RSL10 LPDSP32 Support Manual

M-20844-006
September 2019

ON Semiconductor®

Table of Contents

CHAPTER 1

Introduction & Useful References

1.1 Purpose

This document provides an overview of the techniques involved when writing and integrating code for the LPDSP32 processor included with the RSL10 radio System-on-Chip (SoC).

The document is designed as a tutorial which introduces some common techniques in the context of the implementation of an audio codec development framework. The examples provided assume that the Arm® Cortex®-M3 processor is the master in the system and the LPDSP32 works as a slave, responding to requests. This provides some real world examples and demonstrates how to quickly get LPDSP32 code up and running in tandem with the Arm Cortex-M3 core supervisor code.

1.2 Intended Audience

This document and the associated example code are intended for developers who want to quickly understand the concepts involved in writing LPDSP32 code, and importantly, how this can be integrated into the RSL10 environment. This document does not attempt to explain how to program the LPDSP32 efficiently; other reference documents indicated below provide that detail.

As a minimum, we expect that you understand how to program the LPDSP32, how to program RSL10's Arm Cortex-M3 processor, and how to use the Synopsys® tools. We make use of some third party codec samples but do not provide the detail of how they work, only how these are integrated into a common framework for development, evaluation and test.

1.3 Text Conventions

Two conventions are used when providing non-standard text in this document:

1. Code blocks are always demonstrated with a title and syntax highlighting as follows:

Example Code Block

```
/* This is some sample code */
#define Fred    (3)

int main(int argc, char **argv)
{
    return Fred;
}
```

1. Notes providing additional information will also be shown this way:

NOTE: This is a note, which doesn’t tell you much.

1.4 Further Reading

Additional information regarding the use of the LPDSP32 in RSL10 can be found in the hardware and firmware reference manuals. These provide details such as memory maps and interrupt vectors etc.

- *RSL10 Firmware Reference*
- *RSL10 Hardware Reference*

In addition, specific information related to the LPDSP32 processor can be found in the following documents:

• *LPDSP32-V3 Block Diagram*	(*LPDSP32_V3_arch_v1.3.pdf*)
• *LPDSP32-V3 Hardware Reference Manual*	(*LPDSP32_V3_hwref_man_ver1.3.pdf*)
• *LPDSP32-V3 Interrupt Support Manual*	(*LPDSP32_V3_interrupt_ver1.3.pdf*)
• *User Guide IP Programmer for LPDSP32–V3*	(*user_guide_lpdsp32_v3.pdf*)

Finally, details of the Synopsys ASIP Programmer tools can be located in the

- *ASIP Programmer Overview of the Manuals*

This manual will be installed with the ASIP Programmer.

CHAPTER 2

Getting Started with the LPDSP32 on the RSL10 Evaluation and Development Board

This section contains some very basic information to get up and running with the LPDSP32 development environment.

2.1 INSTALLING THE SYNOPSYS ASIP PROGRAMMER TOOLS

The LPDSP32 toolchain is provided by a third party, not ON Semiconductor. Details of licencing options and enabling access to the tools should initially be discussed with your ON Semiconductor customer support representative, who will help to arrange access. The remainder of this document assumes you have been able to obtain the appropriate installation binaries for your platform.

The ASIP Programmer tools consist of a number of applications, the main one of interest to us being the Chess DE development environment.

There are installation binaries available for both Linux and Windows®, and generally would be specified with a particular version number; for example:

- ASIP Programmer for Linux: *asip_programmer_vN-XXXX.XX_linux64.spf*
- ASIP Programmer for Windows: *ASIP_Programmer_vN-XXXX-XX_win.exe*

There is an installation manual associated with these installers.

- *asip_installation_manual.pdf*

For the purposes of this document, we assume a Windows based installation, using the 2017.09 version, *ASIP_Programmer_vN-2017.09_win.exe.*

2.1.1 ASIP Programmer Installation

Installation of the ASIP Programmer tools is straightforward and is clearly explained in the *asip_installation_manual.pdf* mentioned above. Follow the instructions to install the tools and set up the licence server.

2.1.2 Installing LPDSP32 Processor Package

In order to configure the tools for the LPDSP32, a suitable processor package must be installed. A Windows-compatible Linux package is available for this release of the ASIP programmer.

LPDSP32 Processor Package: *lpdsp32-v3_vN-2017.09_linux.zip*

To install the processor package, follow these steps:

- Unzip the file into the install location of the ASIP Programmer. Assuming the default locations have been selected, this can be found in *C:\Synopsys\ASIP Programmmer\N-2017.09*
- Next we need to build the Instruction Set Simulator (ISS). This requires the installation of Visual Studio 2015 as it is used by the ISS building process.
    - Before building the ISS, we must change the configuration to enable function profiling:
        - Open *model.prx*
        - Right click **iss/fastiss** and select **Open project**
        - **Project settings (F8)** -> **Profilin**g -> **Function profiling: On**

- **Save & close project**
- Right click **iss/debugiss** and select **Open project**
- **Project settings (F8)** -> **Profiling** -> **Profile operations: On**
- **Project settings (F8)** -> **Profiling** -> **Profile hazard: On**
- **Project settings (F8)** -> **Profiling** -> **Function profiling: On**
- **Project settings (F8)** -> **Profiling** -> **Instruction tracing: On**
- **Save & close project**
- **Make (F7)**

This now provides a working installation complete with an ISS. Feel free to build a "Hello World" example and test it out on the simulator.

2.2 Creating a "Hello World" Program for the LPDSP32

The standard way of getting used to a new tool or language is to develop a "Hello World" application and get it running, as this provides us with some confidence that the basic building blocks of the tool are doing what we expect.

For the LPDSP32 tools, this is no different. As this works with a standard C compiler, we can use a classic program to check that the tool installation is working properly.

Having installed the ASIP Programmer tools and the LPDSP32 Processor package, and having built the ISS, we are now ready to start.

1. Create a new folder into which to put the projects.
2. Open the Chess Development Environment.
3. From the menu bar, select **File** > **New** > **Project**.
   - In the **New project** Dialog, set the **Name** as **HelloWorld**, the **Project directory** to the folder you created above, and the **Project type** as **Executable**.

4. From the menu bar, select **File** > **New** > **File**.
   - This creates a new editor window into which we can type text. Add the following:
   - Save the file as *main.c* in the folder you created above.

Sample Code

```
#include <stdio.h>

void main(void)
{
   printf("Hello\n");
}
```

1. From the menu bar, select **Project** > **Add source files...** and in the dialog it provides, select the *main.c* file you have just created.
2. From the menu bar, select **Compile** > **Make**. The program is expected to build with no warnings or errors.

2.2.1 Run the Program in the Simulator

Having successfully built the program, we can now run it using the ISS:

1. From the menu bar, select **Debug** > **Select Debugger** > **lpdsp32**.
2. Now select **Debug** > **Run** in console.

- The Console view is updated, indicating that it is simulating with **lpdsp32**, and providing some information about the command line as well as the text we asked to send to standard output.

Console Output

```
Simulating in console with "lpdsp32"
lpdsp32 -T -e -t C:/Development/TutorialSamples/Hello/_runsim.tcl -p C:/Development/
   TutorialSamples/Hello/Release/HelloWorld
Hello
```

1. Now select **Debug** > **Start debugging**:

- This starts the debugger and changes the view to the **Debug** view, showing the standard startup code assembler file.
- It also shows a console window and a Hosted I/O window. In this case, the output from the program appears in the Hosted I/O window, and the debugger output is shown in the Console window.
- Select the **Run** button or **Debug** > **Go/Continue**. This executes the program.
- As there are no break points set, the program runs to the end. Output similar to the following is produced in the Console window:

Sample Output from the Debugger

```
Starting debugger "lpdsp32"
lpdsp32 -T -s localhost:61533
Disassemble command used: darts -c C:/Development/TutorialSamples/Hello/Release/
   HelloWorld -I"C:/Program Files (x86)/Target Compiler Technologies/lpdsp32-v3-10R1/
   designs/lpdsp32/lib" lpdsp32 -o C:/Development/TutorialSamples/Hello/Release/
   HelloWorld.cmic2 "-IC:/Program Files (x86)/Target Compiler Technologies/
   lpdsp32-v3-10R1/designs/lpdsp32/lib/runtime/include" -B +Mdec +F
Stack range [57344..65528] has been loaded from elf executable for stack memory DMA and
   stack pointer SP
Stack pointer SP for memory DMA has been initialised from elf executable to 65528
Found end of main function.
stepping halted after 45 cycles
Elapsed time = 0.014
#Cycles/(Elapsed) = 3214.285714285714
```

- Select **Debug** > **Stop debugging** to return to the normal IDE.

CHAPTER 3

Setting up the RSL10 Development Board to Debug LPDSP32

At this stage we have installed the LPDSP32 development tools, and have proven that a simple "Hello World" program can be compiled and run using the LPDSP32 simulator.

For some use cases, this would be enough to develop applications or specific algorithms, but for most real world applications we need to make the code run on the RSL10 hardware. This section gives a very quick overview of the process of connecting the Chess development environment to an RSL10 Evaluation and Development board.

3.1 Requirements

There are a number of things we need to be able to make this happen:

1. The ASIP Programmer development environment (as set up in previous examples)
   - Assuming you have followed the instructions in the previous examples, no more setup is required for this.
2. An up to date installation of the RSL10 SDK
   - We need the SDK to provide some code to allow the LPDSP32 JTAG to be visible, so a working knowledge of the RSL10 SDK is required.
   - For additional information on how to use the RSL10 SDK, see the associated manuals.
3. An RSL10 Evaluation and Development board. For the purposes of this example, we are using a WLCSP, not the QFN model, but principles apply to both.
4. The JTalk communications program. This is originally provided by Synopsys:
   - It provides a communications channel between the JTAG programmer and the Chess IDE.
   - Very similar in concept to the GDB server, commonly used when debugging embedded systems using the GNU debugger.
5. Some form of External JTAG programmer. For this example we use a Segger JLink Ultra+.
   - There are a limited number of supported JTAG programmers which can be used, for full details see the user manual for JTalk.

3.2 Some Basic Concepts

The JTAG port on the LPDSP32 is not normally exposed on a development board. In order to use the JTAG port, we need to configure the device to route the signals to the appropriate DIO pins.

We need to connect the EVB to the computer to allow us to load the software which does the configuration. This uses a straightforward USB connection to the onboard JLink programmer.

We then connect the external JTAG programmer (JLink) to the specific DIO pins we are setting. (For this example we use DIOs 1, 2, 3 & 4.)

The RSL10 SDK communicates with the onboard JLink programmer to program the Arm Cortex-M3 core.

The Chess Development environment communicates with JTalk, and hence the external JTAG (JLink) programmer, to program the LPDSP32 core.

3.3 Hardware Setup

1. Enable Arm Communication:
   a. Connect the EVB to the computer using a standard Micro USB cable.
   b. Startup the RSL10 SDK to communicate with the board.
   c. Ensure the board is connected and working by running the *Blinky* sample application provided in the RSL10 SDK.

2. Disconnect the USB to power down the board.
3. When connecting signals for the external JTAG we need to provide the following connections:
   a. `TMS`, `TCK`, `TDI`, `TDO`, `VTref` and `GND`, as shown in Figure 1 and Table 1:

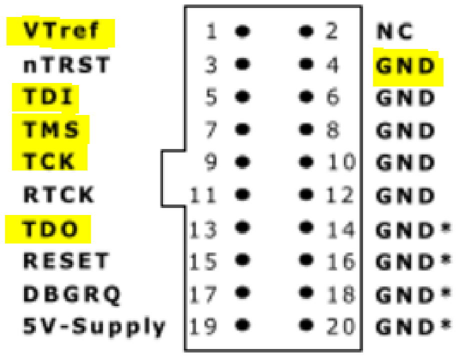


Figure 1. External JTAG Signal Connections

Table 1. JTag and EVB Pins

JTag Pins	EVB Pins
VTref	VDDO
TDI	DIO2
TMS	DIO4
TCK	DIO1
TDO	DIO3
GND	GND

4. Once these are connected, reconnect the USB to the onboard JLink and verify that the *Blinky* sample application still operates as expected.
5. Connect the USB to the external JLink, and verify again that the *Blinky* sample application still works as expected.

NOTE: At this stage you have two JLink devices connected to the PC, causing the JLink driver to pop up a dialog asking which JLink to select. Make sure you connect to the onboard JLink. If you are unsure which JLink is which, there is a serial number on the bottom of the JLink Ultra+ which appears in the dialog. The onboard Jlink is the other one, and always must be used for the RSL10 SDK.

The hardware is now connected, so we need to configure it for use.

3.4 Hardware Configuration

In order to route the connections for the LPDSP32 JTAG to the selected DIO pins, we need to run some code on the Arm Cortex-M3 core.

We have provided a sample application which can be used to configure the JTAG pins as indicated above. This can be found in the samples folder under *LPDSP32_debug_setu*p. This sample application can be run from the RSL10 SDK in the normal fashion and configures the pins accordingly.

Before doing that, let's consider the fundamental operations we need to do, just in case you want to incorporate them into your own code:

1. The pins we are using will be set up using some macros so that they are easy to maintain:

Define the Pin Mapping for the LPDSP32 JTAG

```
/* define the DIOs required for configuring the LPDSP32 JTAG connection */
#define JTAG_LPDSP32_TCK (1)
#define JTAG_LPDSP32_TDI (2)
#define JTAG_LPDSP32_TDO (3)
#define JTAG_LPDSP32_TMS (4)
```

1. A single system call is used to configure the pins for JTAG use:

Enable the Pins We Have Defined Above

```
/* Enable the JTAG for the LPDSP32 using the DIOs we defined above */
Sys_LPDSP32_DIOJTAG(DIO_WEAK_PULL_DOWN, JTAG_LPDSP32_TDI, JTAG_LPDSP32_TMS,
    JTAG_LPDSP32_TCK, JTAG_LPDSP32_TDO);
```

Now load the sample code in the RSL10 SDK and download it to the board. Nothing much visibly happens at this stage, but if you want to ensure the code is running as expected, feel free to step through it in the debugger.

3.5 JTalk Magic

In order for the Chess IDE/Debugger to communicate with the board, it needs a channel to the JTAG. This is achieved using the JTalk utility.

JTalk is distributed as a zip file, with its own installation guide. Follow the installation guide to get JTalk installed in a suitable folder, and make sure the *JLinkARM.dll* is available on the path.

Once JTalk is set up, it can be started from the command line using the following command:

Start JTalk from Command Line

```
> jtalk.exe -c jlink -f 2000
Listening on port 41001
Waiting for traffic ...
```

The -v flag can be added to the command to see details of the communication between the IDE and the device. This is sometimes useful to ensure that the Chess IDE is trying to access the board.

3.6 Nearly There... Let's See if it All Works

So we have now installed two IDEs, downloaded and run the code on the board to configure the JTAG, started the JTalk server... we're ready to test the setup.

Open the Chess IDE and select the **Hello World** program we created earlier.

From the **command** menu, select **Debug** > **select Debugge**r > **lpdsp32_client**.

Try to debug the Hello World program in the same way as if it was on the simulator:

1. Select **Debug** > **Start debugging**.
2. Put a break point on the line in main containing the printf statement (Line 5).
3. Select **Debug** > **Go / Continue**.
4. Verify that it stops at the break point in main.
5. Select **Debug** > **Step Over**.
6. Verify that "Hello" is output to the Hosted I/O console.

Assuming that this has all worked, you now have debug access to the onboard LPDSP32 processor using JTAG.

CHAPTER 4

Arm Cortex-M3 Core – LPDSP32 Communication

The RSL10 device provides two programmable cores: the Arm Cortex-M3 core which hosts the stack and the bulk of any user applications, and the LPDSP32 core which can be used as an accelerator for specific DSP-centric functions.

In order to use the LPDSP32 successfully, it is necessary to understand how these two cores can communicate and share information.

For the purposes of this documentation, we will consider a very simple application which runs on the Arm Cortex-M3 core, but which delegates some key processing to the LPDSP32. This demonstrates how we can notify each core that there is work to be done, and how information is shared between the two cores. To be more specific, we use shared memory to pass context between the cores and interrupts to notify that there is work to be done or has been completed.

We use the Chess IDE to demonstrate the building and running of the LPDSP32 code, and the RSL10 Eclipse IDE to run and debug the Arm Cortex-M3 core code.

4.1 THE SAMPLE APPLICATION

We will create an application that calculates Fibonacci numbers when the user button is pressed on the evaluation board. Each time the button is pressed, the next number in the sequence is calculated. The code on the Arm Cortex-M3 core handles the button presses and tells the LPDSP32 which sequence number to calculate. The code on the LPDSP32 calculates the requested sequence number and passes it back to the Arm Cortex-M3 core. We will use two 32-bit words in shared memory to pass the information back and forth, both defined as `uint32_t`. The Arm Cortex-M3 core notifies the LPDSP32 that there is data requested via an interrupt; the LPDSP32 reciprocates by raising an interrupt on the Arm Cortex-M3 core when the number has been calculated.

The code associated with this sample, for both the Arm Cortex-M3 core and the LPDSP32, is available in the *lpdsp32/samples_lpdsp32/fibonacci* folder of the LPDSP32 package.

Basic flow is as indicated in Figure 2 on page 14, below:

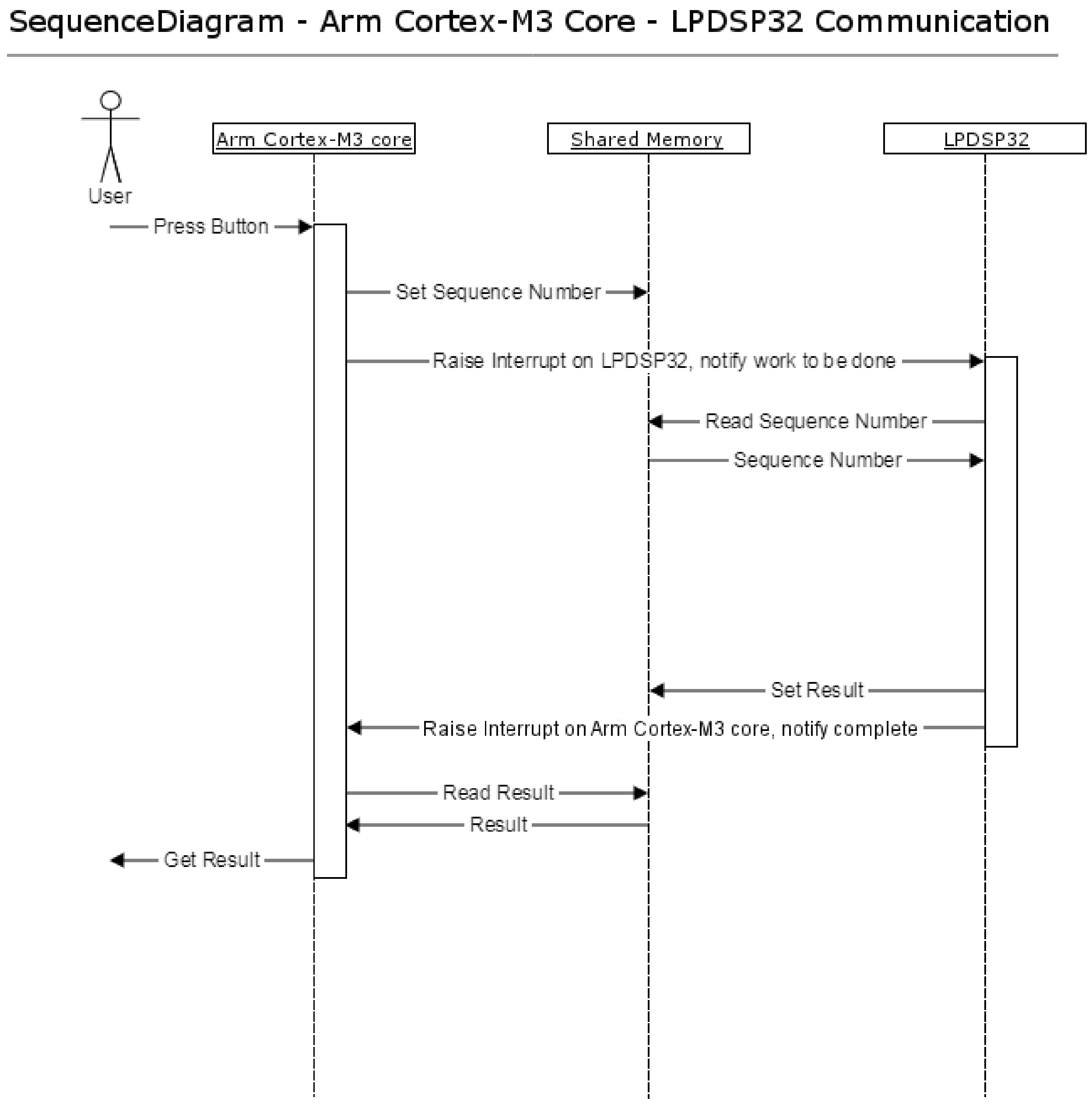

SequenceDiagram - Arm Cortex-M3 Core - LPDSP32 Communication
User
Arm Cortex-M3 core
Shared Memory
LPDSP32
Press Button
Set Sequence Number
Raise Interrupt on LPDSP32, notify work to be done
Read Sequence Number
Sequence Number
Set Result
Raise Interrupt on Arm Cortex-M3 core, notify complete
Read Result
Result
Get Result

Figure 2. Sequence Diagram — Arm Cortex-M3 Core - LPDSP32 Communication

4.2 SETTING UP THE ARM CORTEX-M3 CORE SIDE

The basic setup of the sample code for the Arm Cortex-M3 core is very similar to that for the *Blinky* sample. It uses the LED to indicate some status information and the button as a trigger for the user interaction.

In order to ensure smooth operation when waiting for button presses for the LPDSP32, we have defined a simple state machine with four states, which forms the bulk of the main program. The states are:

- IDLE: The program is waiting for the user to press a button.
- FIRE: The user has pressed a button so the program sets up the next number in the sequence, increments the sequence number, and notifies the LPDSP32 that there is work to be done.
- BUSY: The Arm Cortex-M3 core has indicated to the LPDSP32 that it needs to perform a calculation and is waiting for a response
- DONE: The LPDSP32 has completed the calculation and the result is available.

Pseudo Code for the State Machine

```
ourState = stateIDLE;
while (True)
{
   switch (ourState)
   {
      stateIDLE :
         /* Refresh the watchdog timer, until a button has been pressed */
         break;
       stateFIRE :
         /* Set the sequence number in shared memory */
         /* Calculate the next sequence number */
         /* Change to busy state until LPDSP32 is finished */
         ourState = stateBUSY;
         /* Trigger an interrupt on the LPDSP32 */
         break;
      stateBUSY :
         /* Just wait for the interrupt to occur indicating the LPDSP32 is complete */
         break;
      stateDONE :
         /* Read the result from shared memory and do something with it */
         /* Go back to idle state */
         ourState = stateIDLE;
         break;
   }
}
```

In addition to the state machine which controls the application, we need to define the shared memory locations we will use, and the interrupt handlers for both the button presses and the LPDSP32 response.

4.2.1 Button Press Handler

The button press IRQ Handler is set up in exactly the same manner as the *Blinky* example. In this case, however, we just need the handler to set the program state to FIRE when a button is pressed. Note that we only respond to button presses when in the IDLE state.

Button Press IRQ Handler

```
void DIO0_IRQHandler(void)
{
   static uint8_t ignore_next_dio_int = 0;
   if (ignore_next_dio_int)
   {
       ignore_next_dio_int = 0;
   }
   else if ((DIO_DATA->ALIAS[BUTTON_DIO] == 0) && (ourState == stateIDLE))
   {
       /* Button is pressed: Ignore next interrupt.
       * This is required to deal with the debounce circuit limitations. */
       ignore_next_dio_int = 1;

       /* Set the state to FIRE, indicates something needs to be done */
       ourState = stateFIRE;
   }
}
```

4.2.2 LPDSP32 Response ISR

The LPDSP32 IRQ handler is very straightforward and only needs to set the program state to DONE.

Interrupt Service Routine

```
void DSP0_IRQHandler(void)
{
    if (ourState == stateBUSY)
        ourState = stateDONE;
}
```

In addition to this, we need to enable the interrupt as part of the standard application initialization. This is very similar to the DIO interrupt handler.

In the System Initialisaton Routine

```
/* Enable the interrupt zero from the LPDSP32 */
NVIC_EnableIRQ(DSP0_IRQn);
```

4.2.3 Shared Memory

The firmware reference manual provides an overview of the full memory map of the system from the perspective of both the Arm Cortex-M3 core and the LPDSP32. This should be understood when sharing memory between the two processors.

For the purposes of this example, we need two words that are visible to both the Arm Cortex-M3 core and the LPDSP32. We can choose any suitable locations, provided we take care to avoid other items that may be trying to share the space. For now we will allocate the words at fixed locations that we know are not conflicting. From the Arm Cortex-M3 core, we use two words based at address `0x20011800`. These appear in the memory map of the LPDSP32 at location `0x00803800` in the DMB memory space.

In both cases, the memory locations are accessed via `uint32_t` pointers.

4.3 Setting Up The LPDSP32 Side

On the LPDSP32, the code is somewhat simpler as it only needs to respond to a single event: the interrupt from the Arm Cortex-M3 core telling it that a new number needs to be calculated. Given this, it can spend most of its time asleep and just awaken when prompted.

The code for the main loop is shown below. As you can see, this uses a flag to determine when some work needs to be done. The processor is put to sleep using the `core_halt()` method, and awakens when any events occur. The interrupt service routine handling the interrupt from the Arm Cortex-M3 core sets this flag, indicating that a new number needs to be calculated.

Once the new number has been calculated, the Arm Cortex-M3 core is notified by writing the appropriate value to `CssCmdGen`, which raises the corresponding interrupt.

Main Loop on LPDSP32

```
extern "C" int main(void)
{
    /* Assume nothing ready to be processed yet */
    actionRequired = false;

    /* Enable interrupts so that the Arm Cortex-M3 core can notify us */
    /* of work to do */
    enable_interrupts();

    /* Spin forever, waiting for interrupts from the Arm Cortex-M3 core */
    while (true)
    {
        /* Put the Arm Cortex-M3 core to sleep until we receive an interrupt */
        core_halt();

        /* Only do something if we are responding to an interrupt from */
        /* the Arm Cortex-M3 core */
         if (actionRequired == true)
        {
            actionRequired = false;

            /* Calculate the requested fibonacci number */
            sharedMemory[1] = fibonacci(sharedMemory[0]);

            /* Notify the Arm Cortex-M3 core that we have completed processing */
            CssCmdGen = CSS_CMD_0;
        }
    }

    return 0;
}
```

As indicated above, the ISR for the interrupt from the Arm Cortex-M3 core just needs to set the `actionRequired` flag to true. The main loop continues from the `core_halt()` instruction on termination of this ISR, and the flag indicates that a new value needs to be calculated.

Interrupt Service Routine

```
extern "C" void ARM1_IRQHandler() property (isr)
{
    /* Raise the flag indicating something needs to be processed */
    actionRequired = true;
}
```

In order to tie the interrupt service routine to the actual interrupt being raised, we provide a jump instruction in the Interrupt Vector Table as shown below. It happens that the specific interrupt from the Arm Cortex-M3 core arrives as interrupt 10.

Interrupt Vector Table; Interrupt 10 Refers to Arm Cortex-M3 Core Interrupt

```
// the interrupt vector table with 15 interrupts
.text global 0 _ivt
    jp _main_init                   /*  0 Reset interrupt */
    reti ; nop                      /*  2 DMA channel 0 interrupt */
    reti ; nop                      /*  4 DMA channel 1 interrupt */
    reti ; nop                      /*  6 DMA channel 2 interrupt */
    reti ; nop                      /*  8 DMA channel 3 interrupt */
    reti ; nop                      /* 10 DMA channel 4 interrupt */
    reti ; nop                      /* 12 DMA channel 5 interrupt */
    reti ; nop                      /* 14 DMA channel 6 interrupt */
    reti ; nop                      /* 16 DMA channel 7 interrupt */
    reti ; nop                      /* 18 Arm Cortex-M3 core 0 interrupt */
    jp ARM1_IRQHandler              /* 20 Arm Cortex-M3 core 1 interrupt */
    reti ; nop                      /* 22 Arm Cortex-M3 core 2 interrupt */
    reti ; nop                      /* 24 Arm Cortex-M3 core 3 interrupt */
    reti ; nop                      /* 26 Arm Cortex-M3 core 4 interrupt */
    reti ; nop                      /* 28 Arm Cortex-M3 core 5 interrupt */
    reti ; nop                      /* 30 Arm Cortex-M3 core 6 interrupt */
```

4.4 PUTTING IT TOGETHER

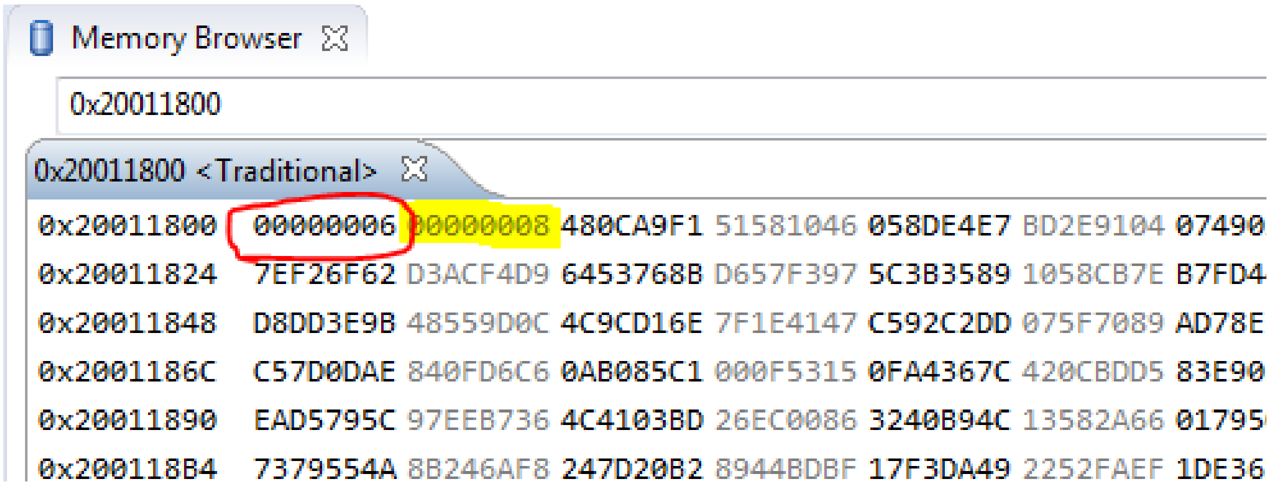
The code for both the Arm Cortex-M3 core and LPDSP32 programs is provided as sample applications in the RSL10 SDK. These need to be opened in their respective IDEs and the projects rebuilt.

4.4.1 LPDSP32

- Ensure JTalk is running and is **Waiting for traffic...**.
- Start the ChessDE software and load the *Fibonacci.prx* project.
- Build the project and ensure there are no errors or warnings.
- Ensure the **lpdsp32_client** debugger is selected.
- Start the debugger from within the IDE and verify that it can start correctly.
- This initializes to the entry point of the program. At this stage, select Run so that the LPDSP32 program is always ready for input.

4.4.2 Arm Cortex-M3 Core

- Start the RSL10 IDE and import the Fibonacci sample code as a project.
- Build the project and ensure there are no errors or warnings.
- Create a standard GDB SEGGER J-Link Debugging session to run the code.
- Start the debugger and ensure it stops at the start of the main code.
- Add a second breakpoint in the `stateDONE` part of the switch statement in the main loop.
- Run the program and verify that when the user button is pressed, the LED flashes on for a very short time and then goes off... and that the breakpoint is hit.
- By examining memory, it will be clear whether the Fibonacci series is being calculated, one number per key press.
    - For example, Figure 3, below, demonstrates that on the seventh button press, the index is 6 (starting at zero), and the seventh Fibonacci number is 8.



```
0x20011800  00000006 00000008 480CA9F1 51581046 058DE4E7 BD2E9104 07490
0x20011824  7EF26F62 D3ACF4D9 6453768B D657F397 5C3B3589 1058CB7E B7FD4
0x20011848  D8DD3E9B 48559D0C 4C9CD16E 7F1E4147 C592C2DD 075F7089 AD78E
0x2001186C  C57D0DAE 840FD6C6 0AB085C1 000F5315 0FA4367C 420CBDD5 83E90
0x20011890  EAD5795C 97EEB736 4C4103BD 26EC0086 3240B94C 13582A66 01795
0x200118B4  7379554A 8B246AF8 247D20B2 8944BDBF 17F3DA49 2252FAEF 1DE36
```

Figure 3. Fibonacci Sample Code Output

4.5 Summary

In this section we created a very simple application which requires communication and co-operation between the Arm Cortex-M3 core and the LPDSP32 core, demonstrating the following principles:

- The use of shared memory to pass information between the cores
- How to trigger an interrupt on the LPDSP32 from the Arm Cortex-M3 core
- The use of interrupts on the LPDSP32 to receive events from the Arm Cortex-M3 core
  - How to define the interrupt service routine (ISR)
  - How to add the ISR to the Interrupt Vector Table for the LPDSP32
  - How to put the LPDSP32 core to sleep when it is not actively busy

- How to trigger interrupts from the LPDSP32 to the Arm Cortex-M3 core
- The use of interrupts on the Arm Cortex-M3 core to receive events from the LPDSP32
  - How to define the DSP ISR
  - How to enable the interrupt so that the ISR is triggered

CHAPTER 5

Arm Cortex-M3 Core – LPDSP32 Integration

So far we have seen how to create and debug simple applications on the LPDSP32, using the Chess development environment and a JTAG connector attached to specified DIOs on the device.

We have also seen a very simple application which uses shared memory and interrupts to allow communication between programs running on the Arm Cortex-M3 core and LPDSP32 processors.

This certainly allows us to get a basic test application up and running, but it relies on the Chess debugger being connected to the device and obviously couldn't be deployed as a real application. In order to do this, we need to be able to have the LPDSP32 program reside on the flash memory of the RSL10, and then have it loaded into RAM when required. This next sections explain some of the techniques involved in doing this.

5.1 Deployment of LPDSP32 Code in an Application

The LPDSP32 is designed to execute its code from DSP PRAM. As such, some mechanism is needed to get the program into PRAM before it can run.

If we bind the LPDSP32 code to the Arm Cortex-M3 core executable in some way, we can utilize the Arm Cortex-M3 core to perform the loading operation. The Arm Cortex-M3 core application would normally reside in flash, so we can store the LPDSP32 program with the Arm Cortex-M3 core and then have the Arm Cortex-M3 core copy the data over as part of the initialization stage. RSL10 also provides an efficient Flash Copier block which can be employed by the Arm Cortex-M3 core to enable this.

In order to achieve this process, the executable LPDSP32 file must be parsed to extract program and data information in much the same way an operating system may load a program from disk. This is quite a complex task for a low power, resource constrained system, and would be wasteful of space, as there is a lot of non-applicable information in the executable file that is not needed. We have therefore opted to use a preprocessing step to extract the relevant data from the executable, and provide it to the Arm Cortex-M3 core application as a series of in memory data structures. This allows us to limit the work done on the RSL10 and ensure that only important information needs to be stored on the device.

The LPDSP32 executable is a fairly standard Executable and Linkable Format (ELF) file, information on which can easily be found on the internet. A brief overview of the component parts can be seen at https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

5.1.1 Process Overview

Before delving into the details of the process, consider the following example as an introduction to the techniques we use:

1. The LPDSP32 program may consist of an ELF file with a single program segment and a single data segment.
2. The program segment has an address offset into the file from where it can be located, as well as a size and an address in PRAM to which it needs to be put.
3. The data segment is constructed similarly except that its load address is in DRAM

We need to extract the data for these two segments in some way, and remember the load address and size of each one. We also need some way of providing this information to our Arm Cortex-M3 core loader program.

We can achieve this by providing two pieces of information and an array for each segment, where the array contains the bytes read from the file. This can easily be generated into C code as follows:

C Code Providing Information and Byte Arrays for Data Segments

```
#define Program_Segment_Size     0x00000000 /* hex value would contain size in bytes */
#define Program_Segment_Address  0x00000000 /* hex value would contain load address  */
uint8_t Program_Segment_Data[] = {         /* one byte value for each byte in segment */
   0x00, 0x00, ...., 0x00
};


#define Data_Segment_Size        0x00000000 /* hex value would contain size in bytes */
#define Data_Segment_Address     0x00000000 /* hex value would contain load address  */
uint8_t Data_Segment_Data[] = {            /* one byte value for each byte in segment */
   0x00, 0x00, ...., 0x00
};
```

These memory structures can be included in a C project for the Arm Cortex-M3 core and compiled with the rest of the code, which would result in the LPDSP32 segments being accessible by the Arm Cortex-M3 core.

This is obviously a simplification of the overall process, but it provides the basic concepts we will employ in the next sections.

5.2 Extracting Program Data From LPDSP32 Binaries

In order to extract the appropriate program and data information from the LPDSP32 executable files, we provide a tool which parses the file, identifies the important pieces, and exports this data as C structures.

This tool has been implemented in Python, and is deployed as source code, so the internal operation of it is available for inspection.

NOTE: This application has been developed in Python 2.7. In addition, it makes use of the pyelftools package. This can be installed using the command `python -m pip install pyelftools`. While this application has been developed using standard Python, we have chosen to distribute it as an Eclipse PyDev project to allow for ease of use.

5.2.1 LPDSP32 Memory Area Overview

Full details of the memory layout for the LPDSP32 are provided in the *RSL10 Hardware Reference*.

For the purposes of this application, when processing the memory segments, there are three areas we need to deal with, as indicated in Table 2.

Table 2. Memory Areas Relevant to This Application

Memory Area	Base Address	Size	Width
Program Memory (PM)	0x00000000	0x00200000*	40 bit
Low Data Memory (DM Lo)	0x00000000	0x00010000	32 bit
High Data Memory (DM Hi)	0x00800000	0x00004000	32 bit

Each of these areas is processed in turn, and a table is generated providing descriptions of the segments which comprise them.

There is a one-to-one mapping between the generated table and any LPDSP32 executable. In this way we can incorporate multiple LPDSP32 programs into a single application and change behavior by loading from different tables.

5.2.1.1 Associated C Files & Structures

Each application is described by C header and source files, which provide the information to access each of these segments. For instance, if we have an `echo` application, then the corresponding source and header files would be *echo_dsp.h and echo_dsp.c*. These two files provide the hooks to describe the memory segments and contain definitions similar to below:

echo_dsp.h

```
#ifndef _ECHO_DSP_H_
#define _ECHO_DSP_H_

/* Auto generated file
*/

typedef struct
{
    void *buffer;
    uint32_t fileSize;
    uint32_t memSize;
    uint32_t vAddress;
} memoryDescription;

typedef struct
{
    memoryDescription *entries;
    uint32_t count;
} memoryOverviewEntry;

typedef struct
    memoryOverviewEntry PM;
    memoryOverviewEntry DMH;
    memoryOverviewEntry DML;
} memoryOverview;

extern memoryOverview __attribute__ ((section (".dsp"))) echo_dsp_Overview;

#endif
```

As can be seen, the contents of the data segments are now available via the `echo_dsp_Overview` symbol. This is available in the Arm Cortex-M3 core application and can be passed to the loader to ensure the application is correctly initialized.

Also, note that the memoryOverview structure contains a single entry for the PM, DMH and DML segments.

The associated C definition code would look similar to this:

echo_dsp.c

```
/* Auto generated file
 */

#include "echo_dsp.h"

memoryOverview echo_dsp_Overview =
{
    { echo_dsp_PM_SegmentList, 3 },
    { echo_dsp_DM_Hi_SegmentList, 1 },
    { echo_dsp_DM_Lo_SegmentList, 4 }
};
```

In this case, there are actually three program segments that need to be loaded, as well as five data segments: one high and four low.

5.2.2 Memory Descriptions

Each memory description is of the same format, regardless of whether it belongs in program or data memory. How the loader needs to handle the segment is different, but the representation is the same.

The memory descriptions for each segment are separated into their own C header and source files, so we have six additional files, all having the same general format:

- *echo_dsp_PM.h* & *echo_dsp_PM.c*
- *echo_dsp_DM_Hi.h* & *echo_dsp_DM_Hi.c*
- *echo_dsp_DM_Lo.h* & *echo_dsp_DM_Lo.c*

As an example, the `echo_dsp_DM_Lo_SegmentList` is defined as follows:

echo_dsp_DM_Lo.h

```
extern memoryDescription __attribute__ ((section (".dsp")))
   echo_dsp_DM_Lo_SegmentList[];
```

The actual implementation of the data appears in the C source file as:

Data Implementation in C Source File

```
/* Auto generated file
 */

#include "echo_dsp_DM_Lo.h"

uint8_t __attribute__ ((section (".dsp"))) echo_dsp_DM_Lo_0[] =
{ };

uint8_t __attribute__ ((section (".dsp"))) echo_dsp_DM_Lo_8[] =
{ };

uint8_t __attribute__ ((section (".dsp"))) echo_dsp_DM_Lo_264[] =
{ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff,
       0xff, 0xff, 0xff, };
```

```
uint8_t __attribute__ ((section (".dsp"))) echo_dsp_DM_Lo_32760[] =
{ };

memoryDescription __attribute__ ((section (".dsp"))) echo_dsp_DM_Lo_SegmentList[] =
{
    { echo_dsp_DM_Lo_0, 0x00000000, 0x00000008, 0x00000000 },
    { echo_dsp_DM_Lo_8, 0x00000000, 0x00000100, 0x00000008 },
    { echo_dsp_DM_Lo_264, 0x00000010, 0x00000010, 0x00000108 },
    { echo_dsp_DM_Lo_32760, 0x00000000, 0x00008000, 0x00007ff8 },
};
```

In this example, it is important to note that we only include data that is actually required to run the program. Some of the data segments have no loadable data associated with them, and these appear as empty array definitions. In addition, the `filesize` field in the `SegmentList` array is set to zero.

This is a very simple example. More complex programs will have many more segments in both the PM and DM tables.

The next section deals with integrating these memory descriptions into the C applications on the Arm Cortex-M3 core, and how we can then have this loaded to the LPDSP32 as part of the system initialization.

5.3 Merging LPDSP32 Code Into An Arm Cortex-M3 Core Application

The previous section indicated how the LPDSP32 memory segments could be extracted by our tool in order to provide C header and source files for inclusion in a Arm Cortex-M3 core program.

Let's walk through the steps required to do this.

1. The first stage is to get an application for the LPDSP32 processor. This can be any application which has been built using the Synopsys tools. You can verify the basic operation prior to use by loading and debugging the application using the Chess IDE and debugger.
2. One of the outputs of the compilation process is an executable file which is stored in the configuration folder of the LPDSP32 project. For the purposes of this discussion, let's assume we are using a project called `echo` and have built the release configuration. This means there will be a file called *Release/echo* under the project folder. This file is the source ELF executable.
3. For simplicity, I have extracted the *elfConverty* Python files to a local folder called *elfConverter*, so the following files are present:

Folder Structure Prior to Generating C Files

```
Z:\>dir
 Volume in drive Z has no label.
 Volume Serial Number is 13DA-8B00

 Directory of Z:\

09/27/2017  05:01 PM    <DIR>          .
09/27/2017  05:01 PM    <DIR>          ..
09/27/2017  05:05 PM    <DIR>          auto
09/19/2017  11:47 AM             4,633 echo.prx
09/18/2017  11:43 AM             3,975 echoWrapper.c
09/27/2017  05:03 PM    <DIR>          elfConvert
```

```
09/27/2017  05:05 PM    <DIR>          Release
               4 File(s)          8,783 bytes
               6 Dir(s)  112,924,954,624 bytes free

Z:\>
Z:\>dir auto
 Volume in drive Z has no label.
 Volume Serial Number is 13DA-8B00

 Directory of Z:\auto

09/27/2017  05:05 PM    <DIR>          .
09/27/2017  05:05 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  112,924,938,240 bytes free

Z:\>
Z:\>dir elfConvert
 Volume in drive Z has no label.
 Volume Serial Number is 13DA-8B00

 Directory of Z:\elfConvert

09/27/2017  05:03 PM    <DIR>          .
09/27/2017  05:03 PM    <DIR>          ..
09/13/2017  07:37 PM               909 DSPIntegrator.py
09/14/2017  02:54 PM            11,344 DSPMemoryIntegrator.py
09/13/2017  07:06 PM               832 DSPSegmentKey.py
09/13/2017  08:17 PM             2,667 DSPSymbolIntegrator.py
09/13/2017  07:58 PM             1,267 elfConverter.py
               5 File(s)         17,019 bytes
               2 Dir(s)  112,924,938,240 bytes free
```

Note that we have an empty folder called *auto*, which is the destination for the generated output files.

4. We can execute the conversion process using python and providing it with the input elf file, output folder, and a tag which defines a prefix for each of the generated files.

Conversion Process Using Python

```
Z:\> python elfConvert\elfConverter.py --elf Release\echo --output auto --tag echo_dsp
```

5. This generates the following files:

Files Generated by Conversion Process

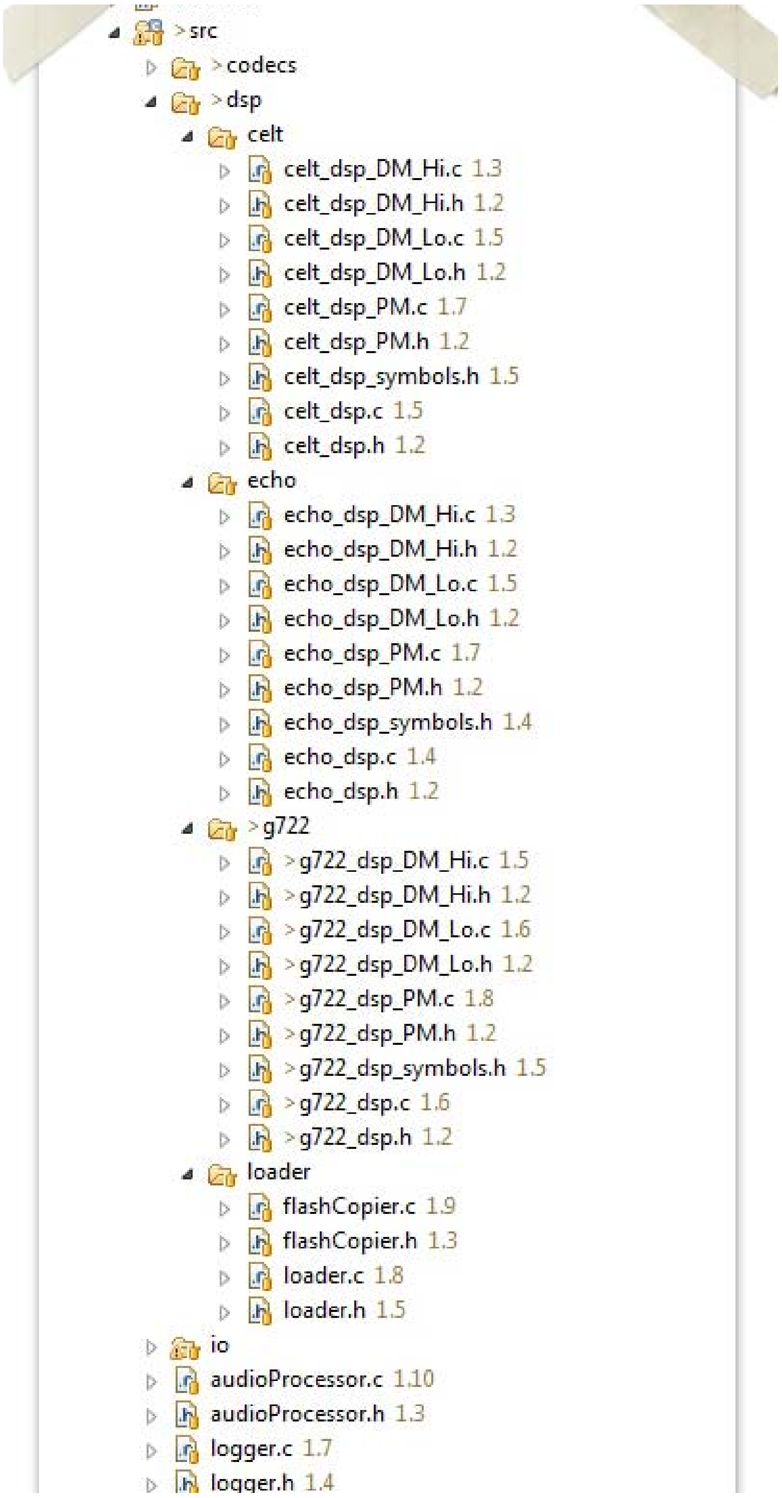
```
Z:\>dir auto
 Volume in drive Z has no label.
 Volume Serial Number is 13DA-8B00

 Directory of Z:\auto

09/27/2017  05:06 PM    <DIR>          .
09/27/2017  05:06 PM    <DIR>          ..
09/27/2017  05:06 PM               210 echo_dsp.c
```

```
09/27/2017  05:06 PM               615 echo_dsp.h
09/27/2017  05:06 PM               305 echo_dsp_DM_Hi.c
09/27/2017  05:06 PM               438 echo_dsp_DM_Hi.h
09/27/2017  05:06 PM               826 echo_dsp_DM_Lo.c
09/27/2017  05:06 PM               438 echo_dsp_DM_Lo.h
09/27/2017  05:06 PM             5,968 echo_dsp_PM.c
09/27/2017  05:06 PM               429 echo_dsp_PM.h
               8 File(s)          9,229 bytes
               2 Dir(s)  112,929,112,064 bytes free
```

6. These files can now be copied into the Arm Cortex-M3 core project in the RSL10 SDK and built as part of that project, making the data and symbols available for use by the Arm Cortex-M3 core to initialize and start the LPDSP32. As an example, Figure 4 demonstrates three different audio codecs incorporated into a single application. Each of these can be loaded and used as needed.



```
◢ >src
    ▷ >codecs
    ◢ >dsp
        ◢ celt
            ▷ celt_dsp_DM_Hi.c 1.3
            ▷ celt_dsp_DM_Hi.h 1.2
            ▷ celt_dsp_DM_Lo.c 1.5
            ▷ celt_dsp_DM_Lo.h 1.2
            ▷ celt_dsp_PM.c 1.7
            ▷ celt_dsp_PM.h 1.2
            ▷ celt_dsp_symbols.h 1.5
            ▷ celt_dsp.c 1.5
            ▷ celt_dsp.h 1.2
        ◢ echo
            ▷ echo_dsp_DM_Hi.c 1.3
            ▷ echo_dsp_DM_Hi.h 1.2
            ▷ echo_dsp_DM_Lo.c 1.5
            ▷ echo_dsp_DM_Lo.h 1.2
            ▷ echo_dsp_PM.c 1.7
            ▷ echo_dsp_PM.h 1.2
            ▷ echo_dsp_symbols.h 1.4
            ▷ echo_dsp.c 1.4
            ▷ echo_dsp.h 1.2
        ◢ >g722
            ▷ >g722_dsp_DM_Hi.c 1.5
            ▷ >g722_dsp_DM_Hi.h 1.2
            ▷ >g722_dsp_DM_Lo.c 1.6
            ▷ >g722_dsp_DM_Lo.h 1.2
            ▷ >g722_dsp_PM.c 1.8
            ▷ >g722_dsp_PM.h 1.2
            ▷ >g722_dsp_symbols.h 1.5
            ▷ >g722_dsp.c 1.6
            ▷ >g722_dsp.h 1.2
        ◢ loader
            ▷ flashCopier.c 1.9
            ▷ flashCopier.h 1.3
            ▷ loader.c 1.8
            ▷ loader.h 1.5
    ▷ io
    ▷ audioProcessor.c 1.10
    ▷ audioProcessor.h 1.3
    ▷ logger.c 1.7
    ▷ logger.h 1.4
```

Figure 4. Three Audio Codecs Incorporated Into a Single Application

5.4 Runtime Loading of the LPDSP32 Code

The basic process of generating the source and header files for an LPDSP32 application has been covered, and we are now ready to look at the code on the Arm Cortex-M3 core which loads this program into memory for use.

5.4.1 The Loader API

As part of our sample code, we provide a codec development framework as an example of how various algorithms can be incorporated into a single Arm Cortex-M3 core application. Part of this framework includes loader routines which employ the RSL10 flash copier block to ensure efficient loading of LPDSP32 memories. The API for this loader is given in the *src/dsp/loader/loader.h* header file.

Extract from loader.h

```
#ifndef LOADER_H_
#define LOADER_H_

/* Normally when loading the LPDSP32 we can just use this routine */
void loadDSPMemory(memoryOverview *overview);

/* Provide a hook to explicitly reset the loop cache */
void resetLoopCache(void);

/* For cases where finer control of the loading process is required, we expose
 * the following helper functions
 */

void loadSinglePRAMEntry(memoryDescription *descriptor);
void loadDSPDRAM(memoryOverviewEntry *hi, memoryOverviewEntry *lo);

#endif /* LOADER_H_ */
```

The main function we need to consider at the moment is the `loadDSPMemory` function; this covers most integration cases.

5.4.2 Using the Loader API

Taking the use of the *echo* sample we introduced in the last section, we can very easily load this to the LPDSP32 by executing the following code:

Loading the echo Sample to LPDSP32

```
loadDSPMemory(&echo_dsp_Overview);
```

This takes the overview structure and loads all the PM and DM memory segments it can to the LPDSP32 memory areas.

There are more complex use cases where different parts of an application may need to be loaded at different times, but these need to be addressed on a case by case basis. As an example of how this may work, the CELT codec loads different parts of the application in at different times, and needs finer control of the loading process. The initialization of that program is shown below:

Initialising the CELT Codec

```
// Define local routine to load sections between known addresses
static void loadPRAM(uint32_t base, uint32_t top)
{
    memoryOverviewEntry *pram = &celt_dsp_Overview.PM;
    for (int i = 0; i < pram->count; i++)
    {
        memoryDescription *description = &pram->entries[i];
        if ((description->vAddress >= base) && (description->vAddress < top))
            loadSinglePRAMEntry(description);
    }
}


...


static void celtDSPInitialise(CODEC codec)
{
    /* load the PRAM for the common and initialisation stages */
    loadPRAM(BASE_COMMON_PM, BASE_INIT_PM);
    loadPRAM(BASE_INIT_PM, BASE_DECODE_PM);

    /* load the DRAM, this is straightforward */
    loadDSPDRAM(&celt_dsp_Overview.DMH, &celt_dsp_Overview.DML);
}
```

CHAPTER 6

Overview of Codec Development Framework

The basic mechanisms of using shared memory to transfer data between the two processors, and interrupts to signal control actions, are very simple concepts that can be built on to make more complex and robust communications structures. For our purposes, however, they provide the basic building blocks for designing some simple applications, which can utilize the extra processing power of the LPDSP32 for specific needs.

One possible application we have found that can benefit from this is to use the LPDSP32 as an audio encoding and decoding engine, implementing some standard audio codecs. The basic premise is that the Arm Cortex-M3 processor can manage the flow of data through the system, while the LPDSP32 is used as a dedicated computational engine performing specific functions. We have provided sample code for the Arm Cortex-M3 core and LPDSP32 demonstrating this. The next few sections of this document describe the overall design and data flow through that system, followed by an explanation of how each of the codecs has been integrated into a common framework.

6.1 Goals of the Application

The primary goal of this sample code is to provide an overview of a possible framework which can be used to develop and integrate different audio codecs into RSL10 applications. The sample takes a fairly high-level view of the behavior of each codec, and focuses instead on how the data can be managed between the Arm Cortex-M3 processor and the LPDSP32.

In addition, a secondary goal of this application is to provide an easy-to-follow mechanism to get developers familiar with programming the two processors in RSL10, showing some of the techniques which need to be managed to make things work smoothly.

6.2 Theory of Operation

Before delving into the code, consider the flow of data as audio is being encoded or decoded. We assume that the audio data is dealt with in fixed length blocks or frames, and that it consists of a number of samples at some rate. For the purposes of this application, we also assume that we can define different sets of parameters to be associated with each frame being processed. The framing and blocking of the data is handled outside of the codec, and complete frames of data are provided to the LPDSP32 when an operation is requested,

Given this information, we can see that we need some form of parameter structure describing the data format, as well as a buffer to hold the encoded and decoded samples. In addition, as this information will need to be known by both the Arm Cortex-M3 processor and the LPDSP32, these three items must exist in shared memory. We can also assume that new data might be arriving whilst the LPDSP32 is busy processing an existing set. Therefore, to ease the management of data into the system, we need some form of queue to store frames before they are passed to the LPDSP32 for processing. By doing this we can maintain throughput, constantly filling the queue as data arrives, handing it off to the LPDSP32 for encoding or decoding when a full frame is available, and then handling the completed data as the LPDSP32 reports that it's finished. Importantly, the LPDSP32 only ever needs to be processing a single frame of data.

As a simplification, we can use ping-pong buffers for the incoming data as we know that the encoder and decoder functions need to be able to keep up with the expected data rates. This is effectively just a queue with max length of two elements: the one being processed and the next one being collected.

In simple terms, the operation of the Arm Cortex-M3 processor main loop would therefore will look something like:

Arm Cortex-M3 Processing Overview

```
in_buffer = 0
out_buffer = 0

status[0] = EMPTY
status[1] = EMPTY

while (true)
{
    if (status[in_buffer] == EMPTY)
    {
        /* TODO load some more data into in_buffer if it is available */
        if (/* TODO we fill the buffer */)
        {
            status[in_buffer] = FULL
            in_buffer = 1 - in_buffer
        }
    }
    if ((status[out_buffer] == FULL) && (status[1 - out_buffer] != PROCESSING))
    {
        status[out_buffer] = PROCESSING
        /* TODO initiate the LPDSP32 to encode or decode the data as required */
    }

    if (status[out_buffer] == READY)
    {
        /* TODO handle the data returned from the LPDSP32 */
        status[out_buffer] = EMPTY
        out_buffer = 1 - out_buffer
    }

}
```

And the Arm Cortex-M3 processor interrupt handler which responds to the LPDSP32 interrupt would deal with indicating that the LPDSP32 has completed any requested encoding or decoding:

Arm Cortex-M3 Service Interrupt Routine

```
void DSP0_IRQHandler(void)
{
    /* When the LPDSP32 is finished, we just need to set the status of the current
     * out_buffer to be ready this will signify that the data can be processed and
     * the LPDSP32 is no longer busy
     */
    status[out_buffer] = READY
}
```

Assuming the LPDSP32 can process the frames of data faster than they arrive, then the system will process samples on a frame by frame basis.

6.3 THE AUDIOCODECS PROJECT

One of the sample code projects associated with this framework holds the code which provides the functionality defined in the previous section. We'll now look at the structure of this project and explain how the project works.

The AudioCodecs project is a standard RSL10 project with all of the source held under the *src* folder. This folder has the following structure:

- AudioCodecs: Top level of project, will have other eclipse content not explained here
  - *src*: The *src* folder contains all of the C source and header files
    - *codecs*: This contains the codec interface definitions and implementations
    - *dsp*: The *dsp* folder contains auto-generated source and header files
    - *io*: This contains test code to feed the harness with different data sets
    - *audioProcessor.c* & *audioProcessor.h*: Provides a standard mechanism for providing audio data to the codecs
    - *logger.c* & *logger.h*: Logger uses GPIO pins to provide status output for debug
    - *lpdsp32_JTAG.c* & *lpdsp32_JTAG.h*: Provides code to establish the JTAG connection to the LPDSP32
    - *sharedBuffers.c* & *sharedBuffers.h*: Defines the structures used to map data in shared memory
    - *main.c*: The main program which drives the framework
    - *linker.lcf*: Linker control file

This is a lot of code to consider, and some of it is really just scaffolding which was useful during development of the code. This has been retained as it may be helpful to other users when investigating how the framework operates. Areas which fall under this category are things like the Logger, the LPDSP32 JTAG which has already been explained, and some of the areas under the *src/io* folder which are effectively mechanisms to get data into and out of the system. Additionally, the dsp code is all auto-generated and the loader mechanisms have been previously explained.

The important code is mainly resident within the *src/codecs* sub-folder. This is where we define the codec interface and provide sample implementations to show how these can be used.

6.3.1 So what is the codec API?

In *src/codecs/codec.h* there is a high-level definition of a codec. It provides some basic definitions as well as a high-level view of the functions or methods that are associated with it.

6.3.1.1 Useful Type Definitions

Known Codecs

```
typedef enum
{
    BASE = 0x01000000, BASE_DSP, ECHO, ECHO_DSP, G722_DSP, CELT_DSP
} KnownCodecs;
```

The KnownCodecs provides a definition of the codecs that the system is aware of. This is the public definition of things that can be created. The first three of these are not relevant to this discussion. The last three (`ECHO_DSP`, `G722_DSP` and `CELT_DSP`) represent three different codecs that behave in different ways, and utilize the LPDSP32 to offload the processing.

EncodeDecode

```
typedef uint8_t EncodeDecode;
#define ENCODE         ((uint8_t) 0x01)  // Perform an encode operation
#define DECODE         ((uint8_t) 0x02)  // Perform a decode operation
#define ENCODE_RESET   ((uint8_t) 0x04)  // Perform a codec encode reset
#define DECODE_RESET   ((uint8_t) 0x08)  // Perform a codec decode reset
#define PACKED         ((uint8_t) 0x10)  // When returning the data, ensure it is packed
#define CONFIGURE      ((uint8_t) 0x80)  // Perform a codec configuration operation
```

The `EncodeDecode` type defines an operation that can be requested of a codec. These are the basic things we will be asking the codec to do.

OperationParameters

```
/* The operational parameters define the parameters to be used for the
 * next operation
 */
typedef struct _OperationParameters
{
    EncodeDecode action;    /* the operation as defined above */
    uint8_t frameSize;      /* the frame size to be used (samples) */
    uint8_t blockSize;      /* the block size or subframe size (samples) */
    uint8_t modeAndChannel; /* for G.722, mode is lower 4 bits, channel is upper */
    uint32_t sampleRate;    /* sample rate associated with the encode/decode */
} OperationParameters;
```

This data block type is used when issuing requests to the LPDSP32.

Note that the mode and channel is defined depending on the codec:

- For G.722, the upper four bits represent the channel number and the lower four bits provide the G.722 mode.
- For CELT, the lower four bits define the channel.

CODEC

```
/* All access to codecs is via the opaque CODEC type */
typedef void *CODEC;
```

When performing operations on a codec, we need some form of handle or context so that we can maintain different types of codecs without worrying too much about the implementation. By using an opaque handle, we can easily change the implementation without affecting the calling routines.

6.3.1.2 Functional API

The creation of new codecs will use a factory pattern, so we don't need to define the specific constructors for these here. All other codec functions will be defined to take a CODEC as their first parameter. This will provide the context for all operations. Table 3, below, shows the methods for working with codecs.

Table 3. Codec Methods

Method	Explanation
`void codecDestructor(CODEC *codec)`	Having created a CODEC of some type, we want to be able to get rid of it using a standard mechanism, so define a destructor method which takes a pointer to a CODEC. This is analogous to the destructor method used in object-oriented programming, but the implementation is dependent on how the CODEC has been constructed. It will not necessarily free memory.
`uint32_t codecIsCodec(CODEC codec)`	Given a CODEC, this method verifies that the CODEC structure underlying the handle seems to be properly configured. This allows us to detect if some invalid memory pointer has been passed or if the memory area used by the handle has been corrupted.

Table 3. Codec Methods (Continued)

Method	Explanation
`void codecInitialise(CODEC codec)`	Initialize a newly created CODEC. This is being provided separately from the CODEC construction as we want a common mechanism to initialize the codec and we want it to be invoked directly from the calling program when necessary.
`KnownCodecs codecGetType(CODEC codec)`	Given a CODEC handle, return the associated `KnownCodec` type.

`void codecSetStatusBuffer(CODEC codec, unsigned char *baseAddress, uint32_t size)`
 	
`void codecSetInputBuffer(CODEC codec, unsigned char *baseAddress, uint32_t size)`
 	
`void codecSetOutputBuffer(CODEC codec, unsigned char *baseAddress, uint32_t size)`
	The `setBuffer` methods allow us to set the base address and size of various data areas used by the CODEC.

`uint32_t codecGetOutputBufferUsed(CODEC codec)`	On completion of the CODEC operation, this retrieves the actual size of the output buffer that has been used.

`void codecConfigure(CODEC codec)`
 	
`void codecEncode(CODEC codec)`
 	
`void codecDecode(CODEC code)`
	These three methods provide basic functionality for encoding, decoding and configuring. Although these operations are also specified as part of the `OperationParameters` control block, having defined methods for them allows us to tune the behavior in some cases.

`void codecSetParameters(CODEC codec, OperationParameters *params)`	Set the operational parameters for the next CODEC operation.

6.3.1.3 Example of Use

Knowing this API, we can define the code to actually perform some codec operations similar to the following:

Example Codec Operation

```
int main(void)
{
    /* Configure the parameter block for the codec. This describes the operation:*/
    OperationalParameters parameters;
    parameters.action = ENCODE_RESET | ENCODE;  /* We'll do an encode in this example */
    parameters.frameSize = SomeFrameSize;       /* Size to be defined somewhere */
    parameters.blockSize = SomeBlockSize;       /* Size to be defined somewhere */
    parameters.modeAndChannel = 0;              /* Not used in an echo operation */
    parameters.sampleRate = 0;                  /* Not used in an echo operation */

    /* Use a factory method to create an ECHO_DSP codec located in the configuration area
   */
    CODEC myCodec = populateCodec(ECHO_DSP, Buffer.configuration,
   CODEC_CONFIGURATION_SIZE);

    /* Make sure the new codec is valid */
    if (! codecIsCodec(myCodec))
        logFatal(0xFFF);
    /* initialize the codec */
    codecInitialise(myCodec);

    /* Configure the static parts of the codec, assuming one input buffer */
    codecSetStatusBuffer(myCodec, Buffer.configuration, CODEC_CONFIGURATION_SIZE);
    codecSetOutputBuffer(myCodec, Buffer.output, CODEC_OUTPUT_SIZE);
    codecSetInputBuffer(myCodec, Buffer.input, CODEC_INPUT_SIZE);

    while (1)
    {
        /* TODO Fill the input buffer */
        ...

        /* Set the parameters for the current frame and turn off the reset */
        codecSetParameters(codec, &parameters);
        parameters.action = ENCODE;

        /* Call encode or decode */
        codecEncode(codec);

        /* TODO Read the output buffer */
        ...
    }

    return 0;
}
```

6.4 Implementing a New Codec

Having defined the framework we will use to communicate with a codec, we now need to consider how we would go about creating a new codec inside this framework.

We have identified a standard API that each codec will provide, but not yet touched on how the codec should be implemented such that it can provide this API in a flexible manner.

We don't want the application code to have to be modified when changing a codec (other than the construction), so we need to provide a function table that redirects the standard API calls to the codec-specific ones as needed. Fortunately, having the opaque handle to the CODEC allows us to implement this function table as part of the handle. This structure is shown in *src/codecs/codecInternal.h*. We won't go into too much detail about the operation of this structure; instead, we will discuss what is involved in creating a new codec and use the Echo implementation as an example. Once that is understood, the operation of the DSP-based Echo, G.722 and CELT examples should be easier to understand.

For this example, we will show how a codec can be implemented solely on the Arm Cortex-M3 core, implementing the same API shown earlier. This demonstrates the basic principles and structure of the code before considering how this needs to interact with the LPDSP32.

6.4.1 Public Interface to a Codec Module

Each module consists of a header file and associated C code file. The header file contains the public interface which generally consists of the means to construct a new module. Nothing else is necessary as all access (once created) is through the standard CODEC API.

For the Echo module, the API is as follows:

Public Interface to Echo Codec

```
#ifndef ECHOCODEC_H_
#define ECHOCODEC_H_

#include "codecs/codec.h"

CODEC makeEchoCodec(void);
CODEC populateEchoCodec(unsigned char *buffer, uint32_t size);

#endif /* ECHOCODEC_H_ */
```

As can be seen, this provides two functions, both of which return a handle to a CODEC structure.

The first function is used when the module is expected to allocate the context block on the heap. In this case, the returned value might be `NULL` if the required space cannot be allocated.

The second function initializes the buffer memory provided and uses that as the handle to the CODEC. The size parameter is checked to ensure there is enough space in which to initailize all of the fields; if not, then `NULL` is returned.

In both cases, on success, the returned CODEC value points to the base of a codec control block as defined in *codecInternal.h*.

These two functions are used in the codec construction factory to create the correct type of codec when requested.

6.4.2 Internal Implementation of a Codec Module

Start by considering the construction of a new codec. In both functions outlined above, the behavior is the same regardless of the codec we are creating. We need to initialize some buffer to a known state and then populate specific fields with information that is specific to the item we are creating. As this is analogous to the derivation of objects, we have provided a base implementation that can be exploited to do much of the common work.

Construction Using the Base Codec Implementation

```
CODEC makeEchoCodec(void)
{
    /* "derive" from the base codec */
    pCodec echo = getCodec(makeBaseCodec());
    initialiseStructure(echo);
    return echo;
}

CODEC populateEchoCodec(unsigned char *buffer, uint32_t size)
{
    /* "derive" from the base codec */
    pCodec echo = getCodec(populateBaseCodec(buffer, size));
    initialiseStructure(echo);
    return echo;
}
```

In both cases:

1. The base implementation is used to get the structure we will use.
2. The structure is initialized.
3. The pointer to the CODEC is returned as the handle.

The initialization routine is used to set up the specific vectors for the ECHO implementation. In this case, it needs to provide implementations for the encode and decode functions. Nothing else is required for this simple example.

Initialising an Echo codec

```
static void initialiseStructure(pCodec structure)
{
    if (codecIsCodec(structure))
    {
        /* In this example, encode and decode will do the same thing so */
        /* we can reuse the same function here for both vectors         */
        structure->type = ECHO;
        structure->encode = &echoEncode;
        structure->decode = &echoEncode;
    }
}
```

This block is indicating that the encode and decode functions have been "over-ridden" from the base implementation and we now need to provide some code implementing the new functionality.

Consider the encode functionality. Remember this is just performing an echo function, so we can just copy the input to the output and we are done. The decode is exactly the same as the echo doesn't change the data, so we can reuse that function directly.

Encode Functionality

```
static void echoEncode(CODEC codec)
{
    /* TODO copy the input buffer to the output buffer */
    ...

    /* TODO setup the size of the used output buffer */
    ...

    /* notify completion of the operation */
    echo->notifyComplete();
}
```

And that's all that is required to implement a very simple transfer function. Of course this is not actually using the LPDSP32 and that will bring more complexities but this provides the basic techniques required.

The next sections demonstrates the implementation of the API when communicating with the LPDSP32.

Complete Implementation of the Echo Codec

```
#include <string.h>

#include "codecs/base/baseCodec.h"
#include "codecs/codecInternal.h"
```

```
/* Define function prototypes for the features we will override from the base */
static void echoEncode(CODEC codec);
static void echoDecode(CODEC codec);

static void initialiseStructure(pCodec structure)
{
    if (codecIsCodec(structure))
    {
        structure->type = ECHO;
        structure->encode = &echoEncode;
        structure->decode = &echoEncode;
    }
}

CODEC makeEchoCodec(void)
{
    /* "derive" from the base codec */
    pCodec echo = getCodec(makeBaseCodec());
    initialiseStructure(echo);
    return echo;
}

CODEC populateEchoCodec(unsigned char *buffer, uint32_t size)
{
    /* "derive" from the base codec */
    pCodec echo = getCodec(populateBaseCodec(buffer, size));
    initialiseStructure(echo);
    return buffer;
}

static void echoEncode(CODEC codec)
{
    pCodec echo = getCodec(codec);
    uint32_t sizeToEcho =
            (echo->inputBufferSize > echo->outputBufferMaxSize) ?
                    echo->outputBufferMaxSize : echo->inputBufferSize;
    memcpy((void *) echo->outputBufferAddress,
            (void *) echo->inputBufferAddress, sizeToEcho);
    echo->outputBufferUsedSize = sizeToEcho;
    echo->notifyComplete();
}
```

CHAPTER 7

The Simplest Codec - Echo Encoder / Decoder

7.1 ARM CORTEX-M3 CORE IMPLEMENTATION

So, having seen how the framework can be used to provide a standard API to a codec and how a very simple echo function could be implemented, we can now extend that example to provide the echo functionality utilizing the LPDSP32.

The same general principles apply when communicating with the LPDSP32, but in this case we derive the modules from the BaseDSP implementation rather than the Base. The BaseDSP implementation is itself derived from the Base implementation, but has additional features added to it, such as automatic triggering of interrupts on the LPDSP32, and the provision of a standard handshake protocol between the two cores.

7.1.1 Public Interface

The header file associated with the ECHO_DSP codec is very similar to the Echo; just the functions are renamed appropriately.

Public Interface to the Echo_DSP Codec

```
#ifndef ECHODSPCODEC_H_
#define ECHODSPCODEC_H_

#include "codecs/codec.h"

CODEC makeEchoDSPCodec(void);
CODEC populateEchoDSPCodec(unsigned char *buffer, uint32_t size);

#endif /* ECHODSPCODEC_H_ */
```

7.1.2 Internal Implementation

When constructing an EchoDSP codec, we need to derive it from the BaseDSP object. This has different default behavior which we can exploit in this implementation.

Note that when we create this module, we don't need to over-ride the encode and decode operations. The default implementation in the base works fine.

Construction

```
static void initialiseStructure(pCodec structure)
{
    if (codecIsCodec(structure))
    {
        structure->type = ECHO_DSP;
        structure->initialise = &echoDSPInitialise;
    }
}


CODEC makeEchoDSPCodec(void)
{
    /* "derive" from the base codec */
    pCodec echo = getCodec(makeBaseDSPCodec());
    initialiseStructure(echo);
    return echo;
}

CODEC populateEchoDSPCodec(unsigned char *buffer, uint32_t size)
{
    /* derive from the base codec */
    pCodec echo = getCodec(populateBaseDSPCodec(buffer, size));
    initialiseStructure(echo);
    return buffer;
}
```

We did over-ride the public initialize method. This is to allow us to be able to load the LPDSP32 with the code for the codec.

The initialization assumes that the LPDSP32 code has been compiled and the relevant data areas extracted into C tables for the Arm Cortex-M3 core as described previously. In this case, the initialization routine just needs to load memory.

Initialization of the LPDSP32 PRAM & DRAM

```
static void echoDSPInitialise(CODEC codec)
{
    pBaseDSPCodec object = getDSPCodec(codec);
    loadDSPMemory(&echo_dsp_Overview);
}
```

That is all that is involved in the Arm Cortex-M3 core side of the implementation.

7.2 LPDSP32 Implementation

So far, we have only considered the Arm Cortex-M3 core side of the implementation, but we also need to provide LPDSP32 code which provides the other side of the implementation.

On the LPDSP32, we have a slightly different model when implementing each codec. As we only have one codec active at any given point in time, we can treat each one as an independent program and build them individually.

We have provided a standard framework on the LPDSP32 side which manages the conversation with the Arm Cortex-M3 core. Then for each specific codec, we just need to implement the API specified in a wrapper interface.

This wrapper can be found in *AudioCodecsDSP/src/codecs/wrapper/wrapper.h*, and it specifies the minimum set of functions that must be implemented for a single codec.

Wrapper Interface for LPDSP32 Codecs

```
/* ---------------------------------------------------------------------------
 * Description : Method to provide a means to configure a codec for first use.
 * Inputs :
 *         pConfig : Pointer to a codec configuration block which has been set up
 *         with any information required to allow the configuration.
 *         Exactly what information needs to be set up is codec dependent.
 *         For instance, sample rates and frame sizes should be pre-initialized.
 * Outputs :
 *         No outputs but the configuration process will potentially change the
 *         internal state of the codec.
 */
void configure(pBaseDSPCodec pConfig);

/* ----------------------------------------------------------------------------
 * Description : Method to provide a mechanism to reset the encode portion of a
 * codec to its initial state.
 * Inputs :
 *         pConfig : Pointer to a codec configuration block
 * Outputs :
 *         No outputs but the configuration process will potentially change the
 *         internal state of the codec.
 */
void encodeReset(pBaseDSPCodec pConfig);

/* ----------------------------------------------------------------------------
 * Description : Method to provide a mechanism to reset the decode portion of a
 * codec to its initial state.
 * Inputs :
 *         pConfig : Pointer to a codec configuration block
 * Outputs :
 *         No outputs but the configuration process will potentially change the
 *         internal state of the codec.
 */
void decodeReset(pBaseDSPCodec pConfig);

/* ----------------------------------------------------------------------------
 * Description : Method to encode a block of samples using the specified codec.
 * Inputs :
 *         pConfig : Pointer to a codec configuration block which specifies the
 *         control parameters for the encode operation
 *         inBuffer : a pointer to a buffer containing the samples to be encoded
 * Outputs :
 *         outBuffer : a pointer to a buffer that can be used to write the encoded
 *         information
 */
void encode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer);

/* ----------------------------------------------------------------------------
 * Description : Method to decode a block of samples using the specified codec.
 * Inputs :
 *         pConfig : Pointer to a codec configuration block which specifies the
 *         control parameters for the decode operation
 *         inBuffer : a pointer to a buffer containing the samples to be decoded
 * Outputs :
 *         outBuffer : a pointer to a buffer that can be used to write the decoded
 *         samples
 */
void decode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer);
```

This interface provides all the hooks used by the main application in response to requests from the Arm Cortex-M3 core.

For the EchoDSP codec, most of these functions do not need any implementation. The structure is shown below:

Structure for EchoDSP Codec

```
void configure(pBaseDSPCodec pConfig)
{
    /* Nothing required here, not used in ECHO codec */
}

void encodeReset(pBaseDSPCodec pConfig)
{
    /* Nothing required here, not used in ECHO codec */
}

void decodeReset(pBaseDSPCodec pConfig)
{
    /* Nothing required here, not used in ECHO codec */
}

void encode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    /* Only copies full words. This is an example implementation. */
    uint32_t togo = pConfig->parent.inputBufferSize >> 2;

    /* Copy the number of words specified above */
    uint32_t *pDst = (uint32_t *) outBuffer;
    uint32_t *pSrc = (uint32_t *) inBuffer;
    while (togo-- > 0)
        *pDst++ = *pSrc++;

    /* Set up the buffer used size to be the same as the input size */
    pConfig->parent.outputBufferUsedSize = pConfig->parent.inputBufferSize;
}

void decode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    /* Encode is the same as decode; just call the other method */
    encode(pConfig, inBuffer, outBuffer);
}
```

As you can see, this is a very simple implementation, but the techniques are much the same as those used in the more complex examples.

CHAPTER 8

G.722 Encoder / Decoder

Moving on to a more complex example, we can consider how a G.722 encoder/decoder may be implemented using similar techniques to the EchoDSP example.

8.1 ARM CORTEX-M3 CORE IMPLEMENTATION

8.1.1 Public Interface

The header file associated with the G722_DSP codec is very similar to the EchoDSP; just the functions are renamed appropriately.

Public Interface to the G722DSP Codec

```
#ifndef G722DSPCODEC_H_
#define G722DSPCODEC_H_
#include "codecs/codec.h"

CODEC makeG722DSPCodec(void);
CODEC populateG722DSPCodec(unsigned char *buffer, uint32_t size);

#endif /* G722DSPCODEC_H_ */
```

8.1.2 Internal Implementation

The internal implementation is also very similar to the EchoDSP example; the construction is exactly the same other than function names.

Construction

```
static void initialiseStructure(pCodec structure)
{
    if (codecIsCodec(structure))
    {
        structure->type = G722_DSP;
        structure->initialise = &g722DSPInitialise;
    }
}

CODEC makeG722DSPCodec(void)
{
    pCodec g722 = getCodec(makeBaseDSPCodec());
    initialiseStructure(g722);
    return g722;
}

CODEC populateG722DSPCodec(unsigned char *buffer, uint32_t size)
{
    pCodec g722 = getCodec(populateBaseDSPCodec(buffer, size));
    initialiseStructure(g722);
    return buffer;
}
```

When over-riding the initialize method, we have chosen in this case to halt and resume the LPDSP32 while we are updating its internal state. This is generally good practice.

Initialization of the LPDSP32 PRAM & DRAM

```
static void g722DSPInitialise(CODEC codec)
{
    pBaseDSPCodec object = getDSPCodec(codec);
    SYSCTRL->DSS_CTRL = DSS_LPDSP32_PAUSE;
    loadDSPMemory(&g722_dsp_Overview);
    SYSCTRL->DSS_CTRL = DSS_LPDSP32_RESUME;
}
```

That is all that is required on the Arm Cortex-M3 core side of the implementation. As you can see, this is very similar to the EchoDSP codec.

8.2 LPDSP32 Implementation

As for the EchoDSP codec, we only need to consider the implementation of the wrapper functions. In this case however, the wrapper is much more complex as we have a real codec that must be integrated into our framework.

For the purposes of this example, we have used an off-the-shelf, open source G.722 implementation and adjusted it where necessary to fit into our structure.

We will show below the high level implementation of the wrapper functions, but for full implementation details, please refer directly to the sample source file. (*AudioCodecsDSP/src/codecs/g722/g722Wrapper.c*)

Setup and Reset Functions for G722

```
void configure(pBaseDSPCodec pConfig)
{
    /* Nothing required here; not used in G.722 codec */
}

void encodeReset(pBaseDSPCodec pConfig)
{
    /* This fetches the current encoder configuration and resets it
     */
    g722_config *myState = getEncoderState(pConfig);
    reset(myState, pConfig);
    g722_reset_encoder(&myState->state);

}

void decodeReset(pBaseDSPCodec pConfig)
{
    /* This fetches the current decoder configuration and resets it
     */
    g722_config *myState = getDecoderState(pConfig);
    reset(myState, pConfig);
    g722_reset_decoder(&myState->state);
}
```

Encode Function for G722

```
void encode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    g722_config *myState = getEncoderState(pConfig);
    if ((myState->bitsPerSample < 8) && (myState->packed != 0))
    {
        /* If we are packing the bits, necessary for mode 2 & 3, encode into a temporary
         * buffer and then pack the buffer. Note that we have two alternative mechanisms
         * for packing the data
         */
        encodeBuffer(myState, (short *) inBuffer, tempBuffer);
        if (myState->packed == PACKED)
            packBuffer(myState, tempBuffer, outBuffer);
        else
            packBufferAlt(myState, tempBuffer, outBuffer);
    }
    else
    {
        /* No packing required; encode straight into the output buffer
         */
        encodeBuffer(myState, (short *) inBuffer, outBuffer);
    }


    /* Update the used buffer size */
    pConfig->parent.outputBufferUsedSize = expectedEncodedBytes(myState);
}
```

Decode function for G722

```
void decode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    g722_config *myState = getDecoderState(pConfig);
    if ((myState->bitsPerSample < 8) && (myState->packed != 0))
    {
        /* If we are unpacking the data, again modes 2 & 3, unpack into a temp
         * buffer and then decode the temp buffer
         */
        if (myState->packed == PACKED)
            unpackBuffer(myState, inBuffer, tempBuffer);
        else
            unpackBufferAlt(myState, inBuffer, tempBuffer);
        decodeBuffer(myState, tempBuffer, (short *) outBuffer);
    }
    else
    {
        /* If not packing, decode straight into output buffer
         */
        decodeBuffer(myState, inBuffer, (short *) outBuffer);
    }

    /* Update the used buffer size */
    pConfig->parent.outputBufferUsedSize = expectedDecodedBytes(myState);
}
```

This demonstrates the mechanisms we need to implement to integrate into our framework. This wrapper, together with the G.722 codec files and the main program implementation, needs to be built to provide the loadable module used by the Arm Cortex-M3 core.

CHAPTER 9

CELT Encoder / Decoder

Finally, we'll talk about the CELT encoder/decoder. This is a much larger piece of code which provides wide-band audio compression. Again, we have taken an open source implementation of the codec with some minor modifications to enable it to perform well on the LPDSP32 and integrated it into the same environment we have been discussing.

As this is a larger piece of code, we found that the encode and decode portions of the codec are too large to exist in the limited LPDSP32 PRAM space. This is a problem if we wish to perform encode and decode operations sequentially, say for bi-directional communication. To get round this, we have carefully located the different parts of the algorithms into separate memory locations and will page these into memory as required. This all happens on the Arm Cortex-M3 core as part of the codec operation, so let's consider the LPDSP32 side first of all, as this is more straightforward.

9.1 LPDSP32 Implementation

Again we only need to consider the implementation of the wrapper functions. In this case, the configure function is important, but the encode and decode reset features are not used.

We will show below the high level implementation of the wrapper functions, but for full implementation details, please refer directly to the sample source file (*AudioCodecsDSP/src/codecs/celt/celtWrapper.c*).

CELT Configuration

```
void configure(pBaseDSPCodec pConfig)
{
    int errorState;

    OperationParameters *op = &pConfig->parent.operation;
    int frameSize = op->frameSize;
    int sampleRate = op->sampleRate;

    /* Only interested in the "mode" component which specifies the number of
     * channels being used
     */
    int channels = op->modeAndChannel & 0x0F;

    encoderMode = opus_custom_mode_create(sampleRate, frameSize, &errorState);
    if (encoderMode < 0)
    {
        errorState = (int) encoderMode;
    }
    if (errorState != OPUS_OK)
    {
        pConfig->dspError = errorState;
        return;
    }

    encoder = opus_custom_encoder_create(encoderMode, channels, &errorState);
    if (errorState != OPUS_OK)
    {
        pConfig->dspError = errorState;
        return;
    }

    decoder = opus_custom_decoder_create(encoderMode, channels, &errorState);
    if (errorState != OPUS_OK)
    {
        pConfig->dspError = errorState;
        return;
    }

    opus_custom_decoder_ctl(decoder, OPUS_GET_LOOKAHEAD(&encoderLookahead));
    opus_custom_encoder_ctl(encoder, OPUS_SET_COMPLEXITY(0));
}
```

CELT Encode

```
void encodeReset(pBaseDSPCodec pConfig)
{
    /* Nothing required here; not used for CELT codec */
}

void encode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    OperationParameters *op = &pConfig->parent.operation;
    int frameSize = op->frameSize;
    int blockSize = op->blockSize;

    int length = opus_custom_encode(encoder, (opus_int16 *) inBuffer, frameSize,
   outBuffer, blockSize);
    if (length != blockSize)
    {
        pConfig->dspError = length;
        return;
    }

    opus_custom_encoder_ctl(encoder, OPUS_GET_FINAL_RANGE(&encoderFinalRange));
    pConfig->parent.outputBufferUsedSize = length;
}
```

CELT Decode

```
void decodeReset(pBaseDSPCodec pConfig)
{
    /* Nothing required here; not used for CELT codec */
}

void decode(pBaseDSPCodec pConfig, unsigned char *inBuffer, unsigned char *outBuffer)
{
    OperationParameters *op = &pConfig->parent.operation;
    int frameSize = op->frameSize;
    int blockSize = op->blockSize;

    int length = opus_custom_decode(decoder, inBuffer, blockSize,
            (opus_int16 *) outBuffer, frameSize);
    if (length != frameSize)
    {
        pConfig->dspError = length;
        return;
    }
    pConfig->parent.outputBufferUsedSize = length << 1;
}
```

Again, the implementation code is relatively light and as far as the LPDSP32 is concerned. There is no need to do anything different to handle the paging of different data sets into memory.

The management of the specific locations of each function has been done inside the LPDSP32 project, using a very specific LPDSP32 linker control specification. This has mapped the configuration, the encode and the decode blocks into unique memory areas which do not overlap or have dependencies on each other. Refer to the LPDSP32 *CELT* project for more details.

9.2 ARM CORTEX-M3 CORE IMPLEMENTATION

9.2.1 Public Interface

Standard header file, providing the constructor and initialization routine:

Public Interface to the CELTDSP Codec

```
#ifndef CELTDSPCODEC_H_
#define CELTDSPCODEC_H_

#include "codecs/codec.h"

CODEC makeCeltDSPCodec(void);
CODEC populateCeltDSPCodec(unsigned char *buffer, uint32_t size);

#endif /* CELTDSPCODEC_H_ */
```

9.2.2 Internal Implementation

The internal implementation is also very similar to the public interface; however, note that we now provide over-ridden methods for the `PrepareConfigure`, `PrepareEncode` and `PrepareDecode` methods. These prepare methods are called on the Arm Cortex-M3 core prior to the invocation of the `Configure`, `Encode` and `Decode` calls allowing for custom set-up to be done, if necessary.

Construction

```
static void initialiseStructure(pCodec structure)
{
    if (codecIsCodec(structure))
    {
        structure->type = CELT_DSP;
        structure->initialise = &celtDSPInitialise;
        structure->prepareConfigure = &celtDSPPrepareConfigure;
        structure->prepareEncode = &celtDSPPrepareEncode;
        structure->prepareDecode = &celtDSPPrepareDecode;
    }
}


CODEC makeCeltDSPCodec(void)
{
    pCodec celt = getCodec(makeBaseDSPCodec());
    initialiseStructure(celt);
    return celt;
}

CODEC populateCeltDSPCodec(unsigned char *buffer, uint32_t size)
{
    pCodec celt = getCodec(populateBaseDSPCodec(buffer, size));
    initialiseStructure(celt);
    return buffer;
}
```

The initialization code now needs to handle setting up the PRAM sections, only taking into account those sections that will be needed to get the basic system up and running. This uses a custom loader to only copy specific areas of memory from Flash to the LPDSP32 PRAM.

As can be seen from the code snippets below, the common and initialization sections of PRAM are loaded, based on known address ranges.

Initialisation of the LPDSP32 PRAM & DRAM

```
static void loadPRAM(uint32_t base, uint32_t top)
{
    memoryOverviewEntry *pram = &celt_dsp_Overview.PM;
    for (int i = 0; i < pram->count; i++)
    {
        memoryDescription *description = &pram->entries[i];
        if ((description->vAddress >= base) && (description->vAddress < top))
            loadSinglePRAMEntry(description);
    }
}

static void celtDSPInitialise(CODEC codec)
{
    pBaseDSPCodec celt = getDSPCodec(codec);

    /* load the PRAM for the common and initialization stages */
    loadPRAM(BASE_COMMON_PM, BASE_INIT_PM);
    loadPRAM(BASE_INIT_PM, BASE_DECODE_PM);

    /* load the DRAM */
    loadDSPDRAM(&celt_dsp_Overview.DMH, &celt_dsp_Overview.DML);

    /* Run LPDSP32 */
    resetLoopCache();
    SYSCTRL->DSS_CTRL = DSS_RESET;
}
```

Now that the basic initialization has been done, we need to be able to handle changing the running code when we swap between the configuration, encode and decode features. This is managed by the `prepare*` functions we mentioned earlier. In each case, the operation is the same, load the memory space defined for the feature. These functions are shown below.

Prepare Functions

```
static void celtDSPPrepareConfigure(CODEC codec)
{
    loadPRAM(BASE_INIT_PM, BASE_DECODE_PM);
}

static void celtDSPPrepareEncode(CODEC codec)
{
    loadPRAM(BASE_ENCODE_PM, BASE_UNUSED_PM);
}

static void celtDSPPrepareDecode(CODEC codec)
{
    loadPRAM(BASE_DECODE_PM, BASE_ENCODE_PM);
}
```

APPENDIX A

Contents of LPDSP32 Codec Support Package

A.1 OVERVIEW

Whilst this package is designed to provide a starting point for future codec development using RSL10, many of the techniques and tools employed are applicable to other domains.

This provides an overview of the Codec Support Package for RSL10.

Five basic components are provided as reference points. These are outlined below (more details of each one are given farther down the page):

- LPDSP32 Startup Files - These provide the basic functionality used in most LPDSP32 programs.
- AudioCodecs Eclipse Project - This provides the Arm Cortex-M3 core functionality wrapped as an Eclipse project.
- AudioCodecsDSP Eclipse Project - This provides the LPDSP32 functionality, and is shipped as an Eclipse project with multiple LPDSP32 projects embedded within it.
- DSPIntegration Eclipse Project - This provides a suite of Python scripts which allow us to integrate the LPDSP32 output code with the Arm Cortex-M3 core.
- AudioIO Eclipse Project - This provides a mechanism to test the various other components, using Python scripts to feed raw and compressed audio though the codecs for testing the encode and decode functions respectively.

Together these five items provide a comprehensive framework demonstrating how the LPDSP32 can be used as a compression/decompression engine for various audio codecs.

A.2 AUDIO CODEC DEVELOPMENT

A.2.1 AudioCodecs project

The AudioCodecs project provides a C-based framework for building and testing codecs as identified in the associated documentation.

This Eclipse project is designed to provide the Arm Cortex-M3 core code to start up a sample application, initialize a specified codec, and then feed test data to it via various test methods.

A common interface is defined to ensure that each codec behaves in a similar manner, and to allow the framework to be easily extended with new codecs as they become available.

For the initial release, three DSP-based codecs are provided:

- Echo, a simple passthrough codec which does not change the data being fed through it. This is provided as a teaching aid and is heavily used in this documentation.
- G722, an example of integrating a third party G722 codec suitable for speech into the development framework, providing encode and decode functionality in modes 1, 2 & 3.
- CELT, an example of a more complex codec suitable for higher bandwidth audio, again providing encode and decode functionality.

As well as the basic codec implementation, this project also includes some additional features demonstrating how the RSL10 device can be used, showing the following:

- Use of the flash copier facilities

- Enabling the LPDSP32 JTAG
- Inclusion of LPDSP32 executable code within an Arm Cortex-M3 core project
- Synchronization between the Arm Cortex-M3 core and LPDSP32

A.2.2 AudioCodecsDSP Project

This project is complementary to the Arm Cortex-M3 core project indicated above, but in this case the focus is on the LPDSP32 implementation of each of the three codecs.

In each case, a common framework is provided into which the specifics of each codec is merged. This provides a common interface to each one, and allows us to switch between different sets of functionalities quickly and easily.

Each codec is a standalone LPDSP32 project, and is designed to be built with the Synopsys LPDSP32 Chess IDE.

A.2.3 AudioIO Project

The Audio IO project is designed as a test harness for the two projects identified above, and allows control of the running code via the GDB debugger.

This project is designed as a small set of Python scripts, which process encoded and decoded files depending on command line parameters, allowing us to drive specific data through the codecs, capture the resultant output, and compare the two sets of data for correctness.

A.3 ADDITIONAL SUPPORT TOOLS

A.3.1 LPDSP32 Startup Files

The startup files provide some header and source files, which manage basic features such as setting up the Interrupt Vector Table for the LPDSP32, and providing some basic memory management functions.

In this release, there is not a great deal of functionality in this package; and implements only a minimum required set.

A.3.2 DSPIntegration

The DSP Integration package provides a suite of Python scripts, which can be used to parse the LPDSP32 compilation output into a form that can be embedded into the Arm Cortex-M3 core programs. This extracts the relevant information to allow the embedded loading code to copy the PRAM and DRAM areas to the LPDSP32 memory space and begin execution.

These are provided as Python scripts because the functionality may need to be slightly different depending on the version of the LPDSP32 tools, and the version of the application being processed.

The scripts provide a starting point that works with our sample codecs and we believe can also work with customer code. If updates are required, or if a customer wishes to integrate the code differently, then the scripts provide a starting point for any further development.

A.4 BASIC INSTALLATION AND VERIFICATION OF PACKAGE

This page provides some simple steps to be taken to verify the installation and use of the LPDSP32 Support package.

A.4.1 Assumptions

1. The RSL10 SDK has been installed and is working.
2. The Synopsys LPDSP32 Development environment has been installed and is working.
   - This includes the installation of the LPDSP32 processor package.

3. Python 2.7 has been installed and is working.
4. Some form of RSL10 development board is available to allow verification of the compiled code.
5. The latest version of JLink tools (or at least a version which fully supports RSL10) is installed.
   - As a guide, J-Link SDK V6.20f is known to be compatible, as should any subsequent release be.

A.4.2 Preparing Python

The DSPIntegration project uses a third party package, pyelftools, to interpret ELF format executable files. This should be installed if it is not already available. The installation command is given below; it checks if the package is already available and only updates it if necessary.

pyelftools Installation Command

```
python -m pip install pyelftools
```

A.4.3 Preparing the RSL10 SDK

A.4.3.1 Installation of PyDev plugin

The Python based projects have been deployed as PyDev compatible projects which can be managed within the RSL10 SDK using the PyDev plugin. This stage explains how to install and set up the PyDev environment within the RSL10 SDK.

This stage is optional as the scripts can all be driven from the command line, but PyDev provides an easy and intuitive development environment allowing exploration of the Python tools and providing a debug environment. The rest of this section assumes this stage has been completed.

1. Start the RSL10 SDK and select a new clean workspace.
2. From the Eclipse IDE, Select **Help** > **Install New Software...**; this pops up the **Available Software** dialog.
3. Select the **Add...** button on the top right to provide a new software site; this pops up the **Add Repository** dialog.
4. Set the **Name:** field to **PyDev** and the **Location:** to **http://www.pydev.org/update_sites/5.2.0**
   - Note that this is referring to a specific version of the PyDev plug-ins. These have been selected as being compatible with the Kepler version of Eclipse on which the RSL10 SDK has been based.

5. This will now provide access to the PyDev development tools. Select the check-box for PyDev to allow installation.
6. Follow the on-screen instructions to install PyDev; this requires restarting the Eclipse workspace.

A.4.3.2 PyDev Configuration

Once Eclipse has restarted, we need to configure the new plug-in so that it is aware of the location of Python.

1. Select **Window** > **Preferences** to get the preferences dialog.
2. Select **PyDev** > **Interpreters** > **Python Interpreter**.
3. On the right hand side of the tabbed window, select the **New...** button to pop up the **Select Interpreter** dialog.
4. Set the Interpreter Name to **python1**.
5. Set the **Interpreter Executable** to the *python.exe* file from the Python2.7 installation using the browse button.

6. Accept the default for the **PythonPath** updates and click **OK** to close the dialog box.

This completes the PyDev configuration.

A.4.4 Importing LPDSP32 Support Package

Having configured Python and the RSL10 SDK, we can now import the relevant projects from the RSL10 LPDSP32 Support Package.

Unzip the package into a clean folder. This provides a structure with six sub-folders as shown in Figure 5, below:

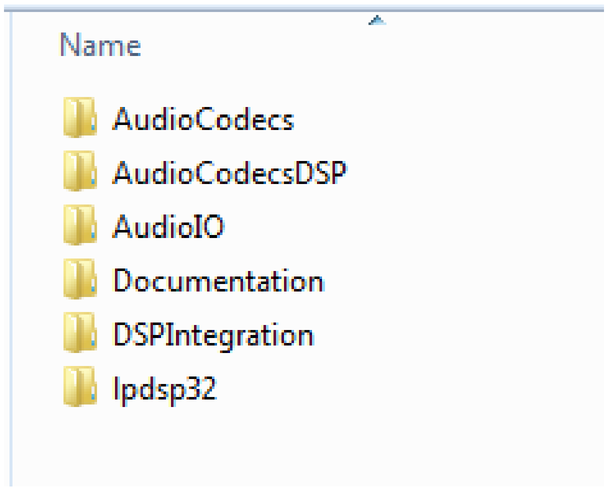


Figure 5. LPDSP32 Support Package Sub-Folders

Each of the folders, with the exception of Documentation, is a standalone Eclipse project and can be imported into the RSL10 SDK using the **File** > **Import...** > **Existing Projects Into Workspace**.

Do this for each of the five project folders, selecting the **Copy into workspace** checkbox. This will give a structure in the workspace, as shown in Figure 6, below:

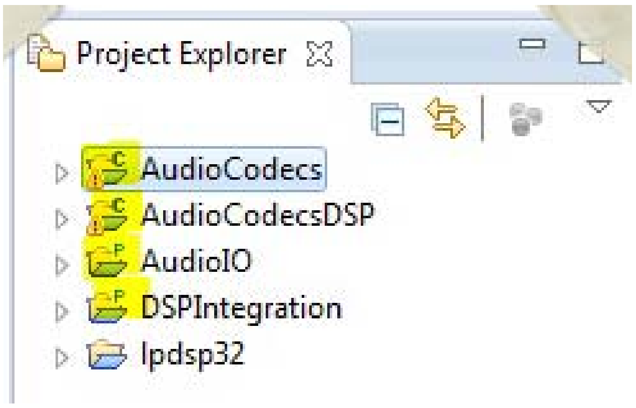


Figure 6. Workspace Structure

Note that two of the projects are marked as C/C++ projects, and two are marked as PyDev projects.

A.4.5 Building the samples

The first thing to check is that the provided sample codes build correctly. This is a two-step process using the RSL10 SDK and the Synopsys LPDSP32 development environment.

A.4.5.1 Building the Arm Cortex-M3 core based projects

From within the RSL10 SDK, right click the **AudioCodecs** project and select **Build Project**.

This should compile and link the delivered project with no warnings or errors.

A.4.5.2 Building the LPDSP32 based projects

To build the LPDSP32 projects, we need to build each one individually:

1. Start the ChessDE Development environment
2. Select **File** > **Open** > **Project...** and then navigate to the *AudioCodecsDSP* folder within the Eclipse workspace.
3. There are three projects which need to be built in the *src/codecs* folder:
    a. *celt/celt.prx*
    b. *g722/g722.prx*
    c. *echo/echo.prx*

4. Open each of these in turn and ensure that they can be successfully compiled using the ChessDE environment.

A.4.6 Checking the Debug connection

A.4.6.1 Basic Debug

The first thing to verify is the basic connection to the board: ensure that the RSL10 development board is attached to the PC.

Now set up a debug configuration to check the AudioCodecs project:

1. From within the RSL10 SDK, right-click the **AudioCodecs** project and select **Debug As** > **Debug Configurations ...**
2. This pops up the **Debug Configurations** dialog which contains a list of possible debug types on the left hand side.
    a. Select **GDB SEGGER J-Link Debugging**
    b. Right-click and select **New**; this will create a debug configuration for the AudioCodecs application.
    c. Select the **Debugger** tab and ensure the **Device Name** is set to **RSL10**.

3. Select the **Debug** button. This will load the program to the board and start the debugger.
4. Select the **Debug** perspective and verify the debugger has stopped at the first line of the main program.
5. Single step a number of times until past the line containing the call to `dspHandshake()`.
    - This verifies the C code on the Arm Cortex-M3 core is working as expected and that the LPDSP32 has been correctly initialized and started.

Stop the debugger; this is good enough for now.

A.4.6.2 Automated Debug

To verify the ability to feed samples through the codecs, we will use the AudioIO project to control the GDB debugger by feeding samples into the program under test.

Again from within the RSL10 SDK, right-click the **AudioIO** project and select **Run As...** > **Run Configurations**. This pops up the **Run Configurations** dialog.

1. Select **Python Run** from the list of run configurations on the left hand side.
2. Right click and select **New**.
3. Provide a name for the configuration: **G722 Encode**.
4. Select **AudioIO** as the project.
5. Select **src** > **com** > **onSemi** > **audioio** > **driver.py** as the main module.
6. Select the **Arguments** tab and then under **Program Arguments**, add the information from the code block below:

Sample Program Arguments for G722 Encode

```
--exe "${workspace_loc}/AudioCodecs/Debug/AudioCodecs.elf" --gdb "${eclipse_home}/../
    arm_tools/bin/arm-none-eabi-gdb.exe" --input ${workspace_loc}/AudioIO/audio/in/G722/
    encodeSource.raw --golden ${workspace_loc}/AudioIO/audio/out/G722/enc_48000.raw
    --gdbServer "C:\Program Files (x86)\SEGGER\JLink_V620f\JLinkGDBServerCL.exe"
    --inBinary --outBinary --inPacketSize 320 --outPacketSize 80  --encode --frameSize 160
    --blockSize 4 --mode 3
```

- Note the path to the `gsbServer` has been fully specified; change this to reflect your own installation.

1. Select **Run**. This should pop up a progress bar that shows the RSL10 device being flashed. The console will show output similar to the following:

Console Output

```
320 / 40960 - 63
640 / 40960 - 63
720 / 40960 - 78
...
39920 / 40960 - 2
40240 / 40960 - 1
40320 / 40960 - 1
40640 / 40960 - 0
40720 / 40960 - 0
```

The actual numbers indicate progress and the last value on each line gives a rough indication of time to go in seconds.

Having been through these steps, we have now verified that the distribution behaves as expected.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for ON Semiconductor
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support: 800-282-9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

ON Semiconductor Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative