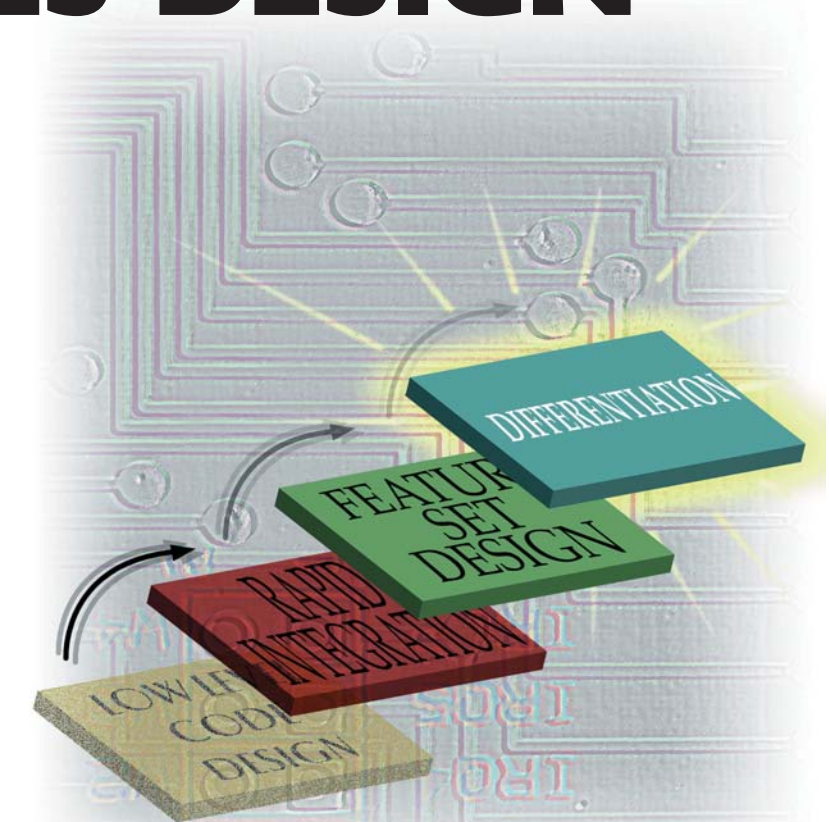


# REFERENCE FRAMEWORK SLASHES DESIGN TIME

**The right commercial framework not only saves time in building applications but provides flexibility in shaping the design.**

*By Steve Poulsen*



**A**ttention embedded software solution providers: You can provide your customers with reference designs without duplicating framework development for each algorithm combination. How? By going to a ready-made production-quality framework. If you select one that's designed for easy customization, you can focus on integration—not framework development—and save substantial design and test time.

Be cautious, however. Not all frameworks can accommodate the required customization, nor will designs necessarily be reliable if the framework is forced to provide functionality outside its original design criterion. If that happens, significant problems can occur later during integration or customer support. Bottom line: Make sure the framework is a positive contributor to the solution, not a hindrance.

Our company, Imagine Technology, reaped many benefits by taking advantage of the Texas Instruments eXpressDSP Reference Framework (RF). In fact, using the RF slashed the development time of a production solution by 75 percent. Here's how we did it:

During the development of an MP3 TCP/IP server solution, it became apparent that we required a framework that was inherently flexible and practical. Typically, designs are based on a streaming codec interface, and the RF provides several codec drivers for that purpose. Support for TCP/IP also was essential, and the RF comes through here also.

In practice, our designers quickly converted the RF design (level 3) to their MP3 encode/decode solution, using only a codec LIO driver. Next, they integrated the TCP/IP into the application and created an LIO driver for TCP/IP that appears like a codec to the application. This enabled two sources for data (codec and TCP/IP) and two sinks (codec and TCP/IP). The final step was to implement the TCP/IP control thread, which allows all combinations of source/sinks to be executed in real time. To understand the process of creating reference designs, let's focus on the TCP/IP LIO driver and control thread. After developing our MP3 algorithm technology, we tapped the RF to quickly develop complete MP3 solutions that could be demonstrated to customers or serve as production quality designs. Integrating Imagine's MP3 encode/decode technology and the RF yielded an MP3 server solution and demonstration kit with robust and tested components, such as the framework, TCP/IP stack and codec support.

The first step in the integration process defines the base control structure, or framework. The framework

allows components to be quickly integrated into a system that manages system resources.

The logical solution is a ready-made framework, designed for easy customization, so we could focus on integration, not framework development. A production quality framework also reduces the amount of testing required. TI's RF satisfies these requirements and, to meet varying complexity needs, supports several levels of integration. We selected Level 3 (RF3) as the baseline for the MP3 server solution because of its built-in multi-channel support, and dynamic algorithm creation capabilities.

The RF3 package centers around a codec driver, associated with an audio thread. In addition, we wanted support for TCP/IP packets, MP3 encode/decode of bitstreams, audio input/output back to TCP/IP and codecs.

RF3 aids in the customizations, which included replacement of the demo algorithms and codec drivers by ours. The first step is to copy the RF3 demo, then replace the FIR and VOL algorithms with our MP3 encoder and decoder algorithms.

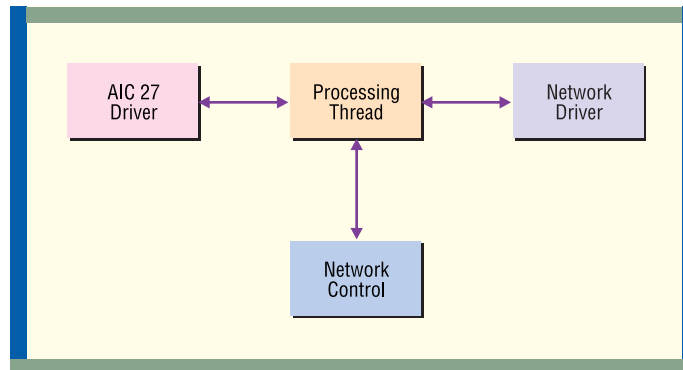
Note that we did not replace the audio processing

thread (`thrAudioproc.c`) with our algorithm processing, which we left for after the integration of all drivers and data feeds. Instead, we removed the `thrAudioproc.c` processing code, except for the pipe processing. Then, we added memcpy from `src` to `dst` to provide for pass through audio. At this point, the solution could create algorithms, but passed audio straight through. The code could be built and tested to verify that the solution was stable and no problems were created.

The next step was to replace the codec from the RF3 demo with Imagine's LIO codec driver, which interfaces our board and the demo. The board transforms the demo into a system capable of recording and playing high-quality stereo audio. In order to replace the driver, we modified the `link.cmd`, replacing:

```
-l dsk6x11_edma_ad535.l62
```

```
with
```



*An MP3 server solution based on a commercial reference framework can accept input audio from a network, via TCP/IP, or a codec*

```
-l dsk6711_edma_aic23.l67
```

This is simply another codec; its development is not unique and therefore is not detailed in this article. In order to initialize the new codec properly, we changed the `applO.c` so that `applOInit` looks like:

```
Void applOInit()
{
/*
 * Initialization the of LIO/PLIO driver:
 * Call the driver's init function, set driver parameters in a copy
 * of the global driver setup data object, call the driver's setup()
 * function with the local setup data object. (If the default setup
 * parameters are OK, NULL can be passed to the setup() func-
tion.
 */
DSK6711_DMA_AIC23_init();

/* now call the driver setup function */
DSK6711_DMA_AIC23_setup( NULL );

/* Initialize PLIO:
```

The PLIO initialization is not shown above.

RF3 provides for splitting and joining of stereo data by default, and posting SWIs to handle each channel independently. For MP3 encoding and decoding, we needed the entire signal, which entails removing the split and join code, splitting and joining SWIs from the DSP/BIOS configuration tool, removing the split and join pipes, and modifying the tx and rx pipes so that they post a SWI to one audio processing thread. (All of these items are detailed in the RF3 documentation.)

In order to integrate TCP/IP into the MP3 solution, we developed a method to convert the driver for network usage. The standard LIO codec drivers for RF3 support the ability to source and/or sink audio samples. Audio samples from codecs are transferred continuously at a periodic rate. For TCP/IP, packets are received sporadically, containing either audio samples or compressed bitstream data. Also, header and mode information are attached to each packet.

Our solution required the ability to send and receive MP3 bitstreams, audio samples, statistics, and control information via such a driver, while using the LIO driver interface. In addition, the TCP/IP driver is expected to run as a DSP/BIOS task, not interrupt driven. Although this difference is minor, it affects the configuration. We concluded that placing the TCP/IP driver in this model would not be straightforward and could

require custom interfaces, something we wanted to avoid, if possible.

The TCP/IP driver required the ability to handle several input and output combinations. Also, the MP3 server solution can accept input audio from a network or codec and optionally encode or decode, then send the output either out through the codec or back to the host via the network. The server also can receive network compressed data bitstreams and decode and send the output back to the host or through the codec. (See Fig. 1.)

The execution level of the TCP/IP driver was the first concern during development. Codec drivers are typically activated by an interrupt, and TCP/IP by a background task. The configuration tool was used to create a task and tie it to the driver task—similar to the codec method of assigning an ISR via the configuration tool. The driver was developed based on this model, requiring one task to be setup. The main task contains all the code to initialize the TCP/IP stack and several network tasks. These new tasks handle receive packets and send packets. Additionally, we investigated the possibility of not requiring any DSP/BIOS configuration by auto creating our tasks during initialization.

The main task creates two sub tasks, consisting of a receive task and a send task. The receive task handles the entire network interface for incoming packets. This task creates a listening network socket that waits for a host connection. Once this connection is received, it triggers the send task to open a connection back to the host via a DSP/BIOS mailbox. The internal state machine consists of packet send and receive tasks. The send/receive packets serve as the LIO interface data.

We decided on the following packet structure to be used for all TCP/IP packets:

```
#define DATA_PACKET_SIZE 1152*2
typedef struct
{
    unsigned int length;
    unsigned int options;
    unsigned int startTicks;
    unsigned int endTicks;
    short sampleRate;
    short data[DATA_PACKET_SIZE];
} DATA_PACKET;
```

This structure required our pipes to be of appropriate size to handle such large packets. Thus, we could send one whole MP3 frame of audio to be encoded and

avoid the need to merge packets. Because RF3 uses the PLIO library to provide a link between a LIO driver and DSP/BIOS pipes, the flow becomes: TCP/IP driver receive task listens for a connection; connection received, receive task signals transmit task and waits for a packet; transmit task creates connection back to the host; receive task places received packets into the LIO submitted buffers; transmit task send packets based on LIO submitted buffers; and the audio thread handles modes and packet information.

The driver is now designed to send packets via the LIO interface. The PLIO (piped LIO interface) mechanism is kept in place, and the audio processing thread is modified to handle the packets. The main issue with the thread is that the current design requires it to be driven by a full input buffer and empty output buffer. When both conditions are met, the thread is called. However, it is more appropriate for it to be called under two conditions, input packet available or input codec audio available and output codec buffer available.

To meet this requirement, we set up two SWIs, one to handle the input packets, the other, the codec. Both SWIs were set to execute the same processing thread, so the thread was required to check which condition occurred. The processing thread was designed to promptly exit if no packet was received to set the mode. Once the packet was received, the mode would be set. This mode supports the processing until the mode is changed. The valid modes of operation are.

```
Net input » encode » net output.
Net input » encode » decode » net output.
Net input » decode » net output.
Net input » encode » decode » codec output.
Net input » decode » codec output
Codec input » encode » net output.
Codec input » encode » decode » net output.
Codec input » encode » decode » codec output.
Codec input » codec output.
Net input » net input.
```

We developed a mechanism to handle all of these cases. For each, various Booleans are set, based upon the input options, then the pipes associated with the mode are processed. This occasionally causes the thread to exit whenever the proper data is not available, especially when the mode signifies use of both the codec and the network. The following pseudocode simplifies the actual code used to perform the processing:

```
if (packetReady())
```

```
{
    getPacket();
    setPacketGlobal();
}

if (!codecDataReady && (codecInput || codecOutput))
    return;

/* Input switch */

if (codecInput)
    pOutput = pInput;
else if (packetInput)
    pOutput = pPacketInput;

if (encode)
{
    if (codecInput)
        enclInput = pCodecInput;
    else if (packetInput)
        enclInput = pPacketInput;
```

## Need to get started on DSP eXtra quick?

Get started NOW with the  
FREE CD enclosed in this issue:

### The Essential Guide to Getting Started with DSP



Install the CD to:

- *eXplore: eXpressDSP™ Guided Tour and DSP Platform Presentation*
- *eXperience: Free 90-Day Trial Code Composer Studio™ IDE*
- *eXciting: 30% Discount Offer on full-featured Code Composer Studio*

Also available at:

[www.dspvillage.ti.com/freetools](http://www.dspvillage.ti.com/freetools)

```

        mp3Encode(encInput, encOutput);

        pOutput = encOutput;
    }

    if (decode)
    {
        if (encode)
            decInput = encOutput;
        else if (packetInput)
            decInput = pPacketInput;

        mp3Decode(decInput, decOutput);

        pOutput = decOutput;
    }

    /* Output switch */
    if (codecOutput)
        pCodecOutput = pOutput;
    else if (packetOutput)
        pPacketOutput = pOutput;

```

The next step is to integrate the driver and the thread. We added the new driver library to the link process (link.cmd) and initialized in applO.c. Additionally, we needed two new pipes linked with the PLIO. First, we created a pipRxNet pipe and a pipTxNet pipe. The sizes are set to be 2304 16-bit words (1152 32-bit words) and two deep. The notifyReader of pipRxNet can be configured to post a new SWI by using a value of 0x1 in the and operation. The notifyReader of pipTxNet was configured to post the same SWI with a value of 0x1, and the new SWI is set to require a mask of 0x1 to be triggered and called by the same audio thread as the codec. Next, we modified the link.cmd by adding -l dsk6711\_tcpip.l67 after the CODEC library line. Finally, we modified the applO.c as follows:

```

Void applOInit()
{
    /*
    * Initialization the of LIO/PLIO driver:
    * Call the driver's init function, set driver parameters in a copy
    * of the global driver setup data object, call the driver's setup()
    * function with the local setup data object. (If the default setup
    * parameters are OK, NULL can be passed to the setup() function.
    */
    DSK6711_DMA_AIC23_init());

```

```

    /* now call the driver setup function */
    DSK6711_DMA_AIC23_setup( NULL );

    /* Initialize PLIO:
    * PLIO_new initializes a PLIO object with the following arguments:
    * 1. address of the PLIO object
    * 2. handle of the associated pipe (pipRx for input and pipTx for output)
    * 3. mode: input or output
    * 4. address of the function table for the low-level driver
    * 5. optional generic arguments
    */
    PLIO_new( &plioRx, &pipRx, LIO_INPUT,
    &DSK6711_DMA_AIC23_ILIO, NULL );
    PLIO_new( &plioTx, &pipTx, LIO_OUTPUT,
    &DSK6711_DMA_AIC23_ILIO, NULL );

    PLIO_new( &plioRxNet, &pipRxNet, LIO_INPUT,
    &DSK6711_TCPIP_ILIO, NULL );
    PLIO_new( &plioTxNet, &pipTxNet, LIO_OUTPUT,
    &DSK6711_TCPIP_ILIO, NULL );
}

```

This allows initialization of the TCP/IP driver and links it with the new pipes. PLIO and the LIO driver handle the pipes, and the SWI is automatically called when the pipRxNet contains data, or when space becomes available in pipRxNet

Just above applOInit() the following two definitions should be added:

```

PLIO_Obj plioRxNet;
PLIO_Obj plioTxNet;

```

The final piece of the project is the host application to drive the process. For that, we used Visual C++ 6.0 and socket programming to build a network GUI application that allows a user to select the input, output, processing, and necessary files. The user can then perform all the designed operations from a simple Windows application. Customers can evaluate the product and use it as a reference design. ◆

Steve Poulsen is the Director of Engineering at Imagine Technology, Lincoln, Nebraska. Mr. Poulsen has over 8 years experience in electrical engineering and holds patents in signal processing for voice and data encryption in a wireless system. His signal processing expertise includes telephony, modem design, audio algorithms, echo cancellation and simulation of CDMA algorithms, with an emphasis in code-tracking loops.