving 456 words of program space for the designer.

- **Remove Real-Time Data Exchange (RTDX).** Like SWIs, RTDX is an exceptionally useful feature; it provides extremely fast, deterministic Real-Time Analysis. This enables designers to solve many hard-to-find bugs. Its program and data footprint, however, is sizable.

  RF1 does leave the low-footprint instrumentation of DSP/BIOS in place. That is, the LOG, STS, and CLK functions are still used to provide valuable trace logging and execution time measurement, but only in target-halted mode.

  RTDX was disabled (and can be re-enabled) by setting the properties in Table 13 with the DSP/BIOS Configuration Tool.

### Table 13. Disabling and Enabling RTDX

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Setting to Disable RTDX</th>
<th>Setting to Enable RTDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Settings</td>
<td>Enable Real-Time Analysis</td>
<td>clear checkbox</td>
<td>check mark in checkbox</td>
</tr>
<tr>
<td>HST - Host Channel Manager</td>
<td>Host Link Type</td>
<td>“None”</td>
<td>“RTDX”</td>
</tr>
<tr>
<td>RTDX - Real-Time Data Exchange</td>
<td>Enable RTDX</td>
<td>clear checkbox</td>
<td>check mark in checkbox</td>
</tr>
</tbody>
</table>
• **Disable dynamic memory heaps.** The need for dynamic memory allocation was eliminated as detailed in Section 4.4, *The XDAIS Buffer Pre-Generator—genbufs*, page 24.

• **Disable driving of PRD module.** To remove the PRD_clock object from the CLK manager, the Use CLK Manager to drive PRD checkbox in the PRD Manager Properties dialog is cleared.

• **Disable the CPU load calculation.** The IDL loop instruction count performed for the CPU load calculation is disabled in the IDL Manager Properties dialog. This reduces code size by a small amount.

• **Remove the SYS handling functions.** Normally, the UTL_doAbort(), UTL_doError(), UTL_halt(), and UTL_doPutc() functions are linked in with applications so that they can be run if the corresponding SYS module functions are called. RF1 uses only the UTL_halt() function for these SYS Manager Properties. In addition, the RF1 application sets the trace buffer used by the UTL functions to a size of zero. (Note: These functions are not a part of the UTL macro library described in Section 4.3, *The UTL Module*, page 22.)

These optimizations resulted in a DSP/BIOS program and data footprint of just 1.6K. This includes use of the HWI (and the HWI dispatcher), IDL, PIP, LOG, STS, and CLK modules. The CLK module is included to provide support for detection of missed real-time deadlines and for gathering of execution time statistics.

### 4.8 Minimizing the XDAIS Memory Footprint

A good framework is defined by the design concepts it adopts as well as the services and system "glue logic" it brings. In RF1 the motivating design constraint was to minimize the memory footprint. This section discusses the major savings made in the XDAIS Algorithm domain.

The most significant factor was the elimination of dynamic memory allocation as outlined in Section 4.4, *The XDAIS Buffer Pre-Generator—genbufs*, page 24. Memory management was reduced from heap-based DSP/BIOS MEM calls with ALGRF wrappers to a generated file with static data buffer declarations. All creation code was eliminated, which saved approximately 1K of memory on the 'C5000.

However, further optimizations were also made to enable the system designer to achieve a truly low-cost DSP solution for potential high volume consumer applications.

#### 4.8.1 Program Code Savings

Program code savings were achieved by following the recommendations in the *Zero Overhead with the TMS320 DSP Algorithm Standard IALG Interface* (SPRA716) application note. The link.cmd command file defines a dummy program section and maps several unused XDAIS IALG functions to it.
The DSECT directive informs the linker to create a dummy section not to be included in the output section memory allocation. It therefore takes up no space. Its size and symbol information are listed in the memory map listing (rf1.map) however its contents are never loaded on the target DSP. Figure 20 illustrates the PRGDUMMY section and the effect of the DSECT directive. Its length of 1 is intended as a hint to the system integrator that the contents are not real.

Why was this code linked in at all if the functions were never called by the application? This phenomenon is called "dead code." It is a thorn in the side of all memory-limited system designers. In this instance it is unavoidable due to v-table references in the FIR algorithm (in RF_DIR\src\fir_ti\fir_ti_vt.c).

References to FIR_TI_IFIR, the algorithm’s interface entry point, result in all of the functions listed being linked into the executable whether they are used in the application or not.

The DSECT directive solves this problem, thus saving 186 words of program memory.
Similar savings arise automatically if more XDAIS algorithms are added to the system. The *wildcard in the linker command file enables all algorithms to benefit from this optimization, assuming this common IALG subsection naming policy is adhered to. If your algorithm vendor assigns different section names to IALG methods, check the documentation, and map the new subsections to PRGDUMMY manually.

**NOTE:** Care should be taken when assigning functions to DSECT. The tools do not prevent users from calling functions placed in dummy sections. Any attempt to do so will lead to unpredictable errors.

### 4.8.2 Data Memory Savings

If algorithms and channels use scratch buffers for processing and are not allowed to preempt one another, you can further optimize memory use by overlaying the scratch buffers. Since only a single algorithm uses the scratch buffer at any one time, an overlay allows both scratch buffers to exist at the same physical address.

The genbufs pre-generator deliberately tags IALG_SCRATCH data buffers with an “Scr” postfix. This enables you to differentiate between scratch and persistent buffers. The link.cmd command file allows you to take advantage of this knowledge by using the UNION and GROUP linker directives. The UNION directive causes the linker to allocate multiple sections at the same address. Nesting the GROUP directive within the UNION directive allows you to accommodate algorithms that request multiple scratch buffers. If a single algorithm uses multiple scratch buffers, its buffers cannot overlay one another. However, other algorithms can overlay such a group of scratch buffers with their own scratch buffer(s). Multiple channels of the same algorithm can also overlay their grouped scratch buffers.

In the default RF1 application, only a single channel of a single algorithm is executed, so no gains can be made using scratch buffer overlays. However, suppose that an echo cancellation algorithm and a G.726 encoder both use scratch buffers. As a result of the UNION directive in Figure 21, the size allocated is only that required for the largest scratch buffer, not the sum of all scratch buffer sizes.

```c
SECTIONS
{
    UNION: run = IDATAOVL PAGE 1, align = 2
    {
        GROUP: {
            .bss:lecChanBufId02Scr: {}
        }
        GROUP: {
            .bss:g726ChanBufId04Scr: {}
        }
    }
} /* end scratch overlay sec */
```

**Figure 21. Effect of UNION Directive in Achieving Data Memory Overlays**
The alignment and block size attributes of a union are the maximum alignment and size attributes of any of its members. This fits perfectly with the allocation scheme previously shown in Figure 7.

Multiple channels of the same algorithm or new algorithms need simply to append their Scr-tagged buffers to the UNION and the overlay becomes automatic. It is a zero-overhead method of reusing shared scratch buffers.

See the *TMS320C54x Assembly Language Tools User’s Guide* (SPRU102) and the extensive comments in the link.cmd file to understand the pre-requisites for this scheme. For example, clearly pure persistent memory can never be placed here, nor can algorithms listed in the same union be allowed to preempt one another.

### 4.9 Minimizing the Chip Support Library Memory Footprint

DSP/BIOS integrates with the Chip Support Library (CSL) to simplify the job of developing device drivers. A developer can either use a set of predefined C macros and functions to control peripherals or set up the associated objects using the DSP/BIOS Configuration Tool.

The CSL architecture has a familiar look and feel. Basic resource management is provided through open (for example, DMA_open() ) and close functions for many of the peripherals. It also adopts the naming conventions of DSP/BIOS.

CSL is used extensively in the device drivers. For example, the DSK5402_DMA_AD50 driver uses the MCBSP, DMA, and IRQ modules. In addition, the TIMER APIs are used by DSP/BIOS itself in the clock functions.

Like DSP/BIOS, the Chip Support Library links in virtually no "dead code" (that is, code that is never used).

However, CSL does not eliminate code that creates initialized peripheral objects. (See item 3 in Appendix C: Issues Reference, page 69.) It generates run-once setup code, but this code is not eliminated from the end-executable. The resulting program memory size when configuring CSL statically is no different from the size when coding it dynamically. This is not the case with DSP/BIOS, which provides true footprint savings when static configuration is used.

The code space used by run-once setup code can be regained through an overlay technique. Each function must be assigned its own .text:function subsection to give visibility to CSL APIs during the link step. This enables the system designer to place/map functions uniquely.

RF1 uses this technique to reclaim 1 K words of memory. In a compact system this represents a significant optimization. Figure 22 illustrates the technique:
The basic premise of this technique is that these CSL functions are run once at initialization in a static system. DMA channels are open, configured, and used in a known configuration for the duration of the application. Likewise for McBSP channels.

Any uninitialized data used only after the completion of IPROGOVL code, can be successfully overlaid at the same physical memory address. The data buffers obliterate the program code but this is of no consequence. This is a useful tool for minimizing the application footprint.

4.9.1 Scope and Limitations

The most unfortunate limitation of overlay techniques is that they are not universally applicable across all 'C5000 and 'C6000 devices. 'C54x devices have on-chip RAM mapped into both program and data spaces, and 'C6x1x devices (such as the 'C6711) have a unified memory map. However, the internal memory on 'C6x0x devices is either program or data, not both. Overlays could be applied in external memory as these spaces are typically unified to a single space via the External Memory Interface (EMIF).
Other limitations include:

- Not applicable in ROM or Flash memory.
- The data overlaying the run-once code must be uninitialized. RF1 places its input and output double buffered pipes in this section along with the XDAIS FIR algorithm’s scratch data buffer. Neither of these is used until completion of all CSL initialization code.
- Order of mapping is important. Data placement must precede program code placement. Otherwise the functions would be overwritten and destroyed prior to execution.

In terms of scope:

- If IPROGOVL contents are larger than IDATAOVL contents, the effects are harmless. It simply means less memory is reclaimed.
- If IDATAOVL exceeds IPROGOVL, the effects are harmless. In this case, all run once program code is reclaimed for system data buffers. It is 100% efficient in memory reuse.
- This technique is not restricted to CSL startup code. In fact, RF1 also maps the following code to IPROGOVL:

```c
/*
 * Run-once init code. A subsection naming policy has been adopted. I/O
 * Drivers and XDAIS create code *ALL* are assigned to .text:init sections
 */
.initOvlcode {
  *(.text:init) /* app specific initialization code */
} > IPROGOVL PAGE 0
```

Table 14 shows the total amount of program and data space reclaimed through use of memory overlays for program code that is run once only.

<table>
<thead>
<tr>
<th>Overlay Memory Space</th>
<th>Size Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPROGOVL</td>
<td>0x513 (decimal 1299 words)</td>
</tr>
<tr>
<td>IDATAOVL</td>
<td>0x51f (decimal 1311 words)</td>
</tr>
</tbody>
</table>

Since IPROGOVL size is less than IDATAOVL, the overlay is 100% efficient. All 1299 words of creation code placed in IPROGOVL are reclaimed for useful system data buffers.

5 **Further Optimizations Available**

RF1 uses a variety of techniques to meet its primary goal—minimizing system footprint. The resulting footprint sizes are listed in Section 7.2, *Framework Footprint*, page 65.

If code and data sizes are still too large for very cost-sensitive, bare-bones applications, further optimizations are available. These optimizations are described in the sections that follow. Note that none of the additional optimizations change the shape of the framework. That is, the structured, modular eXpressDSP-driven design of the framework holds firm.

For details about the calculation of the size of the RF1 footprint, see Appendix A: *RF1 Memory Footprint*, page 67.
5.1 Disable All DSP/BIOS Instrumentation

Since LOG and STS are non-intrusive, it is recommended that production programs retain explicit instrumentation. These can be used, for example, in manufacturing tests and field diagnostics. This adheres to the long-standing practice of “Fly what you test, test what you fly.” Exactly the same set of bits tested in the development lab, run on the production system.

Despite this, users are free to eliminate all instrumentation to further reduce footprint. With two simple changes, you can further reduce the system footprint by 381 words:

- In the Build Options for the project, change the UTL_DBGLEVEL to 0.

![Build Options](image)

- In the DSP/BIOS Configuration Tool, disable the CLK Manager.

As a result of these changes, all the UTL module debugging and diagnostics code is removed. Calls to UTL_assert, UTL_logDebug, UTL_stsStart, and UTL_stsStop expand to nothing. In addition, almost all the DSP/BIOS LOG, STS, and CLK code is eliminated and most of the CSL TIMER module is removed.

The UTL functions save many steps in eliminating DSP/BIOS instrumentation. Had we used LOG, STS, and CLK calls explicitly, each call would need to be bracketed with #if and #endif directives to achieve the same goal. Changing the UTL_DBGLEVEL from 60 to zero is a much cleaner solution.

Note that some parts of the RTA modules cannot be completely removed from the end executable, even with UTL_DBGLEVEL set to zero. The LIO controllers for example, explicitly call DSP/BIOS LOG_message(), hence that code is always present.

If you want to enable real-time DSP/BIOS instrumentation, reverse the changes listed for disabling RTDX in Section 4.7, Minimizing the DSP/BIOS Memory Footprint, page 38.

5.2 Overlay XDAIS algInit and algMoved Functions

You may have noticed the following empty allocation in RF1’s link.cmd:

```c
/* XDAIS IALG code used once only */
.ialgOvlCode: {
}
```

This is left as a placeholder for an advanced optimization, available in limited circumstances.

The algInit() function is mandatory for XDAIS algorithms. It performs all initialization necessary to complete the run-time creation of an algorithm’s instance object.
Despite being run only once, the algInit() function is not placed in IPROGOVL so that its code space can be reclaimed. The reasons for this are two-fold:

- If, for some reason, an algorithm manipulates and modifies its assigned scratch buffers inside algInit(), the framework may fail. Scratch buffers are mapped to the overlaid data section and could therefore conflict with the algInit() code in the same physical memory space.

- Some systems must retain algInit() throughout execution. For example, in telecommunications, voice channels are often switched to fax at the end of a business day. At that time, the same algInit() function must be re-run to reset the system dynamically and switch both algorithm and memory usage to handle fax data.

If the designer is both the algorithm-writer and the system integrator, these factors can be controlled. As algorithm-writer, the designer can take steps to avoid modifying scratch buffers inside algInit(). As system integrator, the designer can commit to never calling algInit() more than once.

If these preconditions are satisfied, the ialgOvlCode section can be mapped as follows to hold the algInit() and algMoved() functions of all XDAIS algorithms. The conditions for including algMoved() are the same as for algInit().

```c
/* XDAIS IALG code used once only */
.ialgOvlCode: {
    *(.text:algInit) /* initialize instance object */
    *(.text:algMoved) /* update ptrs after buffers moved */
} > IPROGOVL PAGE 0
```

If additional data were available to overlay this code, 82 words of program memory could be saved in the FIR algorithm.

### 5.3 Replacing the HWI Dispatcher

The HWI dispatcher is the preferred method for handling hardware interrupts. Its advantages include:

- Interrupt routines can be written entirely in C source code. The dispatcher performs the required prolog and epilog for the ISR.

- It transparently protects ISRs that call DSP/BIOS functions, which may cause a context-switch.

- Using the dispatcher saves code space versus its HWI_enter / HWI_exit macro pair counterpart. Since two ISRs are used in the driver (one each for Receive and Transmit), expanding the macros twice increases the footprint.

As a general rule, it is recommended that you not use the “interrupt” keyword as an alternative to the dispatcher. Otherwise operations such as posting a SWI or a semaphore would not be handled correctly.
On the other hand, optimizations can be made for space constrained systems provided certain prerequisites are met. To replace the dispatcher while preserving the content of the driver, the following code can be used:

```c
interrupt void rxWrap()
{
    C54XX_DMA_MCBSP_isr(0); /* Arg value 0 == LIO_INPUT == Rx channel */
}

interrupt void txWrap()
{
    C54XX_DMA_MCBSP_isr(1); /* Arg value 1 == LIO_OUTPUT == Tx channel */
}
```

In the DSP/BIOS Configuration Tool, you must then change the _C54XX_DMA_MCBSP_isr functions called by the HWI_SINT12 and HWI_SINT13 objects to _rxWrap and _txWrap. You must also uncheck the "Use Dispatcher" box for the HWI manager.

Unfortunately a reference to the dispatcher code remains internally. (See item 4 in Appendix C: Issues Reference, page 69.) To prevent the HWI dispatcher code and data being linked in, a temporary stub is therefore required.

```asm
; ======== hwi_disp_asm.s54 ========
; Stub to eliminate Dispatcher code + data when not actually using it
; This stub file is linked in instead of DSP/BIOS hwi_disp_asm.s54
;
.include hwi.h54

.global _HWI_dispatchTab
.bss _HWI_dispatchTab, 1, 0, 2
```

This optimization reduces system footprint by about 400 words.

Be aware that using the interrupt keyword has the following preconditions and limitations:

- ISR (including any callbacks) may not perform context-switching DSP/BIOS operations such as posting a SWI or a semaphore. Therefore, systems that use the interrupt keyword should not enable the SWI or TSK module. Since RF1 uses only HWI and IDL thread scheduling, this limitation is not a concern.

- Interrupt latency will be larger. DSP/BIOS re-enables other HWI interrupts in the prolog as specified by a mask. The interrupt keyword does not have this granularity.

- On the 'C6000, this optimization is not available. Both the CLK and RTDX hardware interrupts are bound to the HWI dispatcher and cannot be modified.
5.4 Rebuilding Libraries

Libraries used by RF1 (for example, ALGMIN, UTL, and PLIO) 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 (rf1.out) or use Project→Rebuild All in order to build with the new library. CCStudio does not check for dependencies on rebuilt libraries.

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.

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 RF1, CCStudio opens windows with their source code automatically as you step into a library module procedure.

6 Adapting the RF1 Application

This section shows ways to adapt RF1 to your needs. For example, you can add channels, add or change algorithms, and change the frame size. Many of the changes can be made by performing case-sensitive, global search-and-replace operations in several files.

Reference Frameworks are designed to make it easy to add algorithms or extend the framework to run several channels.

Reference Frameworks can 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.

6.1 Switching to Other Algorithms

RF1 is intended for use with a small number of eXpressDSP-compliant algorithms, typically from one to three.

This section provides step-by-step instructions on how to adapt RF1, which initially runs a simple FIR algorithm, to apply a G.726 encoder and decoder. These procedures are a template for customers populating the framework with their own algorithms.
Figure 23. RF1 Adapted as G.726 Encoder/Decoder

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

You can modify the code shown to substitute the eXpressDSP-compliant algorithm or algorithms you are using. In most places, you simply use the name of your algorithm instead of "g726enc".

The phases of this adaptation are as follows:

- Creating the Application Folder (Section 6.1.1, page 50)
- Copying Algorithm Files to the lib and include Folders (Section 6.1.2, page 51)
- Setting Up the Algorithm Application Folder (Section 6.1.3, page 51)
- Modifying and Running the genbufs.pjt Project (Section 6.1.4, page 54)
- Modifying the rf1.pjt Project (Section 6.1.5, page 55)

6.1.1 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 rf1\ folder tree and name the copy "rf1_g726". File paths that follow are relative to this folder (RF_DIR\apps\rf1_g726).

   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 folder you create in this step.

   In file names that follow, mod is the lowercase name of your algorithm and MOD is the uppercase name of your algorithm.
6.1.2 Copying Algorithm Files to the lib and include 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.

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:\nda\src\vocoders` folders. If you are adapting RF1 to use third-party algorithms, see the documentation provided by your vendor(s) for file locations.

1. Copy the following library files.
   Copy from: `TI_DIR\examples\dsk5402\xdais\demo\lib`
   Copy to: `RF_DIR\lib`
   - g726enc_ti.l54
   - g726dec_ti.l54

2. Copy the G.726 interface header files. These files are specific to the vendor’s G.726 algorithm implementation.
   Copy from: `TI_DIR\examples\dsk5402\xdais\demo\include`
   Copy to: `RF_DIR\include`
   - g726enc_ti.h
   - g726dec_ti.h

3. 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:\nda\src\vocoders`
   Copy to: `RF_DIR\include`
   - ig726.h
   - ig726enc.h
   - ig726dec.h

6.1.3 Setting Up the Algorithm Application Folder

Once you have copied the algorithm libraries and headers, you need to set up application files for the algorithm similar to those for the FIR algorithm.

1. In your `RF_DIR\apps\rf1_g726` folder, create a `modapp` folder (for example, `RF_DIR\apps\rf1_g726\g726app`). This is where you will place the algorithm function wrappers and default parameter files.

2. Copy the following source files, which specify the default parameters for the algorithms:
   Copy from: `TI_DIR\c:\nda\src\vocoders`
   Copy to: `RF_DIR\apps\rf1_g726\g726app`
   - ig726enc.c
   - ig726dec.c
3. Copy the following source and header files, and rename them as shown in the table:

<table>
<thead>
<tr>
<th>Original Filename</th>
<th>Change File Name To</th>
<th>Example File Name for G.726 Encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir_setParams.c</td>
<td>mod_setParams.c</td>
<td>g726enc_setParams.c</td>
</tr>
<tr>
<td>firapp.c</td>
<td>modapp.c</td>
<td>g726encapp.c</td>
</tr>
<tr>
<td>firapp.h</td>
<td>modapp.h</td>
<td>g726encapp.h</td>
</tr>
</tbody>
</table>

4. Copy an additional set of the files listed in Step 3 for each algorithm you plan to use. Rename the files using each algorithm name (for example, "g726dec"). The files you copied to various folders should be as shown in Figure 24.

5. Perform a case-sensitive find-and-replace on all the files you renamed in the `modapp` folder (for example, `RF_DIR\apps\rf1\rf1_g726\g726app`). To perform case-sensitive searches in CCStudio, choose Edit→Find/Replace and click Properties and put a checkmark in the Match case box.

<table>
<thead>
<tr>
<th>Find</th>
<th>Replace in Encoder Files</th>
<th>Replace in Decoder Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>G726ENC</td>
<td>G726DEC</td>
</tr>
<tr>
<td>fir</td>
<td>g726enc</td>
<td>g726dec</td>
</tr>
</tbody>
</table>

(For other algorithms, make case-sensitive substitutions using that algorithm name.)

6. Make the following additional changes to the `mod_setParams.c` file (for example, `g726enc_setParams.c`).
   - Remove the line that includes `filterCoeffs.h`. These algorithms do not use this file. (If your algorithm application files uses additional header files, include them here.)

```
#include "filterCoeffs.h"
```
– Open the imod.c file in the same folder and copy the parameters inside the MOD_Params static initialization to your clipboard. For example, copy the text shown in bold from the ig726enc.c file:

```c
const IG726ENC_Params IG726ENC_PARAMS = {
    sizeof(IG726ENC_PARAMS),    /* Size of this structure */
    1,                          /* Sample by sample processing */
    IG726_ALAW,                 /* Input buffer format is A-law */
    IG726_16KBPS,               /* Working rate is 16kbps */
};
```

– Paste the text you copied into mod_setParams.c as shown in bold:

```c
IG726ENC_Params G726ENC_STATIC_PARAMS[] = {
    {
        sizeof(IG726ENC_PARAMS),    /* Size of this structure */
        1,                          /* Sample by sample processing */
        IG726_ALAW,                 /* Input buffer format is A-law */
        IG726_16KBPS,               /* Working rate is 16kbps */
    },
    /* +ENTER NEW CHANNEL PARAMETER SETS BELOW+ */
}; /* end of G726ENC_STATIC_PARAMS[] */
```

– Change the values of algorithm parameters in mod_setParams.c as needed. If you are following the G.726 example, make the changes shown in bold in the following code:

```c
IG726ENC_Params G726ENC_STATIC_PARAMS[] = {
    {
        sizeof(IG726ENC_PARAMS),    /* Size of this structure */
        256,                        /* Process by frames instead of by sample */
        IG726_LINEAR,               /* Input buffer format is LINEAR */
        IG726_16KBPS,               /* Working rate is 16kbps */
    },
    /* +ENTER NEW CHANNEL PARAMETER SETS BELOW+ */
}; /* end of G726ENC_STATIC_PARAMS[] */
```

7. Repeat the changes in Step 6 using g726dec_setParams.c (or the corresponding file for any other algorithms you are using).

To confirm that you have modified all the files, you may want to search the files in RF_DIR\apps\rf1_g726\g726app for any remaining occurrences of "FIR" or "fir" before continuing.
6.1.4 Modifying and Running the genbufs.pjt Project

After you have set up the algorithm application files, you can modify and run the genbufs application. You use the genbufs application to generate a C source file that defines each algorithm's data buffer needs. Generating this file is relatively straightforward, and it reduces the memory footprint significantly.

1. In CCStudio, open the project file, `RF_DIR`apps\rf1_g726\genbufs\dsk5402\genbufs.pjt. Perform the steps that follow once for every algorithm your application will use. For example, to use the G.726 encoder and decoder, you must follow Step 2 through Step 5 twice to generate two separate source files (one for each algorithm).

2. In the `RF_DIR`apps\rf1_g726\genbufs folder, make the following changes for the first algorithm. Note that genbufs can only generate a source file for one algorithm at a time.

   - `genbufs\mod_configure.h`. Change the MODULENAME constant to the name of the first algorithm. For the G.726 encoder, you would change the constant definition as shown in bold below:

   ```
   #define MODULENAME g726enc
   ```

   - `genbufs\dsk5402\link.cmd`. Perform a case-sensitive find-and-replace on this file. For the G.726 encoder, change all instances of "FIR" to "G726ENC" and all instances of "fir" to "g726enc". (For other algorithms, make case-sensitive substitutions using that algorithm name.)

3. In CCStudio, add the following source files to the genbufs project from the algorithm application folder you created in Section 6.1.3, Setting Up the Algorithm Application Folder, page 51 (for example, `RF_DIR`apps\rf1_g726\g726app):

<table>
<thead>
<tr>
<th>File To Add</th>
<th>Files to Add for G.726 Encoder</th>
<th>Files to Add for G.726 Decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_setParams.c</td>
<td>g726enc_setParams.c</td>
<td>g726dec_setParams.c</td>
</tr>
<tr>
<td>imod.c</td>
<td>ig726enc.c</td>
<td>ig726dec.c</td>
</tr>
</tbody>
</table>

Add one set of files the first time you perform Steps 2 through 5. Use the other set of files the second time you perform Steps 2 through 5.

4. Remove the following source files from the genbufs project. (The second time you perform Steps 2 through 5, remove the files you added for the first algorithm.)

   - filterCoeffs.c
   - fir_setParams.c
   - ifir.c

5. Build and run the genbufs application as described in Section 3.3, Running the genbufs Application, page 14. Write the output file to genbufs\mod_staticcfg.c (for example, g726enc_staticcfg.c). This filename convention is suggested, but is not a requirement.

6. Perform Steps 2 through 5 for each additional algorithm your application will use. Be sure to use different names for the file you generate for each algorithm (for example, g726enc_staticcfg.c and g726dec_staticcfg.c).
6.1.5 Modifying the rf1.pjt Project

After you have generated a source file for each algorithm that defines the buffers used by the application, you can modify the RF1 application itself to use these new source files and the algorithms you want your application to use. Follow these steps, to modify the RF1 application:

1. In CCStudio, open the project file, \RF_DIR\apps\rf1\g726\dsk5402\rf1.pjt.

2. In CCStudio, choose Project → Build Options. In the Compiler tab, select the Preprocessor category. Change "firapp" in the Include Search Path field to "modapp". (For the G.726 algorithms, use "g726app" as shown in Figure 25.)

![Build Options for G.726 RF1 Adaptation]

3. In CCStudio, remove the following source files from the rf1.pjt project:

   - filterCoeffs.c
   - fir_setParams.c
   - firapp.c
   - ifir.c
   - fir_staticcfg.c

4. Add the following source files to the rf1.pjt project from the modapp\ folder (for example, \RF_DIR\apps\rf1\g726\g726app):

<table>
<thead>
<tr>
<th>File To Add</th>
<th>Files to Add for G.726 Encoder/Decoder Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_setParams.c</td>
<td>g726enc_setParams.c, g726dec_setParams.c</td>
</tr>
<tr>
<td>modapp.c</td>
<td>g726encapp.c, g726decapp.c</td>
</tr>
<tr>
<td>imod.c</td>
<td>ig726enc.c, ig726dec.c</td>
</tr>
</tbody>
</table>

5. Add the following files to the rf1.pjt project from the RF_DIR\apps\rf1\g726\genbufs folder. These files were the output files generated by the genbufs application.

<table>
<thead>
<tr>
<th>File To Add</th>
<th>Files to Add for G.726 Encoder/Decoder Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_staticcfg.c</td>
<td>g726enc_staticcfg.c, g726dec_staticcfg.c</td>
</tr>
</tbody>
</table>
6. Open the RF_DIR\apps\rf1_g726\genbufs\mod_staticcfg.c and the RF_DIR\include\mod.h files for each algorithm so that you can view their contents while editing other files.

7. In each modapp\modapp.h file, make the following changes (for example, RF_DIR\apps\rf1_g726\g726app\g726encapp.h). You should have already globally changed FIR to your algorithm name in Section 6.1.3, Setting Up the Algorithm Application Folder, page 51.

   – Change the list of external references to channel buffers to match those declared in the mod_staticcfg.c file. For example, to switch from the FIR algorithm to G726ENC, remove the external reference to the scratch buffer. TI’s performance-detuned G.726 algorithms do not request any scratch buffers.

   ```c
   /* Match the buffer references to those in the pre-generated buffers C file */
   extern Char g726encChanBufId00[];
   extern Char g726encChanBufId01[];
   extern Char g726encChanBufId02Scr[];
   ```

   – In the external MOD_apply function definition, change the input and output array arguments to match the arguments of the prototype in the MOD_Fxns structure (in the imod.h file) that applies the algorithm to an input buffer and places the result in an output buffer. For example, ig726enc.h defines its encode function as follows:

   ```c
   XDAS_Int16 (*encode)(IG726ENC_Handle handle, XDAS_Int16 *in, XDAS_Int8 *out);
   ```

   So, in g726encapp.h, make the following change to switch from FIR to G726ENC:

   ```c
   /* *
   *  =========== G726ENC_apply ===========
   *  Apply G726ENC algorithm to input array and place results in output array.
   */
   extern Void G726ENC_apply(G726ENC_Handle handle, Short in[], Short out[]);
   ```

   NOTE: In the G.726 example, remember that the data types for the *in and *out arguments have their sequence switched for the encoder and decoder.

8. In each modapp\modapp.c file, make the following changes. (You should have globally changed FIR to your algorithm name in Section 6.1.3, Setting Up the Algorithm Application Folder, page 51.)

   – In the MOD_apply function definition, change the input and output array arguments to match the arguments of the prototype in the MOD_Fxns structure. For example:

   ```c
   /* *
   *  =========== G726ENC_apply ===========
   *  Apply G726ENC algorithm to input array and place results in output array.
   */
   Void G726ENC_apply(G726ENC_Handle handle, Short in[], Short out[]);
   Void G726ENC_apply(IG726ENC_Handle handle, XDAS_Int16 *in, XDAS_Int8 *out )
   ```
In the MOD_apply function, change the algorithm operation to the one defined in the IMOD_Fxns structure in the RF_DIR/include/mmoodd.h file. For example, to switch from the FIR algorithm to G726ENC, change the code as follows:

```c
handle->fxns->filter( handle, in, out );  /* filter data */
handle->fxns->encode( handle, in, out );  /* encode data */
```

9. Be sure that you have made the modifications described in Steps 4 through 8 for all the algorithms your application will use. For this example, repeat these steps for the G.726 decoder instead of the encoder.

10. In dsk5402\link.cmd, make the following changes:

   - Change the line that links in the FIR algorithm library file to link in the library files for your algorithms. You should link in a separate library for each algorithm. For example, to link in the G.726 encoder and decoder libraries, change the file as follows:

     ```
     /* XDAIS algorithm(s) */
     -l fir_ti.l54
     -l g726enc_ti.l54
     -l g726dec_ti.l54
     ```

   - Change the pointer to the vendor's IMODULE implementation. To use TI's G.726 encoder and decoder, look at RF_DIR/include/g726enc_ti.h and g726dec_ti.h, and change link.cmd as follows:

     ```
     /* Point to the algorithm vendor's IMOD v-table (name located in mod_ven.h) */
     _FIR_IFIR = _FIR_TI_IFIR;
     _G726ENC_IG726ENC = _G726ENC_TI_IG726ENC;
     _G726DEC_IG726DEC = _G726DEC_TI_IG726DEC;
     ```

   - Change the alignment statements for the persistent algorithm buffers to match the values in the #pragma directives in the mod_staticcfg.c files you generated. To use TI's G.726 encoder and decoder change link.cmd as follows:

     ```
     SECTIONS
     {
     .bss:firChanBufId00: align = 4 fill = 0x0 {} > IDATA PAGE 1
     .bss:firChanBufId01: align = 2 fill = 0x0 {} > IDATA PAGE 1
     .bss:g726encChanBufId00: align = 0 fill = 0x0 {} > IDATA PAGE 1
     .bss:g726encChanBufId01: align = 128 {} > IDATA PAGE 1
     .bss:g726decChanBufId00: align = 0 fill = 0x0 {} > IDATA PAGE 1
     .bss:g726decChanBufId01: align = 128 {} > IDATA PAGE 1
     }
     /* end of persistent buffer alignment and fill section */
     
**NOTE:** When item 1 in Appendix C: Issues Reference, page 69, is resolved in CCStudio 2.2, this step will be eliminated. Only the zero fill operation on the first buffer (memTab[0]) will still be required for compliance with the XDAIS specification.
Change the alignment statements for the scratch algorithm buffers to match the values in the #pragma directives in the mod_staticcfg.c files you generated. For example, TI's G.726 encoder and decoder use no scratch buffers, so you simply delete the statement and comment out the entire unit as follows:

```c
/* begin comment
SECTIONS
{
  UNION: run = IDATAOVL PAGE 1, align = 2
  {
    GROUP: {
      .bss:firChanBufId02Scr: {} }
    }
  }
}
end comment */
```

**NOTE**: If your algorithms use no scratch buffers, it is important to comment out or delete the entire SECTIONS block, because the linker does not allow empty UNION statements. If you have multiple algorithms, create a GROUP block for each algorithm and put the scratch buffer(s) for each algorithm in a separate GROUP.

11. In dsk5402\appMain.c, make the following changes:

- Perform a case-sensitive find-and-replace on this file to change all instances of "FIR" to the uppercase name of your first algorithm (for example, "G726ENC") and all instances of "fir" to the lowercase name of your first algorithm (for example, "g726enc").

- Add a statement to the list of included files to #include the modapp.h header file for any additional algorithms. For example, insert the following line in bold to add the G.726 decoder algorithm:

```c
#include "g726encapp.h"           /* algorithm's application interface(s) */
#include "g726decapp.h"           /* algorithm's application interface(s) */
```

- Add declarations to enable instantiation of any additional algorithms. For example, insert the following lines in bold to add the G.726 decoder algorithm:

```c
/* Declaration of XDAIS v-tables. These are the algorithm(s) entry pts */
extern IG726ENC_Fxns      G726ENC_IG726ENC;     /* G726ENC algorithm */
extern IG726DEC_Fxns      G726DEC_IG726DEC;     /* G726DEC algorithm */

/* Handles to the XDAIS algorithm(s). Provides reentrant access to the fxns */
G726ENC_Handle    g726encHandle0;
G726DEC_Handle    g726decHandle0;
```
– In the declaration of the \texttt{modChanBufs} array at the beginning of the main function, change the number of buffers to match the number used by the algorithm. For example, delete the third buffer for the G.726 encoder:

\begin{verbatim}
/* Prepare buffer pointer array for Static XDAIS Algorithm instantiation */
Char *g726encChanBufs[] = {
g726encChanBufId00, g726encChanBufId01, g726encChanBufId02Scr
};
\end{verbatim}

– Add declarations of \textit{modChanBufs} arrays for any additional algorithms. For example, add the following declaration for the G.726 decoder:

\begin{verbatim}
Char *g726decChanBufs[] = { g726decChanBufId00, g726decChanBufId01 }
\end{verbatim}

– In the main function, add a call to the initialization function for any additional algorithms. For example, insert the line in bold to add the G.726 decoder algorithm:

\begin{verbatim}
/* Initialize the XDAIS algorithm(s) */
G726ENC_init();
G726DEC_init();
\end{verbatim}

– Add a call to \textit{MOD\_new} for any additional algorithms. For example, add the following call for the G.726 decoder:

\begin{verbatim}
/* Statically create an instance of the XDAIS algo(s) ie no heaps etc */
g726encHandle0 = G726ENC_new(&G726ENC_IG726ENC, G726ENC_chanParamPtrs[0],
g726encChanBufs, (sizeof(g726encChanBufs) / sizeof(g726encChanBufs[0])));
g726decHandle0 = G726DEC_new(&G726DEC_IG726DEC, G726DEC_chanParamPtrs[0],
g726decChanBufs, (sizeof(g726decChanBufs) / sizeof(g726decChanBufs[0])));
\end{verbatim}

– In the \textit{audioProcess} function, change the source and destination data types for the \textit{MOD\_apply} function arguments to match the input and output data types in \textit{modapp/modapp.c}. Also, add a call to \textit{MOD\_apply} for any additional algorithms. For example, make the following changes to use the G.726 encoder and decoder algorithms:

\begin{verbatim}
/* Execute the XDAIS Algorithm(s) */
G726ENC_apply(g726encHandle0, (\texttt{short} XDAS\_Int16 *)src, (\texttt{short} XDAS\_Int8 *)dst);
G726DEC_apply(g726decHandle0, (XDAS\_Int8 *)src, (XDAS\_Int16 *)dst);
\end{verbatim}
The G.726 example converts a filter application to an encoder/decoder application. The following application modifications are required to cause the application to decode the encoded data.

Before the main function, add the following global declaration to create an intermediate buffer:

```c
XDAS_Int8 bufInterm[256]; /* intermediate buffer to store encoder output */
```

In the audioProcess function, make the following changes to use the intermediate buffer:

```c
/* Execute the XDAIS Algorithm(s) */
G726ENC_apply(g726encHandle0, (XDAS_Int16 *)src, (XDAS_Int8 *)bufInterm);
G726DEC_apply(g726decHandle0, (XDAS_Int8 *)src, (XDAS_Int8 *)bufInterm, (XDAS_Int16 *)dst);
```

Your application is likely to require similar application logic modification.

12. Save all the files you edited and the project.

13. Build and run the RF1 application as described in Section 3.4, Building and Running the RF1 Application, page 16.

You should hear audio with slight degradation compared to the simple FIR filter processing. This degradation is normal given the compression rate of the vocoder. If you do not hear audio at all, check the G726ENC_STATIC_PARAMS and G726DEC_STATIC_PARAMS structures in the g726enc_setParams.c and g726dec_setParams.c files. Missed steps in setting parameters may produce problems with the audio.

**NOTE:** Warnings about unrecognized #pragmas are expected. (See item 1 in Appendix C: Issues Reference, page 69.) Warnings about text:algNumAlloc and text:algControl occur because the G.726 algorithms do not implement these optional XDAIS methods.

In this section, you adapted RF1 to run a G.726 encoder and decoder. You can use this procedure as a template for populating the framework with other algorithms.

### 6.2 Adding a Second Channel

RF1 is designed for applications that use a small number of channels.

Many simple applications may have the need for a second channel. For example, a small MP3 player with stereo would have left and right channels. Or, an audio player might have different static parameters for a channel going to speakers than for a channel going to a headphone jack.

Since the 'C5402 DSK has an AD50 mono codec, this example duplicates a single signal to two channels rather than converting the application to a true multi-channel application.

**NOTE:** The instructions provided for this adaptation assume that you are working from the base RF1 application, which uses the FIR filter. If you have already switched algorithms as described in Section 6.1, Switching to Other Algorithms, page 49, modify the example statements shown as appropriate.

In the file paths that follow, **RF_DIR** is your "referenceframeworks" folder, which you may have unzipped to any location. Other file paths are relative to the location of the folder you create in the first step (for example, **RF_DIR**\apps\rf1_2ch).
In order to convert the default RF1 application to a two-channel application with different static parameters for each channel, follow these steps:

1. In the apps\ folder, duplicate the entire rf1\ folder tree and name the copy "rf1_2ch". We recommend that you copy the application folder so the unmodified application remains available for reference.

2. In CCStudio, open the project file, \RF_DIR\apps\rf1_2ch\genbufs\dsk5402\genbufs.pjt.

3. In genbufs\mod_configure.h, change the NUMCHANNELS constant to the number of channels you will use. For this example, set NUMCHANNELS to 2.

```c
#define MODULENAME          fir
#define NUMCHANNELS           2
```

4. If the algorithm parameters are the same for all channels using this algorithm, skip to Step 5. To use different algorithm parameters for two channels that use the same algorithm, edit the firapp\fir_setParams.c file by adding a block of channel parameters to FIR_STATIC_PARAMS. Modify parameters as necessary for the second channel.

A typical modification for a second channel might be to use different filter coefficients. For example, if you define a filterCoeffs_1 constant array in firapp\filterCoeffs.c and firapp\filterCoeffs.h, you could add the following second parameter set:

```c
IFIR_Params FIR_STATIC_PARAMS[] = {

    sizeof(IFIR_Params),      /* sizeof base params structure */
    (short *)filterCoeffs,    /* coefficient array */
    COEFFSIZE,                /* filter length */
    FILTERFRAMESIZE           /* frame length */
},

/* +ENTER NEW CHANNEL PARAMETER SETS BELOW+ */
{
    sizeof(IFIR_Params),      /* sizeof base params structure */
    (short *)filterCoeffs_1,  /* coefficient array */
    COEFFSIZE,                /* filter length */
    FILTERFRAMESIZE           /* frame length */
},
}; /* end of FIR_STATIC_PARAMS[] */
```

5. In firapp\fir_setParams.c, add the base address of the parameter set for each channel.

- If the parameters for two channels are the same, follow this example by adding a line like that shown in bold:

```c
IFIR_Params *FIR_chanParamPtrs[] = {

    &FIR_STATIC_PARAMS[0],

    /* +UPDATE PARAM POINTERS TO REFLECT NEW CHAN PARAM SET ADDITIONS+ */
    &FIR_STATIC_PARAMS[0],

};
```

- If the parameters for two channels are different, use &FIR_STATIC_PARAMS[1] for the second set of parameters.
6. Build and run the genbufs application as described in Section 3.3, *Running the genbufs Application*, page 14. Write the output file to fir_2ch_staticcfg.c. (This is a suggested filename, but is not a requirement.)

7. Open the *RF_DIR* apps\rf1_2ch\dsk5402\rf1.pjt project in CCStudio. In that project, make the following changes:
   - Remove the fir_staticcfg.c file from the Source folder of the project and add the file you just generated with the genbufs application (for example, fir_2ch_staticcfg.c) to the project.
   - *dsk5402\appMain.c*. Add the text in bold to create a second FIR object and to run the algorithm on the second channel using the buffers declared in fir_2ch_staticcfg.c. Notice that the first digit in the buffer names is different for each channel.

```c
/* Declaration of XDAIS v-tables. These are the algorithm(s) entry pts */
extern IFIR_Fxns    FIR_IFIR;      /* FIR algorithm */

/* Handles to the XDAIS algorithm(s). Provides reentrant access to the fxns */
FIR_Handle          firHandle0;
FIR_Handle          firHandle1;
...

/* Prepare buffer pointer array for Static XDAIS Algorithm instantiation */
Char *firChanBufs0[] = {firChanBufId00, firChanBufId01, firChanBufId02Scr};
Char *firChanBufs1[] = {firChanBufId10, firChanBufId11, firChanBufId12Scr};
...

/* Statically create an instance of the XDAIS algo(s) ie no heaps etc */
firHandle0 = FIR_new(&FIR_IFIR, FIR_chanParamPtrs[0],
   firChanBufs0, (sizeof(firChanBufs0) / sizeof(firChanBufs0[0])));
firHandle1 = FIR_new(&FIR_IFIR, FIR_chanParamPtrs[0],
   firChanBufs1, (sizeof(firChanBufs1) / sizeof(firChanBufs1[0])));
...

/* algorithm execution */
FIR_apply(firHandle0, (Short *)src, (Short *)dst);
FIR_apply(firHandle1, (Short *)src, (Short *)dst);
```

Since the 'C5402 DSK has an AD50 mono codec, this example simply applies the algorithm to the same input buffer twice. Applications with multiple codecs would instead, typically use independent input and output buffers for successive channels.
firapp\firapp.h. Add the lines shown in bold to create external references to the buffers declared in fir_2ch_staticcfg.c. Again, remember that the first digit in the buffer names is different for each channel.

```c
/* Match the buffer references to those in the pre-generated buffers C file */
extern Char firChanBufId00[];
extern Char firChanBufId01[];
extern Char firChanBufId02Scr[];
extern Char firChanBufId10[];
extern Char firChanBufId11[];
extern Char firChanBufId12Scr[];
```

dsk5402\link.cmd. Add the lines in bold to align and initialize the algorithm's persistent and scratch buffers. The comments in this file explain why these statements are required even though the fir_2ch_staticcfg.c file performs similar actions.

```c
SECTIONS
{
    .bss:firChanBufId00:    align = 4   fill = 0x0      {} > IDATA PAGE 1
    .bss:firChanBufId01:    align = 2                   {} > IDATA PAGE 1
    .bss:firChanBufId10:    align = 4   fill = 0x0      {} > IDATA PAGE 1
    .bss:firChanBufId11:    align = 2                   {} > IDATA PAGE 1
}
/* end of persistent buffer alignment and fill section */

...

SECTIONS
{
    UNION: run = IDATAOVL PAGE 1, align = 2
    {
        GROUP: {
            .bss:firChanBufId02Scr: {}
        }
        GROUP: {
            .bss:firChanBufId12Scr: {}
        }
    }
}
/* end of scratch buffer overlay section */
```

8. Build the RF1 application as described in Section 3.4, *Building and Running the RF1 Application*, page 16. Warnings about unrecognized #pragmas are expected.

9. Load and run the application.

In this section, you adapted RF1 by adding a second channel. You can use this procedure as a template for adding additional channels to your own RF1-based applications.
6.3 Changing the Frame Size

Modifying the end-application to use a different frame size is also a typical way to adapt the Reference Framework. To modify the frame size, you should first run the genbufs application with any parameter changes required to change the frame size. This caters to possible dependencies between the total amount of data memory requested by the algorithm and any frame size parameter. For example, in the supplied FIR algorithm, the scratch buffer size is a function of both filter and frame length. After running the genbufs application, change the PIP object framesize property in the DSP/BIOS configuration file for the end-application and any end-application source files that reference such constants.

7 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.

7.1 Performance Characteristics

The performance characteristics of RF1 are shown in Table 15. 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 15. RF1 CPU Usage Statistics

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU Load*</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF 1 as supplied</td>
<td>7.2%</td>
</tr>
<tr>
<td>RF 1 minus application-specific FIR algorithm processing (1)</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

The load value for the application minus the FIR algorithm processing was obtained by removing the algorithm and replacing it with simple direct copying of the source buffer to the destination buffer. Note that the FIR algorithm is not optimized. This enables easy debugging and code traversal of the XDAIS algorithm supplied with RF1.

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: 256
- Optimization flags: none
- Debug flags: -g
- All critical code and data in internal memory
7.2 Framework Footprint

As mentioned earlier, the goal of RF1 is to enable the creation of complete applications that fit in a memory footprint of less than 8 K words. This size allows applications to fit on a TMS320VC5401, TI’s cheapest DSP.

The actual RF1 footprint and the space available for algorithms is shown in Table 16.

<table>
<thead>
<tr>
<th>Framework Footprint (program + data)</th>
<th>Remaining Memory for Algorithms on TMS320VC5401</th>
<th>Remaining Memory for Algorithms on TMS320VC5402</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF1 as supplied</td>
<td>4 K words</td>
<td>8 – 4 = 4 K words</td>
</tr>
<tr>
<td>RF1 with further optimizations (Section 5)</td>
<td>3 K words</td>
<td>8 – 3 = 5 K words</td>
</tr>
</tbody>
</table>

For example, if a third-party 'C5000 XDAIS-compliant MP3 decoder consumes less than 12-13 K words, a compact, portable audio player could be constructed on a TMS320VC5402 using this framework.

In this calculation, the "framework footprint" includes all application, driver, and foundation software (DSP/BIOS, CSL), framework-XDAIS-wrappers, code, and data. This approximation assumes that enough system data can be found to overlay and reclaim initialization code. It also takes into account approximately 200 words of IALG functions in program memory. This value is approximate since it is algorithm-dependent.

For details about the calculation of the size of the RF1 footprint, see Appendix A: RF1 Memory Footprint, page 67.

The further optimizations described in Section 5, Further Optimizations Available, page 45 can be used to further reduce the footprint size. All of these optimization techniques preserve the design characteristics of RF1.

8 Conclusion

Reference Framework Level 1 is a blueprint for compact, static systems that run on DSPs with restricted memory resources. It enables designers to create bare-bones consumer and industrial products on TI DSPs, using proven software components. It retains the advantages of the eXpressDSP Software Technology strategy, including the ability to easily integrate XDAIS-compliant algorithms.

The framework is supplied as highly reusable C language source code to enable switching between 'C5000 and 'C6000 ISAs. It is packaged in a complete application running on TI’s 'C5402 DSP Starter Kit.

This application note includes framework selection criteria, a list of typical applications, explanations of how the framework functions, and two working examples of key framework adaptations. This enables designers to quickly determine if RF1 fits their needs, and also to rapidly prototype the end-equipment following step-by-step instructions.
The initial goal for RF1 was to enable complete applications to fit in a memory footprint of less than 8 K words. This goal is met by RF1, which can be as small as 3 K words excluding algorithms. This size allows applications to run entirely from the on-chip memory of a TMS320VC5401, TI's cheapest DSP.

RF1 teaches a methodology for building many such compact 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.

9 References
For additional information, see the following sources:

Product Documentation
- TMS320 DSP/BIOS User's Guide (SPRU423)
- TMS320C5000 DSP/BIOS API Reference Guide (SPRU404)
- TMS320C54x Chip Support Library API Reference Guide (SPRU420)
- TMS320C55x Chip Support Library API Reference Guide (SPRU433)
- TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352)
- TMS320 DSP Algorithm Standard API Reference (SPRU360)
- TMS320C54x Assembly Language Tools User's Guide (SPRU102)

Application Notes
- Reference Frameworks for eXpressDSP Software: A White Paper (SPRA094)
- Reference Frameworks for eXpressDSP Software: API Reference (SPRA147)
- Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System (SPRA793)
- Writing DSP/BIOS Device Drivers for Block I/O (SPRA802)
- 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: RF1 Memory Footprint

The overall footprint sizes for RF1 are listed in Section 7.2, Framework Footprint, page 65. This appendix provides details on how those numbers were obtained. These numbers are based on measurements taken using CCStudio 2.1.

The breakdown of the RF1 memory footprint is shown in Table 17. This breakdown assists the system designer in planning an overall DSP solution. Given the size of the XDAIS algorithms to be used in the end-application and the size of the RF1 framework container, a decision can be made about which TMS320 DSP to use.

Naturally, the classification of categories can be debated. For example, are Chip Support Library functions such as DMA_open() part of CSL or the I/O driver? Regardless of this, the key number is the total footprint.

<table>
<thead>
<tr>
<th>Table 17. RF1 Memory Footprint Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>DSP/BIOS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>CSL</td>
</tr>
<tr>
<td>(See note 2.)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>I/O Driver</td>
</tr>
<tr>
<td>(See note 3.)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>XDAIS ALGMIN Framework</td>
</tr>
<tr>
<td>(See note 4.)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>XDAIS FIR Algorithm</td>
</tr>
<tr>
<td>(See note 5.)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>General Framework / Application</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Data Pipes</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>XDAIS FIR buffers</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>RTS Library</td>
</tr>
<tr>
<td>Interrupt Vectors</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** The .LOG_system$buf section cannot be removed because LOG_message is used in the driver source code.

**Note 2:** 1024 words of run-once program code were reclaimed with overlays in the .cslOvlCode section.

**Note 3:** 239 words of run-once program code were reclaimed with overlays in the .initOvlCode section.

**Note 4:** Approx 1 K words were saved via static instantiation and genbufs pre-generator application, eliminating the need for a heap and memory allocation functions.

**Note 5:** 186 words of XDAIS IALG algorithm code were reclaimed through static instantiation and the genbufs pre-generator application, which eliminates the need for certain IALG functions.

The total of 6323 words represents the footprint for the supplied Reference Framework Level 1. However, components such as the FIR algorithm will not be in the user's end-application. Therefore, in order to determine whether the XDAIS algorithms will fit in a certain DSP’s on-chip memory, we need to subtract the size of components that will not appear.

XDAIS algorithm vendors must supply the following footprint sizes:

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

This data is provided for the FIR algorithm in *Appendix B: FIR Algorithm Characterization*, page 69.
## Appendix B: FIR Algorithm Characterization

### Module Vendor Variant Arch Mem Model Version Library Name

<table>
<thead>
<tr>
<th>Module</th>
<th>Vendor</th>
<th>Variant</th>
<th>Arch</th>
<th>Mem Model</th>
<th>Version</th>
<th>Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>TI</td>
<td>54</td>
<td>near</td>
<td></td>
<td></td>
<td>fir_ti.l54</td>
</tr>
</tbody>
</table>

**ROMable (Rule 5)**

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

*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)

<table>
<thead>
<tr>
<th>memTab</th>
<th>Attribute</th>
<th>Size (bytes)</th>
<th>Align (MAUs)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Persist</td>
<td>6 * FACTOR8BITBYTES</td>
<td>4</td>
<td>External</td>
</tr>
<tr>
<td>1</td>
<td>Persist</td>
<td>(filterLen – 1) * FACTOR8BITBYTES</td>
<td>2</td>
<td>External</td>
</tr>
<tr>
<td>2</td>
<td>Scratch</td>
<td>(filterLen – 1 + frameLen) * FACTOR8BITBYTES</td>
<td>2</td>
<td>DARAM0</td>
</tr>
</tbody>
</table>

### Stack Space Memory (Rule 20)

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Align (MAUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
</tbody>
</table>

### Static Data Memory (Rule 21)

<table>
<thead>
<tr>
<th>.data</th>
<th>.bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object File</td>
<td>Size (bytes)</td>
</tr>
<tr>
<td>fir_ti_vt.o54</td>
<td>10 * FACTOR8BITBYTES</td>
</tr>
</tbody>
</table>

Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System
Program Memory (Rule 22)

<table>
<thead>
<tr>
<th>Code Sections</th>
<th>Code</th>
<th>Size (bytes)</th>
<th>Align (MAUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:filter</td>
<td></td>
<td>0x74 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:control</td>
<td></td>
<td>0x29 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algAlloc</td>
<td></td>
<td>0x56 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algInit</td>
<td></td>
<td>0x3a * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algFree</td>
<td></td>
<td>0x39 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algActivate</td>
<td></td>
<td>0x15 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algMoved</td>
<td></td>
<td>0x18 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algDeactivate</td>
<td></td>
<td>0x15 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:algNumAlloc</td>
<td></td>
<td>0x2 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:init</td>
<td></td>
<td>0x1 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
<tr>
<td>.text:exit</td>
<td></td>
<td>0x1 * FACTOR8BITBYTES</td>
<td>0</td>
</tr>
</tbody>
</table>

Interrupt Latency (Rule 23)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Typical Call Frequency (microsecond)</th>
<th>Worst-case Instruction Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter()</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>control()</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>

Period / Execution Time (Rule 24)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Typical Call Frequency (microsecond)</th>
<th>Worst-case Cycles/Period</th>
<th>Worst-case Cycles/Period</th>
<th>Worst-case Cycles/Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter()</td>
<td>N/A</td>
<td>24446</td>
<td>No periodic execution</td>
<td>No periodic execution</td>
</tr>
<tr>
<td>control()</td>
<td>N/A</td>
<td>200</td>
<td>No periodic execution</td>
<td>No periodic execution</td>
</tr>
</tbody>
</table>

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 18: This algorithm does not include definitions specific to a debug variant.

• 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 (c55x only).

• (NOTE: All OBJECT files were compiled with [–ml] option to satisfy Rule32) (c55x only)

• Rule 33: This algorithm operates properly with program memory operated in cache mode (c55x only).

• IDMA Rule 1: All data transfers are completed before return to the caller.
Appendix C: Issues Reference

For the latest information regarding issues you may encounter, see the Release Notes provided in the Reference Frameworks area of the DSPvillage website (www.dspvillage.com).

It is recommended that you have the latest version of the CCStudio software, as it may contain important features or problem fixes. To use the Update Advisor, choose Help → CCS on the Web → Update Advisor and follow the instructions for downloading.

The following product issues are referenced in this document:

1. The DATA_ALIGN pragma is not supported on the 'C54x in CCStudio 2.1. It will be supported in CCStudio 2.2. Although the statements generate warnings when you compile, the statements have been left in because they are the best solution moving forward. (SDSsq20538)

   For the 'C54x, the linker command file for the rf1.pjt project forces the alignment requested by the algorithm and initializes the first buffer to zero as required by the preconditions for the XDAIS algInit() function. This workaround can be removed once CCStudio 2.2 is available and installed. Both the 'C55x and the 'C6000 support the DATA_ALIGN pragma.

2. The genbufs technique depends upon an enhancement in CCS 2.0 that allows the LOG_printf() output to be saved to a file and an enhancement in the CCS 2.00.03 Service Pack that allows LOG sequence numbers to be disabled.

3. The ability to overlay CSL run-once code depends upon an enhancement in the CCStudio 2.00.03 service pack. (SDSsq20805)

4. An enhancement request (SDSsq23223) has been filed for 'C5000 to eliminate remaining dead code references that might cause the HWI dispatcher to be linked in when it is not actually used.

5. The LOG_system object cannot be removed because the drivers explicitly call LOG_message() under certain conditions. LOG_message is a macro that calls LOG_event(), which uses the LOG_system object.

Issues in the following list may have been resolved in CCStudio versions or service packs released after the publication of this application note.

### Table 18. Fixes and Enhancement Request Summary

<table>
<thead>
<tr>
<th>ID Number</th>
<th>Description</th>
<th>Entered Against</th>
<th>Fixed In</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSsq20013</td>
<td>Make sequence numbers in LOG removable.</td>
<td>CCStudio 2.0</td>
<td>CCStudio 2.00.03</td>
</tr>
<tr>
<td>SDSsq20389</td>
<td>The HWI module made an assembly code reference to the SWI_F_run function. As a result, some SWI code was linked in with executables even if no SWI or PRD object existed.</td>
<td>CCStudio 2.0</td>
<td>CCStudio 2.1</td>
</tr>
<tr>
<td>SDSsq20536</td>
<td>Message log output file was not released until the log display window was closed.</td>
<td>CCStudio 2.0</td>
<td>CCStudio 2.1</td>
</tr>
<tr>
<td>SDSsq20538</td>
<td>Request support for DATA_ALIGN pragma for 'C54x.</td>
<td>CCStudio 2.0</td>
<td>CCStudio 2.2</td>
</tr>
<tr>
<td>SDSsq20805</td>
<td>Allow CSL initialization code to be overlaid.</td>
<td>CCStudio 2.0</td>
<td>CCStudio 2.00.03</td>
</tr>
<tr>
<td>SDSsq23223</td>
<td>Remove remaining reference to the HWI dispatcher if no HWIs use the dispatcher.</td>
<td>CCStudio 2.0</td>
<td>--</td>
</tr>
</tbody>
</table>
Appendix D: Reference Framework Board Ports

As of the publication date, the board-specific portions of RF1 are provided for the 'C5402 DSK board, which is available as part of the 'C5402 DSP Starter Kit. However, a number of additional device controllers are provided with the Reference Frameworks code distribution. The controllers provided as of the publication date of this application note are as follows:

- 'C5402 DSK, which is available as part of the 'C5402 DSP Starter Kit. The board features a 'C5402 DSP, 64K of external RAM, and an AD50 mono codec.
- 'C5402 DSK board with a daughterboard containing the AIC23 stereo codec.
- 'C5416 LEIA board with the PCM3002 stereo codec.
- 'C6x11 DSK board with a AD535 mono codec.

Additional device controllers may be added in the future. See the readme.txt files in the Reference Frameworks folder tree for updated information.

For details about how to implement device controllers, see the Writing DSP/BIOS Device Drivers for Block I/O (SPRA802) application note.

Reference Framework source code and configuration files refer directly to the driver’s initialization, setup, and ISR functions. It is the job of the PLIO adapter to use LIO 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 RF1 to a new target, other changes are required in addition to creating or using a different device controller. For example, a configuration file must be created for the target with appropriate objects and properties. And, code in the target-specific folder for the framework requires some changes.
IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated