Qualcomm Technologies International, Ltd.

Qualcomm® BlueCore™ 1-mic and 2-mic Example Applications

Application Note

80-CT410-1 Rev. AK

November 6, 2017

Revision history

Revision	Date	Description
AA	March 2013	Original publication of this document. Alternative document number: CS-00235436
AB	December 2013	Updated for ADK 3
AC	January 2014	Packet Loss Concealment updated
AD	February 2014	Minor editorial corrections
AE	March 2014	Updated Table 3.2 to reflect PSKEY_USR6 changes.
AF	July 2014	Editorial updates
AG	January 2016	Updated for ADK 4.0.1 release
AH	January 2016	Added Qualcomm confidentiality statement
AI	September 2016	Editorial updates
AJ	July 2017	Updated document number. Editorial updates
AK	November 2017	Editorial updates

Contents

Tables

Figures

1 Introduction

This document describes the `one_mic_example` application provided in ADKs. Although this document focuses on the `one_mic_example`, its key concepts also apply to the `two_mic_example` and `usb_dongle` DSP applications.

It describes how to use the `one_mic_example` applications as a basis for creating a custom Bluetooth audio application. It includes detailed code excerpts to help understand the application's framework and an example that shows how to add a parametric equalizer to the application.

This document describes:

- How to run the `one_mic_example` application
- How operators move the data between MMU ports and circular buffers and between internal buffers.
- How frames of data are processed using the processing function tables
- How to add processing modules to the application and adjust the Task Scheduler
- How the framework minimizes latency via the Task Scheduler and performs rate matching via dropping and inserting samples

This document assumes that you are familiar with:

- xIDE, QTIL's integrated development environment
- Qualcomm® Kalimba™, QTIL's Digital Signal Processor
- The Kalimba instruction set

2 Data flow

The `one_mic_example` DSP application is a pass-through application. Incoming Bluetooth audio plays out of the DAC, and audio from the ADC is transmitted over Bluetooth. Tones and a sidetone signal from the ADC are mixed with the incoming Bluetooth data and played out of the DAC. A Packet Loss Concealment algorithm runs on the incoming Bluetooth to conceal corruption caused by lost and corrupted packets.

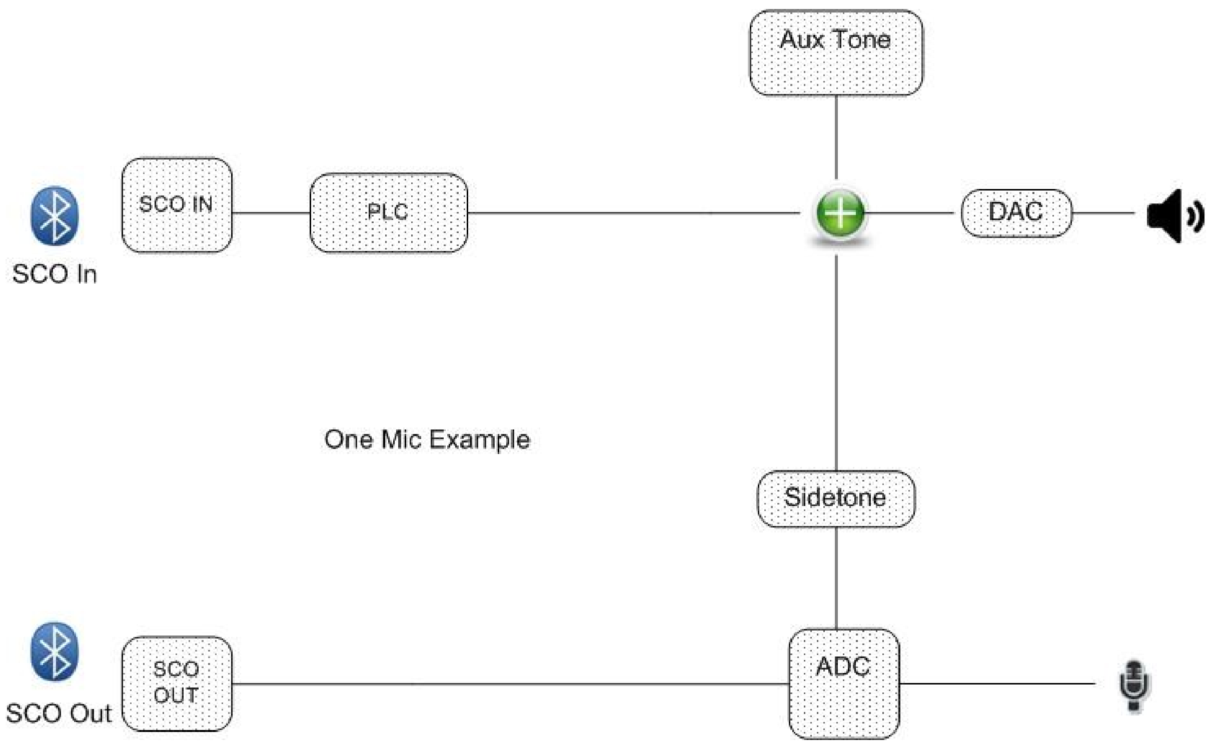

Figure 2-1 Data flow

3 Running the application

This section describes how to use:

- DSP application
- VM application
- VM plug-in
- VM `sink` configuration

3.1 DSP application

The ADK ships with two variants of the `one_mic_example` DSP application:

- `one_mic_example_16k.xip`
- `one_mic_example_cvsd.xip`.

Both projects use the same source files. However, preprocessor symbols are defined within each project's build properties to selectively include source code that is relevant to each application. Specifically, the `one_mic_example_16k.xip` project defines `uses_16kHz` symbol, which makes the application use an mSBC decoder and run with an audio sample rate of 16 kHz. The `one_mic_example_cvsd.xip` project does not define this symbol. It excludes the mSBC decoder and runs at an audio sample rate of 8 kHz. Each project creates a unique `.kap` file, which is loaded by the VM plug-in library `csr_common_example_plugin`.

3.2 VM application

ADK ships with the VM application `sink`. You can configure `sink` to load the `one_mic_example` applications. See section VM Sink configuration for more information.

3.3 VM Plug-in

VM plug-ins are libraries that enable communication between the DSP and the VM. Each DSP application has its own unique plug-in. VM applications do not have direct access to plug-ins. Instead, they make function calls into the audio library. These function calls send messages to plug-ins, which send messages to the DSP. The audio library API is defined in the `audio.h` file, which is in the ADK's **/src/lib/audio** library directory.

one_mic_example applications use the csr_common_example_plugin.

Table 3-1 Plug-in API

VM Application Call to Audio library	VM Plug-in action	DSP Application action
AudioConnect	Loads .kap file Connects streams Sets DAC and ADC gains	Executes initialization code Starts timer event to copy audio data between and DSP
AudioSetMode	Sends pass-through mode message to DSP application	Goes into pass-through mode. **Note**: This application is in pass-through mode before receiving the message. This message is provided as an example only.
AudioSetSoftMute	Mutes/unmutes microphone inputs and audio outputs according to parameters given	-
AudioSetVolume	DAC gain is set to the level specified. Values greater than 0 dB are set to 0 dB.	-
AudioPlayTone	Sets the DAC gain to the value specified and connects the tone stream to the DSP. The DAC gain is restored when the tone completes.	Mixes the tone with the SCO data
AudioStopTone	Disconnects the tone stream from the DSP.	-
AudioDisconnect	Sets ADC and DAC gains to 45 dB Disconnects streams Powers off DSP	-

3.4 VM Sink configuration

To run the one_mic_example DSP application in the VM headset application, on a H13179v2 board:

1. Open headset.xiw located in the ADK's **apps\sink** directory.
2. Insert the one_mic_example DSP applications into the workspace:
   a. Click on **Project/Insert Project into Workspace**
   b. Navigate to the ADK **kalimba\apps\one_mic_example** directory.

   NOTE Perform Step 2 twice, once to insert one_mic_example_16k.xip and again to insert one_mic_example_cvsd.xip.

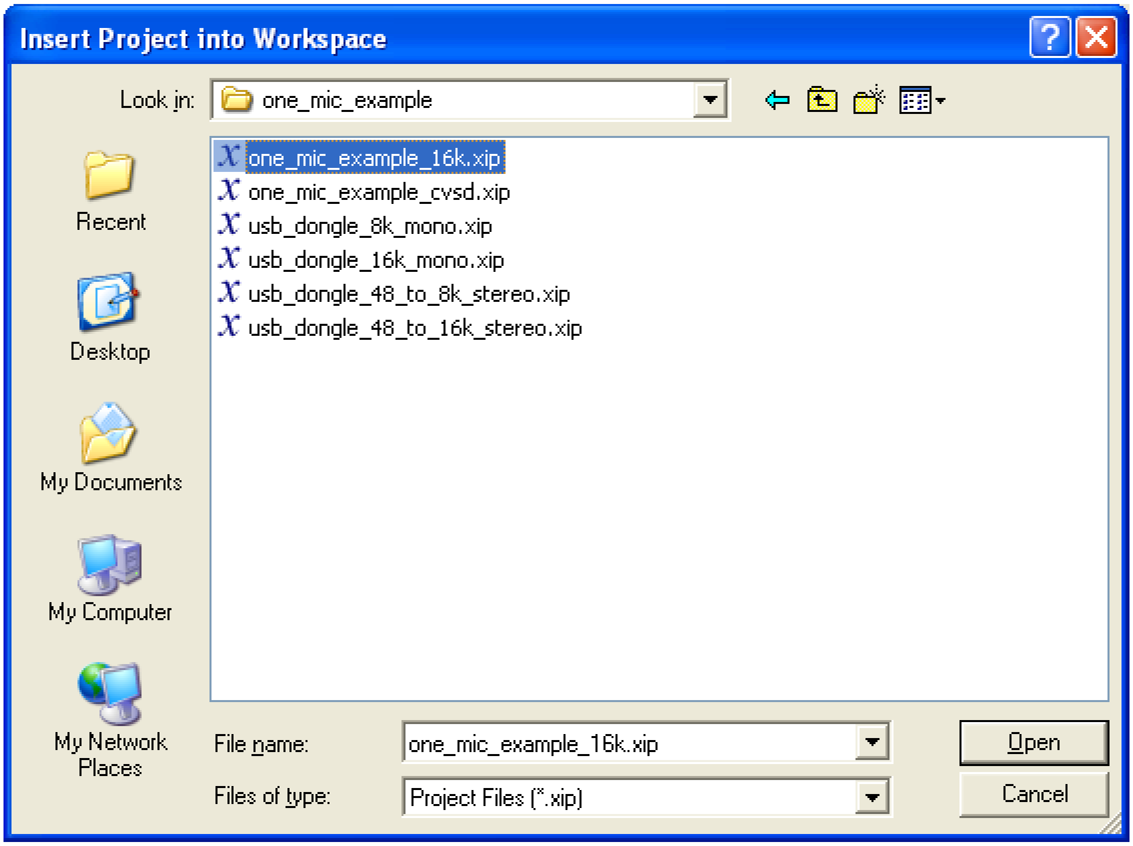

Figure 3-1 Insert project into workspace window

3. Open the **Project Properties** window.

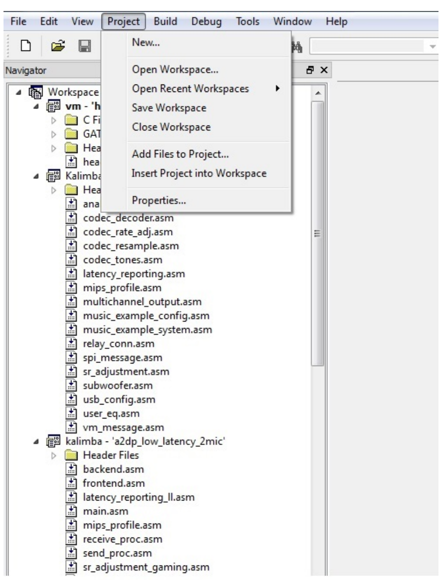

Figure 3-2 Project Properties window

4. Define the `INCLUDE_DSP_EXAMPLES` symbol in the workspace **Define Symbols** field.

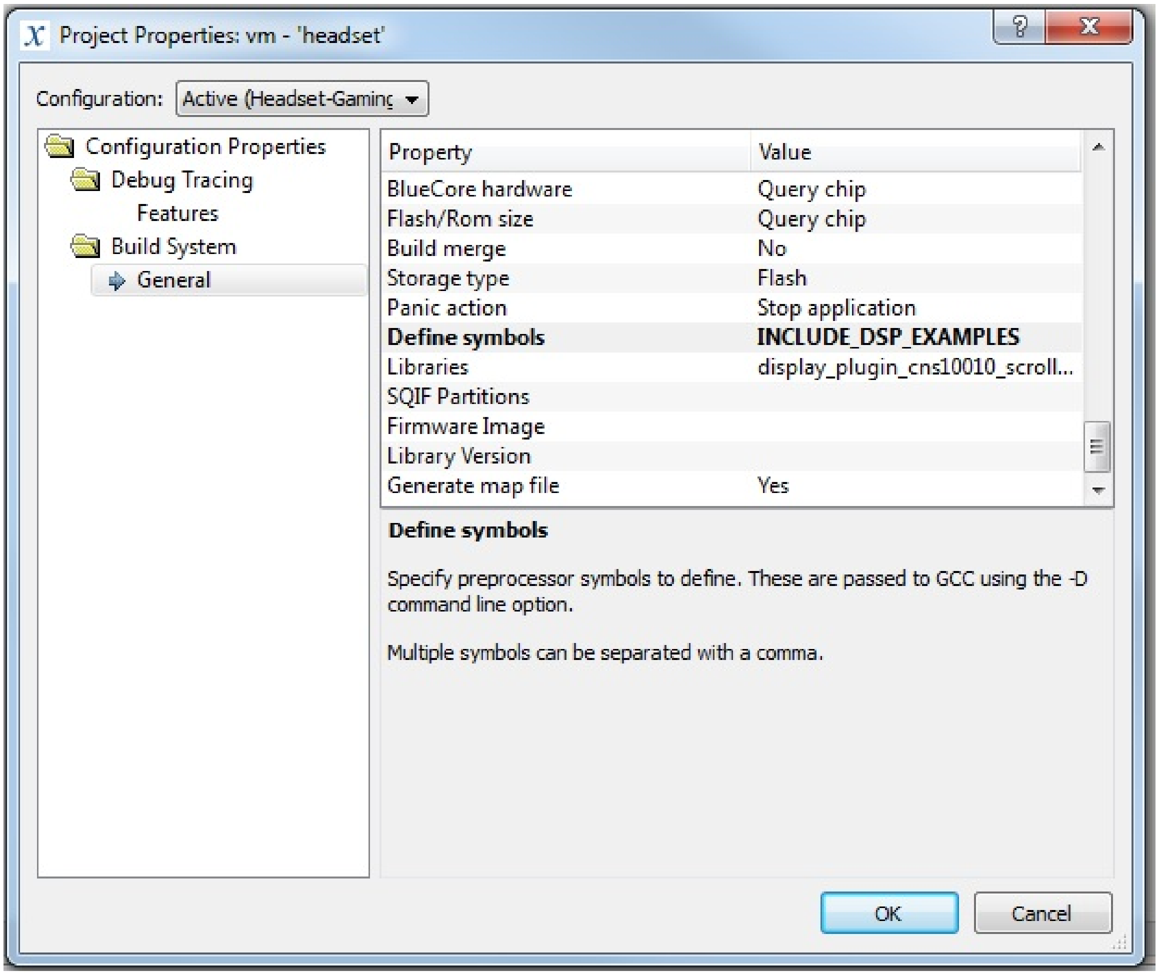

Figure 3-3 INCLUDE_DSP_EXAMPLES Window

Ensure `headset.mak` includes the `one_mic_example` entries. These entries copy the generated `one_mic_example_xxx.kap` files into the sink application image directory.

```
image/one_mic_example_cvsd/one_mic_example_cvsd.kap :
$(mkdir) image/one_mic_example_cvsd
$(copyfile) ../../kalimba/apps/one_mic_example/image/
one_mic_example_cvsd/one_mic_example_cvsd.kap $@
image.fs : image/one_mic_example_cvsd/one_mic_example_cvsd.kap
image/one_mic_example_16k/one_mic_example_16k.kap :
$(mkdir) image/one_mic_example_16k
$(copyfile) ../../kalimba/apps/one_mic_example/image/one_mic_example_16k/
one_mic_example_16k.kap $@
image.fs : image/one_mic_example_16k/one_mic_example_16k.kap
```

5. Turn on the BlueCore development board.

6. Merge PS Keys to load the appropriate one_mic_example DSP application:
   a. Open PSTool: **C:\ADK\tools\bin\PsTool.exe**
   b. Connect to the sink device.
   c. **Merge** the relevant PS Keys for the IC/Development board you are using, for example: `sink_system_csr8675.psr` file followed by `headset_gaming_H13179v2_H13478v2.psr` file. Both are in: **c:\ADK\apps\sink\configurations**

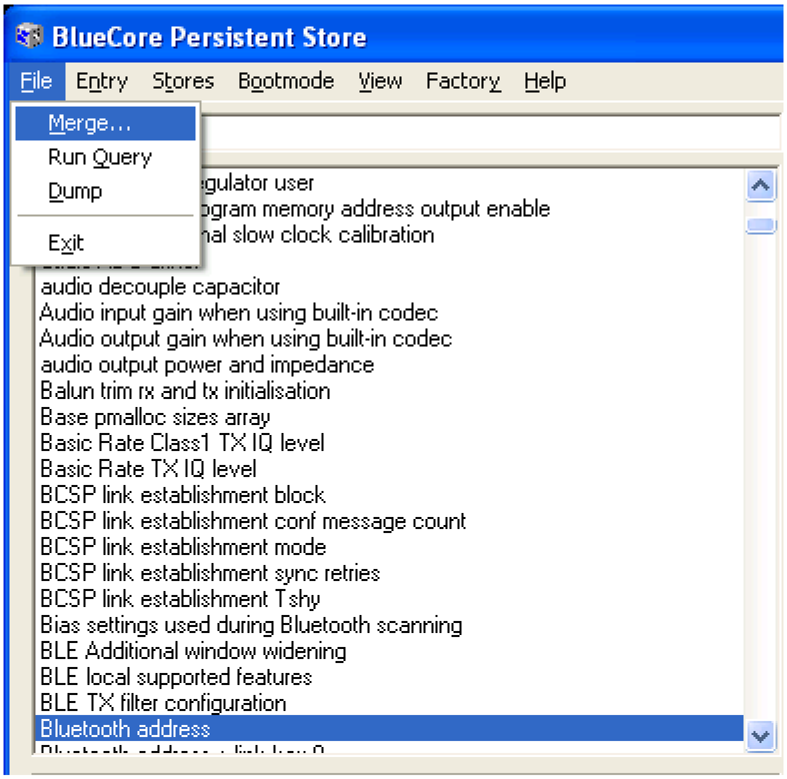

Figure 3-4 Merging PS keys with PSTool

7. Set the VM to load example plug-in on SCO connection:
   a. Open the Sink Configuration Tool:
   b. **C:\ADK\tools\bin\ConfigTool.exe**
   c. At the top level of the **Audio** tab, select **Example 1-Mic** from the Audio Plug-in list.

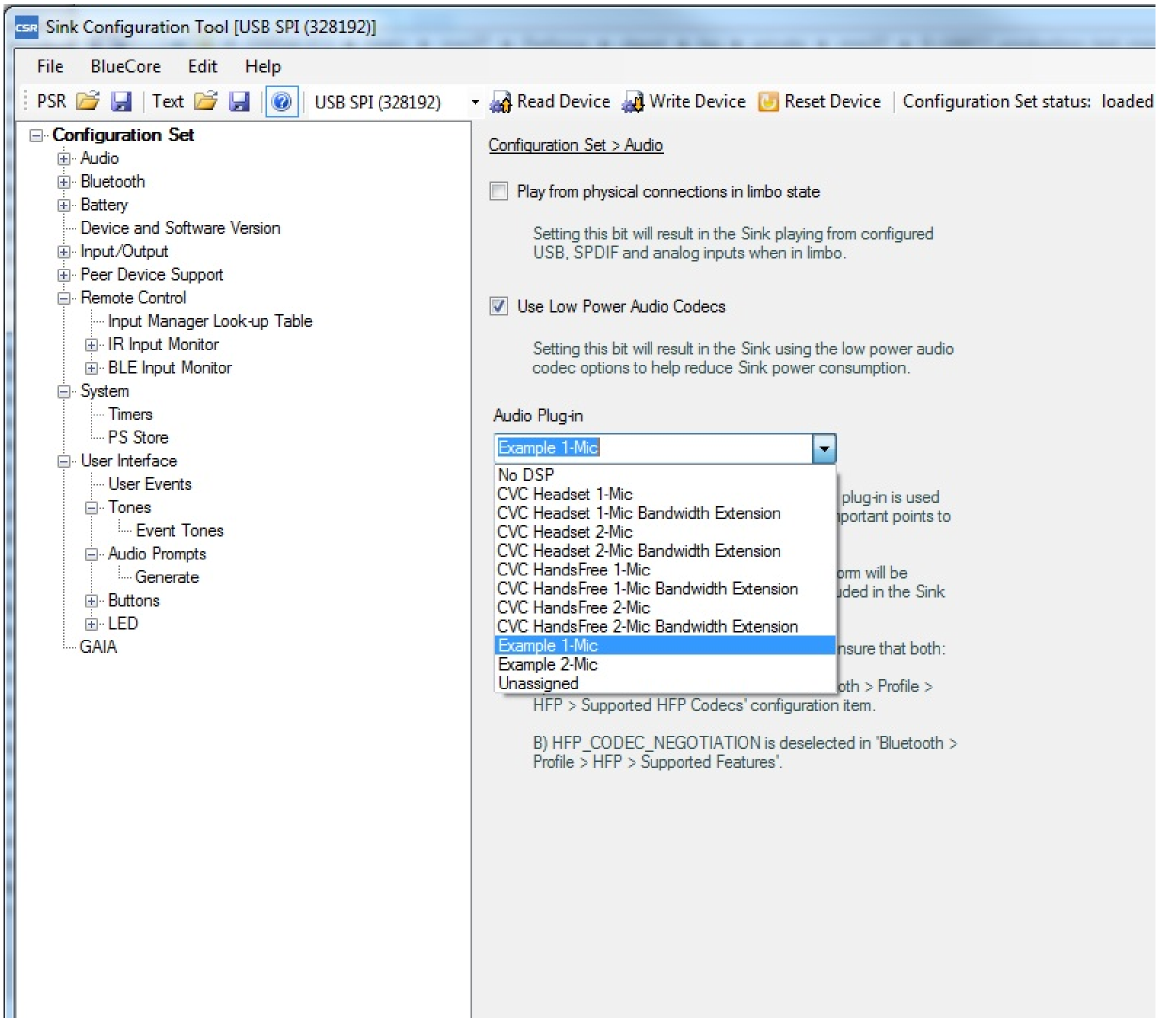

Figure 3-5 Audio plug-in list

8. Return to XIDE and ensure that the BlueCore development board designated as the sink device is connected to the correct transport port for example. LPT1 or USB-SPI corresponding to the **Debug-> Transport** dialogue box in xIDE.
9. Press the **F5** key to build and flash the application.

When the application is AWAKE, the development board is discoverable. This enables other suitable source devices to pair and connect to it.

4 Understanding the DSP code

The `one_mic_example` DSP application is an audio pass-through application. It uses a flexible framework, which enables you to easily add your own processing modules. A Task Scheduler runs in the foreground from within the application's infinite main loop. It calls intensive processing routines, such as echo cancellation and noise reduction algorithms.

An interrupt driven timer task runs in the background, executing small operator functions that copy data in between the MMU ports and DSP buffers.

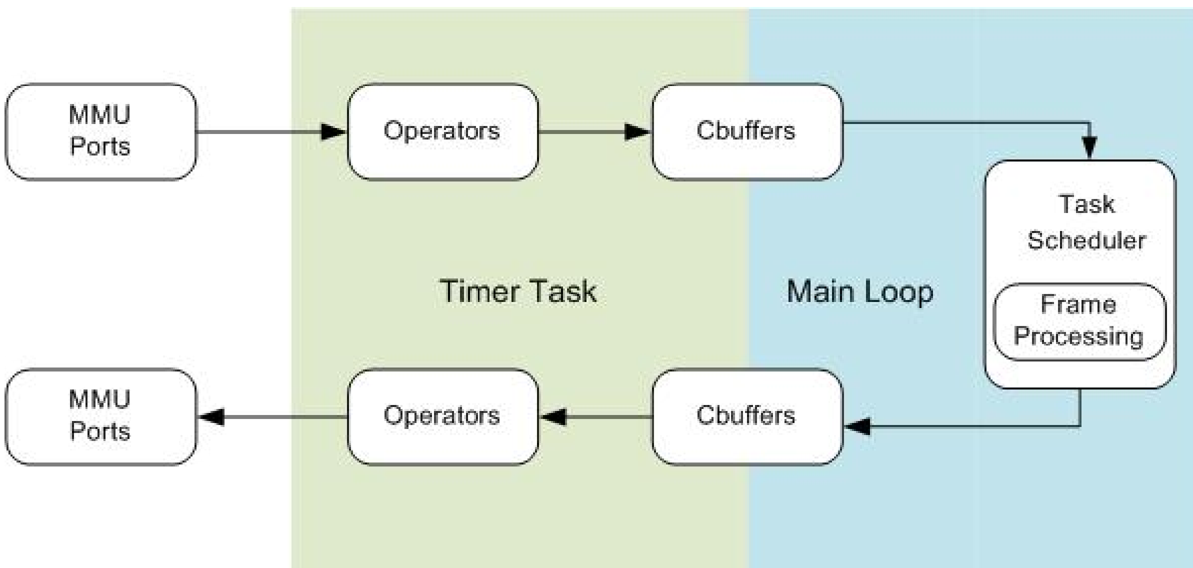

Figure 4-1 Foreground and background tasks

4.1 Task scheduler

The Task Scheduler schedules three main tasks:

1. Housekeeping
2. Receive processing
3. Send processing

The Task Schedulers schedules tasks according to when data is received and transmitted over Bluetooth to minimize buffering latency.

4.1.1 Housekeeping task

The `$main_housekeeping` task contains routines that are unsuitable for the send and receive tasks. The `one_mix_example` application does not use this task.

4.1.2 Send task

The `$main_send` task processes data, which is transmitted to the remote device. This tasks reads data from the ADC cbuffers and copies into its internal buffers with the `stream_copy` function and later encodes with `$frame_sync.sco_encode`.

4.1.3 Receive task

The `$main_receive` task processes data, which is received from the local device. The received data is first decoded with `$frame_sync.sco_decode` and later the `stream_copy` function copies the data to DAC buffers.

4.1.4 Using the task scheduler

Tasks are synchronized to the Bluetooth clock. They have a period of twelve 625 µs long Bluetooth slots to match the `one_mic_example` application's audio processing period of 7.5 ms.

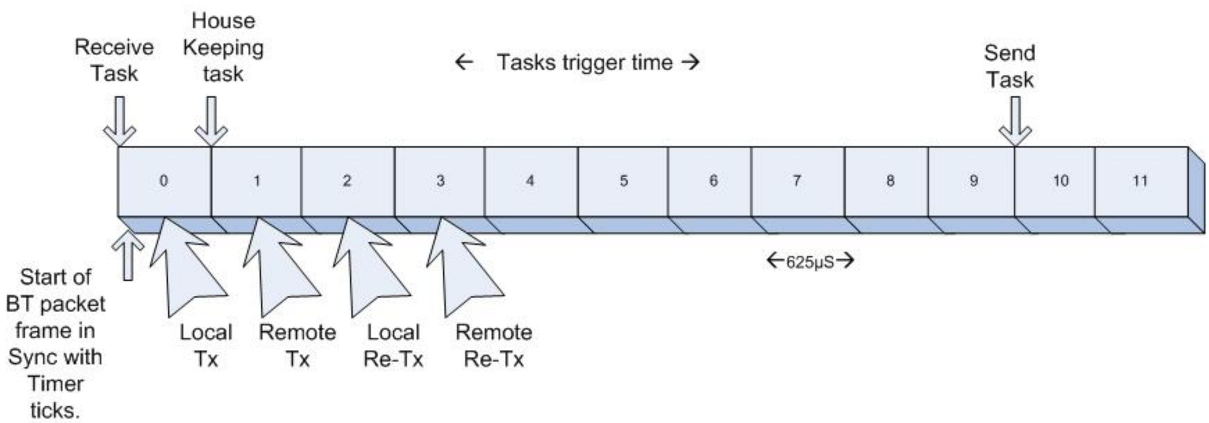

Figure 4-2 Task Scheduler slot numbering

The scheduler gives each slot a number, zero to 11. The local device transmits at slot 0, and can retransmit, if necessary, in slot 2.

The remote device transmits in slot 1, and can retransmit, if required in slot 3.

Tasks are scheduled using slot numbers. The goal is to schedule the send and receive tasks so data lingers in the local device's buffers for as little time as possible. If the send task takes less than one slot's worth of time (625 µs), the Task Scheduler schedules it at slot 11. If it takes more than one slot's worth of time, but fewer than two slots, the Task Scheduler schedules it at slot 10.

The Task Scheduler schedules send tasks with heavy processing loads to run earlier using lower slot numbers, compared with send tasks with light processing loads.

The Task Scheduler schedules the receive task to occur after the data arrives at the local device. It occurs at slot 0, which means received data is buffered for 8 slots before being serviced. This enables time-consuming send tasks to run, which is typical because acoustic echo cancellation algorithms are ubiquitous in `sink` applications. In this case, the send task is scheduled much earlier than shown in Figure 4-2, for example, at slot 4 or 5.

The housekeeping task is scheduled to run after the receive task completes. It is scheduled to run at slot 1, and then runs, as long as the receive task has finished. If the receive task requires more than one slot to run, the housekeeping task runs when the receive task finishes.

The following code shows the data object that the Task Scheduler uses.

```
.MODULE $M.App.scheduler;
   .DATASEGMENT DM;
   // ** Memory Allocation for CVC config structure
   .VAR    tasks[]=
      0,                        //    COUNT_FIELD
      12,                       //    MAX_COUNT_FIELD
      (Length(tasks) - $FRM_SCHEDULER.TASKS_FIELD)/2, // NUM_TASKS_FIELD
      0,                        //    TOTAL_MIPS_FIELD
      0,                        //    SEND_MIPS_FIELD
      0,                        //    TOTALTM_FIELD
      0,                        //    TOTALSND_FIELD
      0,                        //    TIMER_FIELD
      0,                        //    TRIGGER_FIELD
      // Task List (highest to lowest priority) - Limit 23 tasks
(Modulous 12)
      $main_send,               SND_PROCESS_TRIGGER,
      $main_receive,            0,
      $main_housekeeping,       1;
.ENDMODULE;
```

NOTE The value of `SND_PROCESS_TRIGGER` must be decreased if more processing is added to the `$main_send_task`.

4.2 Frame processing

The send and receive tasks process data as groups of samples called *frames.* The `frame_sync` library facilitates this processing.

4.3 Mode tables

This section provides a code excerpt from the `one_mic_example` application's receive task. This code shows the `frame_sync` being invoked.

```
// Get Current System Mode
r0 = M[$one_mic_example.sys_mode];
// Call processing table that corresponds to the current mode
r4 = M[$M.system_config.data.receive_mode_table + r0];
call $frame_sync.run_function_table;
```

The code that uses `$one_mic_example.sys_mode` as an index into the table is: `$M.system_config.data.receive_mode_table`.

The `one_mic_example` application only has one mode, which means that `$one_mic_example.sys_mode` is always zero corresponding to passthrough mode. See the code excerpt for the VAR receive mode table:

```
.VAR receive_mode_table[] =
    &rcv_funcs, // mode 0, receive function
    // more entries can be added here
    0;
```

However, the application can be expanded to have more modes. To do this, add entries into the `receive_mode_table` and create new processing function tables.

The value of `$one_mic_example.sys_mode` is set within the handler function `$one_mic_example.vm_msg.setmode`. This handler runs as a response to the `MESSAGE_SETMODE` message sent by the plug-in:

```
KalimbaSendMessage(MESSAGE_SETMODE , SYSMODE_PSTHRGH , 0, 0, 0 );
```

4.3.1 Processing function tables

Each task makes a call to `$frame_sync.run_function_table`, which executes functions, defined in the send and receive function tables:

The send function table is defined as:

```
.VAR snd_funcs[] =
 // Function                             r7                           r8
    $frame_sync.distribute_streams_ind,   &snd_process_streams,        0,
    $stream_copy,                         &snd_pass_thru_obj,          0,
    $frame_sync.update_streams_ind,       &snd_process_streams,        0,
    0;
```

The receive function table is defined as:

```
   .VAR rcv_funcs[] =
   // Function                           r7                           r8
      $frame_sync.distribute_streams_ind, &rcv_process_streams,        0,
      $stream_copy,                       &rcv_pass_thru_obj,          0,
      $frame_sync.update_streams_ind,     &rcv_process_streams,        0,
      0;
```

These function tables follow the API defined by the `$frame_sync.run_function_table` function. In each row, the first entry must be a function, while the second and third entries are optional data objects for that function. `$frame_sync.run_function_table` loads `r7` and `r8` with the data object addresses prior to calling each function. When `$frame_sync.run_function_table` encounters a zero instead of a function it exits, which is why the table is null terminated.

The first function in the table must be `$frame_sync.distribute_streams_ind`. This function tells subsequent functions in the table where to get and write audio samples.

The last function must be `$frame_sync.update_streams_ind`. This function knows the number of samples that have been processed by the preceding functions, updates read and write pointers to reflect the amount of data that has been read to and from the cbuffers that interface into the frame processing.

Functions in between `$frame_sync.distribute_streams_ind` and `$frame_sync.update_streams_ind` are application-specific, and are referred to as *processing modules.* The `one_mic_example` has one simple copy function in each table, but other applications may have multiple more sophisticated processing modules. The number of processing modules that exists between `$frame_sync.distribute_streams_ind` and `$frame_sync.update_streams_ind` is only limited by the DSP's MIPS and memory. Functions are called in the order that they are listed. To add processing modules, see Adding a processing module.

4.3.2 Stream maps tables

Each processing module used by the `frame_sync` architecture requires a data object. The data object must contain pointers to input and output buffers so that the processing module knows where to read and write data. The data objects must also contain the buffer lengths so the processing modules do not access memory outside the buffers.

The processing module `$stream_copy` and its data object `rcv_pass_thru_obj`, referenced by the `rcv_funcs` function table:

```
.VAR rcv_pass_thru_obj[$stream_copy.STRUC_SIZE] =
 &rcv_stream_map_sco_in,
 &rcv_stream_map_dac;
```

This `rcv_pass_thru_obj consists of all the stream maps in the direction of flow of data namely from SCO IN to the DAC. The stream copy module parses through the stream maps data object to obtain the information of input and output cbuffers on which it needs to operate. Typically there can be other parameters specific to the processing module.`

The `$frame_sync.distribute_streams_ind` is the first function to run each time the receive function table `rcv_funcs` runs.

`$frame_sync.distribute_streams_ind` takes `rcv_process_streams` as input.defined as:

```
.VAR    rcv_process_streams[] =
   &rcv_stream_map_sco_in,
   &rcv_stream_map_dac,
   0;
```

The first element of `rcv_stream_map_sco_in` is `sco_data.sco_in.cbuffer_struc`, a cbuffer structure. It contains current read and write pointers positions into the `sco_in` cbuffer.

Each time `$frame_sync.distribute_streams_ind runs`, it makes use of the read pointer position and cbuffer length from $`sco_data.sco_in.cbuffer_struc` to let the processing module know where to get its input data.

In the example, `rcv_stream_map_dac` is the output cbuffer for the `stream_copy` process as declared in the `rcv_pass_thru_obj`. This data structure contains the address of the DAC cbuffer, `& $dac_out.cbuffer_struc,` as well as references to the output of the `rcv_pass_thru_obj`:

Each time `$frame_sync.distribute_streams_ind` runs, it makes use of the write pointer position and circular buffer length from `$dac_out.cbuffer_struc` to let the processing module know where to put its output data.

After the processing modules run, `$frame_sync.update_streams_ind` updates the input cbuffer read pointers and the output cbuffer write pointers.

4.3.3 Rate matching

Audio processing is scheduled around Bluetooth packet transmissions. However, the ADC and DAC peripherals are driven by a local clock. If the Bluetooth clock is driven by the remote device, for example, if the remote device is master of the link, a drift occurs between these clocks because they are driven by different sources. If this is left unmanaged, local buffers eventually overrun or underrun unless explicit mechanisms are put in place to reduce this mismatch.

In the one mic example app this is made possible by using cbops hardware warp operator. This operator monitors the ADC or DAC data flow and estimates the rate and adjusts the ADC/DAC rate to match the processing rate. The sampling rate is estimated over a period of time determined by the period of a timer task calling this operator. In systems where the SCO is drives the timing to further contain the mismatch, the SCO logic adjusts the period of the timer task to maintain synchronization with the SCO transmissions.

4.3.4 Re-initialization function table

Re-initialization functions and corresponding data objects are placed in the `one_mic_example` application's re-initialization table:

```
.VAR reinitialize_table[] =
 // Function                         r7                      r8
 $frame_sync.sco_initialize,         &$sco_data.object,      0,
 0;
```

An interrupt-driven timer task runs at 625 µs intervals. Its handler, `$audio_copy_handler`, handles data transfers between firmware MMU ports and DSP cbuffers. Operators perform the data transfers.

4.3.5 Operators

Operators are functions that run within the `cbops_multirate` framework. Typically, they run in interrupt handlers and copy data between ports and buffers or between buffers while doing some form of processing. Operator functions are held in a doubly linked list where processed data is fed forward and space available information is fed backward for tighter control of the data flow. The routing of the audio through the operators is determined using buffer indexes, which are defined in the operator's main data structure.

The following code excerpt shows how to configure an operator to copy data from one buffer to another.

NOTE The following code excerpt depicts the cbops operator chain for reading data from the ADC into cbuffers, for more details see `main.asm` of the `one_mic_example_xxx.xip`.

```
DeclareCBuffer (cbuffer_struc,mem,$BLOCK_SIZE * $BUFFER_SCALING);
DeclareCBuffer
(sidetone_cbuffer_struc,sidetone_mem,4*($SAMPLE_RATE / 1000));
```

```
.VAR copy_struc[] =
   $cbops.scratch.BufferTable,      // BUFFER_TABLE_FIELD
   &copy_op,                        //
MAIN_FIRST_OPERATOR_FIELD
   &sidetone_copy_op,               // MTU_FIRST_OPERATOR_FIELD
   1,                               // NUM_INPUTS_FIELD
   $ADC_PORT,
   2,                               // NUM_OUTPUTS_FIELD
   &cbuffer_struc,
   &sidetone_cbuffer_struc,
   1,                               // NUM_INTERNAL_FIELD
   $cbops.scratch.cbuffer_struc2;
```

The copy structure is passed to `$cbops_multirate.copy` to process an operator chain

Fields in the copy structure are:

- Pointer to buffer table
- Main first operator field: Amount to use first operator field
- Number of input streams: Array of input Port IDs and cBuffer pointers
- Number of output streams: Array of output Port IDs and cBuffer pointers
- Number of internal buffers: Array of internal cBuffer pointers

4.4 Operator structure

```
.BLOCK copy_op;
    .VAR copy_op.mtu_next  = $cbops.NO_MORE_OPERATORS;
    .VAR copy_op.main_next = &hw_rate_op;
    .VAR copy_op.func = &$cbops.shift;
    .VAR copy_op.param[$cbops.shift.STRUC_SIZE] =
        ADCINDEX_PORT,         /*INPUT_INDEX_FIELD*/
        ADCINDEX_CBUFFER,      /*OUTPUT_INDEX_FIELD*/
        8;                     /*SHIFT_AMOUNT_FIELD*/
.ENDBLOCK;
```

Field description

- `mtu_next`: Pointer to previous operator in chain. NULL if first operator.
- `main_next`: Pointer to next operator in chain. NULL if last operator.
- `func`: Pointer to operator function block.
- `param`: Operator-specific data.

In this example all the operator chains such as `ADC_IN`, `DAC_OUT`, `TONE_IN` run in the context of the timer interrupt service routine controlled by the `$cbops_multirate.copy` function:

```
  r8 = &$usb_in_rm.copy_struc;
call $cbops_multirate.copy;
```

`$cbops_multirate.copy` calculates the amount of data in the input buffer and the amount of space in the output buffer and copies the minimum of the two.

NOTE The source code for the `Cbops_multirate` is not provided in the ADK. It is provided as a private library and can be used by including `cbops_multirate.h`.

`Cbops_multirate` can be run in the main loop context as well but in this example and in general we call the `$cbops_multirate.copy` in the interrupt context.

4.5 Packet Loss Concealment

The Packet Loss Concealment (PLC) is a QTIL proprietary algorithm that reconstructs corrupt audio packets received over an eSCO link. The `$main_receive` task calls the PLC just before it calls `$frame_sync.run_function_table`.

The source code for the PLC is not provided in the ADK. It is provided as a private library.

PLC in 1-mic and 2-mic example apps is statically enabled. This is done by setting the `CONFIG FIELD` in `$sco_data.object` to `0x2000`. This enables PLC. To have this static assignment, define uses_PLC = 1, in the project properties of the DSP application. In CVC apps this field is assigned to `0x0000` in the object structure and gets re-assigned during the program execution when enabled or disabled from the UFE. Since the example apps do not use any processing modules, static assignment is the only alternative to enable PLC. If the user intends to incorporate any processing module, for example, PEQ, the `CONFIG FIELD` can be rolled back to `0x0000`.

5 Adding a processing module

This section describes how to add a PEQ to the `one_mic_example_16k` DSP application. The PEQ is added to the receive stream directly after the PLC module.

NOTE This section also provides an example of how to add custom third-party processing modules.

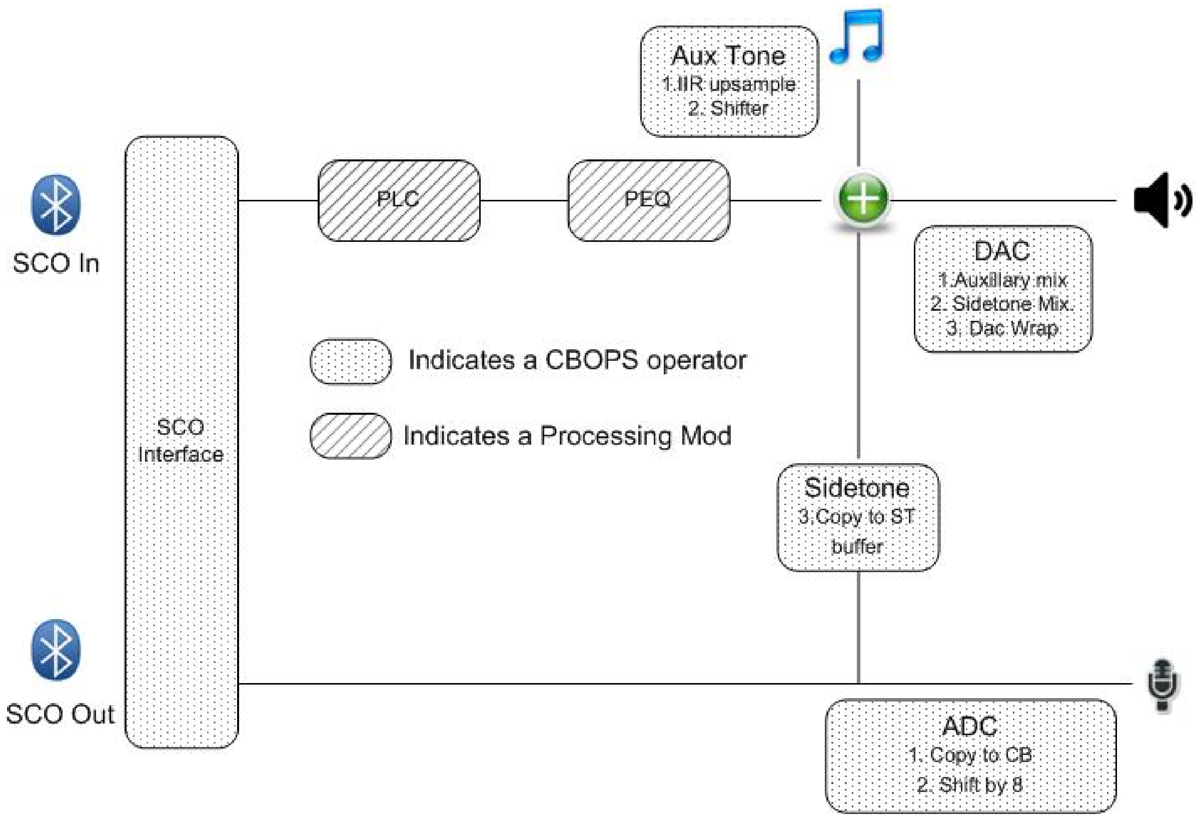

Figure 5-1 1-mic example with PEQ

5.1 Filter coefficient design

The Wideband CVC 1-mic sink UFE generates PEQ coefficients:

1. Launch the UFE application included with the ADK: **Programs\ADK\Tools\UniversalFrontEnd**
2. Select **1Mic Headset WB** from the drop-down menu at the top of the UFE.

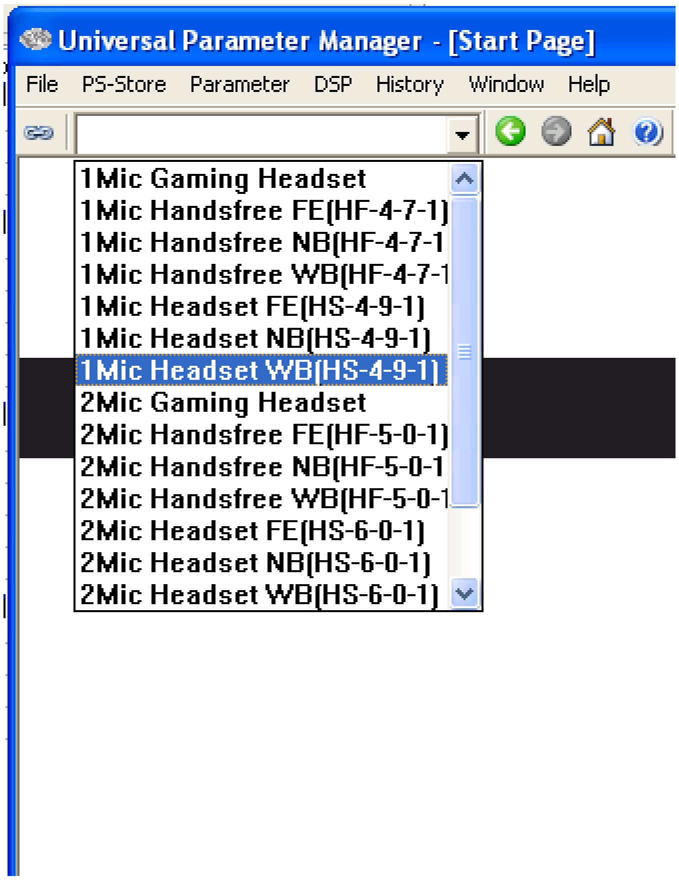

Figure 5-2 Select 1Mic headset WB

3. Click the **receive-path PEQ module** to open the **Receive Path Equalizer Settings** window and enter the settings shown in Figure 5-3.

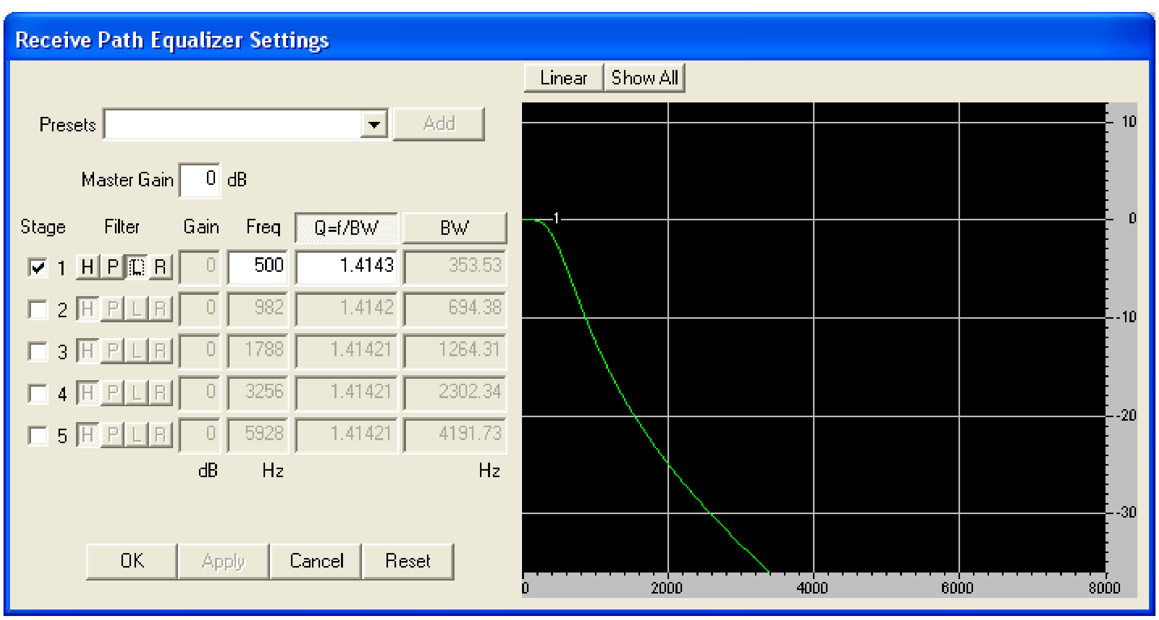

Figure 5-3 Receive Path Equalizer settings window

4. Click **OK**.
5. Click **File\Save params to File**.
6. Save the file as `peq_coeffs.txt`.

The PEQ values are now saved in the file with the prefix `RCV_PEQ`.

5.2 Source code notifications

This section describes the source code modifications required for inserting a 1 stage PEQ object. The following major changes are required:

1. Create a PEQ data object.
2. Add PEQ to the Initialization Function.
3. Add PEQ to the Receive Processing Function Table.

5.2.1 Creating a PEQ data object

To create a PEQ data object, copy values from the file `peq_coeffs.txt` into the `one_mic_example_config.asm` file:

```
#if uses_RCV_PEQ
#ifdef uses_16kHz
#define RCV_PEQ_GAIN_EXP   0x000001
#define RCV_PEQ_GAIN_MANT  0x400000
#define RCV_PEQ_STAGE1_B2  0x3DE6FC
#define RCV_PEQ_STAGE1_B1  0x843207
#define RCV_PEQ_STAGE1_B0  0x3DE6FC
#define RCV_PEQ_STAGE1_A2  0x3BDF94
#define RCV_PEQ_STAGE1_A1  0x8443A2
#define RCV_PEQ_CONFIG     0x000001
#define MAX_NUM_PEQ_STAGES (1) //No OF STAGES
.VAR/DM2CIRC rcv_peq_delaybuf_dm2[2 * (MAX_NUM_PEQ_STAGES + 1)];
// Filter Coefficients
.VAR/DM1CIRC rcv_peq_coeffs[5 * MAX_NUM_PEQ_STAGES] =
RCV_PEQ_STAGE1_B2,RCV_PEQ_STAGE1_B1,RCV_PEQ_STAGE1_B0,RCV_PEQ_STAGE1_A2,RC
V_PEQ_STAGE1_A1;
//Filter Gain
.VAR rcv_peq_gain_exp = RCV_PEQ_GAIN_EXP;
.VAR rcv_peq_gain_mant = RCV_PEQ_GAIN_MANT;
.VAR rcv_peq_scale = 0x000001;

// Other Parametres
   .VAR/DM2 rcv_peq_dm2[PEQ_OBJECT_SIZE(MAX_NUM_PEQ_STAGES)] =
      &rcv_stream_map_sco_in,             // PTR_INPUT_DATA_BUFF_FIELD
      &rcv_stream_map_sco_in,             // PTR_OUTPUT_DATA_BUFF_FIELD
      MAX_NUM_PEQ_STAGES,             // MAX_STAGES_FIELD
      0,
```

```
#ifdef KAL_ARCH2
      &rcv_peq_delaybuf_dm2,           // PTR_DELAY_LINE_FIELD
      &rcv_peq_coeffs,                 // PTR_COEFS_BUFF_FIELD
#endif
      0 ...;
#endif
```

5.2.2 Adding the PEQ to the initialization function

To add PEQ to the initialization function:

```
.VAR reinitialize_table[] =
 // Function                          r7                         r8
 $frame_sync.sco_initialize,          &$sco_data.object,         0,
 $audio_proc.peq.initialize,          &rcv_peq_dm2,              0,
 0;
```

5.2.3 Adding the PEQ to the receive processing function table

The `audio_proc.peq.process` handles *in place processing* after the stream copy operation. It is after the stream copy in the receive function table:

```
.VAR rcv_funcs[] =
 // Function                               r7                      r8
    $frame_sync.distribute_streams_ind,    &rcv_process_streams,   0,
    $stream_copy,                          &rcv_pass_thru_obj,     0,
    $audio_proc.peq.process,               &rcv_peq_dm2,           0,
    $frame_sync.update_streams_ind,        &rcv_process_streams,   0,
    0;
```

5.3 Adding the audio_proc Kalimba public library

The `audio_proc` public Kalimba library provides the PEQ module.

To add this library:

1. Right-click on **Kalimba – 'one_mic_example_16k'** in the **Project Workspace** and select **Properties.**

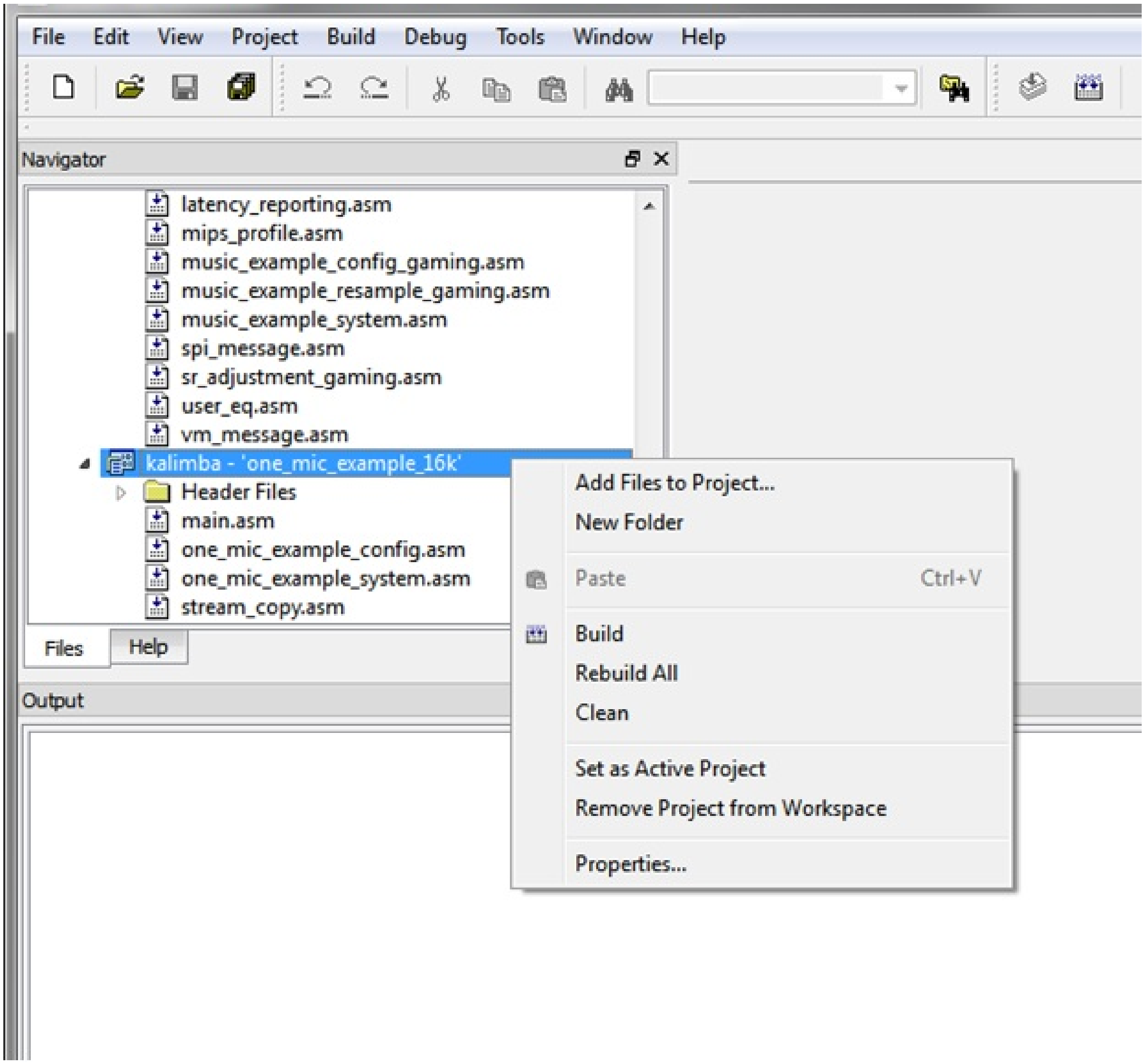

Figure 5-4 Kalimba project properties

1. Add **audio_proc** to the **Libraries** field.

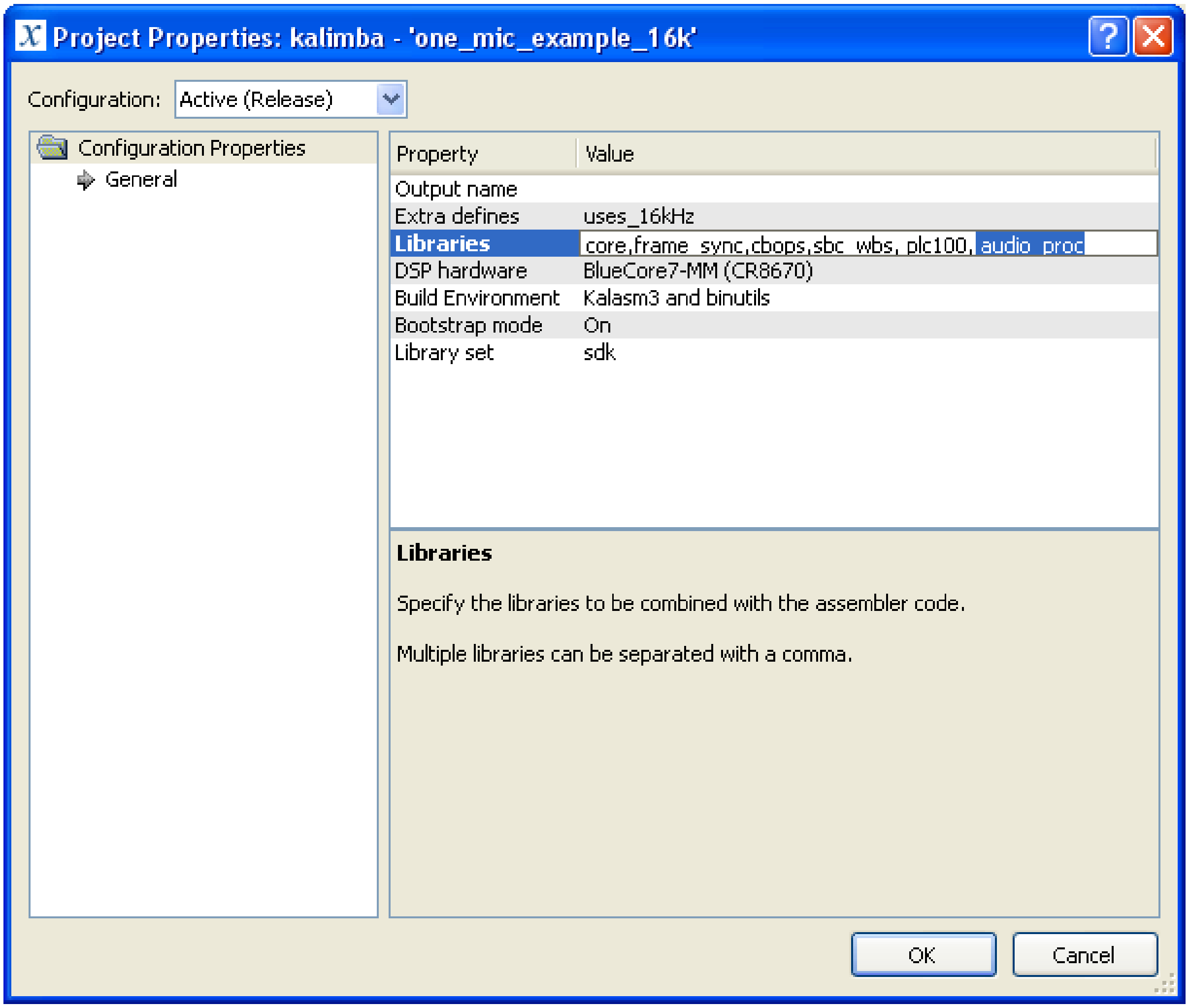

Figure 5-5 Adding the audio_proc library

6 Project variants

This section describes the 2-mic DSP project and USB Dongle Mode.

6.1 2-mic DSP project

The `two_mic_example` DSP application uses two ADCs. An operator copies microphone data from the left and right ADC ports into left and right cbuffers. The `ADC Mixer`, which is in the send function table, mixes the contents of the cbuffers together into one cbuffer.

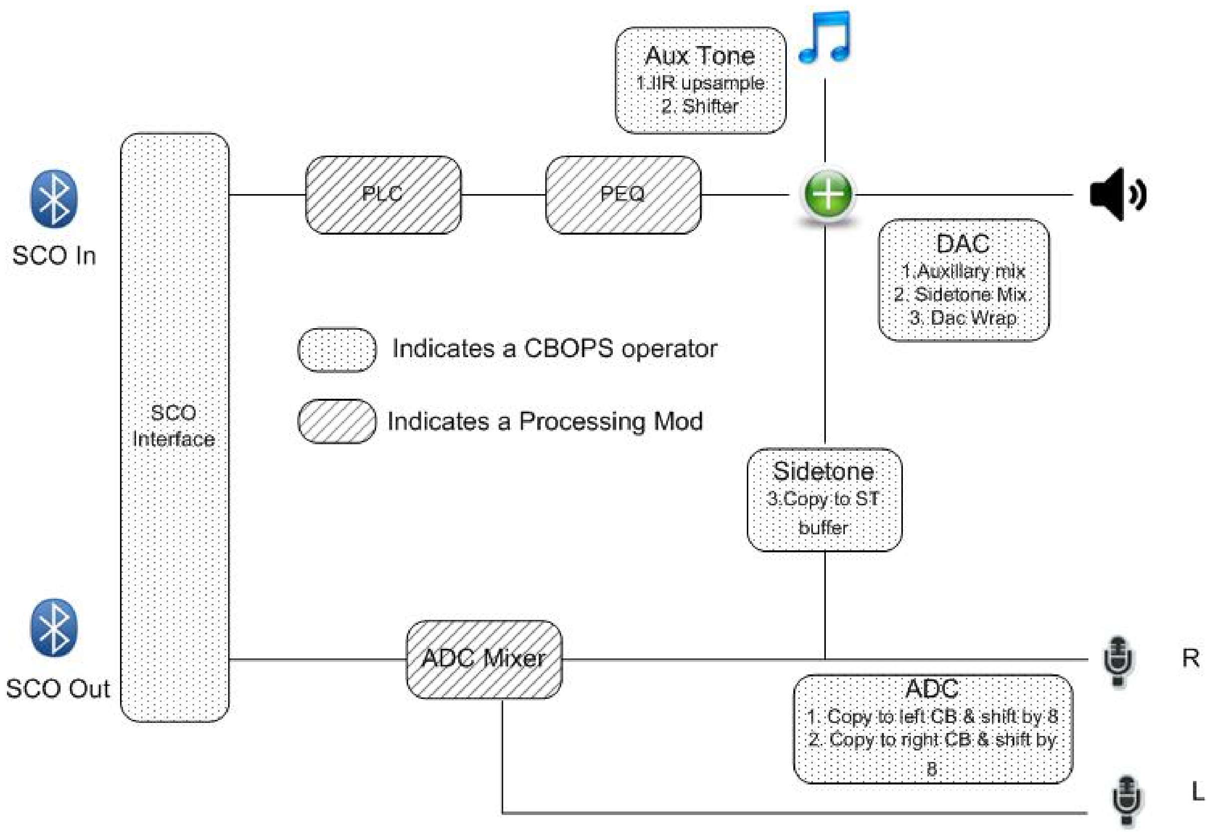

Figure 6-1 2-mic DSP application overview

6.1.1 -mic vs. 2-mic

The differences between the `two_mic_example` application and the `one_mic_example` application are:

- The ADC CBOPS module has two copy and shift operators (instead of one) for copying left and right ADC data.
- The ADC Mixer processing module replaces the simple passthrough object in the send stream. This change is reflected in:
  - The stream map
  - The stream table
  - The initialize function table
  - The processing function table

6.2 USB Dongle mode

The USB Dongle variant of the `one_mic_example` application uses the same source files as the `one_mic_example` project. To enable USB Dongle mode define static symbol `USB_DONGLE`. Defining `USB_DONGLE` replaces the DAC and ADC operators with USB operators.

The four variants of the USB Dongle are:

- `usb_dongle_8k_mono.xip`: Uses mono 8 kHz USB audio and CVSD
- `usb_dongle_16k_mono.xip`: Uses mono 16 kHz USB audio and mSBC
- `usb_dongle_48_to_16k_stereo.xip`: Uses stereo 48 kHz USB audio and mSBC
- `usb_dongle_48_to_8k_stereo.xip`: Uses stereo 48 kHz USB audio and CVSD

NOTE The right channel USB input is discarded.

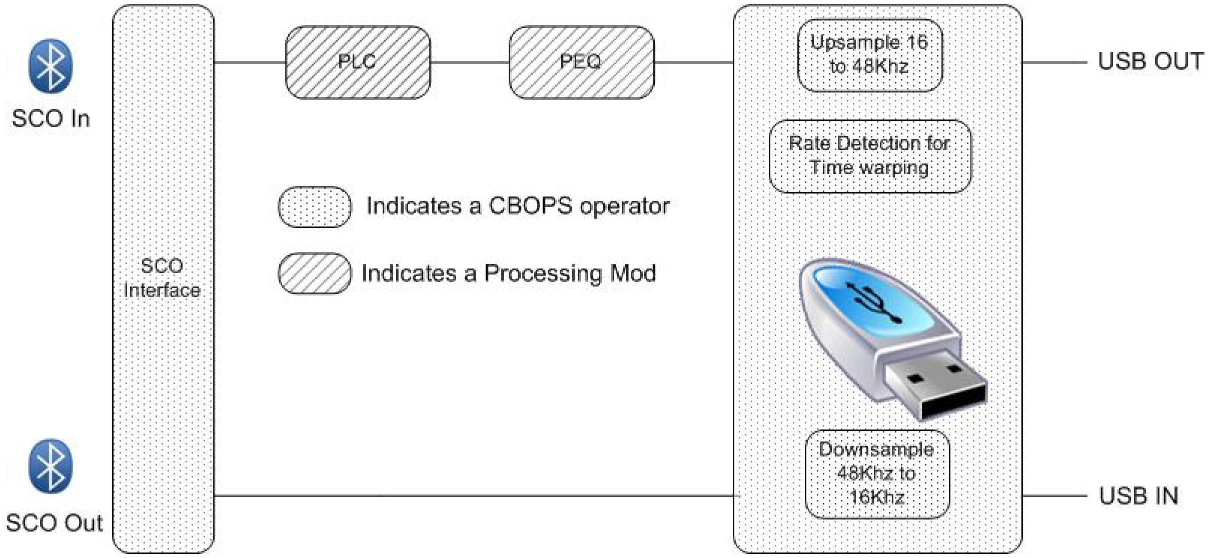

Figure 6-2 USB Dongle mode overview

6.2.1 1-mic vs. USB Dongle

The differences between the `usb_dongle` application and the `one_mic_example` application are:

- The ADC and DAC CBOPS operators are replaced by USB IN and USB OUT operators, which primarily resample data from 48 kHz to 16 kHz for USB IN and 16 kHz to 48 kHz for USB out.
- A rate detection algorithm runs in time warping operations.

7 Resource use

This section describes how the resources are used (shown for the 8675 device).

Table 7-1 1-mic example in wideband 16 K mode

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	4934	11264	32
DM1Group	DATA	3627	32768	24
DM2Group	DATA	3626	32768	24
Flash	DATA	609	-	16

Table 7-2 1-mic example in CVSD mode

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	3850	11264	32
DM1Group	DATA	2499	32768	24
DM2Group	DATA	2499	32768	24
Flash	DATA	0	-	16

Table 7-3 2-mic example in wideband 16 K mode

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	5039	11264	32
DM1Group	DATA	4132	32768	24
DM2Group	DATA	4132	32768	24
Flash	DATA	609	-	16

Table 7-4 2-mic example in CVSD mode

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.codel	0	-	32
CODEGroup	CODE	3918	11264	32

Table 7-4 2-mic example in CVSD mode (cont.)

Memory group	Memory label	Memory used	Total memory	Word width
DM1Group	DATA	2885	32768	24
DM2Group	DATA	2884	32768	24
Flash	DATA	0	-	16

Table 7-5 USB Dongle 8 K mono

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	2718	11264	32
DM1Group	DATA	1513	32768	24
DM2Group	DATA	1513	32768	24
Flash	DATA	0	-	16

Table 7-6 USB Dongle 16 K mono

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	3798	11264	32
DM1Group	DATA	2574	32768	24
DM2Group	DATA	2573	32768	24
Flash	DATA	609	-	16

Table 7-7 USB Dongle 48 K to 8 K stereo

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	2869	11264	32
DM1Group	DATA	1776	32768	24
DM2Group	DATA	1696	32768	24
Flash	DATA	0	-	16

Table 7-8 USB Dongle 48 K to 16 K stereo

Memory group	Memory label	Memory used	Total memory	Word width
CODEFLASHGroup	flash.code	0	-	32
CODEGroup	CODE	3974	11264	32
DM1Group	DATA	2738	32768	24
DM2Group	DATA	2737	32768	24
Flash	DATA	609	-	16

Document references

Document	Reference
Kalimba DSP Assembler User Guide	80-CT425-1 / CS-00212259-UG
BlueCore5-Multimedia Kalimba DSP User Guide	80-CT663-1 / CS-00101693-UG
BlueCore7/8/9 Kalimba DSP User Guide	80-CE519-1 / CS-00202067-UG
My First Kalimba DSP Application	80-CT398-1 / CS-00101420-AN
My Second Kalimba DSP Application	80-CT399-1/ CS-00114287-AN
Frame Sync Architecture	CS-00231405-AN
xIDE User Guide	80-CT405-1 / CS-00101500-UG
CBOPS Multirate	80-CT829-1 / CS-00304657-AN

Terms and definitions

Term	Definition
ADK	Audio Development Kit
ADC	Analogue to Digital Converter
API	Application Programming Interface
BlueCore	Group term for QTIL’s range of Bluetooth wireless technology ICs
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
cbops	Cbuffer Operator
Cbuffer	Circular buffer in Kalimba DSP.
Codec	Coder Decoder
cVc	Clear Voice Clarity
CVSD	Continuous Variable Slope Delta Modulation
DAC	Digital to Analogue Converter
DSP	Digital Signal Processor
HFP	HFP Hands Free Profile
MIC	Microphone
mSBC	modified SBC
MIPS	Millions of Instructions Per Second
MMU	Memory Management Unit
PEQ	Parametric EQualizer
PCL	Packet Loss Concealment
PS Key	Persistent Store Key
QTIL	Qualcomm Technologies International, Ltd.
SBC	Sub-band Coding
SCO	Synchronous Connection-Oriented
SDK	Software Development Kit
SPI	Serial Peripheral Interface
UFE	Universal Front End
USB	Universal Serial Bus
VM	Virtual Machine
WBS	Wide Band Speech

Term	Definition
XAP	Low power silicon-efficient RISC microprocessor
xIDE	QTIL’s Integrated Development Environment