AN-01: Porting Version-5 targets to Version-6 AWE Core libraries

Aug 2017, Rev 02

Introduction

To support multi-core systems, the AWE Core's APIs had to be slightly modified with the release of Audio Weaver 6. This means that Audio Weaver Designer version 6 cannot communicate with targets that have older versions of the AWE Core integrated.

This appnote gives an overview of the changes one must make to an Audio Weaver 5 target to allow it to use Version 6 libraries. It also describes a slight refactoring of our standard reference integrations that was done as part of our continuous improvement of these Board Support Packages [BSPs].

It is recommended to refer to the source code in one of our Reference Integrations and the *AWE Core Integration Guide* for full details on how to integrate the AWE Core into embedded systems.

Target Changes

Generally, there are three files to modify when moving from a Version 5 library to Version 6. Please refer to the appropriate file in one of our Version 6 BSPs.

Platform.c

Refer to `Platform.c` in the reference BSP (Appendix A)

- Remove `TargetInfo` data structure initialization (more on this below)
- Add global `AWEInstance` variable
- Add `IOPinDescriptor` for input and output pins.
- As shown in our reference `Platform.c` add the following methods:
  - `awe_pltGetCore`
  - `awe_pltGetCores`
  - `awe_pltDestroyAll`
  - `AWEInstanceInit` (change the 8-character string to match your identifier)
- In `awe_pltInit` add a call to `AWEInstanceInit`
- In `awe_pltTick` add the instance variable to the `awe_fwTuningTick` call.

AudioDriver.c

Refer to `AudioDriver.c` in the reference BSP. (Appendix B)

- In the `AWEProcessing` method
  - `awe_fwGetInputChannelPtr` – new arguments
  - `awe_fwGetOutputChannelPtr` – new arguments
  - `awe_fwAudioDMAComplete` – new arguments

- In `AudioWeaverPumpIRQ` handlers
    - `awe_fwPump` – new AWE Instance argument

TargetInfo.h

Refer to `TargetInfo.h` in the reference BSP. (Appendix C)

In AWE-5 BSPs, **`Platform.c`** had a **`TargetInfo`** data structure initialized in the beginning of the file. This data structure has now been internalized into the AWE Core library and #defines are now used in TargetInfo.h to perform the same initialization function.

Set the following defines as appropriate for your hardware:

```
// Version Information
#define VER_DAY       01
#define VER_MONTH 07
#define VER_YEAR      17

#define CORE_ID                         0
#define CORE_SPEED            168e6f
#define SAMPLE_SPEED          168e6f
#define HAS_FLOAT_SUPPORT     1
#define HAS_FLASH_FILESYSTEM  1
#define NO_HW_INPUT_PINS      1
#define NO_HW_OUTPUT_PINS     1
#define IS_SMP                0
#define NO_THREADS_SUPPORTED  2
#define FIXED_SAMPLE_RATE     48000.0f
#define IS_COMPLEX            0
#define SAMPLE_SIZE_IN_BYTES  4
```

Reference Code Re-factoring

For ARM-based BSPs GPIO-module support was removed from Platform.c and placed in its own file `GPIO.c` file. USB Descriptors were removed from `usbd_audio.c` and placed into file `USBDescriptors.c`. And a new `ControlDriver.c` component was added to demonstrate how to implement a control interface, as documented in section 5 of the AWE Core Integration Guide.

Version 5 Files	Version 6 Files
Main.c	Main.c
BoardSetup.c	BoardSetup.c
Platform.c	Platform.c
AWEFlash.c	AWEFlash.c
AudioDriver.c	AudioDriver.c
TuningDriver.c	TuningDriver.c
	ControlDriver.c
	USBDescriptors.c
	GPIO.c

Figure 1: BSP source files included in AWE-5 vs. AWE-6

Appendix A. Platform.c

```
/*************************************************************************
*     Platform.c
**************************************************************************
*
*     Description:  AWE Platform Interface
*
*     Copyright:    DSP Concepts, Inc. (c) 2007 - 2016
*                   1800 Wyatt Drive, Suite 14
*                   Sunnyvale, CA 95054
*
*************************************************************************/
#include "Errors.h"
#include "BoardSetup.h"
#include "Platform.h"
#include "TuningHandler.h"
#include "TargetInfo.h"
#include "ControlDriver.h"


/* ------------------------------------------------------------------------
** Global AWE Instance
** ---------------------------------------------------------------------- */
AWEInstance g_AWEInstance;

/* ------------------------------------------------------------------------
** Memory heaps
** ---------------------------------------------------------------------- */
AWE_FW_SLOW_ANY_CONST UINT32 g_master_heap_size = MASTER_HEAP_SIZE;
AWE_FW_SLOW_ANY_CONST UINT32 g_slow_heap_size = SLOW_HEAP_SIZE;
AWE_FW_SLOW_ANY_CONST UINT32 g_fastb_heap_size = FASTB_HEAP_SIZE;

section("awe_heap_fast")
UINT32 g_master_heap[MASTER_HEAP_SIZE];

#pragma section("awe_heap_slow", NO_INIT)
UINT32 g_slow_heap[SLOW_HEAP_SIZE];

section("awe_heap_fastb")
UINT32 g_fastb_heap[FASTB_HEAP_SIZE];

/** Audio Running Flag. */
AWE_FW_SLOW_ANY_DATA static INT32 s_AudioRunning = 0;

extern volatile INT32 g_nPumpCount;

volatile BOOL g_bReboot = FALSE;


/* ------------------------------------------------------------------------
** Module table
** ---------------------------------------------------------------------- */

/* Array of pointers to module descriptors. This is initialized at compile time.
Each item is the address of a module descriptor that we need linked in. The
linker magic is such that only those modules referenced here will be in the
final program. */
AWE_MOD_FAST_ANY_DATA const ModClassModule *g_module_descriptor_table[] =
{
    //------------- The suitably cast pointers to the module descriptors go here.
    LISTOFCLASSOBJECTS
```

```
};

AWE_MOD_SLOW_DM_DATA UINT32 g_module_descriptor_table_size = sizeof(g_module_descriptor_table) /
sizeof(g_module_descriptor_table[0]);


/** The only input pin for this core. */
static IOPinDescriptor s_InputPin[1];

/** The only output pin for this core. */
static IOPinDescriptor s_OutputPin[1];


// Control Driver input demonstration variables
AWE_MOD_SLOW_ANY_DATA volatile int volume = 0;
AWE_MOD_SLOW_ANY_DATA volatile int balance = 0;

//-----------------------------------------------------------------------------
// METHOD:  awe_pltGetCore
// PURPOSE: Return core descriptor
//-----------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
CoreDescriptor * awe_pltGetCore(void *pOwner, UINT32 coreID) { return NULL; }

//-----------------------------------------------------------------------------
// METHOD:  awe_pltGetCores
// PURPOSE: Report number of cores in use on this target
//-----------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
int awe_pltGetCores() { return 1; }

//-----------------------------------------------------------------------------
// METHOD:  awe_pltDestroyAll
// PURPOSE: Destroy all core instances
//-----------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void awe_pltDestroyAll() { awe_fwDestroy(&g_AWEInstance); }


//-----------------------------------------------------------------------------
// METHOD:  AWEInstanceInit
// PURPOSE: Initialize AWE Instance with target details
//-----------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void AWEInstanceInit()
{
    // Third empty string word is to make sure awe_SetPackedName() terminate after first two string words.
Otherwise the
    // loop rolls for 8 times and will lead to memory corruption.
    UINT32 sBoardName[3] = {MAKE_PACKED_STRING('4', '8', '9', 'E'),
                            MAKE_PACKED_STRING('Z', 'K', 'I', 'T'),
                            MAKE_PACKED_STRING('\0', '\0', '\0', '\0')};

    UINT32 sInputPinName[2] = {MAKE_PACKED_STRING('I', 'n', 'p', 'u'), MAKE_PACKED_STRING('t', '\0', '\0',
'\0')};

    UINT32 sOutputPinName[2] = {MAKE_PACKED_STRING('O', 'u', 't', 'p'), MAKE_PACKED_STRING('u', 't', '\0',
```

```
'\0')};

    UINT32 nInputWireInfo = INFO1_PROPS(INPUT_CHANNEL_COUNT, AWE_FRAME_SIZE, IS_COMPLEX, SAMPLE_SIZE_IN_BYTES);

    UINT32 nOutputWireInfo = INFO1_PROPS(OUTUT_CHANNEL_COUNT, AWE_FRAME_SIZE, IS_COMPLEX,
SAMPLE_SIZE_IN_BYTES);

    memset(&g_AWEInstance, 0, sizeof(AWEInstance) );

    // Point to the start and stop functions.
    g_AWEInstance.m_pAwe_pltAudioStart = awe_pltAudioStart;
    g_AWEInstance.m_pAwe_pltAudioStop = awe_pltAudioStop;

    // Point to our private pins.
    g_AWEInstance.m_pInterleavedInputPin = s_InputPin;
    g_AWEInstance.m_pInterleavedOutputPin = s_OutputPin;

    // Point to the global module table.
    g_AWEInstance.m_module_descriptor_table_size = g_module_descriptor_table_size;
    g_AWEInstance.m_pModule_descriptor_table = g_module_descriptor_table;

    // This will be core ID 100.
    g_AWEInstance.m_coreID = CORE_ID;

    awe_fwInitTargetInfo(&g_AWEInstance,
                         CORE_ID,
                         CORE_SPEED,
                         SAMPLE_SPEED,
                         (const char *)sBoardName,
                         PROCESSOR_TYPE_SHARC,
                         HAS_FLOAT_SUPPORT,
                         HAS_FLASH_FILESYSTEM,
                         NO_INPUT_PINS,
                         NO_OUTPUT_PINS,
                         IS_SMP,
                         NO_THREADS_SUPPORTED,
                         FIXED_SAMPLE_RATE,
                         INPUT_CHANNEL_COUNT,
                         OUTPUT_CHANNEL_COUNT,
                         VER_DAY, VER_MONTH, VER_YEAR
                         );

    awe_fwInit_io_pins(&g_AWEInstance, 1);

    s_InputPin[0].sampleRate = FIXED_SAMPLE_RATE;
    s_InputPin[0].wireInfo1 = nInputWireInfo;
    s_InputPin[0].wireInfo3 |= CLOCK_MASTER_BIT;
    s_InputPin[0].m_pinName[0] = sInputPinName[0];
    s_InputPin[0].m_pinName[1] = sInputPinName[1];

    s_OutputPin[0].sampleRate = FIXED_SAMPLE_RATE;
    s_OutputPin[0].wireInfo1 = nOutputWireInfo;
    s_OutputPin[0].m_pinName[0]= sOutputPinName[0];
    s_OutputPin[0].m_pinName[1]= sOutputPinName[1];

    // Allocate the heaps.
    g_AWEInstance.m_master_heap_size = MASTER_HEAP_SIZE;
    g_AWEInstance.m_slow_heap_size = SLOW_HEAP_SIZE;
    g_AWEInstance.m_fastb_heap_size = FASTB_HEAP_SIZE;

    g_AWEInstance.m_master_heap = g_master_heap;
    g_AWEInstance.m_slow_heap = g_slow_heap;
```

```
    g_AWEInstance.m_fastb_heap = g_fastb_heap;

}   // End AWEInstanceInit


///-------------------------------------------------------------------------
/// @name  void awe_pltInit(void)
/// @brief Initialize the AWE Platform
///-------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void awe_pltInit(void)
{
    // Initialize the target info
    AWEInstanceInit();

    // Setup processor clocks, signal routing, timers, etc.
    CoreInit();

    // Setup board peripherals (CODECs, external memory, etc.)
    BoardInit();

    // Setup audio DMA, interrupt priorities, etc.
    AudioInit();

    // Initialize the communications handler
    awe_fwTuningInit(s_PacketBuffer, MAX_COMMAND_BUFFER_LEN);

    // Initialize the Flash File System
    awe_pltInitFlashFileSystem();

    // This needs to be explicitly initialized
    g_bReboot = FALSE;

}   // End awe_pltInit


///-------------------------------------------------------------------------
/// @name  void awe_pltTick(void)
/// @brief Idle loop Platform Tick Processing
///-------------------------------------------------------------------------
AWE_OPTIMIZE_FOR_SPEED
AWE_FW_SLOW_CODE
void awe_pltTick(void)
{
    BOOL bReplyReady;
    static INT32 LEDState = 0;
    static INT32 nCount = 0;

    // Indicate that this idle loop call is getting CPU attention
    g_nPumpCount = 0;

    bReplyReady = awe_fwTuningTick(&g_AWEInstance);

    if (bReplyReady == REPLY_READY)
    {
        UART0SendReply();
    }
     Process any local controls
    ProcessControlIO();
}
```

Appendix B. AudioDriver.c

```
/*****************************************************************************
*    AudioDriver.c
*****************************************************************************
*
*     Description:  AudioWeaver Driver for AD1939 Audio Codec
*
*     Copyright:    DSP Concepts, Inc. (c) 2007 - 2016
*                   1800 Wyatt Drive, Suite 14
*                   Sunnyvale, CA 95054
*
*****************************************************************************/

#include "ProxyIDs.h"
#include "Errors.h"
#include "Framework.h"
#include "VectorLib.h"
#include "Platform.h"
#include "TargetInfo.h"
#include "AD1939_Audio.h"

#include <def21489.h>
#include <cdef21489.h>

#include <signal.h>
#include <sysreg.h>

#ifdef __CCESVERSION__
#include <services/int/adi_int.h>  /* Interrupt Handler API header. */
#endif

#pragma default_section(CODE, "awe_mod_slowcode")
#pragma default_section(ALLDATA, "awe_mod_slowanydata")
#pragma default_section(SWITCH, "awe_mod_slowanydata")

/** Number of audio samples per tick. */
#define SAMPS_PER_TICK      (32u)

/** Number of channels per bank. */
#define CHANS_PER_BANK      (2u)

/** Number of input and output channels that the HW supplies */
#define PLTINCOUNT 4
#define PLTOUTCOUNT 8

/** Convert a pointer to DMA address. */
#define DMA_ADDR(ptr)       ( (UINT32)(ptr) & 0x7FFFFu )

#define MAX_PUMP_COUNT 100

volatile BOOL g_bAudioPump1Active = FALSE;
volatile BOOL g_bAudioPump2Active = FALSE;
volatile UINT32 g_nPumpCount = 0;

#ifdef INCLUDE_DECODERS
extern void RenderDecode(INT32 sig_int);
#endif

/**
 * @brief Enumeration of audio buffers.
```

```
 */
 AWE_FW_SLOW_ANY_DATA
typedef enum
{
    /* Buffer identifiers. */
    ABI_ADC_Bank0_Ping = 0,   /**< ADC Bank 0 Ping Buffer (RX) */
    ABI_ADC_Bank0_Pong = 1,   /**< ADC Bank 0 Pong Buffer (RX) */
    ABI_ADC_Bank1_Ping = 2,   /**< ADC Bank 1 Ping Buffer (RX) */
    ABI_ADC_Bank1_Pong = 3,   /**< ADC Bank 1 Pong Buffer (RX) */
    ABI_DAC_Bank0_Ping = 4,   /**< DAC Bank 0 Ping Buffer (TX) */
    ABI_DAC_Bank0_Pong = 5,   /**< DAC Bank 0 Pong Buffer (TX) */
    ABI_DAC_Bank1_Ping = 6,   /**< DAC Bank 1 Ping Buffer (TX) */
    ABI_DAC_Bank1_Pong = 7,   /**< DAC Bank 1 Pong Buffer (TX) */
    ABI_DAC_Bank2_Ping = 8,   /**< DAC Bank 2 Ping Buffer (TX) */
    ABI_DAC_Bank2_Pong = 9,   /**< DAC Bank 2 Pong Buffer (TX) */
    ABI_DAC_Bank3_Ping = 10,      /**< DAC Bank 3 Ping Buffer (TX) */
    ABI_DAC_Bank3_Pong = 11,      /**< DAC Bank 3 Pong Buffer (TX) */
    ABI_COUNT                 /**< Number of audio buffers */
} tAudioBufferId;

/**
 * @brief DMA Transfer Control Block structure.
 */
 AWE_FW_SLOW_ANY_DATA
typedef struct
{
    UINT32 m_CPSPx;   /**< Chain pointer for next TCB    */
    UINT32 m_CSPx;    /**< Length of source buffer       */
    UINT32 m_IMSPx;   /**< Source buffer step size       */
    UINT32 m_IISPx;   /**< Start address for data buffer */
} tDMATCB;


/* ------------------------------------------------------------------------
 *   Local Variables
 * ------------------------------------------------------------------------ */

/* Audio Buffers. */
AWE_FAST_DMA_DATA static INT32 s_AudioBuffer[ABI_COUNT][SAMPS_PER_TICK * CHANS_PER_BANK];

/** Audio DMA TCBs. */
AWE_FAST_DMA_DATA static tDMATCB s_AudioDMA[ABI_COUNT];

AWE_FW_SLOW_ANY_DATA static INT32 previous_fir_hw_state = 0;

extern UINT32 s_PacketBuffer[MAX_COMMAND_BUFFER_LEN];

/* ------------------------------------------------------------------------
 *   Local Functions
 * ------------------------------------------------------------------------ */

/**
 * @brief Initialize a TCB structure for audio transfer.
 * @param[in] p_tcb         tDMATCB structure to initialize.
 * @param[in] p_nxt         Next tDMATCB in the chain.
 * @param[in] p_buf         Memory Buffer.
 */

AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
static void InitTCB(
```

```
        tDMATCB *p_tcb,
        tDMATCB *p_nxt,
        void *p_buf)
{
    p_tcb->m_CPSPx = DMA_ADDR(((UINT32)p_nxt + 3u)) | 0x80000u;
    p_tcb->m_CSPx  = SAMPS_PER_TICK * CHANS_PER_BANK;
    p_tcb->m_IMSPx = 1;
    p_tcb->m_IISPx = DMA_ADDR(p_buf);
}


/* ---------------------------------------------------------------------------
 *   Functions
 * ------------------------------------------------------------------------- */


/**
 * @brief awe_pltAudioStart
 */
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void AudioStart(void)
{
    /* Start the audio SPORTs. */

    /* Configure SPORT0 for input from ADC. */
    *pSPCTL0 = (OPMODE | SLEN32 | L_FIRST | SPEN_A | SCHEN_A | SDEN_A | SPEN_B | SCHEN_B | SDEN_B);
    *pCPSP0A = s_AudioDMA[ABI_ADC_Bank0_Pong].m_CPSPx;
    *pCPSP0B = s_AudioDMA[ABI_ADC_Bank1_Pong].m_CPSPx;

    /* Configure SPORT1 for output to DAC. */
    *pSPCTL1 = (SPTRAN | OPMODE | SLEN32 | L_FIRST | SPEN_A | SCHEN_A | SDEN_A | SPEN_B | SCHEN_B | SDEN_B);
    *pCPSP1A = s_AudioDMA[ABI_DAC_Bank0_Pong].m_CPSPx;
    *pCPSP1B = s_AudioDMA[ABI_DAC_Bank1_Pong].m_CPSPx;

    /* Configure SPORT2 for output to DAC. */
    *pSPCTL2 = (SPTRAN | OPMODE | SLEN32 | L_FIRST | SPEN_A | SCHEN_A | SDEN_A | SPEN_B | SCHEN_B | SDEN_B);
    *pCPSP2A = s_AudioDMA[ABI_DAC_Bank2_Pong].m_CPSPx;
    *pCPSP2B = s_AudioDMA[ABI_DAC_Bank3_Pong].m_CPSPx;

    /* Reenable HW accelerators */
    if (previous_fir_hw_state & (FIR_DMAEN))
    {
        *pFIRCTL1 |= FIR_DMAEN;
    }

    previous_fir_hw_state = 0;

    /* Start Audio. */
    sysreg_bit_set(sysreg_LIRPTL, SP0IMSK);


}   // End AudioStart


/**
 * @brief awe_pltAudioStop
 */
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void AudioStop(void)
```

```
{
    /* Clear Hardware accelerators */
    if ( (s_PacketBuffer[0] & 0xFFFF) == PFID_Destroy)
    {
        *pFIRCTL1 &= ~(FIR_EN|FIR_DMAEN);

        previous_fir_hw_state = 0;
    }
    else
    {
        previous_fir_hw_state = *pFIRCTL1;

        *pFIRCTL1 &= ~(FIR_DMAEN);
    }

    /* Reset the audio SPORTs. */
    *pSPMCTL0 = 0;    *pSPMCTL1 = 0;    *pSPMCTL2 = 0;
    /* Reset SPORT 0. */
    *pSPCTL0  = 0;    *pDIV0    = 0;    *pCPSP0A  = 0;    *pCPSP0B  = 0;
    /* Reset SPORT 1. */
    *pSPCTL1  = 0;    *pDIV1    = 0;    *pCPSP1A  = 0;    *pCPSP1B  = 0;
    /* Reset SPORT 2. */
    *pSPCTL2  = 0;    *pDIV2    = 0;    *pCPSP2A  = 0;    *pCPSP2B  = 0;

    *pSPCTLN0 = 0;    *pSPCTLN1 = 0;    *pSPCTLN2 = 0;

    /* Disable and clear any pending interrupts. */
    sysreg_bit_clr(sysreg_LIRPTL, SP0IMSK);
    sysreg_bit_clr(sysreg_LIRPTL, SP0I);

}   // End AudioStop


/**
 * @brief Audio Receive ISR handler.
 *
 * This ISR handler is called when the TX/RX DMA operations have
 * processed an entire chunk of data.
 *
 * @param[in] sig_int       Unused.
 */
 AWE_OPTIMIZE_FOR_SPEED
AWE_FW_FAST_CODE
#ifdef __CCESVERSION__
static void AudioReceive_ISR(UINT32 sig_int, void *handlerarg)  //CCES
#else
static void AudioReceive_ISR(INT32 sig_int)     //VDSP
#endif
{
    UINT32 fwInCount, fwOutCount;
    INT32 layoutMask=0;

    INT32 * restrict adc_src[2];
    INT32 * restrict dac_dst[4];
    UINT32 used_chans, chan;
    INT32 *pinPtr;
    INT32 pinStride;

    const UINT32 nPinNdx = 0;

    /* Unused parameter. */
```

```
(void)sig_int;

/* ------------------------------------------------------------------------
** The audio DMA is double buffered.  Identify which DMA buffer should
** be used.
** ------------------------------------------------------------------------ */

/* Identify the buffers. */
/* Pick the adc buffer. */
if (*pIISP0A < DMA_ADDR(s_AudioBuffer[ABI_ADC_Bank0_Pong]))
{
    adc_src[0] = s_AudioBuffer[ABI_ADC_Bank0_Pong];
    adc_src[1] = s_AudioBuffer[ABI_ADC_Bank1_Pong];
}
else
{
    adc_src[0] = s_AudioBuffer[ABI_ADC_Bank0_Ping];
    adc_src[1] = s_AudioBuffer[ABI_ADC_Bank1_Ping];
}

if (*pIISP1A < DMA_ADDR(s_AudioBuffer[ABI_DAC_Bank0_Pong]))
{
    dac_dst[0] = s_AudioBuffer[ABI_DAC_Bank0_Pong];
    dac_dst[1] = s_AudioBuffer[ABI_DAC_Bank1_Pong];
    dac_dst[2] = s_AudioBuffer[ABI_DAC_Bank2_Pong];
    dac_dst[3] = s_AudioBuffer[ABI_DAC_Bank3_Pong];
}
else
{
    dac_dst[0] = s_AudioBuffer[ABI_DAC_Bank0_Ping];
    dac_dst[1] = s_AudioBuffer[ABI_DAC_Bank1_Ping];
    dac_dst[2] = s_AudioBuffer[ABI_DAC_Bank2_Ping];
    dac_dst[3] = s_AudioBuffer[ABI_DAC_Bank3_Ping];
}

/* ------------------------------------------------------------------------
** Determine the number of input and output channels that the current
** audio layout has
** ------------------------------------------------------------------------ */

awe_fwGetChannelCount(&g_AWEInstance, &fwInCount, &fwOutCount);

// If nothing wired up then just copy the input audio to the output
if ((fwInCount == 0) && (fwOutCount == 0))
{
    /* Fill the output with zeros (Mute). */
    awe_vecFill((float *)dac_dst[0], 1, 0, SAMPS_PER_TICK*CHANS_PER_BANK);
    awe_vecFill((float *)dac_dst[1], 1, 0, SAMPS_PER_TICK*CHANS_PER_BANK);
    awe_vecFill((float *)dac_dst[2], 1, 0, SAMPS_PER_TICK*CHANS_PER_BANK);
    awe_vecFill((float *)dac_dst[3], 1, 0, SAMPS_PER_TICK*CHANS_PER_BANK);
}
else
{
    /* ------------------------------------------------------------------------
    ** Process the ADC input data
    ** ------------------------------------------------------------------------ */

    // Process the input channels

    /* Get the input pin channels actually used. */
    used_chans = (fwInCount < PLTINCOUNT) ? fwInCount : PLTINCOUNT;
```

```
for (chan = 0; chan < used_chans; chan++)
{
    /* Determine where the Framework wants the input data written */
    pinPtr=awe_fwGetInputChannelPtr(&g_AWEInstance, nPinNdx, chan, &pinStride);

    if(chan < 2)
    {
        awe_vecCopyFract32( (fract32 *)(adc_src[0] + chan), CHANS_PER_BANK, (fract32 *) pinPtr,
                            pinStride, SAMPS_PER_TICK);
    }
    else
    {
        awe_vecCopyFract32( (fract32 *)(adc_src[1] + chan-2), CHANS_PER_BANK, (fract32 *) pinPtr,
                            pinStride, SAMPS_PER_TICK);
    }
}

// Zero any unused layout inputs
for ( ; chan < fwInCount; chan++)
{
    /* Determine where the Framework wants the input data written */
    pinPtr=awe_fwGetInputChannelPtr(&g_AWEInstance, nPinNdx, chan, &pinStride);
    awe_vecFill((float *) pinPtr, pinStride, 0, SAMPS_PER_TICK);
}

/* ------------------------------------------------------------------------
** Process the DAC output data
** ------------------------------------------------------------------------ */

awe_fwGetChannelCount(&g_AWEInstance, &fwInCount, &fwOutCount);

// Process the two channels for the first stereo output DAC
for (chan=0; chan<2; chan++)
{
    if (fwOutCount > chan)
    {
        pinPtr=awe_fwGetOutputChannelPtr(&g_AWEInstance, nPinNdx, chan, &pinStride);
        awe_vecCopyFract32( (fract32 *) pinPtr, pinStride, ((fract32 *) dac_dst[0]) + chan,
                            CHANS_PER_BANK, SAMPS_PER_TICK);
    }
    else
    {
        awe_vecFill(((float *) dac_dst[0]) + chan, CHANS_PER_BANK, 0, SAMPS_PER_TICK);
    }
}


// Process the two channels for the second stereo output DAC
for (chan=0; chan<2; chan++)
{
    if (fwOutCount > (chan+2))
    {
        pinPtr=awe_fwGetOutputChannelPtr(&g_AWEInstance, nPinNdx, chan+2, &pinStride);
        awe_vecCopyFract32( (fract32 *) pinPtr, pinStride, ((fract32 *) dac_dst[1]) + chan,
                            CHANS_PER_BANK, SAMPS_PER_TICK);
    }
    else
    {
        awe_vecFill(((float *) dac_dst[1]) + chan, CHANS_PER_BANK, 0, SAMPS_PER_TICK);
    }
```

```
        }

        // Process the two channels for the third stereo output DAC
        for (chan=0; chan<2; chan++)
        {
            if (fwOutCount > (chan+4))
            {
                pinPtr=awe_fwGetOutputChannelPtr(&g_AWEInstance, nPinNdx, chan+4, &pinStride);
                awe_vecCopyFract32( (fract32 *)pinPtr, pinStride, ((fract32 *) dac_dst[2]) + chan,
                                    CHANS_PER_BANK, SAMPS_PER_TICK);
            }
            else
            {
                awe_vecFill(((float *) dac_dst[2]) + chan, CHANS_PER_BANK, 0, SAMPS_PER_TICK);
            }
        }

        // Process the two channels for the fourth stereo output DAC
        for (chan=0; chan<2; chan++)
        {
            if (fwOutCount > (chan+6))
            {
                pinPtr=awe_fwGetOutputChannelPtr(&g_AWEInstance, nPinNdx, chan+6, &pinStride);
                awe_vecCopyFract32( (fract32 *)pinPtr, pinStride, ((fract32 *) dac_dst[3]) + chan,
                                    CHANS_PER_BANK, SAMPS_PER_TICK);
            }
            else
            {
                awe_vecFill(((float *) dac_dst[3]) + chan, CHANS_PER_BANK, 0, SAMPS_PER_TICK);
            }
        }

        /* ------------------------------------------------------------------------
        ** Determine if we have enough samples to raise the lower priority
        ** audio interrupt.
        ** ------------------------------------------------------------------- */

        layoutMask = awe_fwAudioDMAComplete(&g_AWEInstance, nPinNdx, SAMPS_PER_TICK);

        if (layoutMask & 0x01)
        {
        if (!g_bAudioPump1Active)
        {
#ifdef __CCESVERSION__
            sysreg_bit_set_nop(sysreg_IRPTL, SFT0I);
#else
            raise(SIG_USR0);
#endif
        }
        }

        if (layoutMask & 0x02)
        {
        if (!g_bAudioPump2Active)
        {
#ifdef __CCESVERSION__
            sysreg_bit_set_nop(sysreg_IRPTL, SFT1I);
#else
            raise(SIG_USR1);
#endif
        }
```

```
        }
    }

}

/**
 * @brief Render Audio
 *
 * This handler is called from the Audio DMA handler
 * Process the audio.
 */
AWE_OPTIMIZE_FOR_SPEED
AWE_FW_FAST_CODE
#ifdef __CCESVERSION__
static void RenderAudio1(UINT32 sig_int, void *handlerarg)
#else
static void RenderAudio1(INT32 sig_int)
#endif
{
    g_bAudioPump1Active = TRUE;

    // If IDLE loop did not get some CPU time then skip the pump
    // g_nPumpCount gets reset in the idle loop method awe_pltTick
    if (g_nPumpCount < MAX_PUMP_COUNT)
    {
        awe_fwPump(&g_AWEInstance, 0);
        g_nPumpCount++;
    }

    g_bAudioPump1Active = FALSE;

} // End RenderAudio

AWE_OPTIMIZE_FOR_SPEED
AWE_FW_FAST_CODE
#ifdef __CCESVERSION__
static void RenderAudio2(UINT32 sig_int, void *handlerarg)
#else
static void RenderAudio2(INT32 sig_int)
#endif
{
    g_bAudioPump2Active = TRUE;

    // Fire the layout at user interrupt level
    awe_fwPump(&g_AWEInstance, 1);

    g_bAudioPump2Active = FALSE;

} // End RenderAudio

/**
 * @brief Audio_Init
 */
AWE_OPTIMIZE_FOR_SPACE
AWE_FW_SLOW_CODE
void AudioInit()
{
    /* Reset the audio SPORTs. */
    *pSPMCTL0 = 0;    *pSPMCTL1 = 0;    *pSPMCTL2 = 0;

    /* Reset SPORT 0. */
```

```
    *pSPCTL0  = 0;    *pDIV0    = 0;    *pCPSP0A  = 0;    *pCPSP0B  = 0;

    /* Reset SPORT 1. */
    *pSPCTL1  = 0;    *pDIV1    = 0;    *pCPSP1A  = 0;    *pCPSP1B  = 0;

    /* Reset SPORT 2. */
    *pSPCTL2  = 0;    *pDIV2    = 0;    *pCPSP2A  = 0;    *pCPSP2B  = 0;

    *pSPCTLN0 = 0;    *pSPCTLN1 = 0;    *pSPCTLN2 = 0;

    /* Configure the ADC-Bank0 RX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_ADC_Bank0_Ping], &s_AudioDMA[ABI_ADC_Bank0_Pong], s_AudioBuffer[ABI_ADC_Bank0_Ping]);
    InitTCB(&s_AudioDMA[ABI_ADC_Bank0_Pong], &s_AudioDMA[ABI_ADC_Bank0_Ping], s_AudioBuffer[ABI_ADC_Bank0_Pong]);

    /* Configure the ADC-Bank1 RX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_ADC_Bank1_Ping], &s_AudioDMA[ABI_ADC_Bank1_Pong], s_AudioBuffer[ABI_ADC_Bank1_Ping]);
    InitTCB(&s_AudioDMA[ABI_ADC_Bank1_Pong], &s_AudioDMA[ABI_ADC_Bank1_Ping], s_AudioBuffer[ABI_ADC_Bank1_Pong]);

    /* Configure the DAC-Bank0 TX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_DAC_Bank0_Ping], &s_AudioDMA[ABI_DAC_Bank0_Pong], s_AudioBuffer[ABI_DAC_Bank0_Ping]);
    InitTCB(&s_AudioDMA[ABI_DAC_Bank0_Pong], &s_AudioDMA[ABI_DAC_Bank0_Ping], s_AudioBuffer[ABI_DAC_Bank0_Pong]);

    /* Configure the DAC-Bank1 TX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_DAC_Bank1_Ping], &s_AudioDMA[ABI_DAC_Bank1_Pong], s_AudioBuffer[ABI_DAC_Bank1_Ping]);
    InitTCB(&s_AudioDMA[ABI_DAC_Bank1_Pong], &s_AudioDMA[ABI_DAC_Bank1_Ping], s_AudioBuffer[ABI_DAC_Bank1_Pong]);

    /* Configure the DAC-Bank2 TX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_DAC_Bank2_Ping], &s_AudioDMA[ABI_DAC_Bank2_Pong], s_AudioBuffer[ABI_DAC_Bank2_Ping]);
    InitTCB(&s_AudioDMA[ABI_DAC_Bank2_Pong], &s_AudioDMA[ABI_DAC_Bank2_Ping], s_AudioBuffer[ABI_DAC_Bank2_Pong]);

    /* Configure the DAC-Bank3 TX ping/pong buffers. */
    InitTCB(&s_AudioDMA[ABI_DAC_Bank3_Ping], &s_AudioDMA[ABI_DAC_Bank3_Pong], s_AudioBuffer[ABI_DAC_Bank3_Ping]);
    InitTCB(&s_AudioDMA[ABI_DAC_Bank3_Pong], &s_AudioDMA[ABI_DAC_Bank3_Ping], s_AudioBuffer[ABI_DAC_Bank3_Pong]);

    /* Enable nested interrupts */
    sysreg_bit_set_nop(sysreg_MODE1, NESTM);

#ifdef __CCESVERSION__
    /* Hook SPORT0 Receive ISR. */
    adi_int_InstallHandler(ADI_CID_SP0I, (ADI_INT_HANDLER_PTR)AudioReceive_ISR, 0, true);

    /* Hook user ISR. */
    adi_int_InstallHandler(ADI_CID_SFT0I, (ADI_INT_HANDLER_PTR)RenderAudio1, 0, true);
    adi_int_InstallHandler(ADI_CID_SFT1I, (ADI_INT_HANDLER_PTR)RenderAudio2, 0, true);
#else
    /* Hook SPORT0 Receive ISR. */
    interruptcb(SIG_SP0, AudioReceive_ISR);

    /* Hook user ISR. */
    interruptcb(SIG_USR0, RenderAudio1);

    interruptcb(SIG_USR1, RenderAudio2);
#endif

#ifdef INCLUDE_DECODERS
    interruptcb(SIG_USR1, RenderDecode);
#endif

    /* Enable interrupts */
    sysreg_bit_set_nop(sysreg_MODE1, IRPTEN);

}   // End AudioInit
```

Appendix C. TargetInfo.h

```
/**************************************************************************
*     TargetInfo.h
**************************************************************************
*
*     Description:  Main header file of ADSP-21489 EZ-Kit to specify which
*                   AudioWeaver modules to be included on the target
*
*     Copyright:    (c) 2017 DSP Concepts, Inc. All rights reserved.
*                   1800 Wyatt Drive, Suite 14
*                   Sunnyvale, CA 95054
*
**************************************************************************/
#ifndef _TARGETINFO_H
#define _TARGETINFO_H

#ifdef __cplusplus
extern "C" {
#endif

/* ------------------------------------------------------------------------
** Default audio I/O mode
** ------------------------------------------------------------------------ */
#define DEFAULT_AUDIO_IO_MODE AudioIOMode_ADC
//#define DEFAULT_AUDIO_IO_MODE AudioIOMode_SPDIF

/* ------------------------------------------------------------------------
** Baud rate info
** ------------------------------------------------------------------------*/
#define DEFAULT_BAUD_RATE (115200)
#define SUPPORTED_BAUD_RATES 115200, 460800

/* ------------------------------------------------------------------------
** Specifies the sizes of each of the heaps on the target
** ------------------------------------------------------------------------ */
#define MASTER_HEAP_SIZE              (25*1024 + 1536)
#define SLOW_HEAP_SIZE        (1024*2900)
#define FASTB_HEAP_SIZE               (25*1024)

#define MAX_COMMAND_BUFFER_LEN  300

/* ------------------------------------------------------------------------
** Specifies the audio modules that will be compiled into the target
** ------------------------------------------------------------------------ */
extern const ModClassModule awe_modAbsClass;
    /*snip for brevity*/
extern const ModClassModule awe_modZeroPaddingClass;

#define LISTOFCLASSOBJECTS \
&awe_modAbsClass, \
    /*snip for brevity*/
&awe_modZeroPaddingClass, \

#ifdef __cplusplus
}
#endif
#endif // _TARGETINFO_H
```