TEXAS INSTRUMENTS

Application Report
SPRA787A – August 2002

Using an LIO-Based UART Driver in an RF3 Application

Arnie Reynoso
Laurent Egu
Texas Instruments, Santa Barbara

ABSTRACT

This application note introduces and describes a UART device controller for use with the 'C5402 DSK board. The 'C5402 DSK has an on-board TL16C550C UART (Universal Asynchronous Receiver and Transmitter). A UART provides a serial asynchronous interface to the DSP. It can be used with the 9-pin connector on the board to create a serial interface, such as RS232, to the board.

This UART controller implements the LIO (Low-level I/O) interface and uses the PLIO device adapter described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note. The application associated with this application note demonstrates the benefit of using a Reference Framework and the DSP/BIOS Low Level I/O driver (LIO) software.

The main challenge in this application is to use two very different transmission channels in the same data flow. The codec streams continuous blocks of data with tight real-time deadlines. In contrast, the UART sends blocks of data discontinuously with more flexible deadlines.

This document also describes a number of techniques that may be applied to other applications. These techniques include: running an application on one or two boards, priming applications that may run on separate boards, adapting the Reference Framework Level 3 (RF3) application, and using the G.726 encode/decode algorithms.

Contents

1 Introduction 3
1.1 About the UART 4
1.2 About the Application 4
2 Setting Up the Application Example 6
2.1 Preparing the Software 6
2.2 Single-Board Demo 7
2.2.1 Preparing the Hardware for the Single-Board Demo 7
2.2.2 Building and Running the Single-Board Application 8
2.3 Two Board Demo 8
2.3.1 Connecting the Boards for the Two-Board Demo 9
2.3.2 Connecting Two Boards to Each Other 9
2.3.3 Using CCStudio Setup for Multiple Boards 9
2.3.4 Building and Running the Two-Board Application 10
3 Adapting RF3 to the RF3_UART_G726 Application 11
3.1 Application Data Sizes 11
3.2 RF3_UART_G726 Application Data Path 12
3.3 Reference Framework Level 3 Data Path 12

3.4 Removing Unnecessary Components 12
3.4.1 Changing Object Names 13
3.4.2 Adding the G.726 Encoder/Decoder to Reduce the Data Flow 14
3.4.3 Adding the UART to the Data Path 17
4 Application Priming 18
4.1 The Encoder 19
4.2 Priming the Encoder 20
4.3 The Decoder 22
4.4 Priming the Decoder 22
4.5 Impact of the Priming Strategy on the Global System 24
4.6 Additional Considerations 25
5 UART LIO Device Controller 26
5.1 UART Controller Modules 26
5.2 DSK5402_UART Controller Module 27
5.2.1 UART Controller Channel Object 28
5.2.2 The DSK5402_UART_setup() Function 29
5.2.3 The open() Function 30
5.2.4 The submit() Function 30
5.2.5 The ISR Functions 32
5.3 Circular Buffer Module (CIRC) 33
5.3.1 Circular Buffer Object 34
5.3.2 The CIRC_new() Function 35
5.3.3 The CIRC_readChar() and CIRC_writeChar() Functions 35
5.3.4 The CIRC_writeBuf() Function 36
5.3.5 The CIRC_readBuf() Function 36
5.4 UART Module 37
5.4.1 UART Attribute Structure 38
5.4.2 The UART_setup() Function 38
5.4.3 The UART_enable/disableRx/Tx Functions 38
5.4.4 The UART_readChar() Function 38
5.4.5 The UART_writeChar() Function 39
5.4.6 The UART_txEmpty() Function 39
5.4.7 The UART_rxFull() Function 39
5.4.8 The UART_clearInt() Function 39
References 39
Appendix A: Connectors and Cables 40

Figures

Figure 1. Data Path 4
Figure 2. Two-Board Approach 5
Figure 3. Loopback Approach 5
Figure 4. Directory Structure 6
Figure 5. Loopback Demo 7
Figure 6. Two Board Demo 8
Figure 7. Sample and Buffer Size Compression 11
Figure 8. Application Data Path with Buffer Lengths 12
Figure 9. RF3 Data Path 12
Figure 10. Data Path for Simplified Version of RF3 13
Figure 11. Data Path After Name Changes 14

Figure 12. Data Path with Added SWI and PIP Objects ... 14
Figure 13. Properties for swiDecode Object ... 15
Figure 14. Properties for pipLink Object ... 15
Figure 15. Switching Algorithms ... 16
Figure 16. swiEncode and swiDecode Buffer Size Reduction ... 16
Figure 17. Final Application Data Path ... 18
Figure 18. Priming in the Default RF3 Application ... 18
Figure 19. Application Execution Graph with RF3 Basic Priming ... 19
Figure 20. Encoder Application ... 19
Figure 21. UART Output Delayed ... 20
Figure 22. Decoder Application ... 22
Figure 23. Decoder Primed with One Buffer of Silence ... 22
Figure 24. Encoder and Decoder Running on Two Boards ... 24
Figure 25. Higher-Priority Thread Can Break the Application ... 25
Figure 26. The DSP/BIOS Device Driver Model ... 26
Figure 27. UART Controller Functions ... 27
Figure 28. Circular Buffer Used by UART Controller ... 33
Figure 29. DB9 Pin Assignments ... 40
Figure 30. DB9 Loopback Connector ... 40
Figure 31. Null Modem Cable with Partial Handshaking ... 40
Figure 32. Null Modem Cable with Full Handshaking ... 40

Tables

Table 1. PLIO, PIP, and SWI Name Changes ... 13
Table 2. Application's Pipe Configuration Parameters ... 17

1 Introduction

The primary motivation for this application is to introduce and explain a UART device controller, which implements the LIO (Low-level I/O) interface as described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note. This application further demonstrates the benefit of using a Reference Framework and the DSP/BIOS Low Level I/O driver (LIO) software.

However, this document also describes the following techniques used by the accompanying application. One or more of these techniques may be applied to your application, depending on its requirements.

- **Asynchronous and synchronous streaming in the same application.** The main challenge raised by this application is to include two very different transmission channels in the same data flow. The codec streams continuous blocks of data with tight real-time deadlines. In contrast, the UART sends blocks of data discontinuously with more flexible deadlines. Basically, the codec stream is synchronous and the UART stream is asynchronous.

- **Modularity for running on one or two boards.** While the two-board approach may match the application requirements, the ability to fully test the application on a single board with a loopback connector simplifies debugging.

- **Priming considerations for two boards.** The priming techniques used in this application solve the synchronous vs. asynchronous and modularization challenges presented by the application.
- **Reference Framework Level 3 adaptation.** Considering the needs of this application—a single channel and two algorithms—as well as the need for DSP/BIOS real-time analysis capabilities, this application is based on Reference Framework Level 3 (RF3). In this case, the application is modified to have a single channel and no control channel. For details about this framework, see the *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793) application note.
- **G.726 encode/decode algorithms.** The application uses the G.726 algorithms provided with the XDAIS component of Code Composer Studio. These are not optimized algorithms and should not be used in a production application. These "performance-detuned" algorithms are provided as an example. Because the G.726 algorithm is used, as well as for performance reasons, the application requires a block-based data path.

1.1 About the UART

The 'C5402 DSK has an on-board TL16C550C UART (Universal Asynchronous Receiver and Transmitter). A UART provides a serial asynchronous interface to the DSP. It is responsible for correctly formatting the data for transmission and decoding it on reception. It performs parallel-to-serial conversion on data received from the 'C5402 DSP and serial-to-parallel conversions on data received from a peripheral device or modem. It can be used with the 9-pin connector on the board to create a serial interface, such as RS232, to the board.

By default, the controller sets the UART to operate at 115.2 kbps (115200 baud) and to use a serial character word length of 8-bits, one stop bit with parity checking and loopback control disabled. However, the UART parameters used by the controller can easily be modified.

1.2 About the Application

The application associated with this document is built for the 'C5402 DSK, and has the data flow shown in Figure 1 and described in the list that follows.

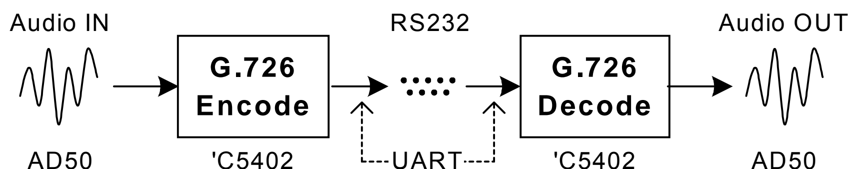

Figure 1. Data Path

1. Audio is acquired using the AD50 mono codec with a sample rate of 8 KHz.
2. The audio is encoded with a G.726 speech coder algorithm.
3. The encoded audio is sent through via the UART interface over a RS232 connection.

4. On the other end of the RS232 connection, the data is received by the UART. (This application may be run on one or two boards. When run a single-board, the same UART is uses simultaneously for input and output.
5. The signal is decoded with a G.726 vocoder algorithm.
6. The decoded signal is sent to the output codec.

For modularity, we divide the data flow in two halves, the *encoder* and the *decoder*. The encoder consists of the audio signal acquisition, the G.726 encoder and data transmission through the UART. The decoder reads the data from the UART before decoding it and playing it back.

This application can be approached in two different ways:

- **Two-board approach.** The approach shown in Figure 2 uses two separate 'C5402 DSK boards connected by a serial cable; one board acts as the encoder and the other as the decoder.

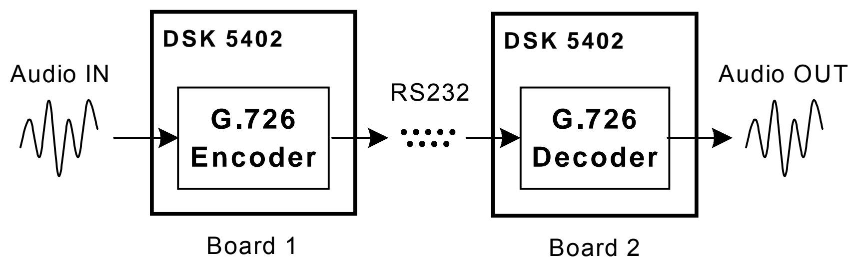

Figure 2. Two-Board Approach

- **Loopback approach.** The approach shown in Figure 3 runs both the encoder and decoder on the same DSP. Data sent through the UART loops back into the same board via a loopback connector. This approach is easier to test and debug.

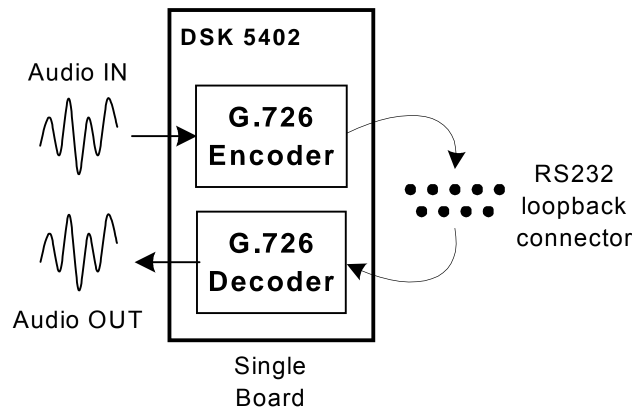

Figure 3. Loopback Approach

The sections that follow focus on the loopback approach. However, switching to the two-board approach is straightforward as long as the encoder and decoder are completely independent of each other.

2 Setting Up the Application Example

The RF3_UART_G726 application is packaged as an addition to the Reference Framework software package. Before you install the RF3_UART_G726 application, first download and install the Reference Framework source code. See the SPRA793 application note for details. You can obtain the source code package and application note regarding Reference Frameworks by going to www.dspvillage.com and following the link to "Reference Frameworks".

2.1 Preparing the Software

Download the spra787.zip file and extract it at the same location where you extracted the Reference Framework zip file. The following files and folders are created when you extract from the spra787.zip file:

- The RF_DIR\apps\RF3_UART_G726 folder is created. It contains the application source.
- The RF_DIR\src\dsk5402uart folder is created. It contains the UART driver source code.
- The UART driver header files are added to the RF_DIR\include folder.
- The UART driver library is added to the RF_DIR\lib folder.

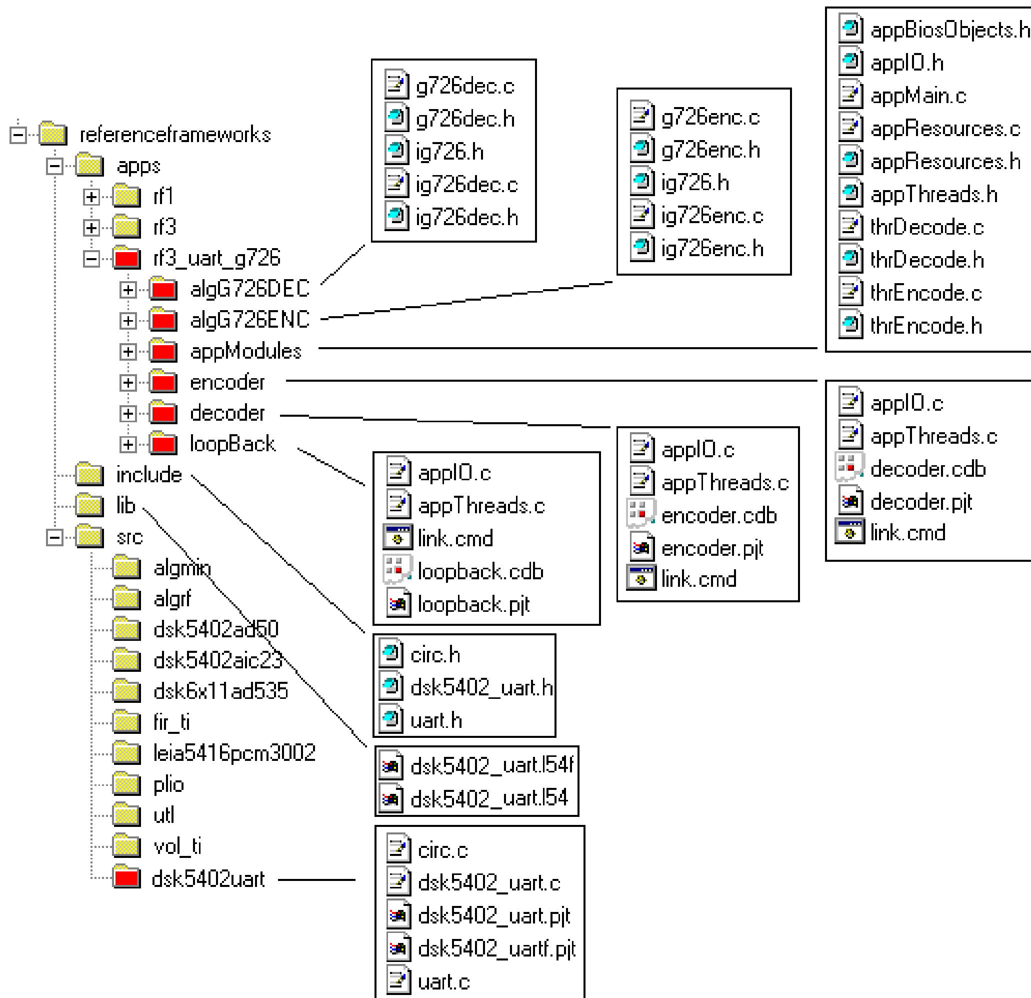

Figure 4. Directory Structure

2.2 Single-Board Demo

This section describes how to set up and run the RF3_UART_G726 application on a single 'C5402 DSK board.

For this approach, you will need:

- One 'C5402 DSP Starter Kit
- One DB9 female loopback connector wired according to the schematic in *Appendix A: Connectors* and Cables.

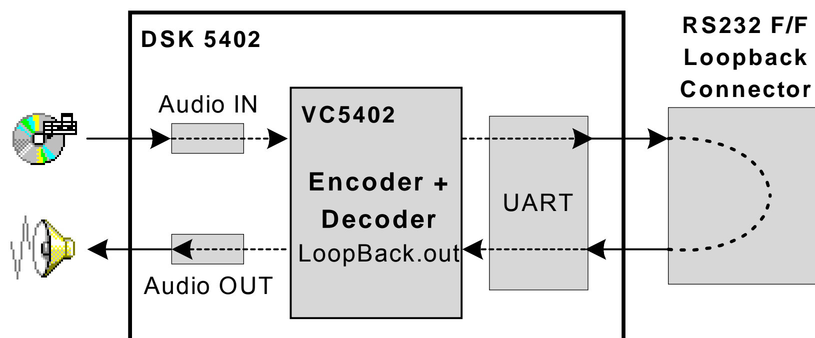

Figure 5. Loopback Demo

2.2.1 Preparing the Hardware for the Single-Board Demo

First install the 'C5402 DSP Starter Kit as described in the documentation provided with the board. The following steps provide an overview of how to connect the hardware. For details and diagrams, see the documentation provided with your board.

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. Plug the DB9 female loopback connector into the DSK's RS232 connector.
5. Connect an audio input device such as the headphone output of a CD player to the audio input jack on the board. You can also connect the audio output of your PC sound card to the audio input of the board.
6. Connect a speaker or other audio output device to the audio output port of the board.
7. Plug the power cable into the board.
8. Plug the other end of the power cable into a power outlet.
9. Start the PC.

2.2.2 Building and Running the Single-Board Application

Follow these steps to build, run, and test the application:

1. Within Code Composer Studio, choose Project→Open, and select the loopback.pjt in RF_DIR\apps\rf3_uart_g726\loopback.

   The library search path in the project file points to G.726 algorithms provided with CCStudio. If you installed CCStudio in a directory other than the default (c:\ti), modify the library search path accordingly so that the build can succeed.

2. Choose Project→Rebuild All to build the application

3. Choose File→Load Program and load the loopback.out file in the Debug subdirectory.

4. Start your CD player or other audio input.

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

6. Choose DSP/BIOS→Statistics View. In the Statistics View, examine the frame transfer counts for the pipRxCodec, pipTxCodec, pipRxUart, and pipTxUart pipe objects.

7. Disconnect the DB9 loopback connector, and notice that the music stops until you reconnect it.

2.3 Two Board Demo

This section describes how to set up and run the RF3_UART_G726 application on two 'C5402 DSK boards.

For this approach, you will need:

- Two 'C5402 DSP Starter Kits
- One null modem female-to-female DB9 cable wired according to the schematic in *Appendix A: Connectors* and Cables.
- One or two PCs to control the application

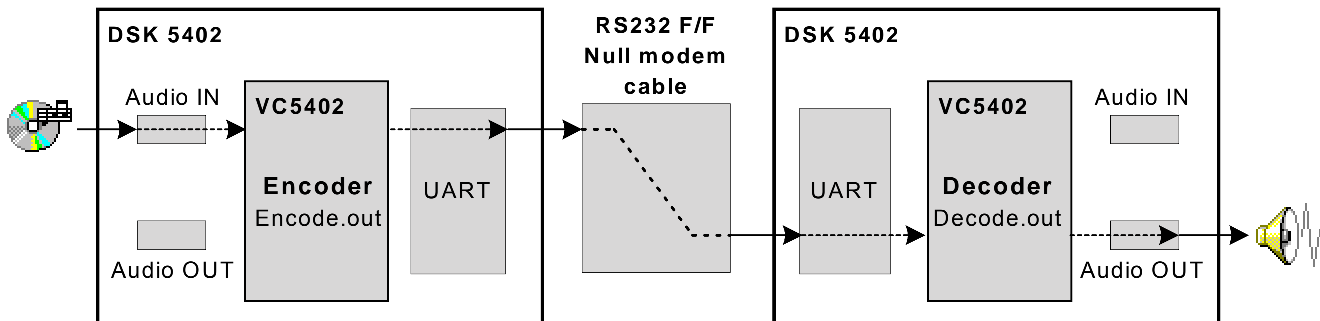

Figure 6. Two Board Demo

2.3.1 Connecting the Boards for the Two-Board Demo

Choose one of the following system configurations to connect the boards to a PC or PCs that will control the boards:

- **Two computers with one board connected to each of them.**

  Follow the steps in Section 2.2.1 for each computer. Refer to the documentation provided with your boards for setup instructions and troubleshooting. In this case, you will use Code Composer Studio independently on both computers.

- **One computer with two boards connected via two different emulators.**

  For example, connect one board through the parallel port and the other through the TI XDS510 emulator connected to an ISA card. (The Spectrum Digital XDS510pp emulator can't be used for the second board since it uses the parallel port as well.) In this case, you will use Code Composer Studio in parallel debugging mode.

- **One computer with two boards connected through the same emulator via an emulator splitter.**

  You can obtain the emulator splitter from Spectrum Digital through TI's third-party network. In this case, you will use Code Composer Studio in parallel debugging mode.

2.3.2 Connecting Two Boards to Each Other

After you have connected the boards to PCs, follow these steps:

1. Connect the two boards using a null modem female-to-female DB9 cable.
2. Connect an audio source to the input line of one of your boards (the encoder)
3. Connect speakers to the output line of the other board (the decoder).

2.3.3 Using CCStudio Setup for Multiple Boards

If you are using a single computer connected to two boards, follows these steps to set up and run Code Composer Studio:

1. Open CCStudio Setup.
2. Import configurations for the boards and emulators you are using.
   – **Different emulators.** For example, you might import the "C5402 DSK via XDS510 Emulator" and "C54x Parallel Port Emulator (DSK5402)" configurations.
   – **Same emulator via a splitter.** Select the driver corresponding to your emulator. Open the properties for the board. In the Processor Configuration tab, add a second CPU.
3. Save your configuration and close CCStudio Setup.
4. Start Code Composer Studio. You will see Parallel Debug Manager window.
5. In the Parallel Debug Manager window, choose from the Open menu twice to open a separate instance of CCStudio for each of your two DSK boards.

2.3.4 Building and Running the Two-Board Application

Follow these steps to build, run, and test the application:

You should have two instances of Code Composer Studio opened: one running on the encoder board and the other on the decoder board. These two windows may be on the same or different PCs, depending on the system configuration you are using.

NOTE: You must start the Decoder program before the Encoder program. The UART initialization will fail if the UART receives data before the Decoder program initializes it.

In the CCStudio window for the decoder board:

1. Choose Project→Open and select the decode.pjt project in RF_DIR\apps\rf3_uart_g726\decode.

   The library search path in the project file points to G.726 algorithms provided with CCStudio. If you installed CCStudio in a directory other than the default (c:\ti), modify the library search path accordingly so that the build can succeed.

2. Choose Project→Build to build the application.
3. Choose File→Load Program and load the decode.out file in the Debug subdirectory.
4. Choose Debug→Run.

In the CCStudio window for the encoder board:

1. Choose Project→Open and select the encode.pjt project in RF_DIR\apps\rf3_uart_g726\encode.
2. Choose Project→Build to build the application.
3. Choose File→Load Program and load the encode.out file in the debug subdirectory.
4. Start your CD player or other audio input.
5. Choose Debug→Run. You should hear the audio output through the speakers connected to the target board.
6. Choose DSP/BIOS→Statistics View. In the Statistics View, examine the frame transfer counts for the pipRxCodec, pipTxCodec, pipRxUart, and pipTxUart pipe objects.
7. Disconnect the female-to-female DB9 cable, and notice that the music stops until you reconnect it.

3 Adapting RF3 to the RF3_UART_G726 Application

This section explains how the application differs from the default RF3 application. Detailed steps for adapting the RF3 application are not provided because the adaptation work has already been done. To see the detailed changes, refer to the source code provided.

To simplify the application, in this section we'll consider only the case where the entire application runs on a single board (the loopback approach). The loopback version of the application is provided in RF_DIR\apps\rf3_uart_g726\loopBack. This simplification allows us to use the default priming method provided with RF3. The priming method developed in Section 4, Application Priming, on page 18, allows the application to function when running on two different boards.

3.1 Application Data Sizes

Data comes from the AD50 codec on the 'C5402 DSK with a sample rate of 8 kHz. This is the rate expected by the G.726 speech coding algorithm, which is widely used for digital transfer of audio. The codec provides 16-bit resolution per sample.

The G.726 algorithm accepts and returns a buffer of 80 samples of 16 bits per sample. However, it encodes only the 14 most significant bits of each sample. In this application, the G.726 vocoder is set to its maximum compression ratio (7:1 ratio, 16 kbps). That is, it accepts 14-bit samples of linear data and returns samples of 2 bits. To summarize, the 16-bit samples coming from the codec are reduced and compressed down to 2-bit samples by the G.726 vocoder.

To minimize the amount of data sent over the UART, the application packs the bits that contain data into a smaller buffer of 20 samples of 8 bits (4 * 2-bit samples). The UART will be configured to transfer 8-bit samples. Figure 7 shows the sample and buffer sizes used by the encoding portion of the application. To unpack the buffers, the decoding portion of the application reverses the process.

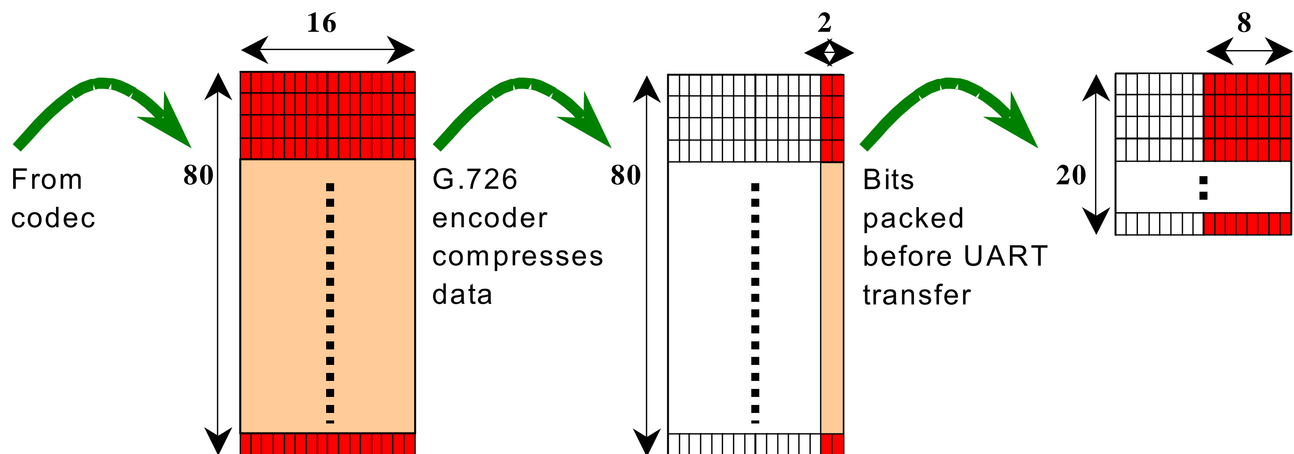

Figure 7. Sample and Buffer Size Compression

3.2 RF3_UART_G726 Application Data Path

Figure 8 modifies the simple data path shown in Figure 1 by specifying the DSP/BIOS objects we plan to use in this application.

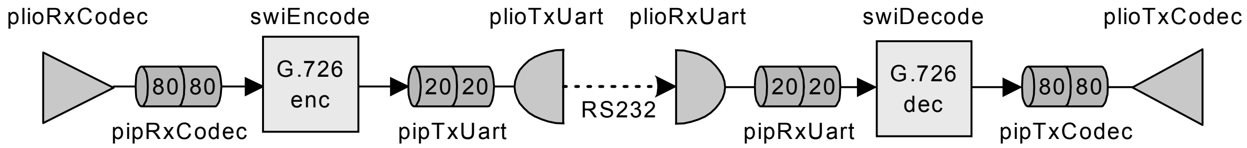

Figure 8. Application Data Path with Buffer Lengths

In this figure, pipe objects (PIP) are represented with one section per frame. The numbers indicate the number of samples contained in a frame. The LIO codec driver and PLIO adapter input and output are shown as triangles. The LIO UART driver and PLIO adapter input and output are shown as half circles.

3.3 Reference Framework Level 3 Data Path

Figure 9 shows data path for the default RF3 application. See the SPRA793 application note for details.

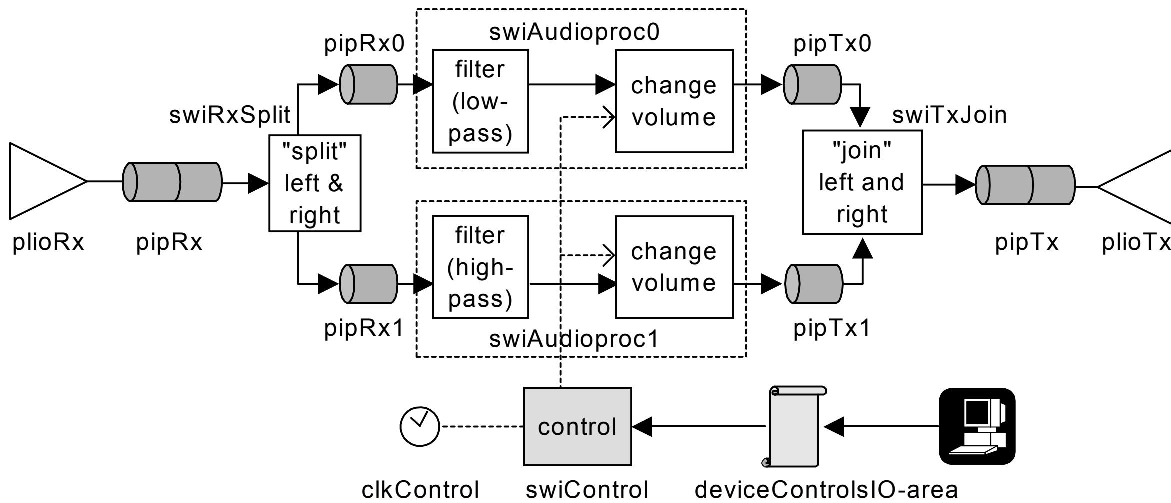

Figure 9. RF3 Data Path

3.4 Removing Unnecessary Components

To convert the data path in Figure 9 to the data path in Figure 8, some components need to be removed or renamed. Others need to be added. In this section, we list components that need to be removed. In later sections, we rename some existing components and add application-specific components.

The following components of the default RF3 application need to be removed:

- The multi-channel capability
- The control function

The SPRA793 application note describes in detail how to remove the second processing channel and the control channel from the RF3 application. See Sections 8.3 and 8.4 of that application note for step-by-step instructions.

While removing some components, we should keep in mind that the following components of the default RF3 application are still needed by the RF3_UART_G726 application:

- The codec LIO driver, which is called via the PLIO adapter
- The ALGRF module, which instantiates algorithms
- The UTL module, which provides debugging features
- The framework “glue” into which we insert our application code

Removing the second channel and the control channel leaves us with the data path shown in Figure 10:

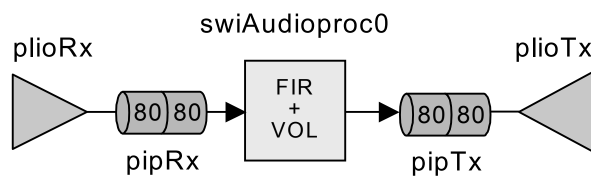

Figure 10. Data Path for Simplified Version of RF3

3.4.1 Changing Object Names

Next, we modify the names of PLIO, PIP, and SWI objects and properties to fit the application names in Figure 8. This prevents name collisions between PLIO and PIP objects and makes understanding the system easier.

Table 1. PLIO, PIP, and SWI Name Changes

Old Name	New Name
plioRx	plioRxCodec
plioTx	plioTxCodec
pipRx	pipRxCodec
pipTx	pipTxCodec
swiAudioproc0	swiEncode
thrAudioprocRun	thrEncodeRun

The easiest way to change these names is to use the DSP/BIOS Configuration Tool to modify the SWI and PIP objects and properties and to perform a find and replace on the following source files and header files:

- appIO.c
- thrAudioproc.c
- thrAudioproc.h
- appBiosObjects.h
- appIO.h

We also rename the thrAudioproc.c and thrAudioproc.h files to thrEncode.c and thrEncode.h. We are separating the processing SWI into an encoder and decoder SWI. Later we will create the thrDecode.c and thrDecode.h files used by the decoder SWI.

For consistency in the appThreads.c, thrEncode.c, and thrEncode.h files, we change all names of functions and variables that contain "thrAudioproc" to corresponding names containing "thrEncode".

These name changes produce the data path shown in Figure 11:

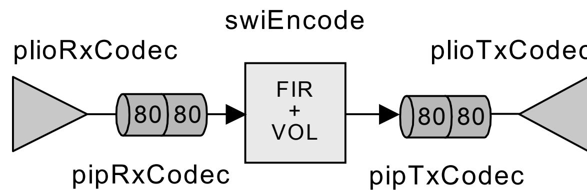

Figure 11. Data Path After Name Changes

3.4.2 Adding the G.726 Encoder/Decoder to Reduce the Data Flow

In this section, we add the G.726 encoder and decoder to the data path, so that we can compress the data before introducing the UART link. This adaptation is similar to the one described in Section 8.4 of the SPRA793 application note. However, in this case, we further modularize the swiEncode and swiDecode components by connecting them with a PIP object called pipLink. This will allow us to further adapt the application so that it can communicate over the UART.

The changes made in this section are:

- **Add SWI and PIP objects.** The first step is to add the swiDecode and pipLink objects to the data flow.

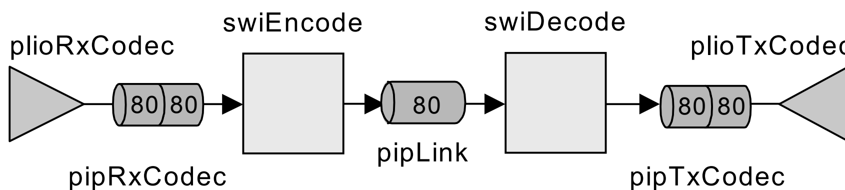

Figure 12. Data Path with Added SWI and PIP Objects

To add these objects, follow these steps:

– In the DSP/BIOS Configuration Tool, add a SWI object called swiDecode. Set the properties as shown in Figure 13. The priority must be the same as that of the swiEncode SWI object.

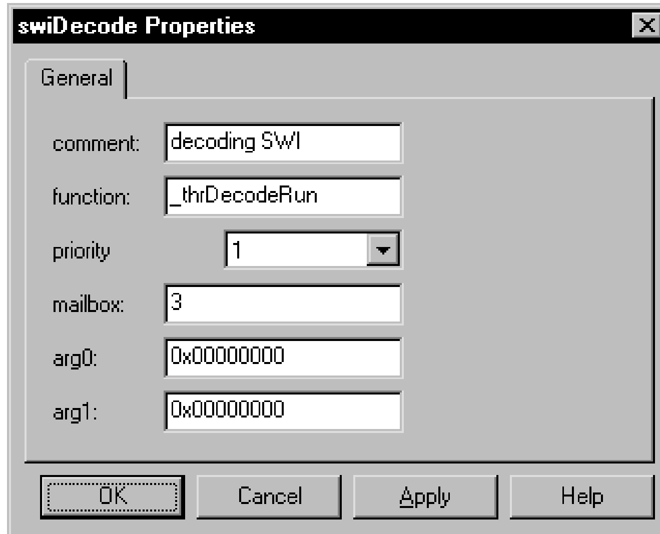

Figure 13. Properties for swiDecode Object

– Add a PIP object called pipLink. Give it the properties shown in Figure 14.

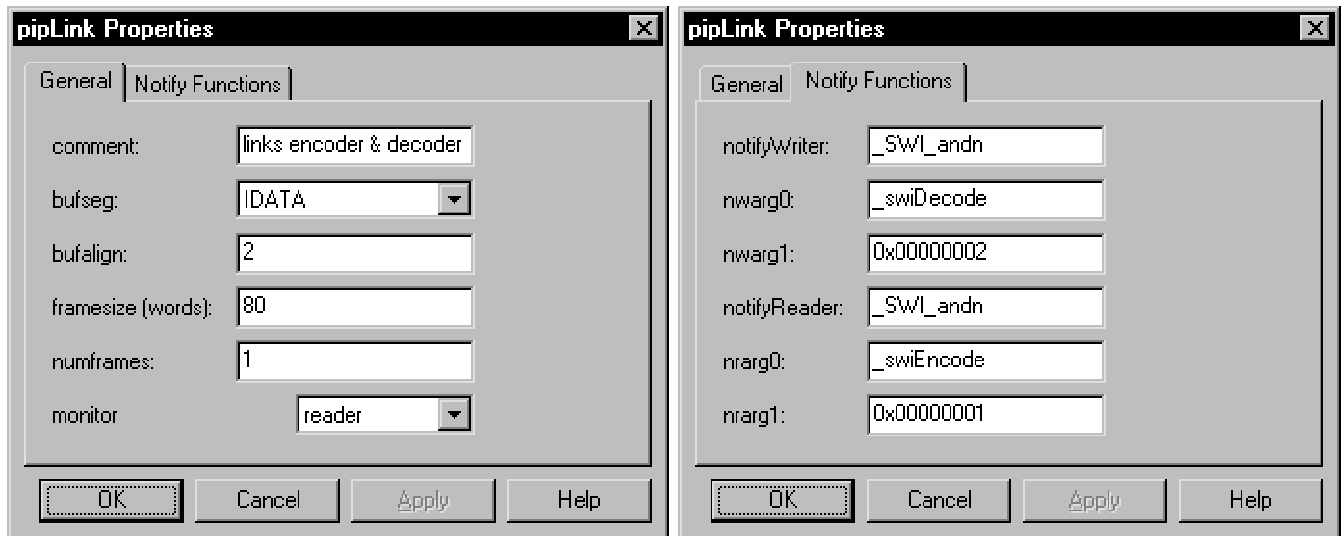

Figure 14. Properties for pipLink Object

– Create the code for the thrDecodeRun() function by coping of thrEncode.c and thrEncode.h files and renaming them thrDecode.c and thrDecode.h. In thrDecode.c and thrDecode.h, do a find-and-replace to change all occurrences of "encode" to "decode".

– Add a call to thrDecodeInit()in appThreads.c after the call to thrEncodeInit().

– Add the new DSP/BIOS objects (swiDecode and pipLink) to appBiosObjects.h.

- **Replace the FIR and VOL algorithms with the G.726 algorithms.** This adaptation is well documented in the RF3 application note (SPRA793, Section 8.4). Refer to that document for details. There are differences between the code in SPRA793 and the code in the thrEncode.c and thrDecode.c files because the actions performed by the thrAudioprocRun() function are split between the thrEncodeRun() and thrDecodeRun() functions.

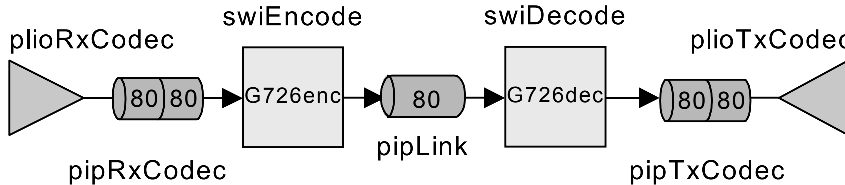

Figure 15. Switching Algorithms

- **Pack the data into a smaller buffer.** The G.726 encoder and decoder do not require any reduction in the data flow size. The G.726 encoder returns a buffer of 80 samples of 16 bits, but only the two lower bits contain data. However, to minimize the amount of data sent over the UART, the thrEncodeRun() function reorders the bits that contain data into a smaller buffer of 20 samples of 8 bits. (The UART will be configured to transfer 8-bit samples.) Figure 16 shows the buffer packing performed by the swiEncode and swiDecode objects.

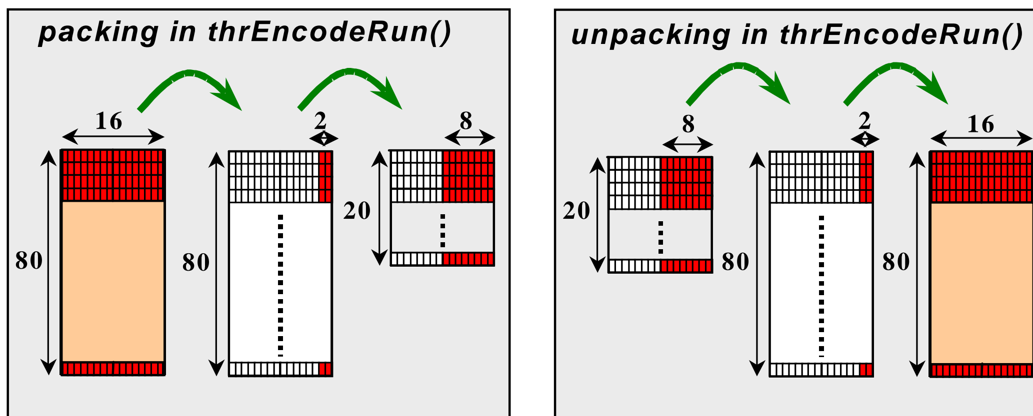

Figure 16. swiEncode and swiDecode Buffer Size Reduction

In the thrEncodeRun() function, the code used to pack the buffer is as follows:

```
/* pack the interim buffer before transmitting it to reduce the data flow */
for (i = 0; i < size; i=i+4) {
    *dst++ =  (Sample) ((thrEncode.bufInterm[i] & 0x03) +
              ((thrEncode.bufInterm[i+1] & 0x03) << 2) +
              ((thrEncode.bufInterm[i+2] & 0x03) << 4) +
              ((thrEncode.bufInterm[i+3] & 0x03) << 6));
}

/* Record the amount of actual data being sent */
PIP_setWriterSize( thrEncode.pipOut, sizeInWords( size / 4) );
```

3.4.3 Adding the UART to the Data Path

The data flow between swiEncode and swiDecode has been divided by 8, which means that the required data rate has been reduced from 16 kbps (kilobits per second) to 2 kbps. The maximum data transfer rate for the application is 11.52 Kbytes per second, so we can use the UART to replace the pipLink pipe. The UART operates at 115.2 kbps.

The UART driver delivered with this application follows the LIO (Low-level Input Output) model and can be used with the PLIO and DLIO adapters. The details of the driver are described in Section 5, *UART LIO Device Controller*, page 26.

The PLIO adapter allows you to connect the driver to a DSP/BIOS pipe. RF3 already uses this adapter to connect the codec LIO driver to pipRxCodec and pipTxCodec. In fact, the application will use the exact same adapter for the codec and UART drivers, thus saving code space.

The appIO.c source file contains the code that initializes the codec driver as well as the PLIO adapter objects for the codec (plioRxCodec and plioTxCodec). Initializing an LIO driver and its PLIO objects is always done with the same function calls. (For details see the SPRA802 application note.) Therefore, we duplicate the code that initializes the codec driver and its PLIO objects to initialize the UART driver and its PLIO objects.

```
PLIO_Obj plioRxCodec, plioTxCodec;
PLIO_Obj plioRxUart, plioTxUart;

Void appIOInit()
{
    DSK5402_DMA_AD50_init();
    DSK5402_DMA_AD50_setup( NULL );

    PLIO_new( &plioRxCodec, &pipRxCodec, LIO_INPUT,  &DSK5402_DMA_AD50_ILIO, NULL );
    PLIO_new( &plioTxCodec, &pipTxCodec, LIO_OUTPUT, &DSK5402_DMA_AD50_ILIO, NULL );

    DSK5402_UART_init();
    DSK5402_UART_setup( NULL );

    PLIO_new( &plioRxUart, &pipRxUart, LIO_INPUT,  &DSK5402_UART_ILIO, NULL );
    PLIO_new( &plioTxUart, &pipTxUart, LIO_OUTPUT, &DSK5402_UART_ILIO, NULL );
}
```

The connection between the UART driver's PLIO objects and the application is created through two new pipes: pipRxUart and pipTxUart, which are defined as shown in the highlighted columns in Table 2.

Table 2. Application's Pipe Configuration Parameters

	pipRxCodec	pipTxUart	pipRxUart	pipTxCodec
framesize	80	20	20	80
numframes	2	2	2	2
notifyWriter	_PLIO_rxPrime	_SWI_andn	_PLIO_rxPrime	_SWI_andn
nwarg0	_plioRxCodec	_swiEncode	_plioRxUart	_swiDecode
nwarg1	0x0000	0x0002	0x0000	0x0002
notifyReader	_SWI_andn	_PLIO_txPrime	_SWI_andn	_PLIO_txPrime
nrarg0	_swiEncode	_plioTxUart	_swiDecode	_plioTxCodec
nrarg1	0x0001	0x0000	0x0001	0x0000

After adding the PIP and PLIO objects and making the appropriate changes to the application source code, the application data path is as shown in Figure 17.

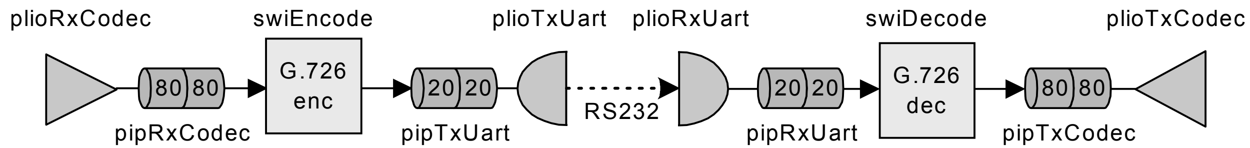

Figure 17. Final Application Data Path

4 Application Priming

Priming is used in an audio application to guarantee that the processor will have a sufficient amount of time to process data from the moment it receives the buffer until the moment the processed buffer is needed. This amount of time is called latency.

One advantage of RF3 is the reliability of the latency. RF3 outputs two buffers of silence to the codec output before starting the codec input. This guarantees that the processor will have at least a frame's worth of time to handle each frame of data. The RF3 priming strategy is relatively simple because there is a strong link between the incoming and the outgoing flow of data in the default application.

However, in the RF3_UART_G726 application, the timing link between the incoming and outgoing data flows is not as strong. In order to provide a reliable data flow, a modified priming strategy is needed.

Figure 18 shows how the priming strategy works in the default RF3 application. This strategy is further explained in Section 7.2.4.7 of the SPRA793 application note

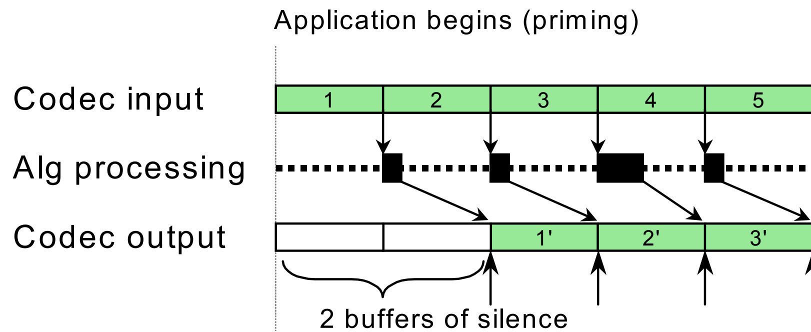

Figure 18. Priming in the Default RF3 Application

The result of priming the codec output with two buffers of silence is that processing can take up to one buffer-worth of time without missing any real time deadlines. Applying that priming strategy to the RF3_UART_G726 application produces the execution graph in Figure 19:

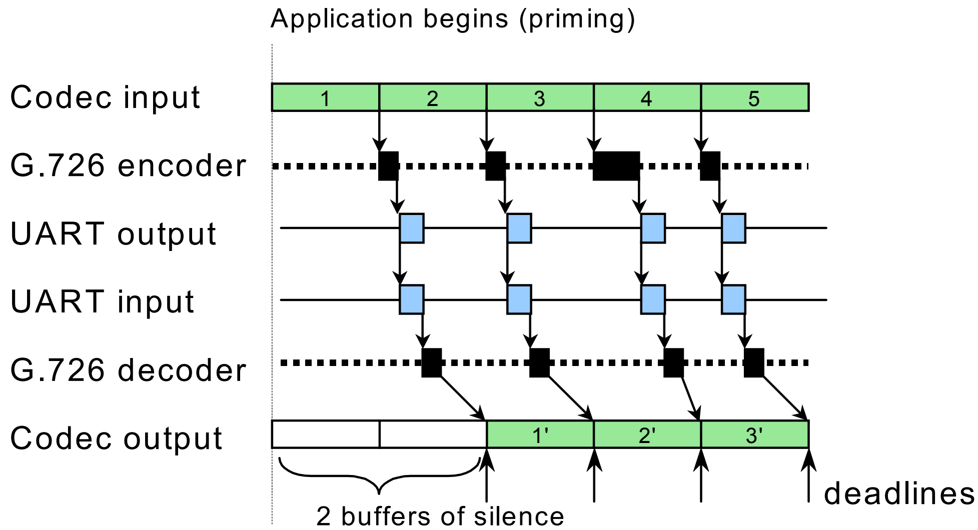

Figure 19. Application Execution Graph with RF3 Basic Priming

Using this simple priming mechanism, the application has one buffer of time to encode the data, transmit through the UART (and receive from the UART at the same time), and to decode the data. This is sufficient time when the application is running on one board. However, there are two problems with this strategy:

- One of the application goals is to make the encoder and decoder completely independent—so they can run on separate boards. This is not the case in this execution graph; the encoder portion of the application primes the output codec.
- If the encoder processing time is not constant, UART transfers do not happen at a regular interval.

To solve these problems, we need to view the encoder and decoder as two separate applications, each of which needs its own priming mechanism. The first step is to define the boundary between the two halves of the application (encoder and decoder) and the expected result of priming. Then the resulting strategy will be implemented.

4.1 The Encoder

Figure 20 shows the data flow and execution graph for the encoder portion of the application.

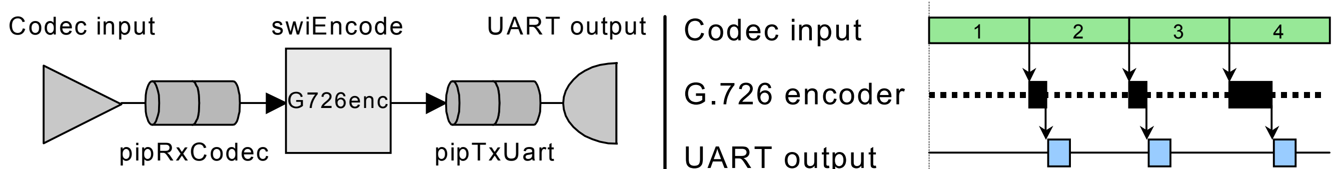

Figure 20. Encoder Application

Notice that the UART does not start at a regular interval. In order to start the UART at a regular interval, we can delay the UART output start until the next full buffer is received by the codec. This also has the effect of providing a full buffer of time for the encoder to process the data before the deadline to transmit the processed buffer through the UART.

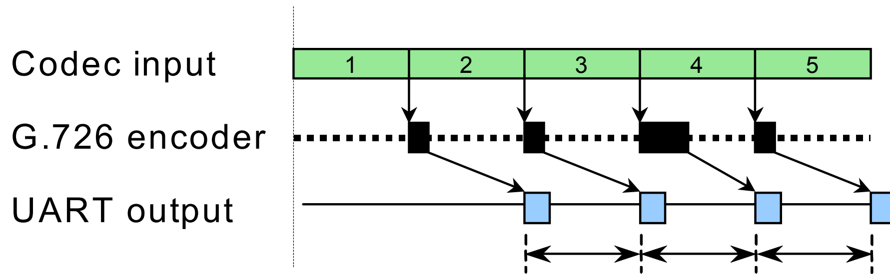

Figure 21. UART Output Delayed

This configuration is similar to the RF3 execution diagram (see Figure 18) in the sense that a full buffer of time is always available for processing the buffer. The major difference is that in the RF3 application, the codec output is started manually only when the application first starts. In contrast, the UART output needs to be started manually by the application each time.

4.2 Priming the Encoder

Based on Figure 21, we need to find a way to delay the UART transmission until the codec has finished receiving the next buffer instead of starting the transmission immediately after the encoder has processed the buffer.

The mechanism that starts the UART transmission is rather simple. Each time a call to PIP_put() writes a frame to the pipRxUart pipe, the notifyReader function of the pipe, PLIO_txPrime, is called. Calling PLIO_txPrime submits a frame in the pipe via the corresponding driver. That is, calling PIP_put() for the pipTxUart pipe causes the frame just written to pipTxUart to be output through the UART driver. This mechanism is documented in the SPRA802 application note.

Typically, the call to PIP_put() occurs near the end of a function that transfers data from one pipe to another. To delay the PIP_put() call until the next buffer is received by the codec, the call to PIP_put() can be moved to the beginning of the thrEncodeRun() function run by the swiEncode object.

The following code shows the thrEncodeRun() function with key calls to PIP functions in bold. In thrEncode.c, thrEncode.pipIn is declared as &pipRxCodec, and thrEncode.pipOut is declared as &pipTxUart.

```
Void thrEncodeRun()
{
    Sample *src, *dst;
    Int     size, i;        /* in samples */

    /* Put frame containing previous buffer, to output it via the UART */
    PIP_put( thrEncode.pipOut );

    /* Check that preconditions are met */
    UTL_assert( PIP_getReaderNumFrames( thrEncode.pipIn  ) > 0 );
    UTL_assert( PIP_getWriterNumFrames( thrEncode.pipOut ) > 0 );

    /* Get the full buffer from the input pipe */
    PIP_get( thrEncode.pipIn );
    src = PIP_getReaderAddr( thrEncode.pipIn );
    /* Get the size in samples (PIP_getReaderSize returns it in words) */
    size = sizeInSamples( PIP_getReaderSize( thrEncode.pipIn ) );

    /* Get the empty buffer from the output pipe */
    PIP_alloc( thrEncode.pipOut );
    dst = PIP_getWriterAddr( thrEncode.pipOut );

    /* Encode the signal in the src buffer and store result in the Interim buf */
    G726ENC_apply( thrEncode.algG726ENC,
               (XDAS_Int16 *)src, (XDAS_Int8 *)(thrEncode.bufInterm));

    /* Pack the interim buffer before transmitting it to reduce the data flow */
    for (i = 0; i < size; i=i+4) {
        *dst++ =  (Sample) ((thrEncode.bufInterm[i] & 0x03) +
                  ((thrEncode.bufInterm[i+1] & 0x03) << 2) +
                  ((thrEncode.bufInterm[i+2] & 0x03) << 4) +
                  ((thrEncode.bufInterm[i+3] & 0x03) << 6));
    }

    /* Record the amount of actual data being sent */
    PIP_setWriterSize( thrEncode.pipOut, sizeInWords( size / 4) );

    /* Free the receive buffer */
    PIP_free( thrEncode.pipIn  );
}
```

Reordering this code requires a change to the start up sequence. The first time thrEncodeRun() runs, PIP_put() would be called without a prior call of PIP_alloc(), causing the system to fail. To correct this, the applOPrime() initialization function needs to call PIP_alloc() once and simply clear the allocated frame.

4.3 The Decoder

With the assumption that the Encoder sends data through the UART respecting a constant period, the actual Decoder placed on a separate board (for the 2 boards application) will behave as follow:

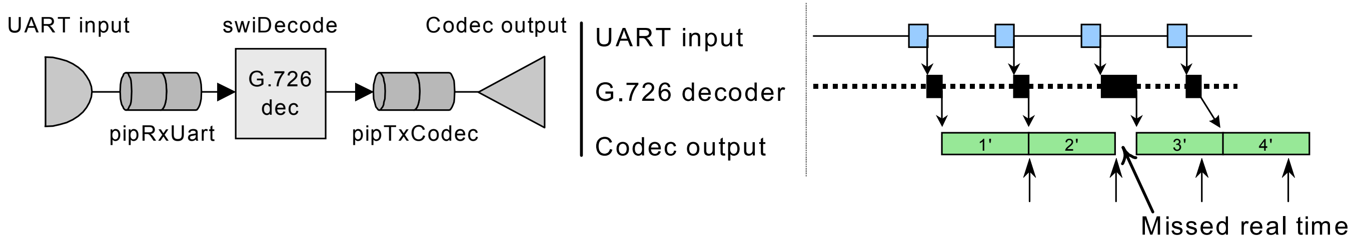

Figure 22. Decoder Application

Clearly, the lack of codec output priming will cause missed real time deadlines if the decoder processing time is not constant. The solution is to prime the codec output with one buffer of silence after the first buffer is received from the UART. This ensures that the processor has a full buffer of time to process each buffer.

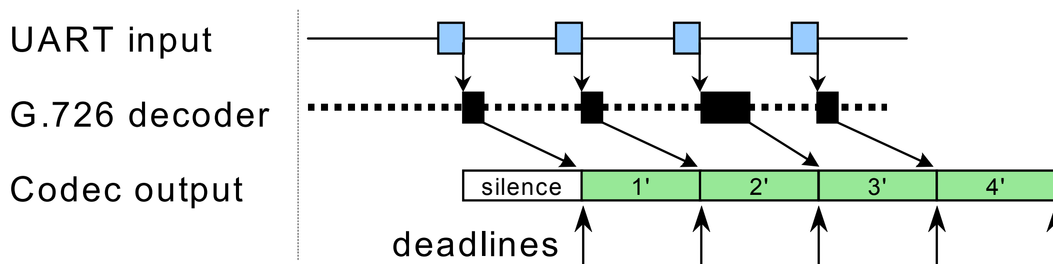

Figure 23. Decoder Primed with One Buffer of Silence

Once again, the results are similar to those for priming the default RF3 application as shown in Figure 18. In this case, the decoder application is primed with a single buffer of silence after the first buffer has been received through the UART.

4.4 Priming the Decoder

The first time the swiDecode object runs its function, thrDecodeRun(), the two frames of the output codec pipe (pipTxCodec) are empty since they have not been used. Therefore, each time we enter thrDecodeRun(), we check the number of frames available in pipTxCodec, if two frames are available, we allocate one, clear it, and output it (with PIP_put()). This primes the application.

The following code shows the thrDecodeRun() function with key calls to PIP functions in bold. In thrDecode.c, thrDecode.pipIn is declared as &pipRxUart, and thrDecode.pipOut is declared as &pipTxCodec.

```
Void thrDecodeRun()
{
    Sample *src, *dst;
    Int     size, i;       /* in samples */

    /* If codec output pipe is completely empty, allocate one buffer of silence */
    if (PIP_getWriterNumFrames(thrDecode.pipOut) == 2) {
        PIP_alloc ( thrDecode.pipOut );
        memset(PIP_getWriterAddr (thrDecode.pipOut),
                 0 ,PIP_getWriterSize (thrDecode.pipOut));
        PIP_put ( thrDecode.pipOut );
    }

    /* Check that the preconditions are met */
    UTL_assert( PIP_getReaderNumFrames( thrDecode.pipIn  ) > 0 );
    UTL_assert( PIP_getWriterNumFrames( thrDecode.pipOut ) > 0 );

    /* Get the full buffer from the input pipe */
    PIP_get( thrDecode.pipIn );
    src = PIP_getReaderAddr( thrDecode.pipIn );
    /* Get the size in samples (PIP_getReaderSize returns it in words) */
    size = sizeInSamples( PIP_getReaderSize( thrDecode.pipIn ) );

    /* Get the empty buffer from the output pipe */
    PIP_alloc( thrDecode.pipOut );
    dst = PIP_getWriterAddr( thrDecode.pipOut );

    /* Unpack received buffer to an intermediate buffer before processing it */
    for (i = 0; i < size * 4; i = i + 4) {
        thrDecode.bufInterm[i]   =  *src       & 0x03;
        thrDecode.bufInterm[i+1] = (*src >> 2) & 0x03;
        thrDecode.bufInterm[i+2] = (*src >> 4) & 0x03;
        thrDecode.bufInterm[i+3] = (*src >> 6) & 0x03;
        src++;
    }

    /* Apply G726 decoder algorithm on the intermediate buffer */
    G726DEC_apply(thrDecode.algG726DEC,
               (XDAS_Int8 *)thrDecode.bufInterm, (XDAS_Int16 *)dst );

    /* Record the amount of actual data being sent */
    PIP_setWriterSize( thrDecode.pipOut, sizeInWords( size * 4 ) );

    /* Free the receive buffer, put the transmit buffer */
    PIP_free( thrDecode.pipIn  );
    PIP_put ( thrDecode.pipOut );
}
```

The advantage of this solution is to prime the codec output each time the UART input starts to receive data. If the UART link is broken for any reason, the output codec will play the samples contained in its pipe until all frames are empty. If the UART link is later reconnected, this function re-primes the application.

4.5 Impact of the Priming Strategy on the Global System

The primary consequence of priming the encoder and decoder separately is that a minimum available processing time can be guaranteed for both the encoder and the decoder.

- Maximum compute time for the Encoder < 1 buffer period
- Maximum compute time for the Decoder < 1 buffer period
- Maximum UART transmission time < 1 buffer periodThe fact that the UART transmission can take up to one buffer period is a side effect of priming the application as two independent applications.

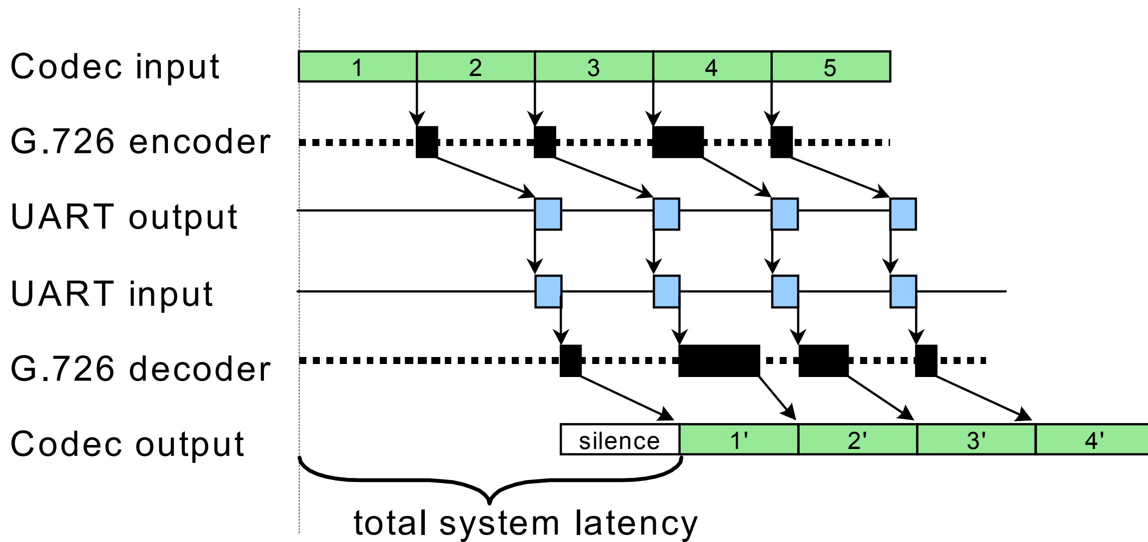

Figure 24. Encoder and Decoder Running on Two Boards

But this strategy has a cost. The total system latency between the codec input and the codec output has increased from 2 buffer periods (Figure 18) to 3 buffer periods plus the time needed to transmit the data through the UART.

4.6 Additional Considerations

The following issues should be considered when using this application or techniques used in this application:

- **Additional threads.** The encoder priming strategy relies on the fact that the swiEncode object runs its function immediately after a buffer is received by the codec. If you want to add another thread to this application, that thread must have a priority lower than the swiEncode object. If a higher-priority SWI or HWI were added, the situation shown in Figure 25 may occur. The encoder is ready to run but is pre-empted by the higher-priority thread. As a result, the UART transmission is started late.

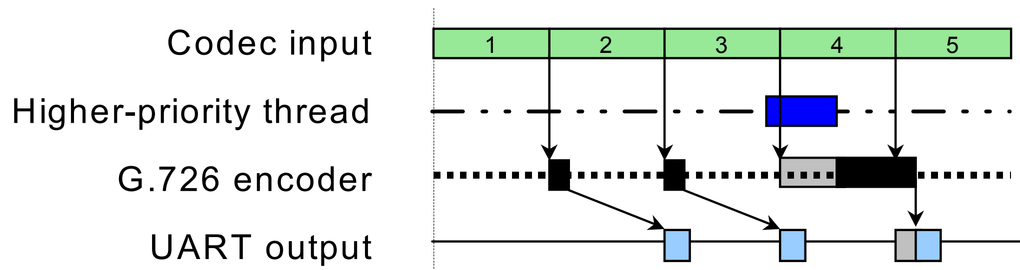

Figure 25. Higher-Priority Thread Can Break the Application

  To solve this problem, you could create another SWI object with the highest priority. This SWI would be posted by the notifyReader of pipRxCodec (instead of swiEncode) just to call PIP_put() and then post swiEncode.

- **Difference between the encoder codec quartz and the decoder codec quartz.** When running on two boards, it may happen that the codec on the encoder board is not running at the same speed as the codec on the decoder board. This situation will soon or later cause an application dysfunction. This type of problem is usual when an application is based on the reception of an asynchronous data flow, which is then used to produce a synchronous output. This problem can be solved temporally by increasing the size of the output buffers, or by adding extra code to artificially slow down or accelerate the output (by adding or removing samples).

5 UART LIO Device Controller

The UART device controller for the 'C5402 DSK board is based on the LIO (Low-level I/O) interface model described in the *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802) application note.

The UART controller has various operation modes. Details about different modes will be discussed in the sections that follow.

5.1 UART Controller Modules

The UART controller is divided into three separate modules. The client application interfaces only with the LIO level functions. However, it is important to understand the underlying modules. The modules are related as shown in Figure 26.

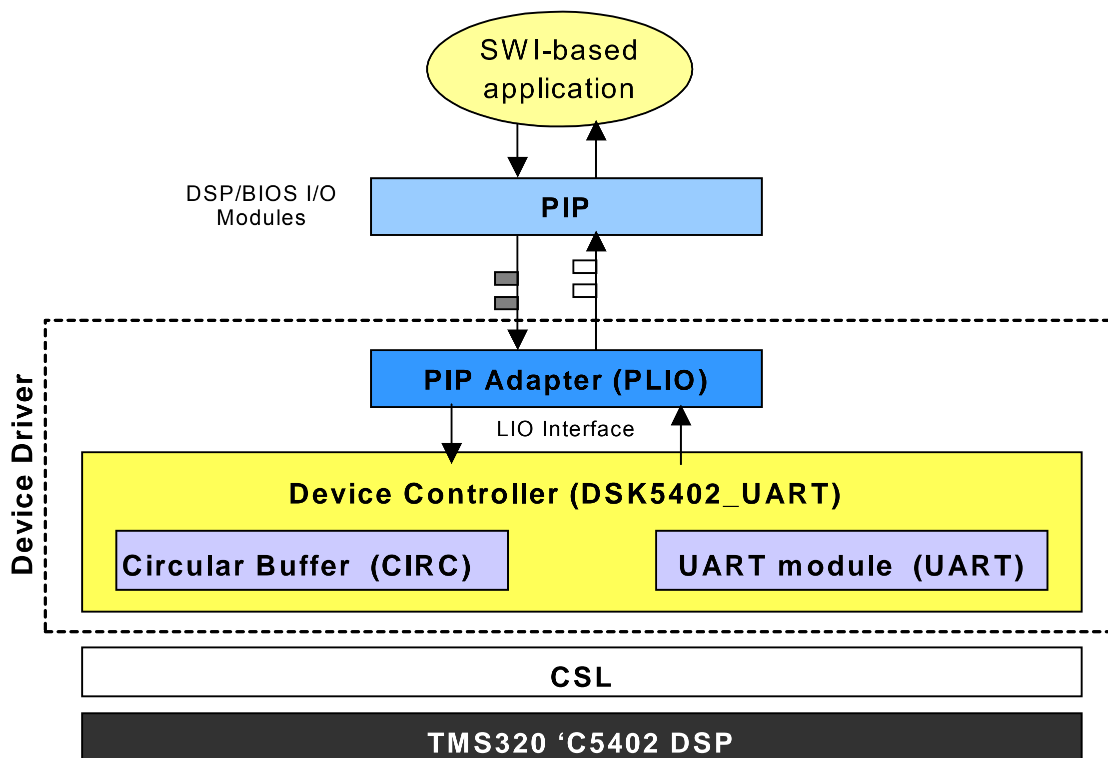

Figure 26. The DSP/BIOS Device Driver Model

The client application interfaces directly with the DSK5402_UART controller module (see Section 5.2). Other UART drivers can reuse this module. The CIRC module is a circular buffer manger that is used as an internal buffer for UART controller (see Section 5.3). The UART module contains UART-specific register settings and control (see Section 5.4). Note there is no communication between the UART and CIRC modules.

5.2 DSK5402_UART Controller Module

The DSK5402_UART device controller sends and receives buffers of samples to and from the UART located on the 'C5402 DSK. It uses a single interrupt service routine (ISR) to send and receive characters. The controller supports two channels and has two channel objects that are initialized globally. It also uses an internal controller buffer managed as a circular buffer (CIRC module).

As specified for all LIO-based controllers, the DSK5402_UART controller contains the functions shown in Figure 27. Global functions and ISR functions have controller-specific names. The channel control functions are contained in the LIO_Fxns table and are required to have the names shown.

Global Functions	Channel Control Functions	ISR Functions
	LIO_Fxns table	
DSK5402_UART_init()	cancel()	DSK5402_UART_isr()
DSK5402_UART_setup()	close()	which calls:
	ctrl()	rxIsr()
	open()	txIsr()
	submit()	

Figure 27. UART Controller Functions

The submit function manages the input and output of data flow. For output, the submit() function receives a buffer and copies the buffer content to an internal controller buffer that is visible to the ISR for processing. For input, the submit() function reads the internal controller buffer and calls the callback function when the application buffer is full and ready to be processed. All communication is done through the channel object. If neither input nor output can satisfy the submit() function, the current buffer address and the current buffer size are stored in the channel object.

When the ISR runs as the result of a receive or transmit interrupt from the UART, it checks to see whether the receive or transmit occurred and then calls the appropriate function. The function then reads or writes a character into its internal buffer and checks to see if the application buffer is empty or full to call the callback function.

5.2.1 UART Controller Channel Object

The channel object shares information between the submit() function and the controller's ISR. Here is the data structure for the channel object used by the DSK5402_UART controller:

```
typedef struct ChanObj
{
    Uns         inuse;          /* TRUE => channel has been opened */
    LIO_Mode    mode;           /* LIO_INPUT or LIO_OUTPUT */

    Char        *bufptr;        /* pointer *within* current buffer */
    Uns         bufcnt;         /* remaining samples to be handled */
    Uns         bufsize;        /* size of this buffer */

    LIO_Tcallback callback;     /* used to notify client when I/O complete */
    Arg         callbackArg;

    CIRC_Obj    circ;           /* Circular buffer */
} ChanObj, *ChanHandle;
```

- The **inuse** variable is set to TRUE (1) by the open() function and should be set to FALSE (0) by the close() function.
- The **mode** determines the direction of the channel and what hardware is allocated for the controller. Each channel object's mode property is set to either LIO_INPUT or LIO_OUTPUT.
- The **bufptr** and **bufsize** are used to communicate with the ISR.
- The **bufcnt** keeps track of the number of buffers that have been submitted to the controller. Although the client application could use multiple submit() calls to pass several buffers to a device controller, the controller processes only one buffer of data at a time. If bufcnt is not equal to zero at the beginning of submit(), submit() ends and returns a failure to the client. The client uses this information to stop calling submit().
- The **callback** and callback argument (**callbackArg**) are channel properties used by the ISR to synchronize with the client.
- The **circ** object is used as a circular buffer to store the character until the application requests them.

This device controller statically declares an array of channel objects with an element for each channel that it supports as follows:

```
#define NUMCHANS        2       /* LIO_INPUT and LIO_OUTPUT */
static ChanObj chans[NUMCHANS] = {
    {FALSE, LIO_INPUT, 0, 0, 0, NULL, NULL},
    {FALSE, LIO_OUTPUT, 0, 0, 0, NULL, NULL}
};
```

5.2.2 The DSK5402_UART_setup() Function

The DSK5402_UART_setup() function needs to set up the UART and the circular buffer used by the controller. This setup is needed for both the transmit and receive channels and should only be performed once. Therefore, these actions should be done in the setup function, rather than the device controller's open function. If the system needed more flexibility, these actions could be performed in the open function, instead. The choice of using the setup or open function is a design decision that has to be made for each device controller.

Here is the code for the setup() function:

```
Void DSK5402_UART_setup(DSK5402_UART_Setup *setup)
{
    if (setup == NULL) {
        setup = &DSK5402_UART_SETUP;
    }

    /* Sets up UART baud rate, bit, parity, etc., ... */
    UART_setup( &(setup->uartAttrs) );

    /* initialize the circular buffer structures */
    CIRC_new(&chans[LIO_INPUT].circ, &(setup->circAttrs) );
    CIRC_new(&chans[LIO_OUTPUT].circ, &(setup->circAttrs) );

    /* Initializes echo mode */
    echo = setup->mode;
}
```

The DSK5402_UART_setup() function calls the UART_setup() function (in uart.c) to set up the UART properly for the particular application. The UART_setup() function sets controller registers so that the UART will use the proper baud rate, word length, stop bits, parity, and lookback. See Section 5.4.2, *The UART_setup() Function*, page 38 for details.

The DSK5402_UART_setup() function then calls the CIRC_new() function to set up circular buffers in the proper modes. The CIRC_new() function initializes the circular buffer structure and sets it up to a particular mode (e.g. CHARMODE, LINEMODE, DATAMODE). See Section 5.3.2, *The CIRC_new() Function*, page 35 for details.

The DSK5402 UART controller also supports an echo mode. This feature is useful when using the controller in LINEMODE or CHARMODE. The feature can be turn on and off.

The DSK5402_UART_Setup structure is passed to the DSK5402_UART_setup() function. This structure contains all the parameters you want to set for the UART and circular buffers. This structure is defined in dsk5402_uart.h as follows:

```
typedef struct DSK5402_UART_Setup {
    UART_Attrs uartAttrs;        /* uart parameters (registers) */
    CIRC_Attrs circAttrs;        /* circular buffer mode */
    DSK5402_UART_Echo mode;      /* echo mode */
} DSK5402_UART_Setup;
```

When NULL is passed to the function the defaults parameters are assigned. These defaults are described in the sections that follow, except for the DSK5402_UART_Mode, which has DSK5402_UART_ECHOMODE_OFF as its default parameter.

5.2.3 The open() Function

The DSK5402_UART_setup() function sets up the hardware for the entire controller and is only called once. The device controller's open() function is called when the application calls PLIO_new(). The open() function creates a channel instance of the controller. The UART controller supports two channels and open() is called for each channel.

The open() function enables a particular channel by changing the state of the channel's object. The open() function sets the channel object's inuse property to TRUE. If a channel is already in use, then it cannot be opened again. Here is the piece of code that checks this condition:

```
if (ATM_setu(&chan->inuse, TRUE)) {
    return (NULL);          /* ERROR! channel is already open! */
}
```

Notice that a DSP/BIOS API call, ATM_setu(), is used to set the condition only if it is not already set. If it is currently set, the open() call returns a null pointer to the calling function in the client. The ATM call disables interrupts while the inuse variable is being modified. This ensures that two threads of different priority do not accidentally open the same channel twice.

Recall that the channel object holds information for the device controller. One of the channel object's parameters is the callback function. The callback function is called by the controller when it completes a buffer. An optional argument to this function is also passed to open(). The open() function copies these values in the channel object, as shown here:

```
chan->callback = cb;
chan->callbackArg = cbArg;
```

The open() function should also perform any hardware initialization that is channel-specific and not performed by the setup() function. The UART controller has a channel for both receive and transmit sides. Each of these channels uses a single ISR and hardware interrupt. The UART controller enables the interrupts as shown here:

```
if (mode == LIO_INPUT) {
    UART_enableRx();
}
else {
    UART_enableTx();
}

IRQ_map(IRQ_EVT_INT1);
IRQ_enable(IRQ_EVT_INT1);
```

The UART function calls used here are described in Section 5.4, *UART Module*, page 37.

5.2.4 The submit() Function

Once a channel has been opened successfully, it can be used by the application to send or receive buffers of data. When the client application receives a new buffer from the application, it calls the controller's submit() function. This function receives three arguments: a pointer to the channel object, a pointer to the new buffer, and the size of the buffer in MAUs (minimum addressable units).

The first thing the submit() function does is to make sure that bufcnt is equal to zero. Otherwise, there should not have been a call to submit() because the controller is already processing a buffer. If bufcnt is not equal to zero, submit() returns failure.

```
if (chan->bufcnt) {
    return (-1);            /* buffer already queued, return FALSE */
}
```

If bufcnt is equal to zero and the mode is set to LIO_INPUT, submit() calls CIRC_readBuf(). If CIRC_readBuf() return non-zero, it calls the callback function once the application buffer is full and ready to be processed.

```
/* read entire circular buffer that can be handled and call the callback */
if (chan->mode == LIO_INPUT) {
    count = CIRC_readBuf(circ, (Char *)bufp, nmaus);

    if (count > 0) {
        chan->callback(chan->callbackArg, count);
    }
    else { /*  Store the buffer pointer and size */
        chan->bufptr = bufp;
        chan->bufsize = nmaus;
        chan->bufcnt = nmaus;
    }
}
```

If the mode is LIO_OUTPUT, submit() calls CIRC_writeBuf(), and then checks to see if any characters are available for transmission by the UART. If so, it calls the transmit side of the ISR. It checks to see if the number of characters in the application buffer is equal to the number of characters written to the circular buffer. If so, it calls the callback.

```
else {
    count = CIRC_writeBuf(circ, (Char *)bufp, nmaus);

    imask = HWI_disable();

    if (UART_txEmpty()) {  /* transmit buffer empty bit */
        txIsr();
    }

    HWI_restore(imask);

    /* when buffer requested is full call the callback */
    if (count == nmaus) {
        chan->callback(chan->callbackArg, count);
    }
    else {
        chan->bufptr = (Char *)bufp + count;
        chan->bufsize = nmaus;              /* size of entire buffer */
        chan->bufcnt = nmaus - count;       /* remaining char count */
    }
}
```

In any mode, if the submit can’t call the callback function, the channel's buffer pointer, size and count are stored.

5.2.5 The ISR Functions

The DSK5402_UART_isr() function runs as a response to either a receive or transmit interrupt from the UART. The ISR determines whether it was interrupted due to receiving a character or being ready to transmit a character and calls the appropriate function.

```
Void DSK5402_UART_isr(Void)
{
    /*  Clear the Interrupt register by reading it */
    UART_clearInt();

    if (UART_rxFull()) {  /* receive buffer full */
        rxIsr();
    }
    if (UART_txEmpty()) {  /* transmit buffer empty bit */
        txIsr();
    }
}
```

The controller's ISR function, DSK5402_UART_isr(), needs to be plugged into **HWI_INT1**. This can be accomplished with the DSP/BIOS Configuration Tool under the HWI - Hardware Interrupt Service Routine Manager.

5.2.5.1 The txIsr() Function

If the UART is ready to transmit, the txIsr() function is called within the DSK5402_UART_isr() function. If the internal circular buffer is not full, the txIsr() function places a character in the UART transmit registers by calling UART_writeChar().

```
/*  Place character in transmit register */
if (CIRC_fullCount(circ) > 0) {
    c = CIRC_readChar(circ);
    UART_writeChar(c); /* write character to the transmit register */
}
```

If other characters arrive while transmitting the previous character, the txIsr() function writes the characters into the circular buffer. If the bufcnt reaches zero, the ISR has filled the application buffer, and it needs to notify the client that a buffer is complete. The ISR could simply call the client to notify it that a buffer is complete. However, this would make the controller client-specific. To avoid this, the controller uses a callback type of signaling. The channel object contains a pointer to a function initialized by the client application. This function pointer is used by the ISR to call the function specified by the client. There are two arguments to the callback, the buffer size and an optional callback argument that is also specified in the channel object.

```
/* Write to circular buffer until no more samples need to be handled */
if (chan->bufcnt > 0) {
    count = CIRC_writeBuf(circ, chan->bufptr, chan->bufcnt);
    chan->bufcnt = chan->bufcnt - count;
    chan->bufptr = chan->bufptr + count;

    /* then call the callback function */
    if (chan->bufcnt == 0) {
        chan->callback(chan->callbackArg, chan->bufsize);
    }
}
```

5.2.5.2 The rxIsr() Function

If the UART is ready to receive, the rxIsr() function is called within the DSK5402_UART_isr() function. The rxIsr() calls UART_readChar() to read a character from the receiver register.

If echo mode is on, it writes the character into the LIO_OUTPUT circular buffer if space is available.

```
if (echo) {
    /* if there's room, put character into output fifo */
    if (CIRC_emptyCount(&chans[LIO_OUTPUT].circ)) {
        CIRC_writeChar(&chans[LIO_OUTPUT].circ, c);
    }
}
```

If the buffer has space available, it writes the character it received to the input circular buffer.

```
if (CIRC_EmptyCount(circ)) {
    CIRC_writeChar(circ, c);
}
```

If other characters arrive while receiving the previous character, the rxIsr() function reads the characters from the circular buffer.

If bufcnt reaches zero, the ISR has emptied the application buffer, and notifies the client that a buffer is complete by calling the callback function with the appropriate arguments.

```
if (chan->bufcnt) {
    count = CIRC_readBuf(circ, chan->bufptr, chan->bufcnt);
    if (count > 0) {
        chan->bufcnt = 0;
        chan->callback(chan->callbackArg, count);
    }
}
```

5.3 Circular Buffer Module (CIRC)

This section describes the circular buffer (CIRC) module. The UART controller uses the CIRC module for its own circular buffer. It is important to understand the CIRC module since it plays a critical role in the functionality of the UART controller. Figure 28 shows the structure of the circular buffer.

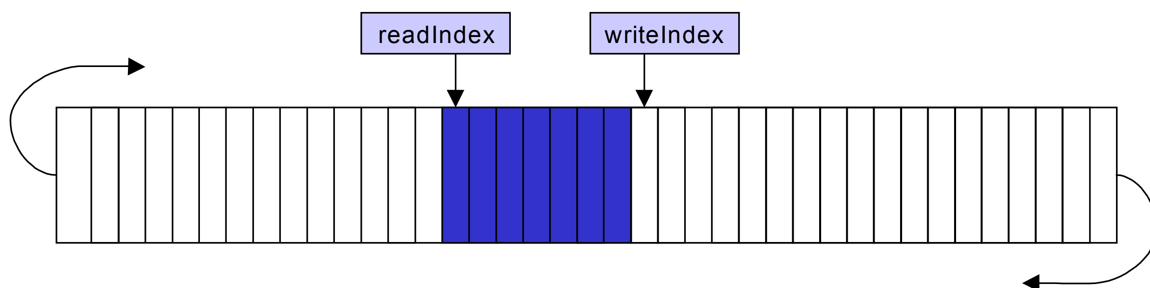

Figure 28. Circular Buffer Used by UART Controller

The CIRC module provides the following functions:

- CIRC_backspace(). Adjusts the contents of the circular buffer by removing the last character and decrementing the count of characters in the buffer. This function is used if a backspace character is encountered.
- CIRC_new(). Initializes the circular buffer structure.
- CIRC_readBuf(). Reads the requested number of characters from the buffer.
- CIRC_readChar(). Reads a single character from the buffer.
- CIRC_writeBuf(). Writes the requested number of characters to the buffer.
- CIRC_writeChar(). Writes a single character to the buffer.
- CIRC_fullCount(). Returns the count of characters in the buffer.
- CIRC_emptyCount(). Returns the count of empty character positions in the buffer.
- CIRC_nextIndex(). Returns the index of the next position in the buffer.
- CIRC_prevIndex(). Returns the index of the previous position in the buffer.

5.3.1 Circular Buffer Object

The circular buffer object shares information between the submit() function and the UART ISR. Here is the type definition for the circular buffer object used in the DSK5402_UART controller:

```
typedef struct CIRC_Obj {
    CIRC_Mode   mode;           /* circular object mode */
    Uns         writeIndex;     /* write pointer for the buffer */
    Uns         readIndex;      /* read pointer fro the buffer */
    Uns         charCount;      /* buffer character count */
    Uns         lineCount;      /* carriage return count */
    Char        buf[CIRC_BUFSIZE]; /* Circular buffer */
} CIRC_Obj, *CIRC_Handle;
```

- The **mode** determines the mode in which the circular buffer in configured. The following options are available:
  - CIRC_CHARMODE – processes character by character
  - CIRC_LINEMODE – processes a string of characters
  - CIRC_DATAMODE – processes a block of data
- The **writeIndex** is a pointer to write to the buffer. The writer only modifies the pointer.
- The **readIndex** is a pointer to read from the buffer. The reader only modifies the pointer.
- The **charCount** is a counter to keep track of the number of characters in the circular buffer. ATM_dec and ATM_inc are used to modify this variable since both the reader and the writer modify it.
- The **lineCount** is a counter to keep track of the number of carriage return line feeds entered. ATM_dec and ATM_inc are used to modify this variable since both the reader and the writer modify it.
- The **buf[CIRC_BUFSIZE]** is the circular buffer.

5.3.2 The CIRC_new() Function

The CIRC_new() function initializes the CIRC object and sets the appropriate mode for the buffer. This function is called by the DSK5402_UART_setup() function.

```
 Void CIRC_new(CIRC_Handle circ, CIRC_Attrs *attrs)
{
    circ->mode = attrs->mode;
    circ->writeIndex = 0;
    circ->readIndex = 0;
    circ->charCount = 0;
    circ->lineCount = 0;
}
```

The available modes are CIRC_CHARMODE, CIRC_LINEMODE and CIRC_DATAMODE. The module provides a default attribute structure (CIRC_Attrs) set to CIRC_DATAMODE. The modes are explained in Section 5.3.5, *The CIRC_readBuf() Function*, page 36.

5.3.3 The CIRC_readChar() and CIRC_writeChar() Functions

The CIRC_readChar() and CIRC_writeChar() functions operate in a symmetrical fashion.

CIRC_writeChar() writes a single character to the circular buffer and increments the charCount variable. The lineCount flag is incremented if a carriage return-line feed is detected in CIRC_LINEMODE.

```
Void CIRC_writeChar(CIRC_Handle circ, Char c)
{
    /* write character and decrement the character count */
    circ->buf[circ->writeIndex] = c;
    circ->writeIndex = CIRC_nextIndex(circ->writeIndex);
    ATM_incu(&circ->charCount);

    /* if enter/return key increment character control flag */
    if (c == '\n' && circ->mode == CIRC_LINEMODE) {
        ATM_incu(&circ->lineCount);
    }
}
```

CIRC_readChar() reads a single character from the circular buffer and decrements the charCount variable. The lineCount flag is decremented if a carriage return line feed is detected in CIRC_LINEMODE.

```
Char CIRC_readChar(CIRC_Handle circ)
{
    Char c;

    /* read character and increment the character count */
    c = circ->buf[circ->readIndex];
    circ->readIndex = CIRC_nextIndex(circ->readIndex);
    ATM_decu(&circ->charCount);

    /* if enter/return key decrement character control flag */
    if (c == '\n' && circ->mode == CIRC_LINEMODE) {
        ATM_decu(&circ->lineCount);
    }
    return (c);
}
```

Note that both the reader and writer modify the charCount and lineCount, therefore ATM_inc and ATM_dec are used to modify these variables**.** Both of these functions are called by the CIRC_writeBuf() and CIRC_readBuf() functions described in the following sections.

5.3.4 The CIRC_writeBuf() Function

The CIRC_writeBuf() function writes as many characters as can fit into the buffer**.** If the buffer is full, the function returns the number of characters it was able to write to the buffer.

```
Uns CIRC_writeBuf(CIRC_Handle circ, Char *buf, Uns nmaus)
{
    Uns count = 0;

    /* write characters into buffer until no more character to write */
    while (circ->charCount < CIRC_BUFSIZE && count < nmaus) {
        CIRC_writeChar(circ, *buf++);
        count++;
    }
    return (count);
}
```

5.3.5 The CIRC_readBuf() Function

The CIRC_readBuf() function performs different actions depending on the mode defined in the CIRC_new() function. The function begins by initializing a count variable to a value of zero.

In CIRC_CHARMODE, the function reads a single character at a time until the requested number of characters has been copied or there are more characters in the buffer to read.

```
if (circ->mode == CIRC_CHARMODE) {
    while (circ->charCount > 0 && count < nmaus) {
        *buf++ = CIRC_readChar(circ);
        count++;
    }
}
```

In CIRC_LINEMODE, characters are read one at a time until the requested number of characters has been copied or there are more characters in the buffer to read. In this mode, if the function detects a carriage return line feed, it returns immediately.

```
else if ( circ->mode == CIRC_LINEMODE ) {
    if ( (nmaus <= circ->charCount) || (circ->lineCount > 0) ) {
        while (circ->charCount > 0 && count < nmaus) {
            *buf = CIRC_readChar(circ);
            count++;
            if (*buf++ == '\n') {
                break;
            }
        }
    }
}
```

In CIRC_DATAMODE, the function checks to make sure the requested number of characters are available before copying the characters to the buffer.

```
else {      /* CIRC_DATAMODE */
    if (nmaus <= circ->charCount) {
        for (; count < nmaus; count++) {
            *buf++ = CIRC_readChar(circ);
        }
    }
}
```

In all modes, the CIRC_readBuf() function returns the number of characters read.

5.4 UART Module

To keep the UART controller as generic as possible, all the UART-specific calls are defined in the UART module. All access to and setup of the registers is performed in this module. This section describes all the functions and prototypes used by the DSK5402_UART controller.

The UART module provides the following functions:

- UART_setup(). Sets up the UART registers and related hardware registers.
- UART_disableRx(). Disables the interrupt for the receiver.
- UART_disableTx(). Disables the interrupt for the transmitter.
- UART_enableRx(). Enables the interrupt for the receiver.
- UART_enableTx(). Enables the interrupt for the transmitter.
- UART_readChar(). Reads a character from the receiver buffer register.
- UART_writeChar(). Writes a character to the transmitter holding register.
- UART_txEmpty(). Returns true if the transmit buffer is empty.
- UART_rxFull(). Returns true if the receive buffer is full of data.
- UART_clearInt(). Clears the interrupt status.

All functions other than UART_setup(), are defined as preprocessor macros. In addition to these functions, the UART module provides the UART_Attrs structure and a number of enumerated types for setting up the hardware.

5.4.1 UART Attribute Structure

The UART_Attrs structure is used to set up the UART hardware into a physical mode. Here is the UART_Attrs structure definition:

```
typedef struct UART_Attrs {
    UART_Baud           baud;
    UART_WordLen        wordLength;
    UART_StopBits       stopBits;
    UART_Parity         parity;
    UART_Loop           loopEnable;
} UART_Attrs;
```

The types used in this structure are defined in RF_DIR\include\uart.h file. In addition, that file defines the following default parameter settings:

```
#define UART_DEFAULTATTRS {                                          \
    UART_BAUD_115200,                                                \
    UART_WORD8,                                                      \
    UART_STOP1,                                                      \
    UART_DISABLE_PARITY,                                             \
    UART_NO_LOOPBACK                                                 \
}
```

5.4.2 The UART_setup() Function

The UART_setup() function sets up the UART registers (such as DLL, DLM, and LCR) for the appropriate baud rate, word length, stop bits, parity, and loopback mode. It also sets the interrupt enable register (IER) and clears the line status register (LSR). It finally reads the UART receive buffer to clear it. This function is called by the DSK5402_UART_setup() function of the UART controller.

5.4.3 The UART_enable/disableRx/Tx Functions

The following functions clear and set the proper bits in the IER register to enable and disable the proper interrupts for the receiver and transmitter:

- UART_disableRx(). This function is called by the close() function of the UART controller.
- UART_disableTx(). This function is called by the close() function of the UART controller.
- UART_enableRx(). This function is called by the open() function of the UART controller.
- UART_enableTx(). This function is called by the open() function of the UART controller.

5.4.4 The UART_readChar() Function

The UART_readChar() function reads a character from the receiver buffer register (RBR). This function is called by the rxIsr() function of the UART controller.

5.4.5 The UART_writeChar() Function

The UART_writeChar() function writes a character to the transmitter holding register (THR). This function is called by the txIsr() function of the UART controller.

5.4.6 The UART_txEmpty() Function

The UART_txEmpty() function reads the value of the line status register (LSR) and masks it with 0x20 (hex). This function is called by the submit() and DSK5402_UART_isr() functions of the UART controller. It returns true if the transmit buffer is empty.

5.4.7 The UART_rxFull() Function

The UART_rxFull() function reads the value of the line status register (LSR) and masks it with 0x01 (hex). This function is called by the DSK5402_UART_isr() function of the UART controller. It returns true if the receive buffer is full of data.

5.4.8 The UART_clearInt() Function

The UART_clearInt() function reads the value of the interrupt identification status register (IIR). Reading this register has the effect of clearing it. This function is called by the DSK5402_UART_isr() function of the UART controller to clear the interrupt status.

References

Writing DSP/BIOS Device Drivers for Block I/O (SPRA802)

Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System (SPRA793)

TL16C450 Asynchronous Communications Element (SLLS037B)

TMS320C54x DSP Enhanced Peripherals Reference Set Volume 5 (SPRU302)

TMS320C54x DSP CPU and Peripherals Reference Set Volume 1 (SPRU131)

Appendix A: Connectors and Cables

The pin assignments for RS232 DB 9 connectors are shown in Figure 29. Diagrams for connectors and cables used in this application note are provided here. More details about RS232 cables and connectors are provided at numerous locations on the web.

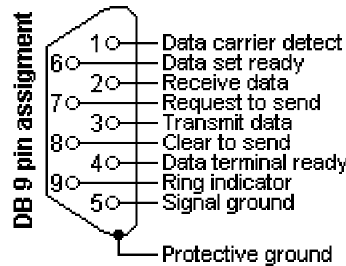

Figure 29. DB9 Pin Assignments

To run the RF3_UART_G726 application on one 'C5402 DSK board, you need a DB9 female loopback connector. This connector should be wired as shown in Figure 30.

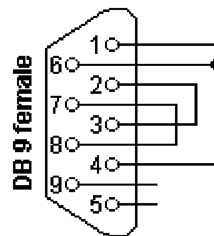

Figure 30. DB9 Loopback Connector

To run the RF3_UART_G726 application on two 'C5402 DSK boards, need a null modem female-to-female DB9 cable. This cable should be wired as shown in Figure 31 or Figure 32. Either of these configurations can be used for this application.

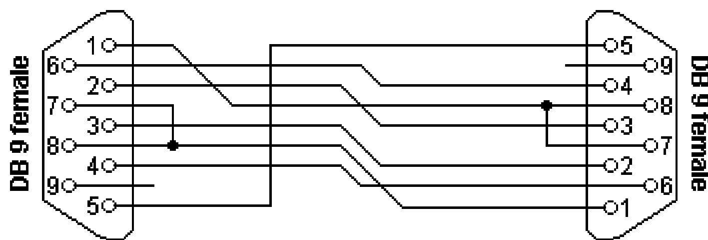

Figure 31. Null Modem Cable with Partial Handshaking

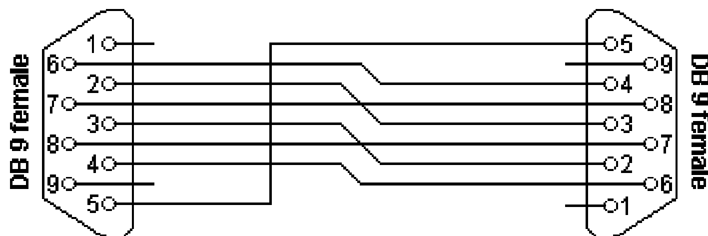

Figure 32. Null Modem Cable with Full Handshaking

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