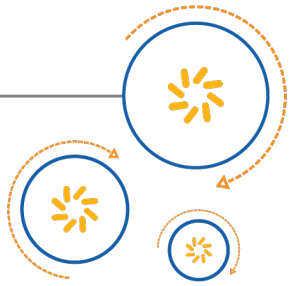
Qualcomm Technologies International, Ltd.



My First Qualcomm Kalimba DSP Application

Application Note

80-CT398-1 Rev. AM

October 23, 2017

Revision history

Revision	Date	Description
1	JUN 2006	Initial release. Alternative document number CS-101420-AN.
2	JUL 2007	Removed reference to obsolete document.
3	NOV 2009	Minor correction to VM and DSP source code files. Updated to latest style guidelines.
4	DEC 2009	Update the gains on the output codec in the VM_app.
5	JUL 2010	Changes to VM code to support the new SDK. Project names now in lower case only.
6	FEB 2012	■ Major changes have been made on VM code to accommodate the changes in xIDE ADK and hardware (CNS10001 Development board) ■ Minor changes have been made on DSP code Removed Sections 2.1 and 2.2 ■ Added step-by-step steps to build and run Kalimba DSP on CSR8670 and BlueCore5-Multimedia boards ■ Added code explanation about mic bias and speaker amplifier shut-down control on CSR8670 board ■ Updated Document references ■ Updated Appendix A VM Application Code ■ Updated Appendix B Kalimba DSP Code ■ Updated Terms and Definitions Updated document to latest style
7	DEC 2013	Updated for ADK 2.5, minor style corrections.
8	DEC 2013	Updated for ADK 3 and to New CSR™ style
9	JUL 2014	Updated for CSR8675
10	JUN 2015	Updates to code examples
11	AUG 2016	Update code examples and associated text for ADK 4.1. Updated to conform to QTI standards.
AM	OCT 2017	Added to the Content Management System. DRN updated to use Agile number. No technical content was changed in this document revision

Contents

Tables

Figures

1 my_first_dsp_app

`my_first_dsp_app` is an example audio application. It uses the Qualcomm® Kalimba™ DSP and the xIDE development environment. The application routes stereo audio through the DSP and describes how to use the Kalimba libraries.

2 Build and run a Kalimba DSP application on a CSR8670 board

Kalimba DSP applications are developed within the xIDE development environment ADK. Applications are structured as a workspace that can be made up of a number of projects. When developing a DSP application the workspace should contain both a VM application and a DSP application. For instructions on how to create projects in xIDE, refer to the *Qualcomm® BlueCore™ xIDE User Guide*.

1. Connect the CNS12002v1 interface daughter board to the CSR8670 board, as shown in Figure 2-1. Also connect an aerial and 3.7 V battery.

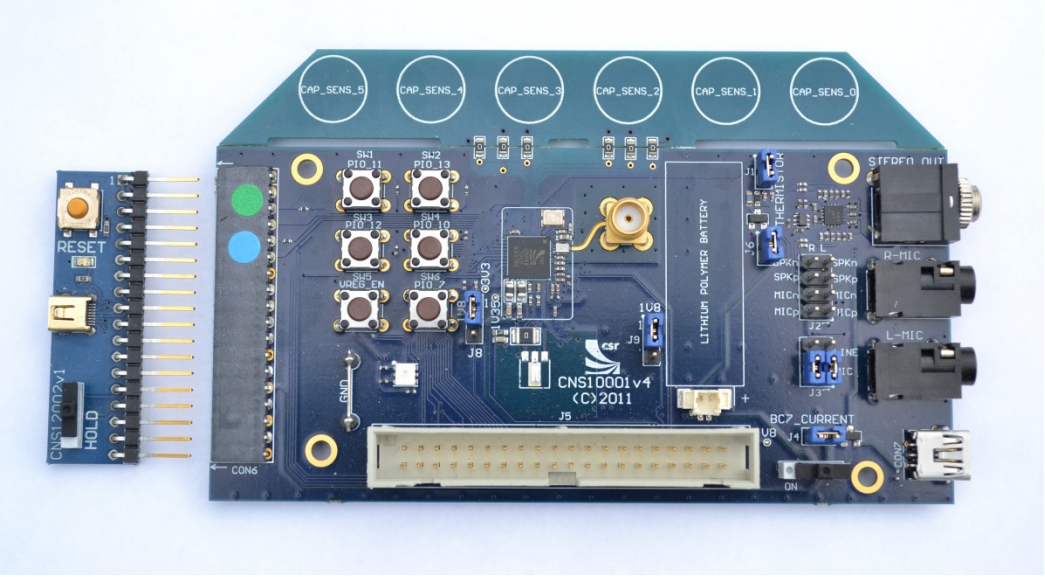
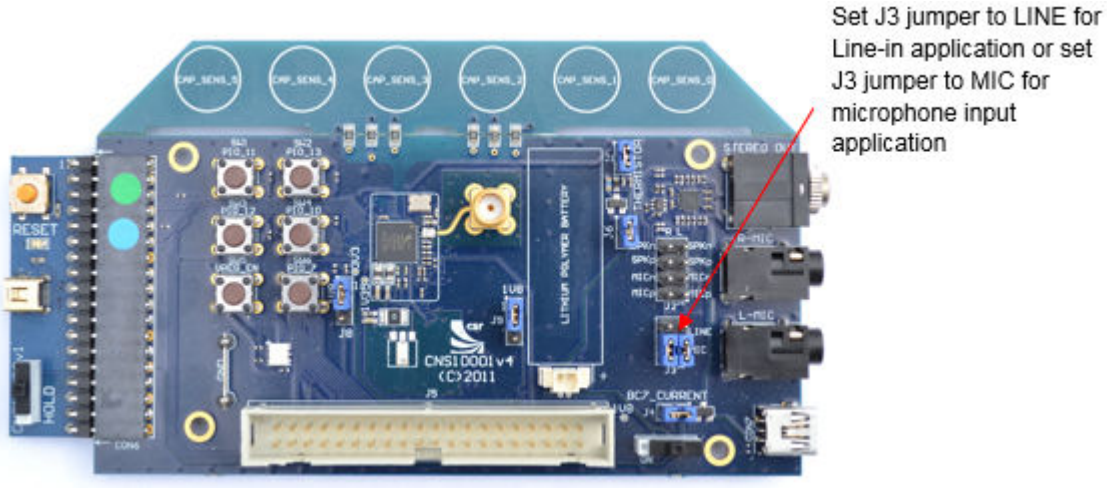


Figure 2-1 Connecting CNS12002v1 interface daughterboard to the CNS10001 (CSR8670) board

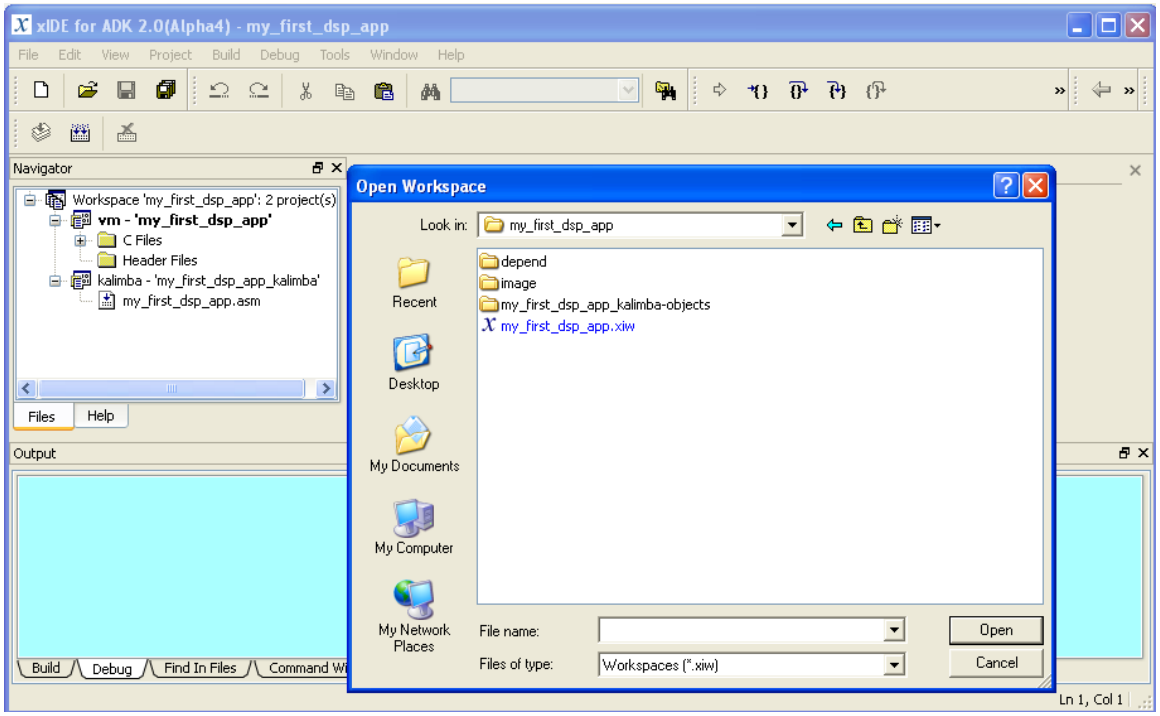
2. Connect the USB-SPI converter (CNS10020) to the CNS12002v1 interface daughter board using the supplied cable. Connect the other end of USB-SPI converter to the PC USB port using the

USB cable:

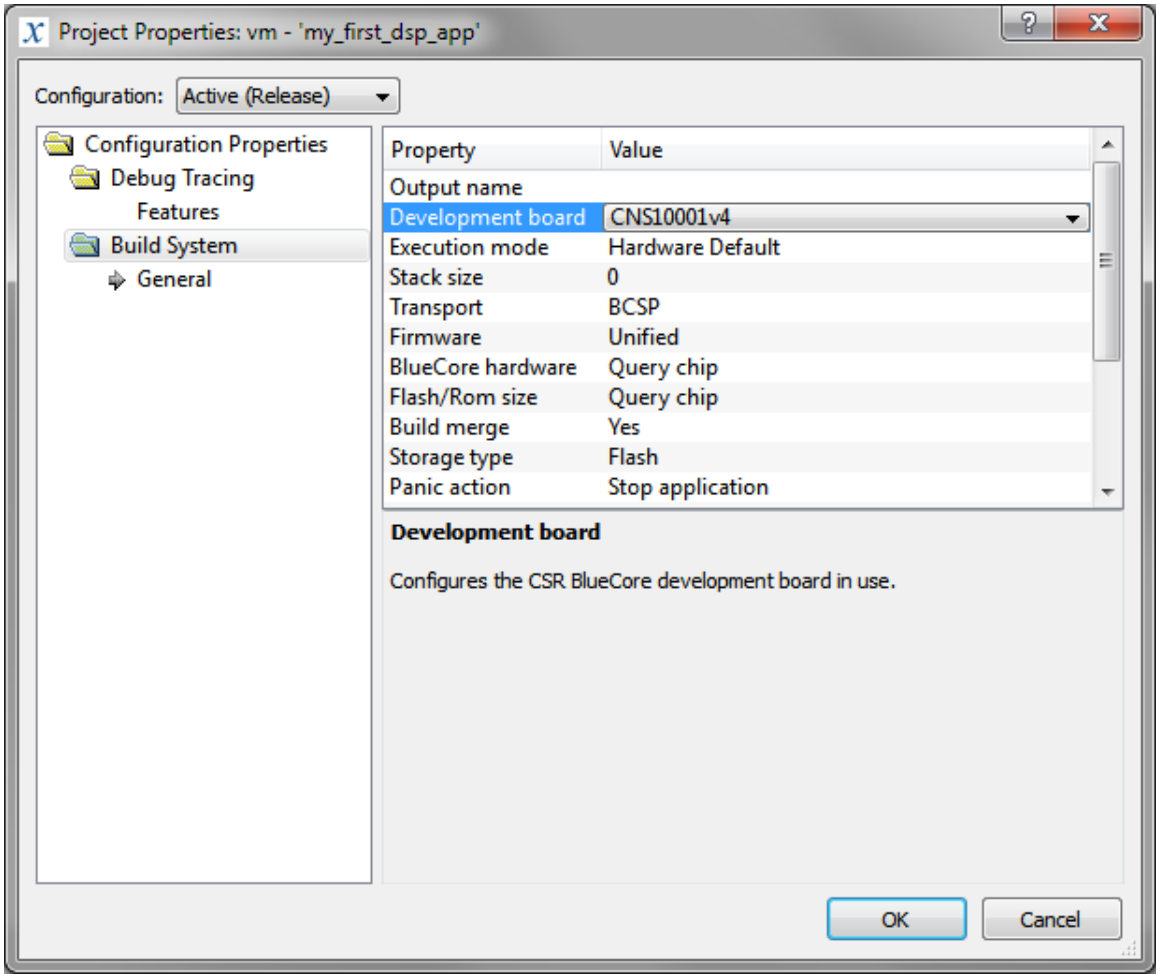
3. Connect the STEREO OUT jack to the speaker or headphone.
   For the single-ended stereo LINE input, connect the R-MIC and L-MIC jacks to the audio Line-out jack of the PC or any portable media player through the Y-type audio cable. Set the J3 jumper to LINE.
   a. For the 2-mic application, connect the R-MIC and L-MIC jacks to the stereo microphones through the Y-type audio cable.
   b. For the 1-mic application, connect either R-MIC or L-MIC jack to the mono microphone. Set the J3 jumper to MIC.



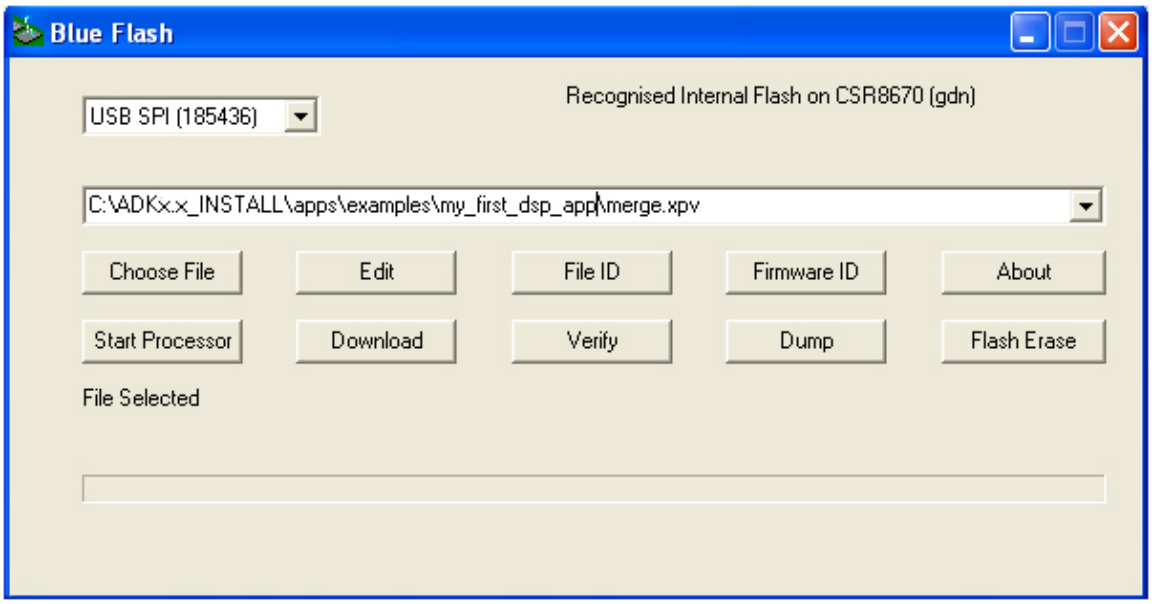
4. Connect a USB cable to the CSR8670 board to provide 5 V power supply.
5. Turn the **Power** switch **ON**.
6. Launch ADK.li
7. Locate the project workspace `my_first_dsp_app.xiw` by selecting **Project | Open Workspace** within xIDE.
   The project file `my_first_dsp_app.xiw` is in `C:\ADK_INSTALL\apps\examples\my_first_dsp_app`. ADK is installed in the **C:\ADK_INSTALL** directory by default. You can copy and save it anywhere or create your own Kalimba DSP project. For instructions, see the *BlueCore Technology xIDE User Guide*.



8. Check that the development board you are using is configured correctly in the project settings. To do this, right-click `vm – 'my_first_dsp_app` and select **Properties... > Build System**:



9. Ensure that the correct Debug Transport is selected, the setting for this is under the **Debug > Transport...** menu in xIDE. Select the USB-SPI adapter connected to the development board, the serial number of the USB-SPI adapter is printed on the underside of the unit.
10. Build the application by pressing the **F7** key, which compiles and links the code for both VM and DSP.
    Alternatively, right-click `kalimba - 'my_first_dsp_app_kalimba` and select **Build and** right-click `vm - 'my_first_dsp_app` and select **Build**.
11. Run the Kalimba DSP application by pressing the **F5** key. This step flashes the executable binary image to the hardware and runs the application in debugging mode.
    Alternatively, use the xIDE to generate the executable binary image. To do this, press the **F7** key and use the flash tool **BlueFlash** to flash and run the application.
    You must set the **Build merge** to **Yes** in the VM **Project Properties** window or the executable image `merge.xpv (.xdv)` does not generate:



After completing Step 12, a speaker or headphone can play music. This example code routes the audio from ADC to DSP and then from DSP to DAC. As an exercise, you can bypass the DSP and route audio directly from ADC to DAC without passing through the DSP. To do this, re-insert the macro definition on line 30 of main.c: `#define BYPASS_KALIMBA`. Alternatively, the symbol `BYPASS_KALIMBA` can be added to the **Project Properties** in **Define symbols** described in Step 9 (separate multiple symbols with commas). Steps 11 and 12 should then be repeated to rebuild the application.

To use a microphone input, re-insert the macro on Line 32 of main.c: `#define MIC_INPUT` and follow steps 1-13. Alternatively, this macro can be added via the **Project Properties** in **Define symbols**. You must also set Jumper **J3** to **MIC**, as described in Step 4.

3 Build and run a Kalimba DSP application on a CSR8675 development board

Kalimba DSP applications are developed within the xIDE development environment ADK. Applications are structured as a workspace which can be made up of a number of projects. When developing a DSP application the workspace should contain both a VM application and a DSP application. For instructions on how to create projects in xIDE, see the *BlueCore xIDE User Guide*.

1. Connect the H13223 headphone amplifier to the H13179 board. Also connect an aerial and 3.7 V battery. SPI interface is built into the development board.

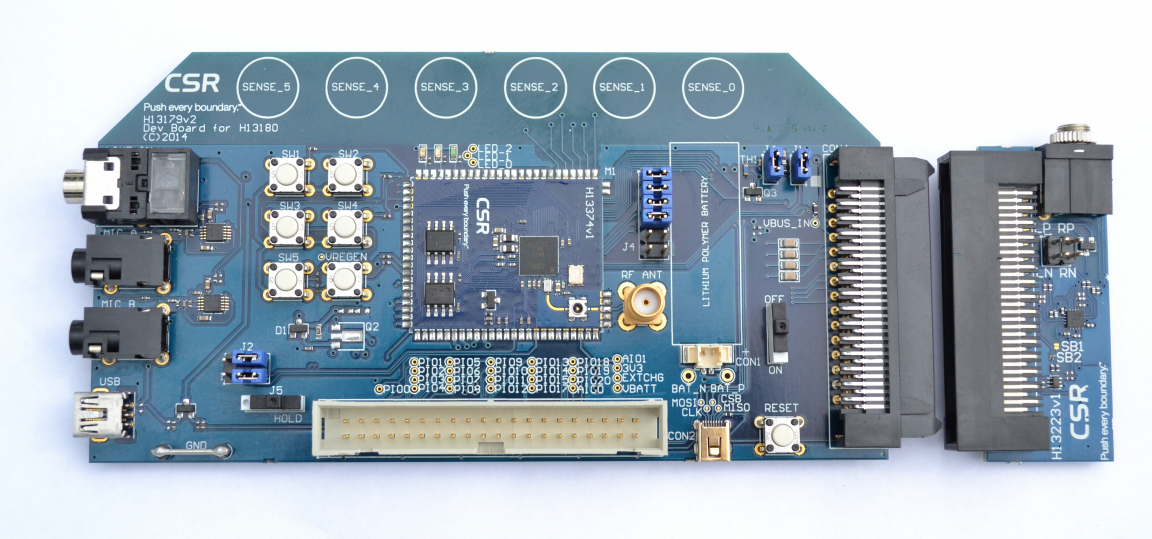
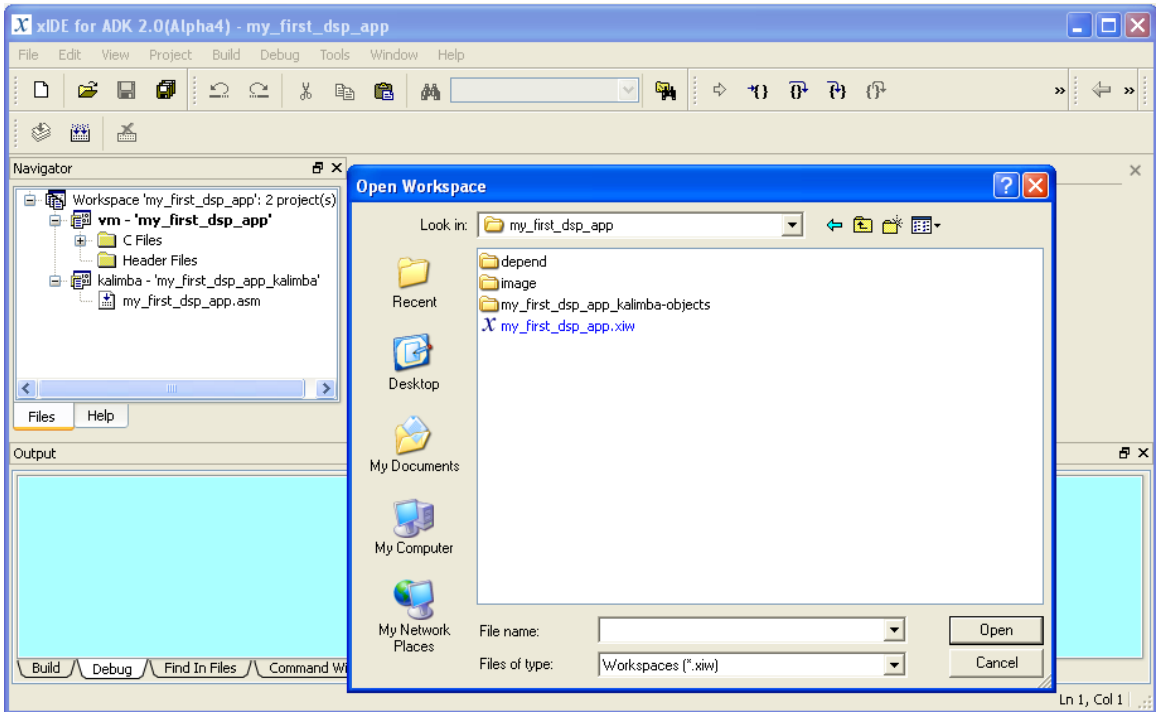


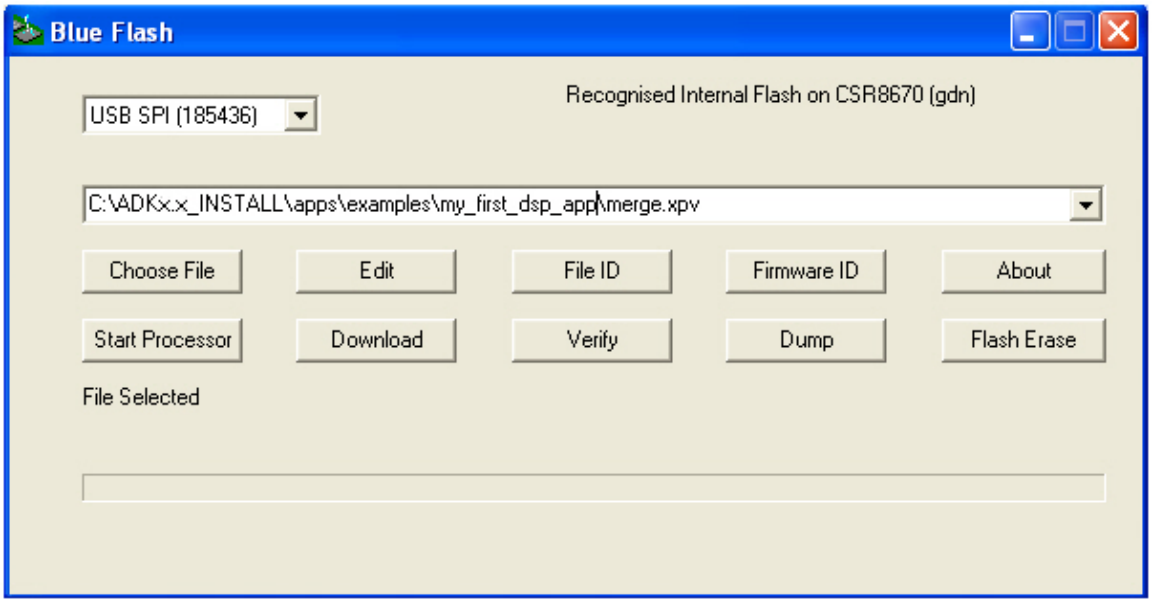
Figure 3-1 Connecting H13223 headphone amplifier board to the H13179/H13374 (CSR8675) board

2. Connect the USB-SPI converter (CNS10020) to CON2 on the H13179 board using the cable provided. Connect the other end of USB-SPI converter to the PC USB port using the USB cable.
3. Connect the AUDIO OUT jack on the H13223 headphone amplifier to the headphones.
4. For the single-ended stereo LINE input, connect the LINE IN jack to the audio Line-out jack of a PC or any portable media player.
   a. For the 2-mic application, connect the MIC A and MIC B jacks to the stereo microphones through a Y-type audio cable.
   b. For the 1-mic application, connect either MIC A or MIC B jack to the mono microphone.
5. Connect USB cable to development board to provide 5 V power supply.
6. Turn the Power switch **ON**.

7. Launch ADK.
8. Locate the project workspace `my_first_dsp_app.xiw` by selecting **Project | Open Workspace ...** within xIDE.
9. The project file `my_first_dsp_app.xiw` is in `C:\ADK_INSTALL\apps\examples\my_first_dsp_app`. By default the ADK is installed in the **C:\ADK_INSTALL** directory. You can copy and save it to anywhere or create your own Kalimba DSP project. For instructions see the *xIDE User Guide*.



10. Check that the development board you are using is configured correctly in the project settings. To do this, right-click `vm – 'my_first_dsp_app` and select **Properties... | Build System**.
11. Ensure that the correct Debug Transport is selected, the setting for this is under the **Debug | Transport...** menu in xIDE. Select the USB-SPI adapter connected to the development board, the serial number of the USB-SPI adapter is printed on the underside of the unit.
12. Build the application by pressing the **F7** key, which compiles and links the code for both VM and DSP.
    Alternatively, right-click `kalimba – 'my_first_dsp_app_kalimba'` and select **Build**.
    Then right-click `vm – 'my_first_dsp_app'` and select **Build**.
13. Run the Kalimba DSP application by pressing the **F5** key. This step flashes the executable binary image to the hardware and runs the application in debugging mode.
14. Alternatively, use the xIDE to generate the executable binary image. To do this, press the **F7** key and use the flash tool **BlueFlash** to flash and run the application.
15. You must set the **Build merge** to **Yes** in the VM **Project Properties** window or the executable image `merge.xpv (.xdv)` does not generate.



After completing Step 12, a speaker or headphone can play music. This example code routes the audio from ADC to DSP and then from DSP to DAC. As an exercise, you can bypass the DSP and route audio directly from ADC to DAC without passing through the DSP. To do this, re-insert the macro definition on line 30 of main.c: `#define BYPASS_KALIMBA`. Alternatively, the symbol `BYPASS_KALIMBA` can be added to the **Project Properties** in **Define symbols** described in Step 9 (separate multiple symbols with commas). Steps 11 and 12 should then be repeated to rebuild the application.

To use a microphone input, re-insert the macro on Line 32 of main.c: `#define MIC_INPUT` and follow steps 1-13. Alternatively, this macro can be added via the **Project Properties** in **Define symbols**. You must also set Jumper J3 to MIC, as described in Step 4.

4 my_first_dsp_app VM application

The VM code runs in the main processor. It loads the DSP executable image to the DSP processor, connects the ADC and DAC to the Kalimba DSP ports, starts the DSP code and runs an infinitive loop. VM code runs in master processor and DSP code runs in slave Kalimba DSP processor.

In the example application `my_first_dsp_app`, the VM application is responsible for:

1. Setting up the Kalimba DSP
2. Connecting the audio stream path
3. Starting the main DSP application loop

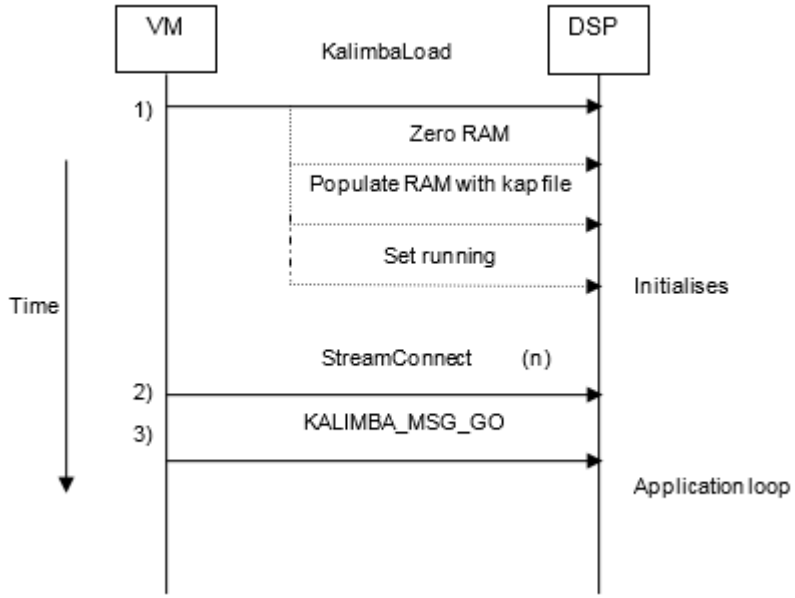


Figure 4-1 Initializing Kalimba from the VM

The VM also initializes the input and output gain of the codecs.

Loading the DSP application

To load the DSP application, the VM application code must do the following:

Find the appropriate file in the file system:

The root is the `\image` directory.

```
  static const char kal[] =
                "my_first_dsp_app_kalimba/my_first_dsp_app_kalimba.kap";
  FILE_INDEX index = FileFind( FILE_ROOT,(const char *)kal, strlen(kal) );
Load the specified DSP code into Kalimba.
  KalimbaLoad( index );
```

4.1 Connect the audio stream path

The `my_first_dsp_app` application utilizes the stream library to quickly route data from the PCM input ports through the DSP and to the PCM output ports using streams.

Streams, as implemented in the firmware, include two halves, a source and a sink. A source produces a byte stream while a sink consumes it.

Streams can represent various data sources from PCM audio to UART data. Section xx describes how to use the `StreamConnect()` function to route PCM audio between the PCM subsystem and the DSP. For more information about the stream library functions, see the VM and Native Reference Guide files listed in Table 4-1.

Table 4-1 Stream library functions reference documents

Document Name	Directory Location
`stream.8h.html`	`<ADK Installation Path>\doc\reference\html\`
`stream.h`	`<ADK Installation Path>\tools\include\firmware\`

4.1.1 Define the source and sinks

The following function calls define the audio sources. The left and right channels are assigned to channel A and B, respectively:

```
    Source audio_source_a = StreamAudioSource( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0, AUDIO_CHANNEL_A );
    Source audio_source_b = StreamAudioSource( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0, AUDIO_CHANNEL_B );
```

The following function calls define the audio sinks. The left and right channels are assigned to channel A and B, respectively:

```
    Sink audio_sink_a = StreamAudioSink( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0, AUDIO_CHANNEL_A );
    Sink audio_sink_b = StreamAudioSink( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0, AUDIO_CHANNEL_B );
```

4.1.2 Configure the sampling rate and synchronization

The following function calls configure the sampling rates of the left and right input channels to 44.1 kHz:

```
    SourceConfigure(audio_source_a, STREAM_CODEC_INPUT_RATE, 44100);
    SourceConfigure(audio_source_b, STREAM_CODEC_INPUT_RATE, 44100);
```

The following function call synchronizes the left and right input channels:

```
SourceSynchronise(audio_source_a, audio_source_b);
```

The following function calls configure the sampling rates of the left and right output channels to 44.1 kHz:

```
SinkConfigure(audio_sink_a, STREAM_CODEC_OUTPUT_RATE, 44100);
SinkConfigure(audio_sink_b, STREAM_CODEC_OUTPUT_RATE, 44100);
```

The following function call synchronizes the left and right output channels:

```
SinkSynchronise(audio_sink_a, audio_sink_b);
```

4.1.3 Configure the ADC gain settings

The following function calls configure the input channel gains to -3.0 dB.

To configure the input gain for the left channel use:

```
SourceConfigure(audio_source_a, STREAM_CODEC_INPUT_GAIN, 8);
```

To configure the input gain for the right channel use:

```
SourceConfigure(audio_source_b, STREAM_CODEC_INPUT_GAIN, 8);
```

NOTE Table 4-2 shows the `input_gain` given in the form of an index (0 to 22) that is mapped to an actual gain/attenuation value.

Table 4-2 Index to gain values mappings

Index	Gain (db)
0	-27.0
1	-23.5
2	-21.0
3	-17.5
4	-15.0
5	-11.5
6	-9.0
7	-5.5
8	-3.0
9	0.0
10	3.0
11	6.0
12	9.0
13	12.5
14	15.0
15	18.0
16	21.5
17	24.0
18	27.5

Table 4-2 Index to gain values mappings (cont.)

Index	Gain (db)
19	30.0
20	33.5
21	36.0
22	39.5

The following function calls control the preamps for the left and right input channels (disabled in the example). When enabled this adds an extra 24.5 dB in addition to the selected input gain from Table 4-2.

```
SourceConfigure(audio_source_a, STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 0);
SourceConfigure(audio_source_b, STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 0);
```

For microphone pre-amp circuitry to work, the BlueCore chips' BIAS_MIC pins must be properly DC biased. To do this, use the following code:

```
MicbiasConfigure(MIC_BIAS_0, MIC_BIAS_ENABLE, MIC_BIAS_FORCE_ON);
MicbiasConfigure(MIC_BIAS_1, MIC_BIAS_ENABLE, MIC_BIAS_FORCE_ON);
```

You only need the first line of code for the BlueCore5-Multimedia board because only one microphone is used.

NOTE In the provided source code, the `MicbiasConfigure()` function call is conditionally compiled according to the definition of the `MIC_INPUT` macro. To use this function, include the `micbias.h header` file in the VM application.

In addition to enabling the current, you can configure both the current and voltage of the microphone bias using the function calls:

```
MicbiasConfigure( MIC_BIAS_0, MIC_BIAS_CURRENT, current );
MicbiasConfigure( MIC_BIAS_0, MIC_BIAS_VOLTAGE, voltage );
```

NOTE The bias `current` and `voltage` are given in the form of indices (0 to 15) that are mapped to actual values. See the table below.

Table 4-3 Input indices to current and voltage settings mappings

Index	Voltage (V)	Current (mA)
0	1.72	0.32
1	1.77	0.40
2	1.83	0.48
3	1.89	0.56
4	1.97	0.64
5	2.53	0.72
6	2.12	0.80
7	2.20	0.88
8	2.34	0.97
9	2.44	1.05

Table 4-3 Input indices to current and voltage settings mappings (cont.)

Index	Voltage (V)	Current (mA)
10	2.58	1.13
11	2.71	1.21
12	2.92	1.29
13	3.10	1.37
14	3.34	1.45
15	3.69	1.53

The development board speaker/headphone output is powered by stereo headphone amplifier with shutdown function. When the speaker amplifier shutdown pin is driven down by active-low voltage, the amplifier is disabled and the audio is muted. On the CSR8670 board control, this shutdown pin by setting the PIO pin `PIO_4`, on the CSR8675 board this is PIO pin `PIO_14`: e.g.

```
PioSetPio(PIO, TRUE);
```

These PIO is automatically configured for the relevant board when the Project Setting of Development Board type is set.

4.1.4 Configure DAC gain setting

The following function calls configure the output channel gains to 0.0 dB.

To configure the output gain for the left channel use:

```
SinkConfigure(audio_sink_a, STREAM_CODEC_OUTPUT_GAIN, 15);
```

To configure the output gain for the right channel use:

```
SinkConfigure(audio_sink_b, STREAM_CODEC_OUTPUT_GAIN, 15);
```

NOTE Table 4-4 provides the `output_gain`, given in the form of an index (0 to 22) that is mapped to an actual gain/attenuation value.

Table 4-4 Index to gain values mappings

Index	Gain (dbFS)
0	-45.0
1	-41.5
2	-39.0
3	-35.5
4	-33.0
5	-29.5
6	-27.0
7	-23.5
8	-21.0
9	-18.0
10	-15.0

Table 4-4 Index to gain values mappings (cont.)

Index	Gain (dbFS)
11	-12.5
12	-9.0
13	-6.0
14	-3.0
15	0.0
16	3.5
17	6
18	9.5
19	12
20	15.5
21	18
22	21.5

NOTE Index values greater than 15 can cause saturation in the digital gain stage of the DAC output.

4.1.5 Use of the StreamConnect function

Table 4-5 provides guidelines on using the `StreamConnect()` function to route PCM audio between the PCM subsystem and the DSP:

Table 4-5 Using the StreamConnect function

Function	Description
`StreamConnect(audio_source_a, StreamKalimbaSink(0));`	Connects the left channel audio source to the DSP read port 0
`StreamConnect(audio_source_b, StreamKalimbaSink(1));`	Connects the right channel audio source to the DSP read port 1
`StreamConnect(StreamKalimbaSource(0), audio_sink_a);`	Connects the DSP write port 0 to the left channel audio sink
`StreamConnect(StreamKalimbaSource(1), audio_sink_b);`	Connects the DSP write port 1 to the right channel audio sink

4.2 Start the main DSP application loop

To start the main DSP application loop, use the following two functions:

1. DSP:

   ```
   $message.send_ready_wait_for_go
   ```

This function sends a message to the VM firmware indicating that it has finished its initialization and is ready to begin processing. The DSP remains in a wait loop until it receives the `KALIMBA_MSG_GO` from the VM application.

2. VM:

   ```
   KalimbaSendMessage(KALIMBA_MSG_GO,0,0,0,0);
   ```

   The function starts the DSP application. After sending this message, the VM application enters the `MessageLoop()` routine to service any received messages.

4.3 Add a preprocessor conditional include

The preprocessor allows conditional compilation of source code by using the `#ifdef, #else`, and `#endif` directives. This is useful for quickly switching between different versions of code at compile time. For example, the VM application uses the preprocessor directives to compile code that enables the microphone input by defining `MIC_INPUT`, or to route the audio around the DSP by defining `BYPASS_KALIMBA`.

These preprocessor statements can be defined in the source code as:

```
#define MIC_INPUT
#define BYPASS_KALIMBA
```

These statements can also be included in the **Define Symbols** field on the **Project Properties** window. Figure 4-2 uses the `#define MIC_INPUT` statement.

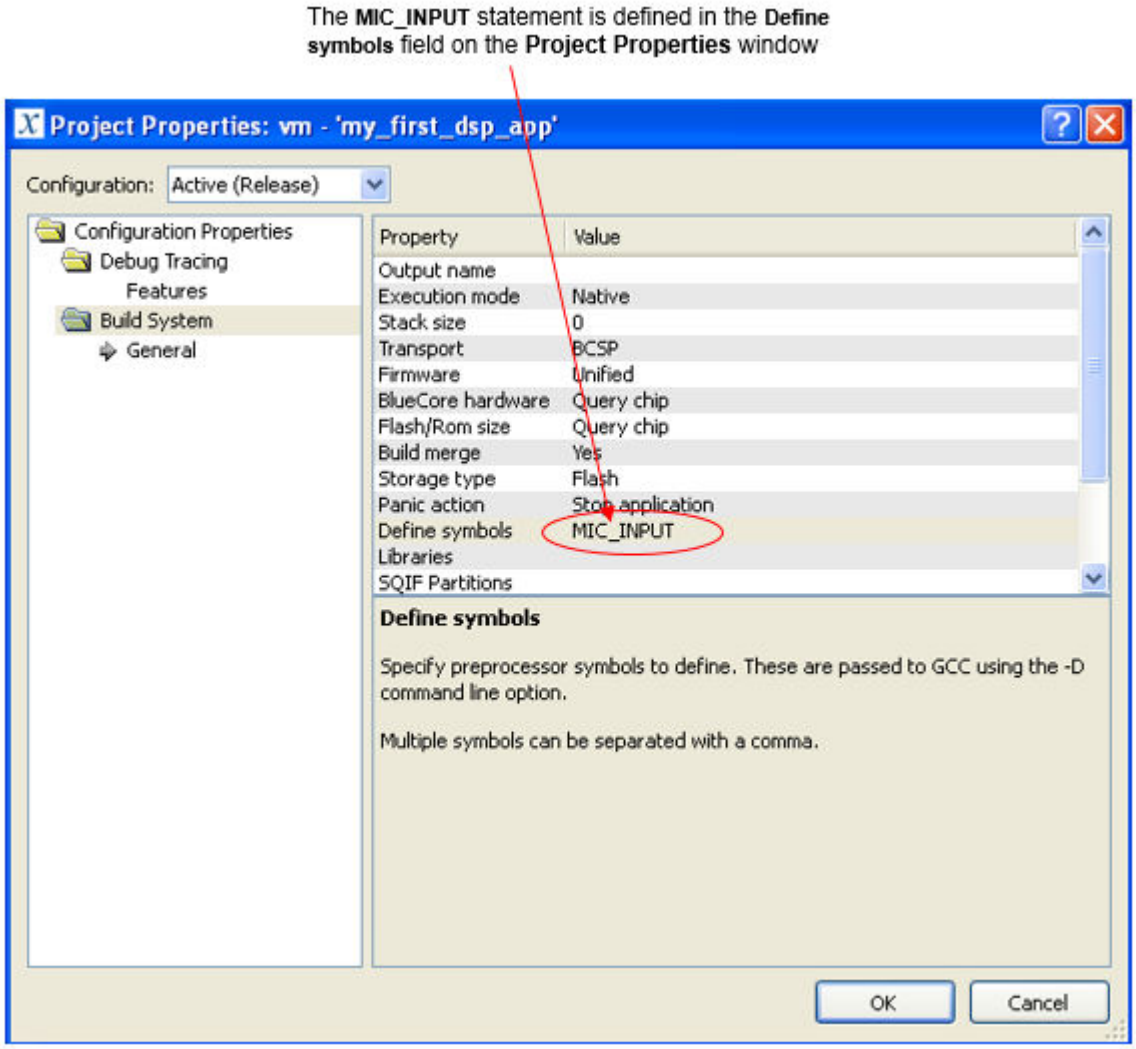


Figure 4-2 Project Properties window: define symbols

5 my_first_dsp_app Kalimba application

All the DSP code for the example application is in the module `my_first_dsp_app.asm`. Each function is contained within the following instructions:

```
.MODULE $module_name.function_name;
        .CODESEGMENT PM;
             .DATASEGMENT DM;
            // … Function implementation here
.ENDMODULE;
```

These instructions specify the name of the function in the module, and which memory banks the code and data segments should reside in.

Table 5-1 describes the modules that contain all DSP code for the example application.

Table 5-1 Example application DSP code

Module	Description
`$M.main (my_first_dsp_app.asm)`:	Contains the main DSP application that initializes the DSP, defines the linked list of operators, starts the audio timers, and enters `copy_loop`
`$M.audio_in_copy_handler (my_first_dsp_app.asm)`	Calls `$cbops.copy` for the `$audio_in_copy_struc` and schedules the next input audio timer
`$M.audio_out_copy_handler (my_first_dsp_app.asm)`	Calls `$cbops.copy` for the `$audio_out_copy_struc` and schedules the next output audio timer
`$M.loopback_copy (my_first_dsp_app.asm)`	Calls `$cbops.copy` for the `$audio_loopback_copy_struc`

5.1 my_first_dsp_app application operators

The `my_first_dsp_app code` uses the current `cbops` operator structure to route data in and out of the DSP ports through cbuffers. This section describes the operator structure.

Operators can be separated into two levels:

1. Stream Object
2. Operator Object

5.1.1 Stream object level

The Stream Object level is the highest level. It defines the number of inputs and outputs and identifies these possible PORTS or cbuffers. It also contains the pointer to the first operator in the linked list. Figure 5-1 shows the generic Stream Object definition and FIR filter specific Stream Object declaration.

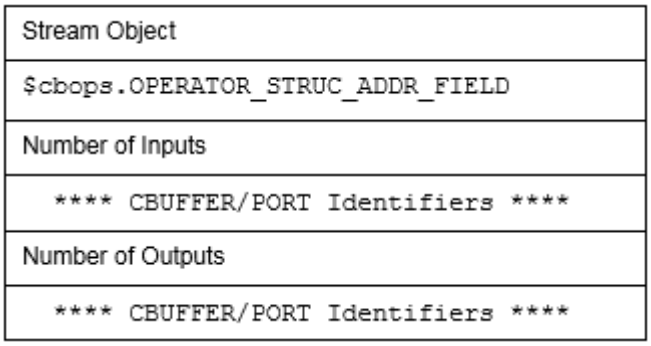


Figure 5-1 Stream object definition

```
.VAR $audio_in_copy_struc[] =
      &$audio_in_left_shift_op,        // First operator block
      2,                               // Number of inputs
      $AUDIO_LEFT_IN_PORT,             // Input
      $AUDIO_RIGHT_IN_PORT,            // Input
      2,                               // Number of outputs
      &$audio_in_left_cbuffer_struc,   // Output
      &$audio_in_right_cbuffer_struc;  // Output
```

Figure 5-2 $audio_in_copy_struc declaration

Figure 5-2 shows that the first operator in the linked list is `$ audio_in_left_shift_op`. This Stream Object uses two inputs (input ports):

1. `$AUDIO_LEFT_IN_PORT`
2. `$AUDIO_RIGHT_IN_PORT,`

and two outputs (cbuffers)

1. `$audio_in_left_cbuffer_struc`
2. `$audio_in_right_cbuffer_struc.`

NOTE The cbuffer outputs are not the actual cbuffers, but the cbuffer structures. The cbuffer structure contains the read and write pointer locations as well as the size of the cbuffer. Figure 5-3 shows the `$audio_in_left_cbuffer_struc cbuffer` structure declaration.

```
// ** allocate memory for cbuffer structures **
.VAR $audio_in_left_cbuffer_struc[$cbuffer.STRUC_SIZE] =
   LENGTH($audio_in_left),           // size
   &$audio_in_left,                  // read pointer
   &$audio_in_left;                  // write pointer
```

Figure 5-3 $audio_in_left_cbuffer_struc declaration

5.1.2 Operator object level

The Operator object level is below the Stream Object level. It contains the address to the next operator in the linked list, the current operator function vector, and the list of operator-specific parameters. To terminate the linked list set the next operator field to `$cbops.NO_MORE_OPERATORS`. Figure 5-4 and Figure 5-5 show the generic Operator Object and FIR filter specific Operator Object declaration.

Operator Object
$cbops.NEXT_OPERATOR_ADDR_FIELD
$cbops.FUNCTION_VECTOR_FIELD
$cbops.PARAMETER_AREA_START_FIELD
< operator specific parameters >

Figure 5-4 Operator object definition

```
.BLOCK $audio_in_left_shift_op;
      .VAR $audio_in_left_shift_op.next =
&$audio_in_left_dc_remove_op;
      .VAR $audio_in_left_shift_op.func = &$cbops.shift;
      .VAR $audio_in_left_shift_op.param[$cbops.shift.STRUC_SIZE] =
         0,                             // Input index
         2,                             // Output index
         8;                             // Shift amount
   .ENDBLOCK;
```

Figure 5-5 : $audio_in_left_shift_op declaration

The first element in Figure 5-5 points to the next operator in the linked list, which is:

`$audio_in_left_dc_remove_op`.

The second element points to the function vector (`&$cbops.shift`) of the current operator (`$audio_in_left_shift_op`).

The third element defines the shift -specific parameters.

- The input index 0, refers to `$AUDIO_LEFT_IN_PORT` (the input to the `cbops.shift` operator is `$AUDIO_LEFT_IN_PORT`).
- The output index, 2, refers to `$audio_in_left_cbuffer_struc` (the output of the `cbops.shift` operator is `$audio_in_left_cbuffer_struc`).
- The shift amount number 8 left shift input data by 8 bits that converts the 16-bit data into 24-bit data.

5.2 Libraries used by my_first_dsp_app

The application uses two Kalimba libraries:

- The `core` library provides an API for buffers in Kalimba memory (which are called `cbuffers`) and the Memory Management Unit (MMU) ports, which streams data between the BlueCore subsystem and the Kalimba.
  `Core` libraries have functions that:
  - Set-up the interrupt controller
  - Send and receive messages from the VM application
  - Manage a simple stack
- The `cbops` library provides routines to copy data between buffers/ports while doing some form of processing. Operator functions, which are held in a linked list, carry out processing. The library contains a number of operators to perform frequently required functionality. You can also write custom functions, as needed.

5.2.1 To use Kalimba libraries

To use the Kalimba libraries the application must include the relevant library header files such as:

```
#include "core_library.h"
```

This define is included in the source file. In addition, the libraries must be listed in the project's properties. These can be added in xIDE by selecting **Properties…** in the xIDE **Project** menu.

Add libraries in the **Build System** section of the **Project Properties** window, see Figure 5-6. Multiple libraries must be comma separated.

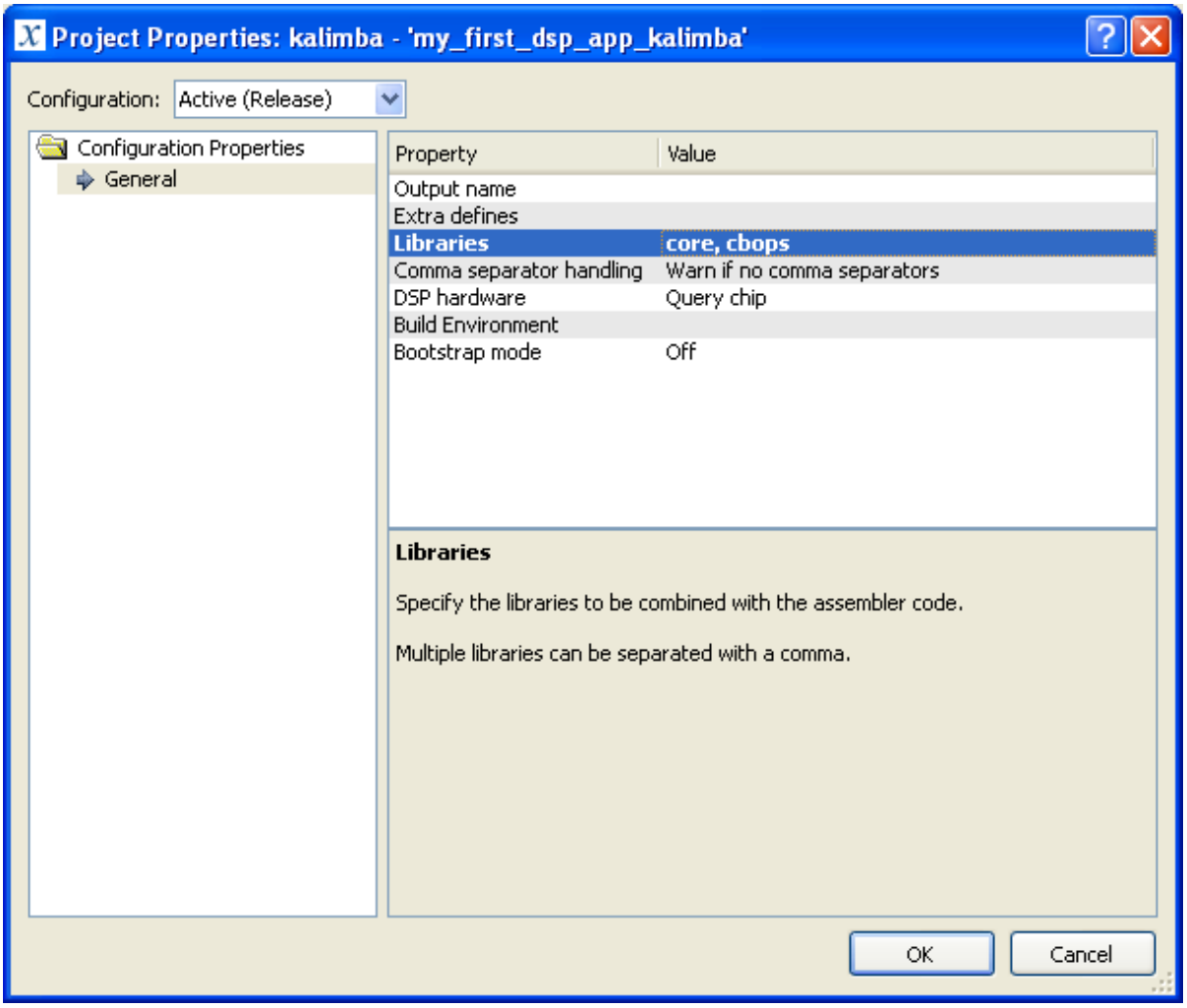


Figure 5-6 Using Kalimba libraries

5.3 Audio routing in my_first_dsp_app

The example application, `my_first_dsp_app`, routes stereo audio from the ADC to the DAC. The ADC and DAC are free-running and controlled by hardware. Their buffers are managed by the Memory Management Unit (MMU). The DSP has its own input and output circular connection buffers known as cbuffers.

The example Kalimba application has three parts. See Figure 5-7.

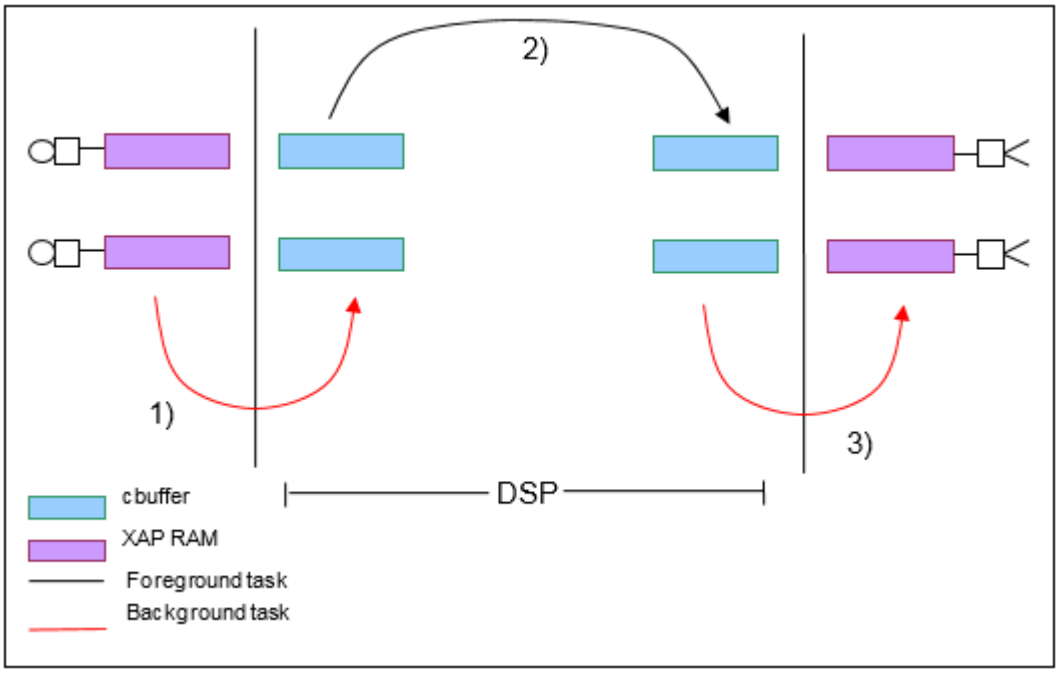


Figure 5-7 Audio buffering

The three parts of the application are:

1. Copying audio from the MMU to input `cbuffers`
2. Copying audio from the input `cbuffers` to the output `cbuffers`
3. Copying audio from the output `cbuffer` to the MMU

5.4 my_first_dsp_app application details

The input and output processes (steps 1 and 3 in Figure 5-7) are handled in interrupt routines. These input and output processes are effectively handled in the background.

Each of the three parts of the application involves copying audio between buffers. Each step is a separate linked list of operations that copy and process the audio. This linked list of operations (or `cbops`) specifies audio processing. The operators in the list are configurable; you can add custom processes by writing new operations.

In `my_first_dsp_app`, the operator list for the input copy in Step 1 of Figure 5-7 includes a shift of the audio and removal of any DC. The shift is carried out because the data memory is 24 bits wide and the PCM audio supplied by the codec is 16 bits wide.

NOTE Further operations could be carried out during this input copy, such as amplifying or attenuating the signal by a further shift or checks for overflow.

In the foreground, the main application loop calls a function to copy audio from the input `cbuffer` to the output `cbuffer`, step 2 in Figure 5-7. It then sleeps for 1 ms (time permitting) to save power.

During the copy routine call interrupts must be blocked to prevent memory corruption. As with the input and output steps, this copying is carried out as part of a linked list of operators. It is possible to insert processing tasks as required into the list.

Figure 5-8 shows the structure of the Kalimba application. This code is in `my_first_dsp_app.asm`.

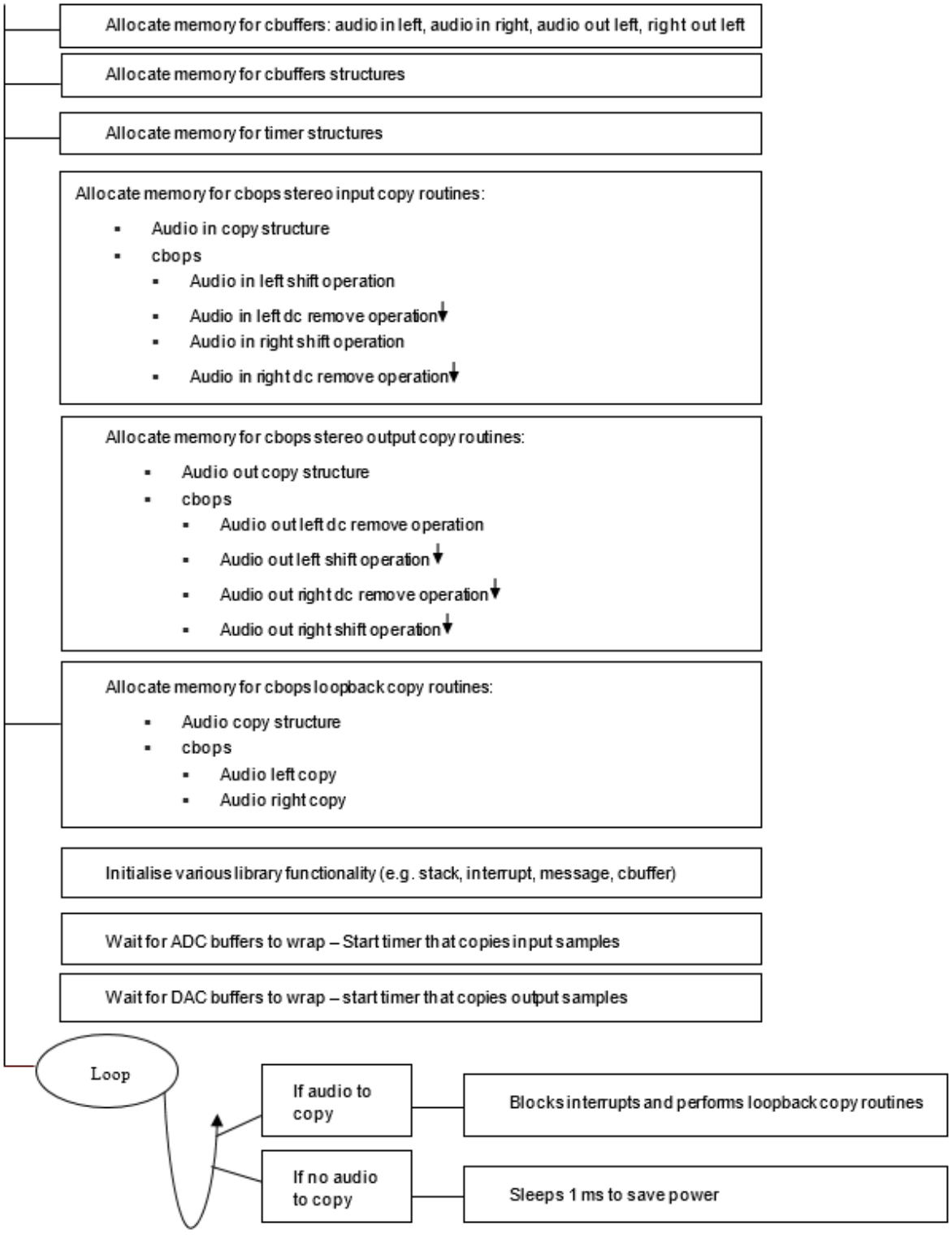


Figure 5-8 my_first_dsp_app outline

A my_first_dsp_app VM code

The following C source code is provided in:

<ADK Installation folder>\Examples*my_first_dsp_app* folder

Do not cut and paste the following sample code for use.

```
/*
Copyright (c) 2006 - 2015 Qualcomm Technologies International, Ltd.
An example app for routing audio through the Kalimba DSP from ADC to DAC
*/
#include <kalimba.h>
#include <kalimba_standard_messages.h>
#include <file.h>
#include <string.h>
#include <panic.h>
#include <source.h>
#include <sink.h>
#include <stream.h>
#include <connection.h>
#include <micbias.h>
#include <pio.h>
void PioSetPio (uint16 pPIO , bool pOnOrOff);
/* Select Amp PIO depending on board used. If not defined, assume the
CNS10001v4 board is assumed. */
#ifdef H13179V2
#define POWER_AMP_PIO 14
#else /* Assume CNS10001v4 */
#define POWER_AMP_PIO 4
#endif
```

```
/* Define the macro "BYPASS_KALIMBA" to bypass Kalimba DSP otherwise direct
ADC->DAC */

/* #define BYPASS_KALIMBA */

/* Define the macro "MIC_INPUT" for microphone input otherwise line-in
input */

/* #define MIC_INPUT */

/* Location of DSP kap file in the file system */

static const char kal[] = "my_first_dsp_app_kalimba/
my_first_dsp_app_kalimba.kap";

uint16 sampleRate = 48000;

void start_kalimba(void);

void connect_streams(void);

/* Main VM routine */

int main(void)

{

/* Load the Kalimba */

start_kalimba();

/* Connect up the ADCs and DACS */

connect_streams();

/* Start the Kalimba */

PanicFalse( KalimbaSendMessage(KALIMBA_MSG_GO,0,0,0,0) );

/* Remain in MessageLoop (handles messages) */

MessageLoop();

return 0;

}

void start_kalimba(void)

{

/* Find the codec file in the file system */

FILE_INDEX index = FileFind( FILE_ROOT, (const char *)kal, strlen(kal) );

/* Did we find the desired file? */

PanicFalse( index != FILE_NONE );

/* Load the codec into Kalimba */

PanicFalse( KalimbaLoad( index ) );

}
```

```
void connect_streams(void)

{

/* Access left and right ADC and DAC */

Source audio_source_a = StreamAudioSource( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0,

AUDIO_CHANNEL_A );

Source audio_source_b = StreamAudioSource( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0,

AUDIO_CHANNEL_B );

Sink audio_sink_a = StreamAudioSink( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0,

AUDIO_CHANNEL_A );

Sink audio_sink_b = StreamAudioSink( AUDIO_HARDWARE_CODEC,
AUDIO_INSTANCE_0,

AUDIO_CHANNEL_B );

/* Configure sampling rate for both channels and synchronise left and right
channels */

PanicFalse( SourceConfigure(audio_source_a, STREAM_CODEC_INPUT_RATE,
sampleRate) );

PanicFalse( SourceConfigure(audio_source_b, STREAM_CODEC_INPUT_RATE,
sampleRate) );

PanicFalse( SourceSynchronise(audio_source_a, audio_source_b) );

PanicFalse( SinkConfigure(audio_sink_a, STREAM_CODEC_OUTPUT_RATE,
sampleRate) );

PanicFalse( SinkConfigure(audio_sink_b, STREAM_CODEC_OUTPUT_RATE,
sampleRate) );

PanicFalse( SinkSynchronise(audio_sink_a, audio_sink_b) );

/* Set up codec gains */

#ifdef MIC_INPUT

PanicFalse( SourceConfigure(audio_source_a,
STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 1) );

PanicFalse( SourceConfigure(audio_source_b,
STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 1) );

PanicFalse(MicbiasConfigure(MIC_BIAS_0, MIC_BIAS_ENABLE,
MIC_BIAS_FORCE_ON));

PanicFalse(MicbiasConfigure(MIC_BIAS_1, MIC_BIAS_ENABLE,
MIC_BIAS_FORCE_ON));

#else
```

```
PanicFalse( SourceConfigure(audio_source_a,
STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 0) );

PanicFalse( SourceConfigure(audio_source_b,
STREAM_CODEC_MIC_INPUT_GAIN_ENABLE, 0) );

#endif

PanicFalse( SourceConfigure(audio_source_a, STREAM_CODEC_INPUT_GAIN, 10) );

PanicFalse( SourceConfigure(audio_source_b, STREAM_CODEC_INPUT_GAIN, 10) );

PioSetPio(POWER_AMP_PIO, TRUE);

PanicFalse( SinkConfigure(audio_sink_a, STREAM_CODEC_OUTPUT_GAIN, 15) );

PanicFalse( SinkConfigure(audio_sink_b, STREAM_CODEC_OUTPUT_GAIN, 15) );

#ifdef BYPASS_KALIMBA

/* Plug Left ADC directly into left DAC */

PanicFalse( StreamConnect(audio_source_a, audio_sink_a) );

/* Plug Right ADC directly into right DAC */

PanicFalse( StreamConnect(audio_source_b, audio_sink_b) );

#else

/* Plug Left ADC into port 0 */

PanicFalse( StreamConnect(audio_source_a, StreamKalimbaSink(0)) );

/* Plug Right ADC into port 1 */

PanicFalse( StreamConnect(audio_source_b, StreamKalimbaSink(1)) );

/* Plug port 0 into Left DAC */

PanicFalse( StreamConnect(StreamKalimbaSource(0), audio_sink_a) );

/* Plug port 1 into Right DAC */

PanicFalse( StreamConnect(StreamKalimbaSource(1), audio_sink_b) );

#endif

}

void PioSetPio (uint16 pPIO , bool pOnOrOff)

{

uint16 lPinVals = 0 ;

uint16 lWhichPin = (1<< pPIO) ;

if ( pOnOrOff )

{

lPinVals = lWhichPin ;
```

```
}

else

{

lPinVals = 0x0000;/*clr the corresponding bit*/

}

/*(mask,bits) setting bit to a '1' sets the corresponding port as an
output*/

PioSetDir32( lWhichPin , lWhichPin );

/*set the value of the pin*/

PioSet32 ( lWhichPin , lPinVals ) ;

}
```

B my_first_dsp_app Kalimba DSP code

The following Kalimba code is provided in the:

<ADK Installation folder>\Examples*my_first_dsp_app* folder

Do not cut and paste the following sample code for use.

```
//
*****************************************************************************

// Copyright (c) 2006 - 2015 Qualcomm Technologies International, Ltd.

//
*****************************************************************************

//
*****************************************************************************

// DESCRIPTION

// An example app for routing audio through the Kalimba DSP from ADC to DAC

//

// NOTES

//

// What the code does:

// Sets up cbuffers (circular connection buffers) for reading audio from

// the ADC and routing to the DAC. Cbuffers are serviced by timer

// interrupts.

//
*****************************************************************************

// 1.5ms is chosen as the interrupt rate for the audio input/output

// because: ADC/DAC MMU buffer is 512 bytes = 256samples

// = 256 samples = 5.0 ms @ 48kHz (approx)

// Assume absolute worst case jitter on interrupts = 1ms
```

```
// Hence 1.5ms (much less than 5 ms) would be OK for audio input/output
interrupts

#define $TMR_PERIOD_AUDIO_COPY 1500

#define $AUDIO_CBUFFER_SIZE 512

#define $DATA_COPIED 0

#define $DATA_NOT_COPIED 1

#define $TONES_BLOCK_SIZE $AUDIO_CBUFFER_SIZE/2

// Standard includes

#include "core_library.h"

#include "cbops_library.h"

.MODULE $M.main;

.CODESEGMENT PM;

.DATASEGMENT DM;

$main:

// ** Setup ports that are to be used **

.CONST $AUDIO_LEFT_IN_PORT ($cbuffer.READ_PORT_MASK + 0);

.CONST $AUDIO_RIGHT_IN_PORT ($cbuffer.READ_PORT_MASK + 1);

.CONST $AUDIO_LEFT_OUT_PORT ($cbuffer.WRITE_PORT_MASK + 0);

.CONST $AUDIO_RIGHT_OUT_PORT ($cbuffer.WRITE_PORT_MASK + 1);

// ** Allocate memory for cbuffers **

// cbuffers are 'circular connection buffers'

.VAR/DMCIRC $audio_in_left[$AUDIO_CBUFFER_SIZE];

.VAR/DMCIRC $audio_in_right[$AUDIO_CBUFFER_SIZE];

.VAR/DMCIRC $audio_out_left[$AUDIO_CBUFFER_SIZE];

.VAR/DMCIRC $audio_out_right[$AUDIO_CBUFFER_SIZE];

// ** Allocate memory for cbuffer structures **

.VAR $audio_in_left_cbuffer_struc[$cbuffer.STRUC_SIZE] =

LENGTH($audio_in_left), // Size

&$audio_in_left, // Read pointer

&$audio_in_left; // Write pointer

.VAR $audio_in_right_cbuffer_struc[$cbuffer.STRUC_SIZE] =

LENGTH($audio_in_right), // Size

&$audio_in_right, // Read pointer
```

```
&$audio_in_right; // Write pointer
.VAR $audio_out_left_cbuffer_struc[$cbuffer.STRUC_SIZE] =
LENGTH($audio_out_left), // Size
&$audio_out_left, // Read pointer
&$audio_out_left; // Write pointer
.VAR $audio_out_right_cbuffer_struc[$cbuffer.STRUC_SIZE] =
LENGTH($audio_out_right), // Size
&$audio_out_right, // Read pointer
&$audio_out_right; // Write pointer
// ** Allocate memory for timer structures **
.VAR $audio_in_timer_struc[$timer.STRUC_SIZE];
.VAR $audio_out_timer_struc[$timer.STRUC_SIZE];
// Input:
// ------
// ** Allocate memory for cbops stereo input copy routines **
.VAR $audio_in_copy_struc[] =
&$audio_in_left_shift_op, // First operator block
2, // Number of inputs
$AUDIO_LEFT_IN_PORT, // Input
$AUDIO_RIGHT_IN_PORT, // Input
2, // Number of outputs
&$audio_in_left_cbuffer_struc, // Output
&$audio_in_right_cbuffer_struc; // Output
.BLOCK $audio_in_left_shift_op;
.VAR $audio_in_left_shift_op.next = &$audio_in_left_dc_remove_op;
.VAR $audio_in_left_shift_op.func = &$cbops.shift;
.VAR $audio_in_left_shift_op.param[$cbops.shift.STRUC_SIZE] =
0, // Input index
2, // Output index
8; // Shift amount
.ENDBLOCK;
.BLOCK $audio_in_left_dc_remove_op;
```

```
.VAR $audio_in_left_dc_remove_op.next = &$audio_in_right_shift_op;
.VAR $audio_in_left_dc_remove_op.func = &$cbops.dc_remove;
.VAR $audio_in_left_dc_remove_op.param[$cbops.dc_remove.STRUC_SIZE] =
2, // Input index
2, // Output index
0; // DC estimate
.ENDBLOCK;
.BLOCK $audio_in_right_shift_op;
.VAR $audio_in_right_shift_op.next = &$audio_in_right_dc_remove_op;
.VAR $audio_in_right_shift_op.func = &$cbops.shift;
.VAR $audio_in_right_shift_op.param[$cbops.shift.STRUC_SIZE] =
1, // Input index
3, // Output index
8; // Shift amount
.ENDBLOCK;
.BLOCK $audio_in_right_dc_remove_op;
.VAR $audio_in_right_dc_remove_op.next = $cbops.NO_MORE_OPERATORS;
.VAR $audio_in_right_dc_remove_op.func = &$cbops.dc_remove;
.VAR $audio_in_right_dc_remove_op.param[$cbops.dc_remove.STRUC_SIZE] =
3, // Input index
3, // Output index
0; // DC estimate
.ENDBLOCK;
// Output:
// ------
// ** Allocate memory for cbops stereo output copy routines **
.VAR $audio_out_copy_struc[] =
&$audio_out_left_dc_remove_op, // First operator block
2, // Number of inputs
&$audio_out_left_cbuffer_struc, // Input
&$audio_out_right_cbuffer_struc, // Input
2, // Number of outputs
```

```
$AUDIO_LEFT_OUT_PORT, // Output
$AUDIO_RIGHT_OUT_PORT; // Output
.BLOCK $audio_out_left_dc_remove_op;
.VAR $audio_out_left_dc_remove_op.next = &$audio_out_left_shift_op;
.VAR $audio_out_left_dc_remove_op.func = &$cbops.dc_remove;
.VAR $audio_out_left_dc_remove_op.param[$cbops.dc_remove.STRUC_SIZE] =
0, // Input index
0, // Output index
0; // DC estimate
.ENDBLOCK;
.BLOCK $audio_out_left_shift_op;
.VAR $audio_out_left_shift_op.next = &$audio_out_right_dc_remove_op;
.VAR $audio_out_left_shift_op.func = &$cbops.shift;
.VAR $audio_out_left_shift_op.param[$cbops.shift.STRUC_SIZE] =
0, // Input index.
2, // Output index
-8; // Shift amount
.ENDBLOCK;
.BLOCK $audio_out_right_dc_remove_op;
.VAR $audio_out_right_dc_remove_op.next = &$audio_out_right_shift_op;
.VAR $audio_out_right_dc_remove_op.func = &$cbops.dc_remove;
.VAR $audio_out_right_dc_remove_op.param[$cbops.dc_remove.STRUC_SIZE] =
1, // Input index
1, // Output index
0; // DC estimate
.ENDBLOCK;
.BLOCK $audio_out_right_shift_op;
.VAR $audio_out_right_shift_op.next = $cbops.NO_MORE_OPERATORS;
.VAR $audio_out_right_shift_op.func = &$cbops.shift;
.VAR $audio_out_right_shift_op.param[$cbops.shift.STRUC_SIZE] =
1, // Input index
3, // Output index
```

```
-8; // Shift amount
.ENDBLOCK;
// Input to Output:
// ----------------
// ** Allocate memory for cbops stereo loopback copy routines **
.VAR $audio_loopback_copy_struc[] =
&$audio_loopback_left_copy_op, // First operator block
2, // Number of inputs
&$audio_in_left_cbuffer_struc, // Input
&$audio_in_right_cbuffer_struc, // Input
2, // Number of outputs
&$audio_out_left_cbuffer_struc, // Output
&$audio_out_right_cbuffer_struc; // Output
.BLOCK $audio_loopback_left_copy_op;
.VAR $audio_loopback_left_copy_op.next = &$audio_loopback_right_copy_op;
.VAR $audio_loopback_left_copy_op.func = &$cbops.copy_op;
.VAR $audio_loopback_left_copy_op.param[$cbops.copy_op.STRUC_SIZE] =
0, // Input index
2; // Output index
.ENDBLOCK;
.BLOCK $audio_loopback_right_copy_op;
.VAR $audio_loopback_right_copy_op.next = $cbops.NO_MORE_OPERATORS;
.VAR $audio_loopback_right_copy_op.func = &$cbops.copy_op;
.VAR $audio_loopback_right_copy_op.param[$cbops.copy_op.STRUC_SIZE] =
1, // Input index
3; // Output index
.ENDBLOCK;
// Initialise the stack library
call $stack.initialise;
// Initialise the interrupt library
call $interrupt.initialise;
// Initialise the message library
```

```
call $message.initialise;
// Initialise the cbuffer library
call $cbuffer.initialise;
// Tell VM we're ready and wait for the go message
call $message.send_ready_wait_for_go;
// Left and right audio channels from the MMU have been synced to each
// other by the VM app but are free running in that the DSP doesn't tell
// them to start. We need to make sure that our copying between the
// cbuffers and the MMU buffers starts off in sync with respect to left
// and right channels. To do this we make sure that when we start the
// copying timers that there is no chance of a buffer wrap around
// occurring within the timer period. The easiest way to do this is to
// start the timers just after a buffer wrap around occurs.
// Wait for ADC buffers to have just wrapped around
wait_for_adc_buffer_wraparound:
r0 = $AUDIO_LEFT_IN_PORT;
call $cbuffer.calc_amount_data;
// If the amount of data in the buffer is less than 32 bytes then a
// buffer wrap around must have just occurred.
Null = r0 - 32;
if POS jump wait_for_adc_buffer_wraparound;
// Start timer that copies input samples
r1 = &$audio_in_timer_struc;
r2 = $TMR_PERIOD_AUDIO_COPY;
r3 = &$audio_in_copy_handler;
call $timer.schedule_event_in;
// Wait for DAC buffers to have just wrapped around
wait_for_dac_buffer_wraparound:
r0 = $AUDIO_LEFT_OUT_PORT;
call $cbuffer.calc_amount_space;
// If the amount of space in the buffer is less than 32 bytes then a
// buffer wrap around must have just occurred.
```

```
Null = r0 - 32;
if POS jump wait_for_dac_buffer_wraparound;
// Start timer that copies output samples
r1 = &$audio_out_timer_struc;
r2 = $TMR_PERIOD_AUDIO_COPY;
r3 = &$audio_out_copy_handler;
call $timer.schedule_event_in;
// Start a loop to copy the data from the input through to the output
// buffers
copy_loop:
call $loopback_copy;
Null = r0 - $DATA_NOT_COPIED;
if Z call $timer.1ms_delay;
jump copy_loop;
.ENDMODULE;
//
*****************************************************************************
***
// MODULE:
// $audio_in_copy_handler
//
// DESCRIPTION:
// Function called on an interrupt timer to copy samples from MMU
// input ports to internal cbuffers.
//
//
*****************************************************************************
***
.MODULE $M.audio_in_copy_handler;
.CODESEGMENT PM;
.DATASEGMENT DM;
$audio_in_copy_handler:
// Push rLink onto stack
$push_rLink_macro;
```

```
// Copy data whatever mode we are in to keep in sync
// transfer data from mmu port to internal cbuffer
r8 = &$audio_in_copy_struc;
call $cbops.copy;
// Post another timer event
r1 = &$audio_in_timer_struc;
r2 = $TMR_PERIOD_AUDIO_COPY;
r3 = &$audio_in_copy_handler;
call $timer.schedule_event_in;
// Pop rLink from stack
jump $pop_rLink_and_rts;
.ENDMODULE;
//
*****************************************************************************
***
// MODULE:
// $audio_out_copy_handler
//
// DESCRIPTION:
// Function called on an interrupt timer to copy samples from internal
// cbuffers to output MMU ports.
//
//
*****************************************************************************
***
.MODULE $M.audio_out_copy_handler;
.CODESEGMENT PM;
.DATASEGMENT DM;
$audio_out_copy_handler:
// Push rLink onto stack
$push_rLink_macro;
// Transfer data from internal cbuffer to MMU port
r8 = &$audio_out_copy_struc;
call $cbops.copy;
```

```
// Post another timer event
r1 = &$audio_out_timer_struc;
r2 = $TMR_PERIOD_AUDIO_COPY;
r3 = &$audio_out_copy_handler;
call $timer.schedule_event_in;
// Pop rLink from stack
jump $pop_rLink_and_rts;
.ENDMODULE;
//
*****************************************************************************
***
// MODULE:
// $loopback_copy
//
// DESCRIPTION:
// Routine to copy data from both input channels into the corresponding
// output buffer. Routine is on a long delay between calls so need to
// ensure we copy enough data.
//
// INPUTS:
// none
//
// OUTPUTS:
// r0 = DATA_COPIED / DATA_NOT_COPIED
//
// TRASHED REGISTERS:
// r8
// Called buffer routines called also trash:
// r1, r2, r3, r4, I0, L0, I1, L1, r10, DO LOOP
//
//
*****************************************************************************
***
.MODULE $M.loopback_copy;
```

```
.CODESEGMENT PM;
.DATASEGMENT DM;
$loopback_copy:
// push rLink onto stack
$push_rLink_macro;
// Check if there is enough data in the input buffer
r0 = &$audio_in_left_cbuffer_struc;
call $cbuffer.calc_amount_data;
Null = r0 - $TONES_BLOCK_SIZE;
if NEG jump dont_copy;
r0 = &$audio_in_right_cbuffer_struc;
call $cbuffer.calc_amount_data;
Null = r0 - $TONES_BLOCK_SIZE;
if NEG jump dont_copy;
// Check if there is enough data in the output buffer
r0 = &$audio_out_left_cbuffer_struc;
call $cbuffer.calc_amount_space;
Null = r0 - $TONES_BLOCK_SIZE;
if NEG jump dont_copy;
r0 = &$audio_out_right_cbuffer_struc;
call $cbuffer.calc_amount_space;
Null = r0 - $TONES_BLOCK_SIZE;
if NEG jump dont_copy;
// Block interrupts when copying sample data
call $interrupt.block;
// Copy the data between the buffers
r8 = &$audio_loopback_copy_struc;
call $cbops.copy;
// Now unblock interrupts
call $interrupt.unblock;
// Indicate DATA_COPIED and return
r0 = $DATA_COPIED;
```

```
// pop rLink from stack
jump $pop_rLink_and_rts;
// Indicate DATA_NOT_COPIED and return
dont_copy:
r0 = $DATA_NOT_COPIED;
// pop rLink from stack
jump $pop_rLink_and_rts;
.ENDMODULE;
```

Document references

Document	Reference
Qualcomm BlueCore xIDE User Guide	80-CT405-1/CS-00101500-UG
Qualcomm BlueCore Classic vs Native VM vs Assisted Native VM Application Note	80-CT403-1/CS-00122636-AN

Terms and definitions

Term	Definition
ADC	Analog to Digital Converter
ADK	Audio Development Kit
BC	BlueCore
BlueCore	Group term for the range of QTIL Bluetooth wireless technology ICs
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
CODEC	COder DECoder
DAC	Digital to Analog Converter
DSP	Digital Signal Processor
IC	Integrated Circuit
MMU	Memory Management Unit
QTIL	Qualcomm Technologies International, Ltd.
VM	Virtual Machine
xIDE	The BlueCore Integrated Development Environment