APPLICATION NOTE

# How to Simulate I2S Interface With USART Port On SAM9G15 eMPU

**Atmel | SMART SAM9G15 Series**

## Scope

This application note introduces an example on how to use the USART and Timer Counter in the Atmel® | SMART ARM®-based SAM9G15 embedded microprocessor unit (eMPU) to simulate an interface with an Integrated Inter-IC Sound (I2S) formatted audio codec.

This simulation can be used to provide an audio interface in addition to the I2S peripheral included in the SAM9G15 device.

## References

| Type | Item | Atmel Lit. No. |
|---|---|---|
| Datasheet | SAM9G15 Datasheet | 11052 |
| | Wolfson® WM8731 Datasheet | – |
| Software Library | SAM9G15 Software Package | – |

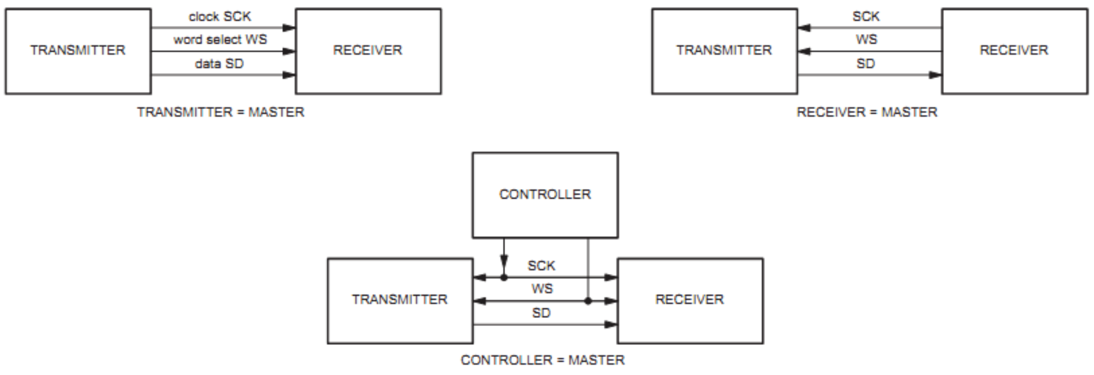
# 1. I2S Introduction

I2S (also known as Inter-IC Sound, Integrated Inter-chip Sound, or IIS) is an electrical serial bus interface standard introduced in 1986 and used for connecting digital audio devices together. It is used to communicate PCM audio data between integrated circuits in an electronic device. The I2S bus separates clock and serial data signals, resulting in a lower jitter than the typical communication systems that recover the clock from the data stream.

The I2S transmitter and receiver have the same clock signal for data transmission. The transmitter can be a master to generate the clock signal or be a slave to use the clock signal from the receiver which acts as a master, or either of the transmitter and the receiver can be salve and the other controller will act as a master to generate the clock signal for both.

Figure 1-1 shows the different use-cases.

**Figure 1-1. I2S System Configurations**

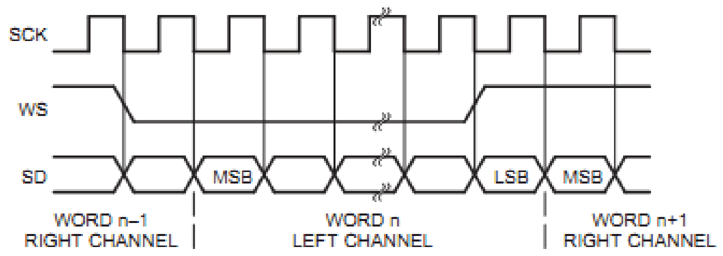


The I2S bus consists of at least three lines:

- Bit clock (also called continuous serial clock): SCK
- Word select: WS (also called left right clock LRCLK)
- Serial data (SD)

Figure 1-2 shows the typical I2S waveform.

**Figure 1-2. I2S Interface Timing**



In this application, SAM9G15 will act as a master to provide the two clocks: SCK and WS. During playback, SAM9G15 acts as a transmitter and when recording, SAM9G15 acts as a receiver. The codec works as a slave to be clocked by SAM9G15.

# 2. Device Overview

## 2.1 SAM9G15 Overview

SAM9G15 is a high performance eMPU with an ARM926 core, running up to 400 MHz.

The SAM9G15 features a graphics LCD controller with 4-layer overlay and 2D acceleration (picture-in-picture, alpha-blending, scaling, rotation, color conversion), and a 10-bit ADC that supports 4/5-wire resistive touchscreen panels. Multiple communication interfaces include a soft modem supporting exclusively the Conexant SmartDAA line driver, HS USB Host and Device and FS USB Host with dedicated onchip transceivers, two HS SDCard/SDIO/MMC interfaces, USARTs, SPIs, I2S and TWIs.

The 10-layer bus matrix coupled with multiple DMA channels ensures uninterrupted data transfers with minimal processor overhead.

The External Bus Interface incorporates controllers offering support for 4-bank and 8-bank DDR2/LPDDR, SDRAM/LPSDRAM, static memories, as well as specific circuitry for MLC/SLC NAND Flash with integrated ECC up to 24 bits.

## 2.2 Wolfson WM8731 Codec Overview

The Wolfson WM8731 is a low-power stereo codec with an integrated headphone driver. It supports sample frequency from 8 kHz to 96 kHz (8/32/44.1/48/88.2/96 kHz) and various word length (16/20/24/32-bit). It also can be programmed to support different types of Audio Data Interface such as I2S, Left-justified, Right-justified and DSP mode and it can work in Master or Slave clocking mode. The clock can be generated by the oscillator connected with the codec if it works in Master mode.

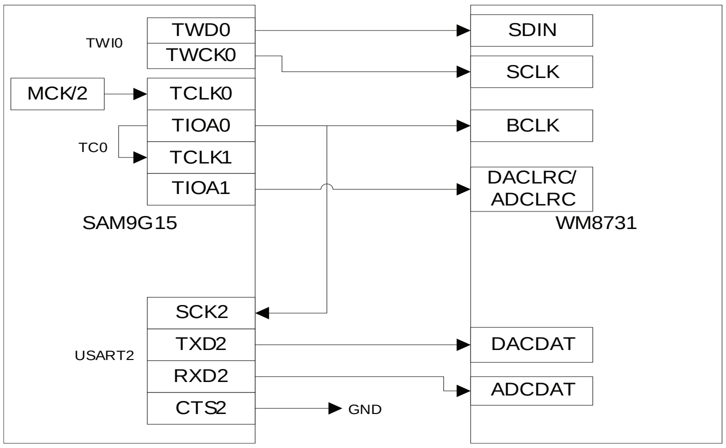
For detailed information of this codec, refer to the Wolfson WM8731 datasheet available on www.cirrus.com.

# 3. Implementation

## 3.1 Overall Introduction of Solution

In the application, we use two channels of Timer Counter 0 (TC0) to generate BCLK and LRC clock for the codec, and use USART2 to transmit or receive the PCM data to or from the codec. Figure 3-1 gives the overall diagram of the signal connection between SAM9G15 and WM8731.

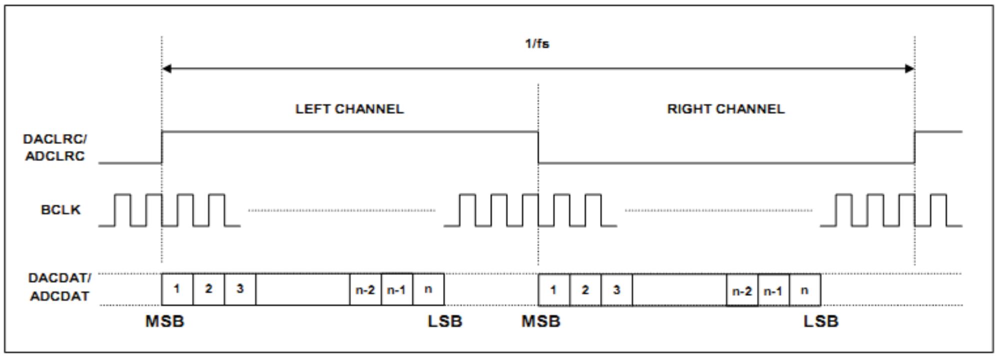
**Figure 3-1. TC USART Signal**



## 3.2 Clock Generation

In this application, we select Left-justified mode for the data transfer and receiving. Figure 3-2 illustrates the waveform of this mode.

**Figure 3-2. Left-justified Waveform**



As can be seen from the diagram, the DACDAT or ADCDAT (SD) is available as long as LRC (WS) level changes, and is sampled from the first rising edge of BCLK (SCK). The data is MSB first ordered.

The Timer Counter (TC) in SAM9G15 includes six identical 32-bit TC channels. Each channel can be configured to perform a wide range of functions including frequency measurement, event counting, interval measurement, pulse generation, delay timing and pulse width modulation. In this application, we select two TC channels (TIOA0 and TIOA1) to generate the waveform for BCLK and DACLRC/ADCLRC.

TIOA0 is selected to generate BCLK and TIOA1 is selected to generate LRC. Both channels are configured as Waveform mode to provide wave generation.

The TC's clock source is optional. We select TIMER_CLOCK1 (MCK/2) as the clock source for TIOA0, and use the TIOA0 output waveform as the clock source for TIOA1.

Both TIOA0 and TIOA1 use the RA and RC register to set the compare value and every time when the TC counter reaches the RA or RC value, the TIOA output level will invert. When the TC counter reaches RC value, its value will be reset to zero, and then continue to increase and so on. With this mode, we can generate the desured frequency by setting the register value of RA and RC.

To ensure the accuracy of the data transfer, the code is programmed to set the initial status of TIOA0 and TIOA1 output to high level.

Note: The first time the TC begins to output the waveform, the initial output level of TIOA is low level. Consequently, in the initialization part, the TC output will be enabled once to make sure that when restarted the TC output in the following time, its initial status is high level.

How to calculate the RA and RC register value of TIOA0 to generate BCLK with the required frequency:

Freq (tcsrc) = Frequency of clock source of the TC channel

Freq (bclk) = Frequency of the generated waveform

DutyCycle = The duty cycle percent of the positive level of the generated waveform

RC = Freq(tcsrc)/Freq(bclk)

In this application: Freq(tcsrc) = Board_MCK/2

RA = RC * DutyCycle/100

In this application, DutyCycle = 50, so RA = RC/2

How to calculate the RA and RC register value of TIOA1 to generate LRC with the required frequency:

From the waveform diagram, we can see the relationship between BCLK and LRC. Normally in one LRC cycle, the number of BCLK cycles is not less than two times of the number of sample length (bits per sample). For example, for the stereo PCM data with 16-bit sample length, every LRC cycle includes at least 32 (2*16) BCLK cycles. So if we take the TIOA0 as the clock source of TIOA1, then the RA register value is set to bit number per each sample, and RC register value is set to double of bit number per each sample.

RA = Sample length

RC = 2 * Sample length

## 3.3 Data Transfer

The Universal Synchronous Asynchronous Receiver Transceiver (USART) in SAM9G15 supports specific operating modes, providing interfaces on SPI, RS485, LIN, ISO7816 and infrared transceivers. In this application, we select SPI mode of USART to transmit the PCM data to the codec and receive the PCM data from the codec.

To ensure the clock synchronization, the clock of the SPI transfer is connected to the output of TIOA0 (BCLK), so we need to set the SPI in slave mode.

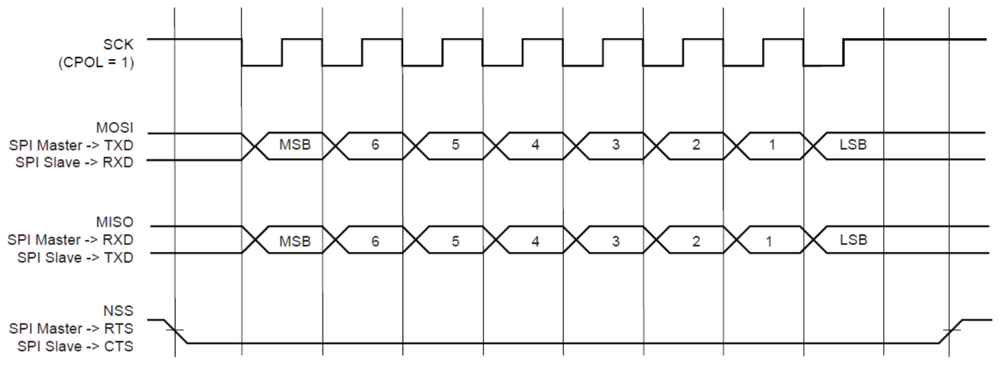
The maximum data length that can be transferred in SPI mode is 9 bits. Since the normal PCM data length is times of 8 bits (16, 24, or 32 bits per each sample), we set the data length for SPI transfer to 8 bits.

The data transfer order of MSB first is compliant with the data order requirement of the codec.

We also need to set the SPI transfer mode to read the data in each rising edge of the clock.

Figure 3-3 illustrates the waveform generated for the USART SPI Slave mode.

**Figure 3-3. USART SPI Slave Mode Waveform**

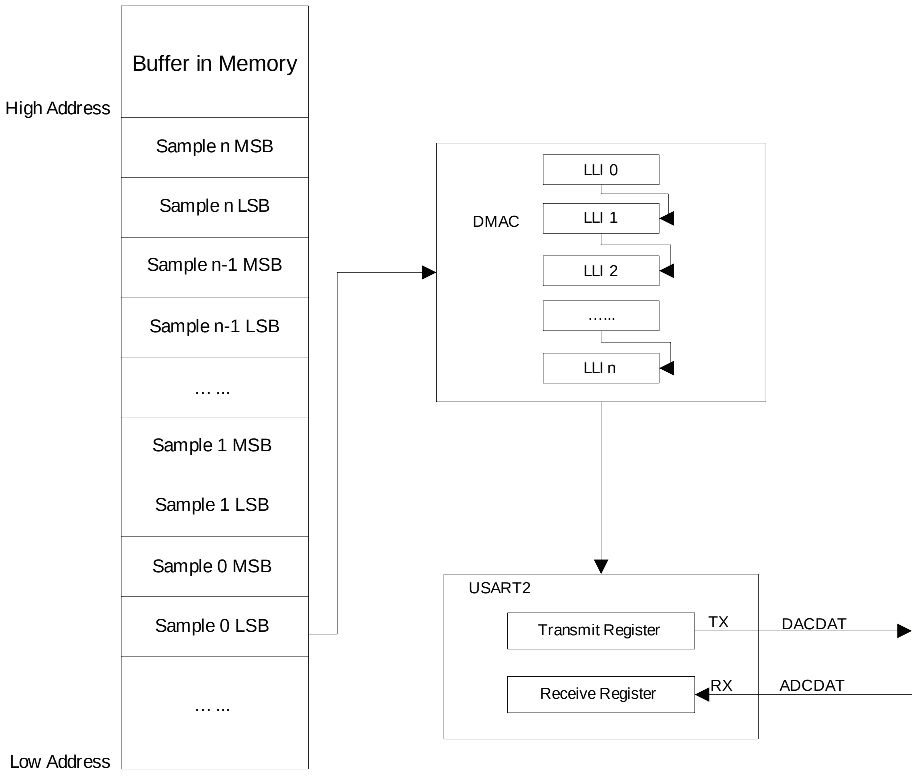


## 3.4 DMA Configuration

The data transfer of the USART SPI mode can use DMA mode, which will save processor bandwidth and significantly improve the system efficiency. SAM9G15 DMA can use Single-buffer transfer and Multi-buffer transfer. In this application, we use the Multi-buffer DMA transfer method: Linked Lists (buffer chaining). The buffers are linked by the pointer like a chain. The previous buffer contains the descriptor pointer which points to the next buffer's location in the system memory. The LLI (Linked List Item) registers should be initialized before DMA transfer.

Figure 3-4 gives the diagram overview of how to use DMA to transfer data.

**Figure 3-4. DMA Audio Transfer**



For detailed description of how LLI DMA works, please refer to section "DMA Controller (DMAC)" in the SAM9G15 datasheet.

## 3.5 Communication with the Audio Codec

Before operating the codec (WM8731), we need to configure it to the right working mode. This device can be programmed with 3-wire or 2-wire interface to access all feature settings such as volume control, mutes, de-emphasis, data transfer mode, data format, sample rate, etc.

In this application, we use 2-wire (I2C) mode to communicate with WM8731 to control the audio interface, volume adjustment, data format, sample frequency and other relative features for the audio playback and recording.

For a detailed description of the codec feature setting, refer to the Wolfson WM8731 datasheet.

## 3.6 Hardware Connection

This application is realized on the SAM9G15-EK. Hardware modifications need to be made by installing flying leads before running this demo on the board.

The instructions to modify SAM9G15-EK are the following.

1. Disconnect the original connection of the following signals:
   - Remove R147 connected with PA7 (refer to Figure 3-7)
   - Remove R156 connected with PA21 (refer to Figure 3-7)
   - Remove R157 connected with PA22 (refer to Figure 3-7)
   - Remove RR17 connected with PB1 and PB2 (refer to Figure 3-8)
   - Remove R86, R87, R88, R89 connected with pin3, pin6 and pin7 of WM8731 (refer to Figure 3-6)
   - Disconnect PA26 with pin4 of WM8731 (refer to Figure 3-6)
   - Disconnect PA25 with pin5 of WM8731 (refer to Figure 3-6)
2. Connect the corresponding signals according to the following direction:
   - TC0 $\rightarrow$ TIOA0 (PA21) connects to USART2 $\rightarrow$ SCK (PB2) and BCLK (pin3) of WM8731
   - TC0 $\rightarrow$ TIOA1 (PA22) connects to DACLRC (pin5) and ADCLRC (pin7) of WM8731
   - USART2 $\rightarrow$ SCK (PB2) connects to TC0 $\rightarrow$ TIOA0 (PA21)
   - USART2 $\rightarrow$ TXD (PA7) connects to DACDAT (pin4) of WM8731
   - USART2 $\rightarrow$ RXD (PA8) connects to ADCDAT (pin6)
   - USART2 $\rightarrow$ CTS (PB1) connects to GND

Figure 3-5, Figure 3-6, Figure 3-7 and Figure 3-8 are the schematics with the signals that need to be modified.

**Figure 3-5. SODIMM Schematics**

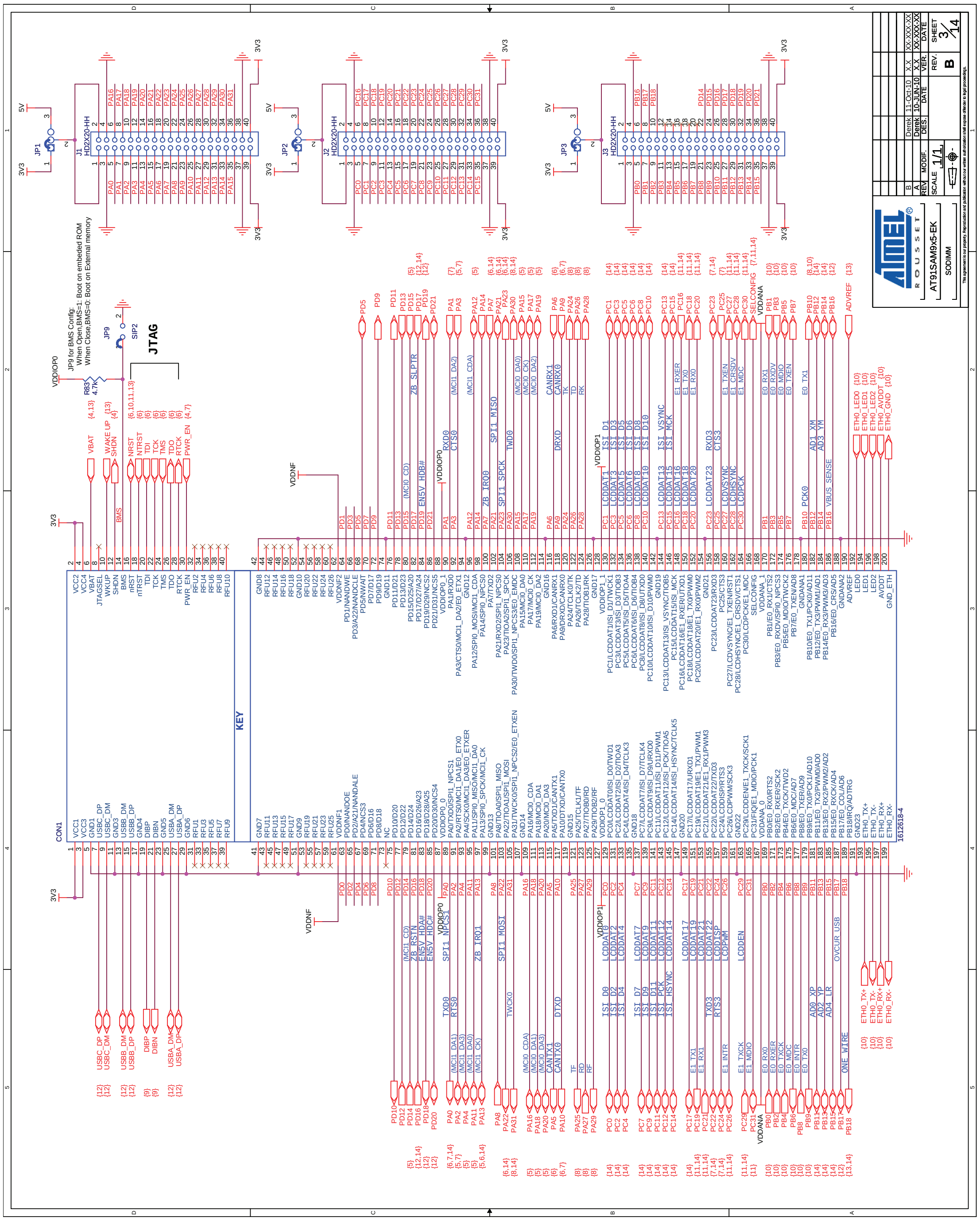


**Figure 3-6. AUDIO Schematic**

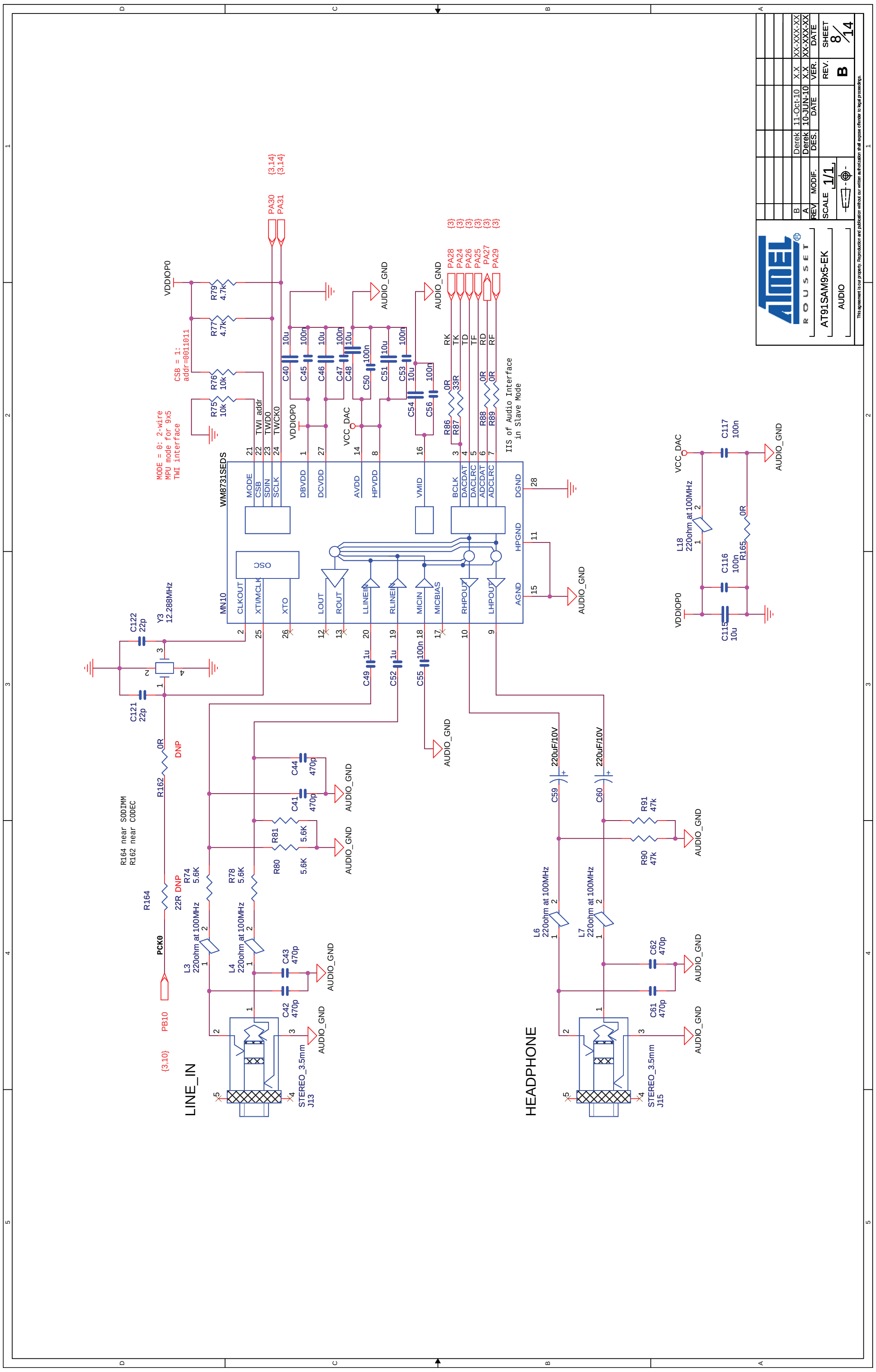


**Figure 3-7. LCD&ISI Schematic**

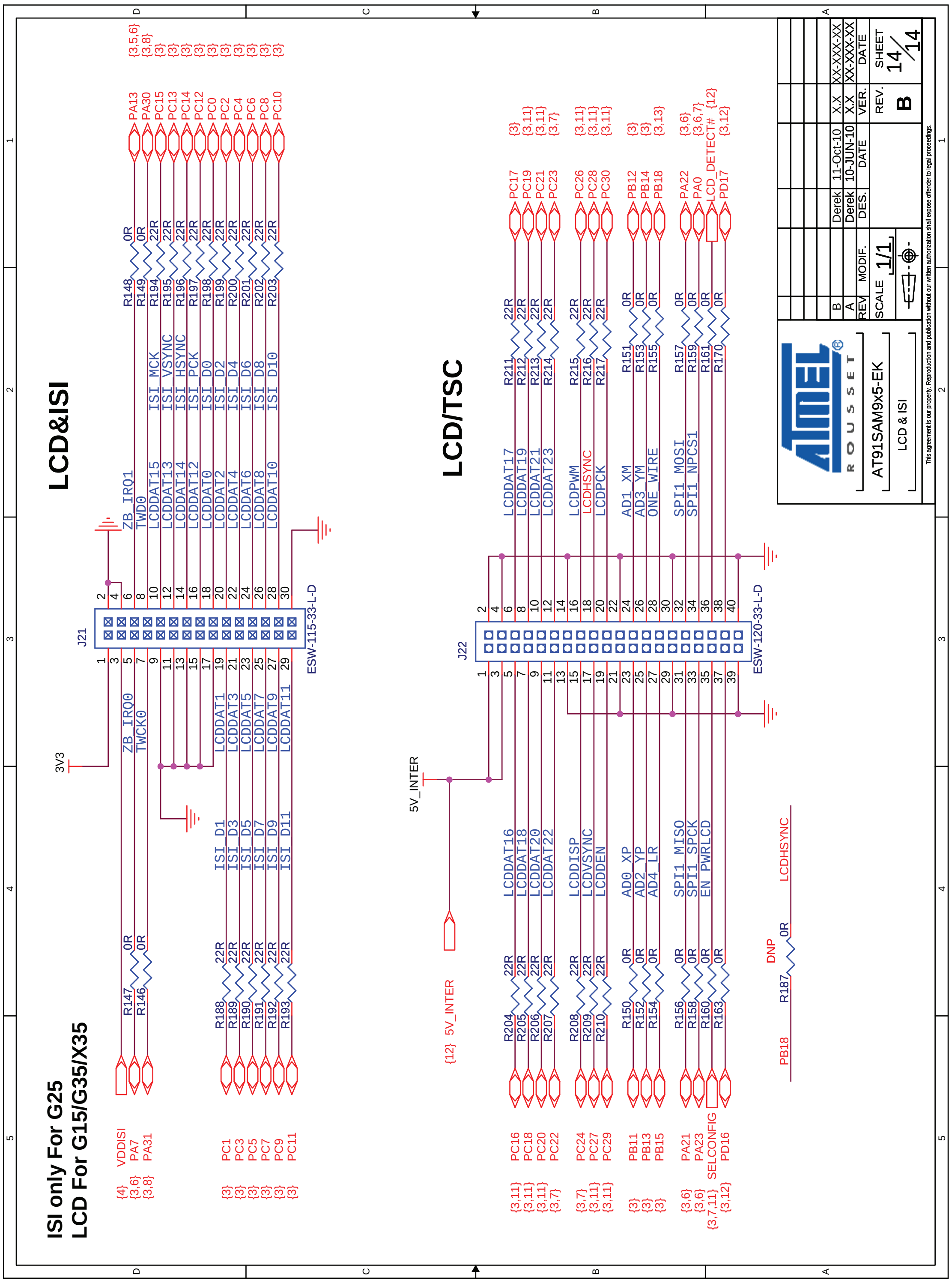
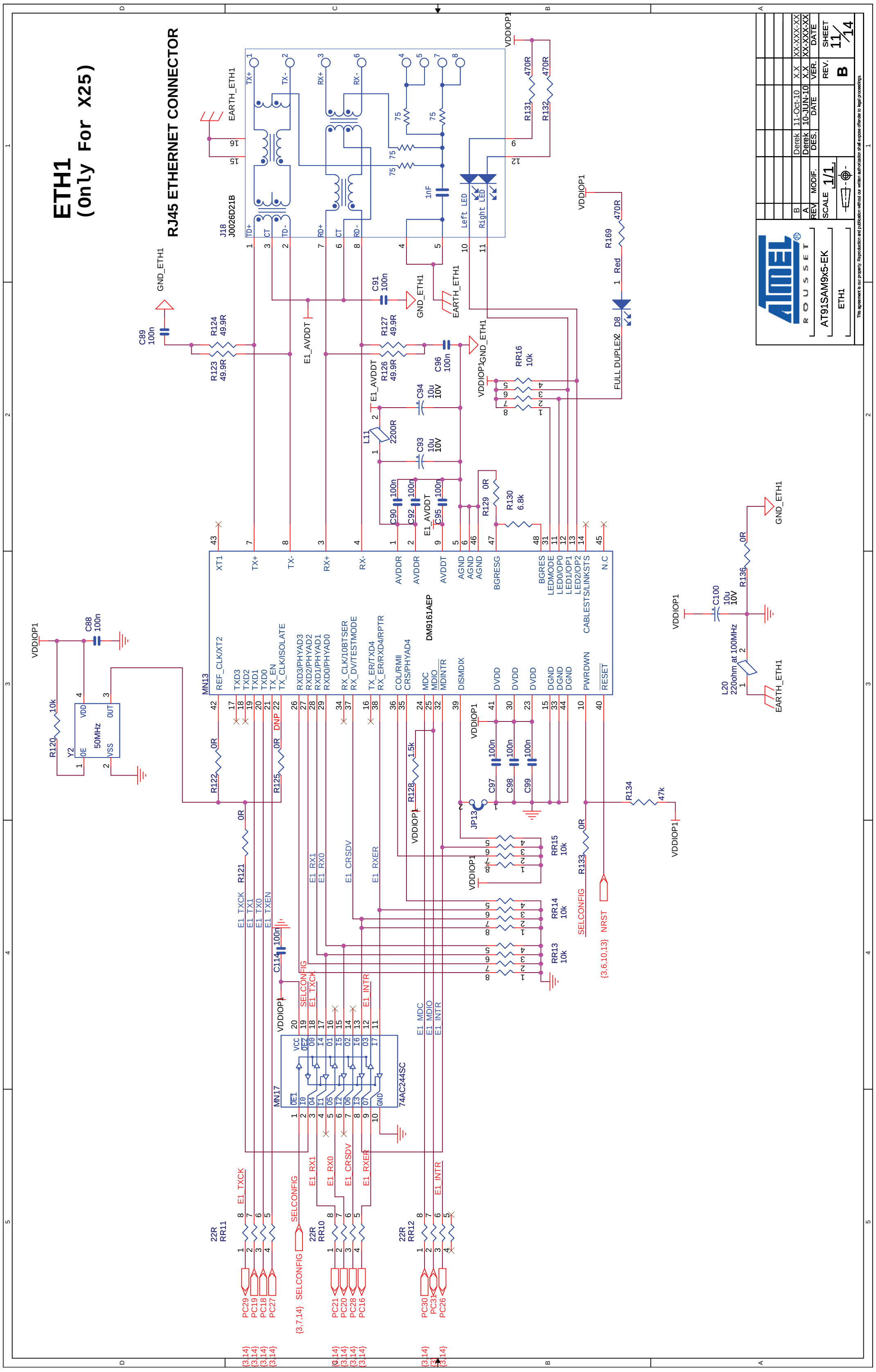


**Figure 3-8. ETH Schematic**



## 3.7 Software Implementation

The software demo provided for this application includes the contents listed in Table 3-1.

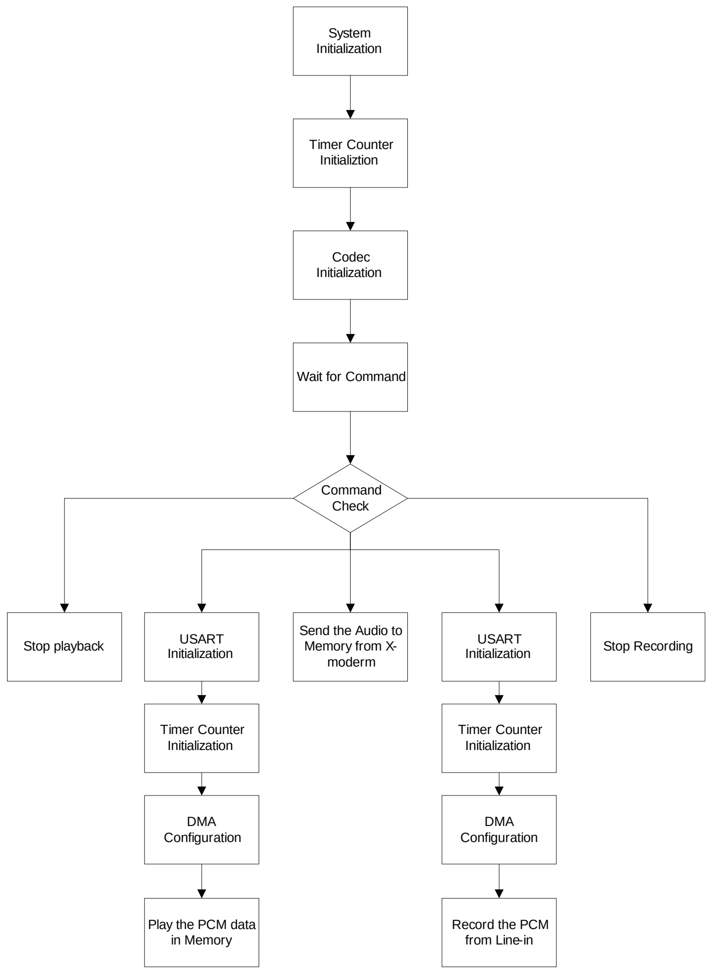
**Table 3-1. Contents of Provided Software Demo**

| Directory | Description |
|---|---|
| libraries | Includes the library files to support the SAM9G15 chip, EK and other peripherals such as USB, storage. Users can simply call the functions in the libraries. |
| usart_i2s_dma_audio | Includes the application IAR project |
| Readme.txt | A brief introduction document to tell users how to make the code run on SAM9G15-EK |

## 3.8 Code Overview

Users can use commands through the debug interface to start and stop playing or recording the audio, download the PCM data into the memory of SAM9G15 by Xmodem of HyperTerminal. Figure 3-9 is the overview for the functions of this demo.

**Figure 3-9. Function Flow Chart**


System Initialization
Timer Counter Initializtion
Codec Initialization
Wait for Command
Command Check
Stop playback
USART Initialization
Send the Audio to Memory from X-moderm
USART Initialization
Stop Recording
Timer Counter Initialization
Timer Counter Initialization
DMA Configuration
DMA Configuration
Play the PCM data in Memory
Record the PCM from Line-in

### 3.8.1 System Initialization

In this part, it is necessary to disable the system watchdog, configure the system clock, and configure the GPIO for the corresponding peripherals such as pins of USART2, TIOA0, TIOA1, etc.

Here is the code for reference:

```
/* Disable watchdog */
    WDT_Disable( WDT ) ;

/* System tick configuration */
    _ConfigureTick(BOARD_MCK);

/* Configure all pins used for this application */
             PIO_Configure(pins, PIO_LISTSIZE(pins));
```

As for the pins structure, it includes all the PIOs used in this demo:

```
/** List of pins to configure. */
static const Pin pins[] = {
                                                   PIN_GPIO_PA26,    //used as
GPIO
                                                   PIN_GPIO_PB7,     //used as
GPIO
                                                   PINS_TWI0,
                                                   PIN_TC0_TIOA0,
PIN_TC0_TIOA1,
                                                   PIN_USART2_RXD,
                                                   PIN_USART2_TXD,
                                                   PIN_USART2_SCK,
                                                   PIN_USART2_CTS,
                                                   PIN_PCK0
};
```

### 3.8.2 Timer Counter Initialization

In this part, mainly set the clock source and working mode for the two TC channels according to the default frequency firstly. And also need to define the start level status for the two waveform outputs.

```
/* Initialize TC, to make sure TC0 Ch0 TIOA output initial status is high level,
so pls make sure this initialization done before any other action */
   _TcWaveformInitialize(wavFrequency,wavBitsPerSample);
 /* Start the TC once, to make sure the initial output level of TIOA0&TIOA1 are
high when next time TC starts */
   _TcClkEnable();
   _TcClkDisable();
```

In function _TcWaveformInitialize(), there are two functions to set each TC channel in detailed.

```
static void TcCh0WaveformConfigure(uint32_t frequency)
{
    const uint32_t divisors[5] = {2, 8, 32, 128, BOARD_MCK / 32768};
    uint32_t ra, rc;

    /* Set channel 0 as waveform mode */
                   REG_TC0_CMR0 =
waveformConfigurations[configuration].clockSelection
             /* Waveform Clock Selection */
                   | TC_CMR_WAVE  /* Waveform mode is enabled */
                   | TC_CMR_ACPA_CLEAR /* RA Compare Effect: Clear */
                   | TC_CMR_ACPC_SET /* RC Compare Effect: Set */
                   | TC_CMR_WAVSEL_UP_RC /* UP mode with automatic trigger on
RC Compare */
                                                   | TC_CMR_ASWTRG_SET; /* Soft
trigger effect on TIOA: Set */

             /* Configure the BCLK frequency based on the wav file's sample
frequency */

    if (frequency == 8000)
                                configuration = 0;
    else if (frequency == 32000)
                                configuration = 1;
    else if (frequency == 44100)
                                configuration = 2;
    else if (frequency == 48000)
                                configuration = 3;
             else
    {
                                printf("This frequency: %d is not supported
\r\n", frequency);
                                return;
    }

             /* Set the RA and RC compare register value */
             rc = (BOARD_MCK /

divisors[waveformConfigurations[configuration].clockSelection]) /
waveformConfigurations[configuration].frequency;
    REG_TC0_RC0 = rc;
    ra = (100 - waveformConfigurations[configuration].dutyCycle) * rc / 100;
    REG_TC0_RA0 = ra;
}

/**
 * \brief Configure clock, frequency and duty cycle for TC0 channel 1 in
waveform mode.
 */
static void TcCh1WaveformConfigure(uint8_t DatLen)
{
             uint32_t ra, rc;
             /* Set channel 1's clock source as TIOA0 */
             REG_TC0_BMR = TC_BMR_TC1XC1S_TIOA0;
             /* Set channel 1 as waveform mode */
             REG_TC0_CMR1 = TC_CMR_TCCLKS_XC1  /* Waveform Clock Selection */
                                                   | TC_CMR_CLKI /*counter is
increamented on falling edge of the clock */
                    | TC_CMR_WAVE /* Waveform mode is enabled */
                    | TC_CMR_ACPA_CLEAR /* RA Compare Effect: clear */
                    | TC_CMR_ACPC_SET /* RC Compare Effect: set */
                  | TC_CMR_WAVSEL_UP_RC /* UP mode with automatic trigger on RC
Compare */
                                                   | TC_CMR_ASWTRG_SET; /* Soft
trigger effect on TIOA: Set */
    ra = DatLen;
    REG_TC0_RA1 = ra;
    rc = DatLen*2;
    REG_TC0_RC1 = rc;
}
```

### 3.8.3 Codec Initialization

In this part, SAM9G15 will configure the WM8731 with the I2C interface to set the headphone output volume, PCM data format, and configure the sampling frequency, etc.

```
/**
 * Initialize PCK & WM8731 for Audio
 * 12MHz PCK
 */
static void _ConfigureAudio(uint32_t sampleRate, uint8_t sampleBit)
{

    /* -- WM8731 Initialize -- */

    /* Enable TWI peripheral clock */
    PMC_EnablePeripheral(ID_TWI0);
    /* Configure and enable the TWI (required for accessing the DAC) */
    TWI_ConfigureMaster(TWI0, TWI_CLOCK, BOARD_MCK);
    TWID_Initialize(&twid, TWI0);


             /* WM8731 as slave */
             printf("Initialize WM8731 in Slave mode\r\n");

             /* If not use OSC on Codec, use 12M PCK, we select this mode in this
demo */
             /* USB mode, can support 8k/32k/44.1k/48k */
             WM8731_Init_1(&twid, WM8731_SLAVE_ADDRESS, 1, 0,
0,sampleRate,sampleBit);

             /* Enable the DAC master clock (Uses 12M, USB mode) */
             REG_PMC_PCK = PMC_MCKR_CSS_MAIN_CLK | PMC_MCKR_PRES_CLOCK;
             /* Programmable Clock 0 Output Enable */
             REG_PMC_SCER = PMC_SCER_PCK0;
             /* Wait for the PCKRDYx bit to be set in the PMC_SR register */
             while ((REG_PMC_SR & PMC_SR_PCKRDY0) == 0);


             /* -- Load WAV file information if wav file is available -- */
             CheckWavFile();

}
```

### 3.8.4 USART Initialization

In this part, it is necessary to configure the USART to work in SPI slave mode, set the transferred data length to 8-bit per each transfer with MSB firstly, and sample the data in the rising edge of the clock, etc.

Please refer to the following code for detailed settings.

```
/**
 * \brief Configure USART2 in SPI slave mode.
 */
static void _ConfigureUsart( void )
{
    uint32_t mode = US_MR_USART_MODE_SPI_SLAVE /* USART works as SPI Slave mode
*/
                                                   | US_MR_CHRL_8_BIT
/*Transmit data length is 8-bit*/
                    | US_MR_CHMODE_NORMAL
                                                   | US_SPI_BPMODE_3;


    /* Enable the peripheral clock for USART2 in the PMC */
    PMC_EnablePeripheral( ID_USART2 );

    /* Configure the USART in the desired mode @USART_SPI_CLK bauds */
    USART_Configure( USART2, mode, USART_SPI_CLK, BOARD_MCK );

}
```

### 3.8.5 DMA Configuration

In this part, it is necessary to create the DMA channels and set callback function for both USART transmitting and receiving.

```
/**
 * \brief DMA driver configuration.
 */
 static void _ConfigureDma(void)
 {
    uint32_t dwCfg;
    uint8_t iController;
    /* Driver initialize */
    DMAD_Initialize( &dmad, 0);
    /* IRQ configure */
    IRQ_ConfigureIT(ID_DMAC0, 0, DMA_IrqHandler);
    IRQ_ConfigureIT(ID_DMAC1, 0, DMA_IrqHandler);
    IRQ_EnableIT(ID_DMAC0);
    IRQ_EnableIT(ID_DMAC1);

    /* Allocate DMA channels for USART2 */
    usart2DmaTxChannel = DMAD_AllocateChannel( &dmad,
                                              DMAD_TRANSFER_MEMORY, ID_USART2);
    usart2DmaRxChannel = DMAD_AllocateChannel( &dmad,
                                              ID_USART2, DMAD_TRANSFER_MEMORY);
    if (   usart2DmaTxChannel == DMAD_ALLOC_FAILED
        || usart2DmaRxChannel == DMAD_ALLOC_FAILED )
    {
        printf("DMA channel allocate error\n\r");
        while(1);
    }

    /* Set RX callback */
    DMAD_SetCallback(&dmad, usart2DmaRxChannel,
                    (DmadTransferCallback)_DmaRxCallback, 0);
    /* Set TX callback */
    DMAD_SetCallback(&dmad, usart2DmaTxChannel,
                    (DmadTransferCallback)_DmaTxCallback, 0);

    /* Configure DMA RX channel */
    iController = (usart2DmaRxChannel >> 8);
    dwCfg = 0
           | DMAC_CFG_SRC_PER(
                                            DMAIF_Get_ChannelNumber( iController,
ID_USART2, DMAD_TRANSFER_RX ))
           | DMAC_CFG_SRC_H2SEL
           | DMAC_CFG_SOD
           | DMAC_CFG_FIFOCFG_ALAP_CFG;
    DMAD_PrepareChannel( &dmad, usart2DmaRxChannel, dwCfg );
             printf("USART2 DMA RX channel number:
%d\r\n",DMAIF_Get_ChannelNumber( iController, ID_USART2, DMAD_TRANSFER_RX ));

    /* Configure DMA TX channel */
    iController = (usart2DmaTxChannel >> 8);
    dwCfg = 0
           | DMAC_CFG_DST_PER(
                                                DMAIF_Get_ChannelNumber( iController,
ID_USART2, DMAD_TRANSFER_TX ))
           | DMAC_CFG_DST_H2SEL
           | DMAC_CFG_SOD
           | DMAC_CFG_FIFOCFG_ALAP_CFG;
    DMAD_PrepareChannel( &dmad, usart2DmaTxChannel, dwCfg );
             printf("USART2 DMA TX channel number:
%d\r\n",DMAIF_Get_ChannelNumber( iController, ID_USART2, DMAD_TRANSFER_TX ));

}
```

### 3.8.6 Audio PCM Data Playback

In this part, SAM9G15 will send the PCM data in its memory to the codec with the USART port. Before sending the data to codec, the system must perform two actions:

- ensure that each PCM sample stored in the memory can be sent to the codec with the MSB first order
- configure the LLI DMA pointer for each DMA transfer buffer:

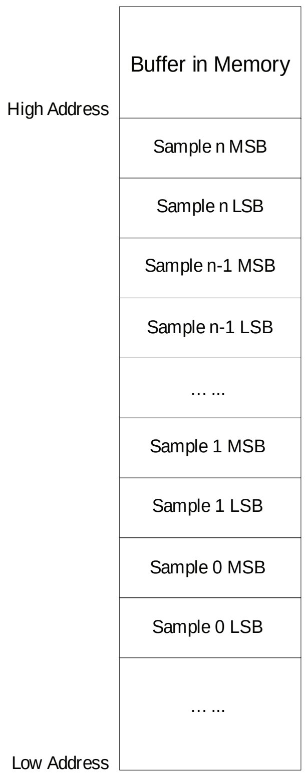
The 16-bit PCM data from WAV file is stored in the memory as described below.

The following is the storage order in PCM files:

| | | | |
|---|---|---|---|
| Sample 0<br>High Byte 0, Low Byte 0 | Sample 1<br>High Byte 1, Low Byte 1 | ...<br>... | Sample n<br>High Byte n, Low Byte n |

When the PCM files are sent to SAM9G15's memory, they will be stored in the order shown in Figure 3-10.

**Figure 3-10. DMA Data Storage**



For the USART DMA transfer, each time it will send one byte because the maximum data length for transfer is 8-bit. So every time when transferring one sample (e.g., 16-bit), DMA will transfer the bytes stored in the lower address firstly, and then the bytes stored in the higher address. To make sure the 16-bit sample can be transferred with MSB firstly, we need to convert the Low bytes and High bytes of each sample before starting the DMA transfer. For the 24-bit or 32-bit sample, the data also need to be converted with the same rule.

Note: For the PCM data recorded in the memory, there is no need to do the inversion because the recorded data received from USART is directly from codec and it can be sent back to the codec without data processing.

After having ensured that each PCM sample stored in the memory can be sent to the codec with the MSB first order and after having configured the LLI DMA pointer for each DMA transfer buffer, the system can start the DMA transfer by enabling the TC clock generation and the recorded data becomes audible from the headphone.

The following is the detailed code for reference:

```
/**
 * \brief Play a WAV file pre-loaded in DDRAM.
 */
static void PlayWav(uint32_t wavDataAddress, uint32_t wavSize)
{
    uint32_t size;
    uint32_t src;
    int i;

    remainingBytes = wavSize;
    transmittedBytes = 0;

      /* For the wav file, the sample storage is like: */
      /*Sample 0                      - Sample 1                      - Sample
n */
      /*High Byte0, Low Byte0 - High Byte1, Low Byte1 - High Byten, Low Byten
*/

      / *For SAM9G15 DMA transmission, it sends the bytes with lower address
firstly. But for I2S, MSB should be transmitted firstly. */
      /* So before starting the DMA transmission, need to reverse the high and
low bytes of the samples in the source file to adapt the DMA transmission order.
*/
      /* For the recorded PCM data in the flash, there is no need to do the data
reversion. */
      /* Here only provide the data conversion sample for 16-bit data, users
can add their own data processing code here. */
      if (userWav->bitsPerSample == 16)
      {
            //change the wav file contents value
            uint16_t *pWords = (uint16_t*)(wavDataAddress);
            uint16_t tmp1, tmp2;

            /* Audio Data Processing: only for 16-bit sample now */
            /* Convert the high and low bytes of one sample, because DMA
transfers the lower bytes firstly and WM7831 requires MSB first. */
            if (DatConverted ==0)
            {
                  for (i =0; i<(wavSize/2);i++ )
                  {
                        tmp1 = pWords[i]&0x00ff;
                        tmp2 = pWords[i]&0xff00;
                        pWords[i] = (tmp1<<8)|(tmp2>>8);

                        //printf("2. pWords[%d] = %x\r\n", i, pWords[i]);
                        }
                DatConverted = 1;
                }
        }


        /* Set the DMA LLI to transmit the audio data */
        src = wavDataAddress+ transmittedBytes;
        TX_LLI_NO = 0;
        for ( i = 0; (i<MAX_LLI_SIZE&& remainingBytes>0); i++)
        {
                        size = min(remainingBytes, 65535);

                        txLLI[i].dwSrcAddr = (uint32_t)src;
                        txLLI[i].dwDstAddr = (uint32_t)(&USART2->US_THR);
                        txLLI[i].dwCtrlA  = DMAC_CTRLA_BTSIZE(size)
                                                                |DMAC_CTRLA_SRC_WIDTH_BYTE |
DMAC_CTRLA_DST_WIDTH_BYTE;
                        txLLI[i].dwCtrlB  = DMAC_CTRLB_FC_MEM2PER_DMA_FC
                         | DMAC_CTRLB_SRC_INCR_INCREMENTING
                         | DMAC_CTRLB_DST_INCR_FIXED;


                        txLLI[i].dwDscAddr =((uint32_t)&txLLI[ i + 1 ].dwSrcAddr);
                        src += size;
                        remainingBytes -=size;
                        transmittedBytes +=size;
                        TX_LLI_NO++;
                        //printf("i = %d\r\n",i);
        }
        txLLI[i-1].dwDscAddr = 0;

        /*Start DMA transmit*/
        isWavPlaying = 1;
        DMAD_PrepareMultiTransfer(&dmad, usart2DmaTxChannel, &txLLI[0]);
        DMAD_StartTransfer(&dmad, usart2DmaTxChannel);
        USART_SetTransmitterEnabled(USART2, 1 ) ;
        Wait (100);
        _TcClkEnable();

}
```

### 3.8.7 Record PCM Data from Line-in

In this function, we need to set the LLI DMA pointers for the receive buffer and then we can start the recording by enabling the DMA transfer and TC clock generation.

```
/* Record the audio from Line-in input */
static void RecordWav(uint32_t recordDataAddr)
{
      uint32_t src;
      uint32_t size;
      uint8_t i;
      ReceivedBytes = 0;


      /* Use LLI to transmit data with DMA */
      src = recordDataAddr+ ReceivedBytes;
      RX_LLI_NO = 0;
      for ( i = 0; (i<MAX_LLI_SIZE&& ReceivedBytes<MAX_RECORD_SIZE); i++)
      {
            size = 65535;

            rxLLI[i].dwSrcAddr = (uint32_t)(&USART2->US_RHR);
            rxLLI[i].dwDstAddr = (uint32_t)src;
            rxLLI[i].dwCtrlA = DMAC_CTRLA_BTSIZE(size)
                                                  | DMAC_CTRLA_SRC_WIDTH_BYTE
                                                  | DMAC_CTRLA_DST_WIDTH_BYTE;
            rxLLI[i].dwCtrlB = DMAC_CTRLB_FC_PER2MEM_DMA_FC
                         | DMAC_CTRLB_SRC_INCR_FIXED
                         | DMAC_CTRLB_DST_INCR_INCREMENTING;


            rxLLI[i].dwDscAddr =((uint32_t)&rxLLI[ i + 1 ].dwSrcAddr);
            src += size;
            ReceivedBytes +=size;
            RX_LLI_NO++;

      }
      rxLLI[i-1].dwDscAddr = 0;

      /* Start DMA Receive */
      isRecording = 1;
      DMAD_PrepareMultiTransfer(&dmad, usart2DmaRxChannel, &rxLLI[0]);
      DMAD_StartTransfer(&dmad, usart2DmaRxChannel);
      USART_SetReceiverEnabled(USART2, 1 );
      Wait (100);
      _TcClkEnable();

}
```

### 3.8.8 Callback Function for DMA

The callback function is used to handle condition checking when the DMA interrupt happens. Normally we will check if the LLI buffer reaches the end or if the PCM data in the memory is finished.

```
/**
 *  \brief DMA TX callback.
 */
static void _DmaTxCallback(uint8_t status, void* pArg)
{

    pArg = pArg;
    //printf("Call back \r\n");
    if (status >= DMAD_ERROR)
    {
        isWavPlaying = 0;
        isWavPlaying = 0;
        return;
    }
   INT_NO++;

/* Check if read the last LLI, if yes, stop the playing */
    if (INT_NO == TX_LLI_NO)
    {
       isWavPlaying = 0;
                         INT_NO = 0;
                         /* Disable USART2 SPI transmitting */
                         USART_SetTransmitterEnabled( USART2, 0 ) ;
                         /*Stop DMA*/
                         DMAD_StopTransfer(&dmad, usart2DmaTxChannel);
                         /* Stop BCLK and LRCLK */
                         _TcClkDisable();
                         DisplayMenu();

    }

}

/**
 *  \brief DMA RX callback.
 */
static void _DmaRxCallback(uint8_t status, void* pArg)
{
            pArg = pArg;
            if (status >= DMAD_ERROR)
            {
        isRecording= 0;
        return;
            }
            INT_NO++;

            /* Check if read the last LLI, if yes, stop the recording */
    if (INT_NO == RX_LLI_NO)
    {
                   isRecording = 0;
                         INT_NO = 0;
```

```
                                        /* Disable USART2 SPI receiving */
                                        USART_SetReceiverEnabled( USART2, 0 );
                                        /*Stop DMA*/
                                        DMAD_StopTransfer(&dmad, usart2DmaRxChannel);
                                        /*Stop BCLK and LRCLK*/
                                        _TcClkDisable();
                                        DisplayMenu();

                    }
                }
```

### 3.8.9 Stop Playback or Recording

Users can stop audio playing or recording by inputting commands from the Debug window.

## 4. Conclusion

With this application demo, users can add one more audio interface in their application if more than one I2S interface is required. Although in this application we only provide the Left-justified mode demo code, users can very easily modify the code to generate the waveform they want to keep compliant with I2S or Right-justified mode.

## 5. Revision History

**Table 5-1. Revision History**

| Document Rev. 11260A | Changes |
|---|---|
| 17-Mar-15 | First issue. |