AND9529/D

# Music Playback Software of LC823450 Series for Audio Applications

ON Semiconductor®

www.onsemi.com

APPLICATION NOTE

## Introduction

This application note describes a software outline of sample application for standalone music playback implemented in LC823450XGEVK, which is an evaluation board kit, to help customers to understand and reuse the sample application.

Intended audience is customers who are building audio application using LC823450 Series (called LC823450 hereafter).

## BACKGROUND

LC823450XGEVK is an audio processing system evaluation board kit used to demonstrate LC823450. This part can record and playback, and offers High-Resolution 32-bit & 192 kHz audio processing capability. It is possible to cover most of the functions necessary for a portable audio with only this LSI.

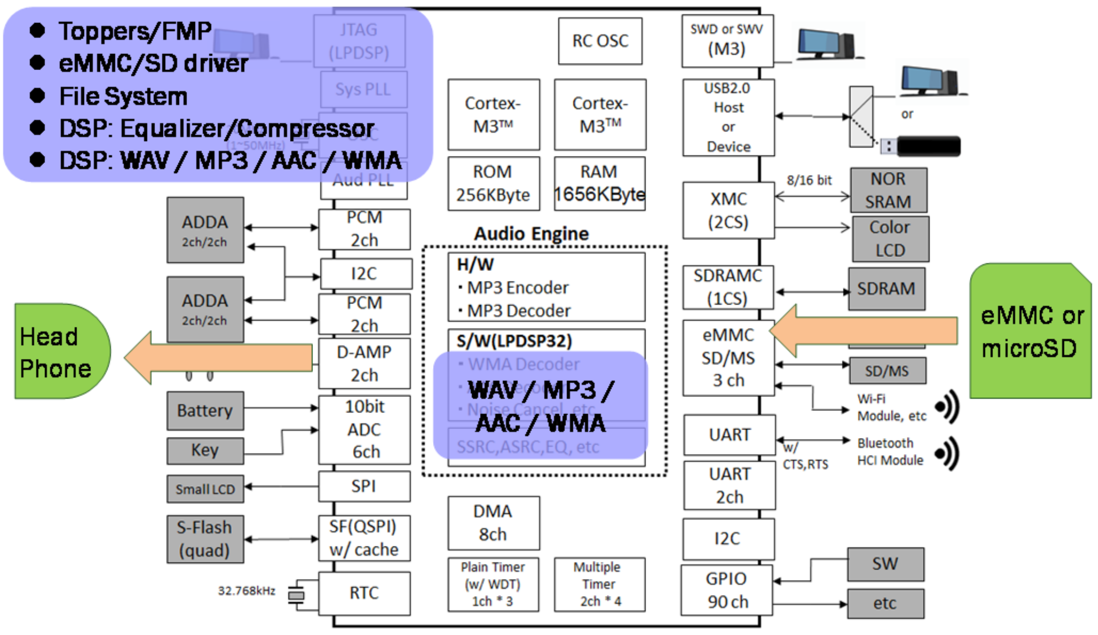
This application note focuses on the standalone music playback application implemented in LC823450XGEVK and describes a software outline of the sample code.

## STANDALONE MUSIC PLAYBACK

### System structure

Figure 1 shows a system structure of the standalone music playback mode. The Audio Engine block in the center is composed of audio hardware blocks and a software block which is LPDSP32. Audio music files to be played back are stored in the eMMC device or microSD card which is connected to SD interface in LC823450. The data of the audio music files is decoded by LPDSP32 inside based on the codec type of the files. LPDSP32 can decode the data of the 4 codec types such as WAV, MP3, AAC or WMA, and the decoded music data is sounded at the head phone through D-AMP in LC823450.

**Figure 1. Standalone Music Playback mode**



Software mainly used in the standalone music playback mode is Toppers / FMP, eMMC / SD driver, File system, Equalizer / Compressor and WAV / MP3 / AAC / WMA decoder. Toppers / FMP is a multitask OS used in the system. eMMC / SD driver is a device driver which controls SD interface. File system is included in libraries for Cortex®-M3. Equalizer/Compressor and WAV / MP3 / AAC / WMA decoder are included in libraries for LPDSP32.

## Audio structure

Figure 2 shows details in the Audio Engine block. It has audio buffers in the center and some audio functions in the right upper side as hardware blocks and LPDSP32 in the right lower side as a software block.

Figure 2 also shows 3 sound routes at the standalone music playback mode in the Audio Engine block. They are the Music Play route, Notification Play route and Ambient Noise route.

The Music Play route is a basic route which shows music playback in light blue in Figure 2. The Notification Play route in light green has a role to add some notification sound from activity trackers, etc. to music playback. The Ambient Noise route in light orange also has a role to add some ambient sound to music playback so as to hear ambient sound picked up from microphone.

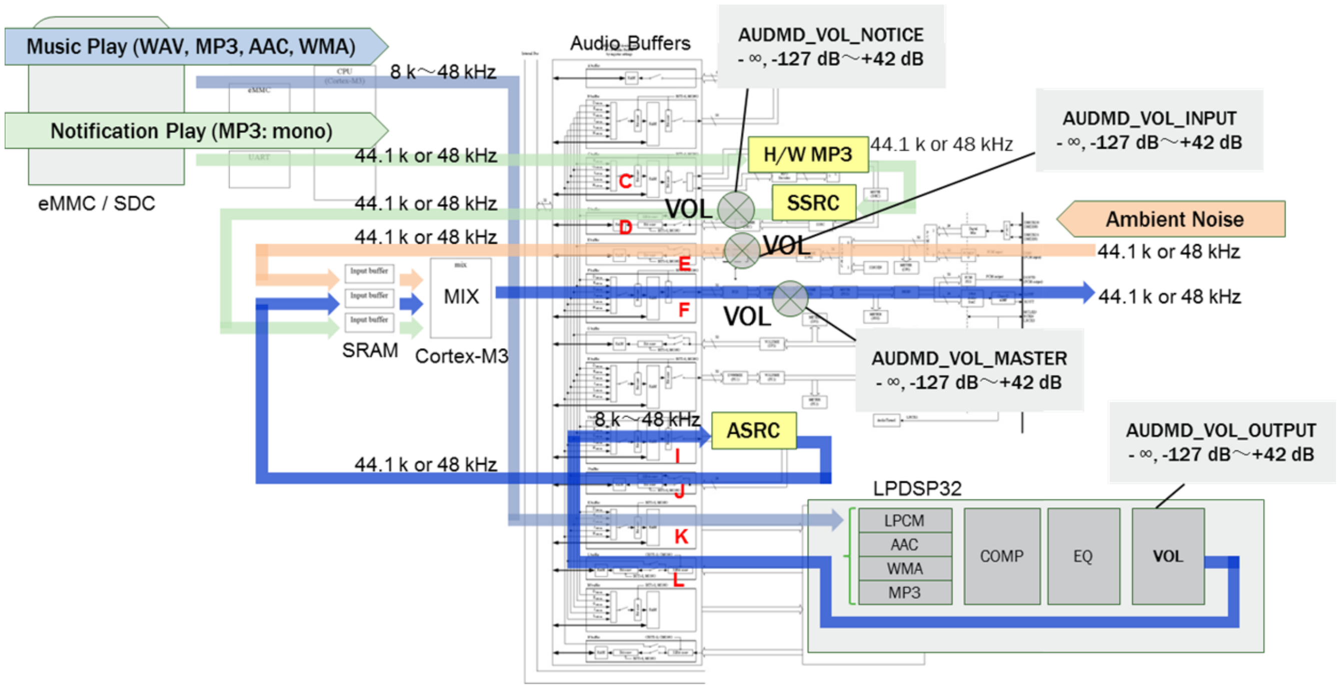
In the Music Play route, audio music files, which have WAV, MP3, AAC, or WMA codec type with sampling rate from 8 to 48 kHz, are stored in the eMMC device or SD card. They are input to LPDSP32 through the K buffer in the audio buffers, and they are decoded by LPDSP32 based on the codec type and processed by LPDSP32 with the Equalizer / Compressor library and volume function if necessary. The processed data by LPDSP32 is input to ASRC function through the L and I buffer in order to convert sampling rate into 44.1 or 48 kHz which is selected in the menu setting in the board. The data is input to internal SRAM through the J buffer and processed by Cortex®-M3 to mix it with notification sounds and ambient sounds. The mixed data is input to the F buffer and passed through the audio output functions and output to the headphone via D-AMP.

In the Notification Play route, notification sound files, which have MP3 codec type with sampling rate of 44.1 or 48 kHz with only monaural sounds, are stored in the eMMC device or SD card. They are input to the Hard-Wired MP3 decoder and decoded. The decoded data is converted at SSRC into either 44.1 or 48 kHz sampling rate which is selected in the menu setting, and output to the internal SRAM for mixing via the D buffer.

In the Ambient Noise route, ambient sounds are input from the microphone on the board with the sampling rate 44.1 or 48 kHz which is selected in the menu setting. They are passed through the audio input functions and output to the internal SRAM for mixing via the E buffer.

Each route has a volume function independently before getting to the internal SRAM on the route and the last volume function is on the mixed route after the F buffer.

**Figure 2. Sound route at the Standalone Music Playback mode**



## Software flow

Figure 3 shows a rough software flow of the standalone music playback mode in light orange. Some keypoints which customers should know in order to reuse the sample application are described in this chapter.

In the step(#1), the OS is set up, and initial routines which should be used only once through the system are executed. In the step(#2), some blocks which are related to the whole system are initialized. In the step(#3), the file system is initialized, and this step has some branching points to the each operating mode decided in the menu display or to the default mode assigned after power on which is standalone music playback mode. In the step(#4), some routines used at music stop state in the standalone music playback mode are executed. In the step(#5), a music file selected in the standalone music playback mode is played back. In the step(#6), the menu is displayed on the LCD, and an operating mode which should be executed at next step can be selected. Whenever getting away from the menu, the step(#3) is executed again, because the file system needs to be initialized so as to access to a music storage again when the music storage is changed from one of either the eMMC device or the microSD card to the other in the menu.

**Figure 3. Standalone music playback flow**

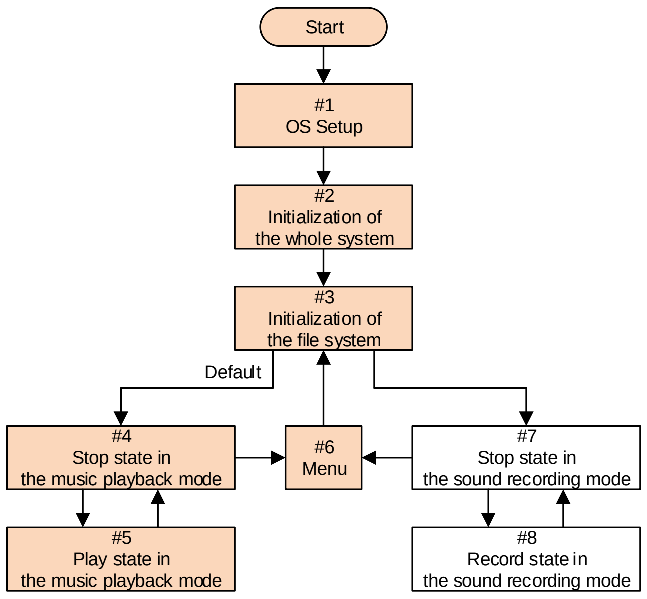


Figure 4 shows a part of a source code in the step(#1), which is the “global_inirtn” routine in the “SysAp.c” file. This routine initializes internal power domain, GPIO, DMA and so on, and it is executed only once through the system.

Especially, the “port_init” routine in red initializes GPIO setting such as port direction, drive capability switching, pull resister setting and so on. If the customers reuse the sample application, they have to edit the “port_init” routine to meet their system.

Figure 5 shows a part of a source code in the step(#2), which is the “SysApInit” routine in the “SysAp.c” file. The “SysUartInit” routine in red initializes a UART function and this initialization enables the customers to use “Printf” routines through the UART function on PC at debug. If the customers need to use I$^2$C functions in their system, they should activate the “I2c0Init” and/or “I2c1Init” routine in red by removing the comment-out mark. The “LcdInit” routine in red initializes to clear up the LCD display as a blank on LC823450XGEVK. If the customers don’t need to use the LCD display in their system, they should inactivate the routine by adding the comment-out mark. In addition, The “MediaDrIdentfyFixed” routine in red recognizes the music storage by establishing a connection to it by sending commands. This is a preparation so as to access to music data and LPDSP codes in the storage.

**Figure 4. OS Setup**

Step(#1) : SysAp.c

```
void global_inirtn( intptr_t exinf )
{
        :
        :
   AdcInit();
   AdcExit();

   port_init();

   StgCoreValtageChange(TRUE_T);
        :
        :
   return;
}
```

**Figure 5. Initialization of the whole system**

Step(#2) : SysAp.c

```
static SINT_T SysApInit(void)
{
        :
        :
   SysUartInit();

   SysMdInitTimer();

//   I2c0Init();
//   I2c1Init();

   LcdInit();

   if(MediaDrInit(PwrDrGetAHBClock(), STG_VDDIF_33V)){
      console_Printf(USE_CONSOLE_CH,"Err Init¥n");
         goto ERR_END;
   }
   SysMdChangeClock(100000000, TRUE_T, FALSE_T);

   if(MediaDrIdentfyFixed(PwrDrGetAHBClock())){
      console_Printf(USE_CONSOLE_CH,"Err Idetfy¥n");
         goto ERR_END;
   }
        :
        :
   return(ret);
}
```

Figure 6 shows a part of a source code in the step(#3), which is the “app_Init_Init” routine in the “app_init.c” file. This routine initializes the file system and branches according to the operation modes in the menu. The default shows the standalone music playback mode.

The “FsInit” routine in red initializes internal SRAM area assigned in order to mount the file system. The next grouped routines in red mount the file system. The file system corresponds to FAT12, FAT16, FAT32 and exFAT.

The “app_MakePlayList” routine in red makes a playlist of music files stored in the “music” folder in the storage. The “app_SetMPMode” routine in red initializes the audio function in LC823450, for example, PLL setting, DAC setting, volume setting and so on.

**Figure 6. Initialization of the file system**

Step(#3) : app_init.c

```
SINT_T app_Init_Init(t_SYS_MASTER_TBL *m_tbl)
{
        :
        :
   FsInit();

   if(FsMount(0)){
      FsFormat(0);
      FsMount(0);
   }
   dbg_warning_value( FsMount( 1 ) );
        :
        :
   switch(SetParaGetMode()){
      case MODE_REC:
         :
         :
         break;
      default:
         dbg_warning_value( app_MakePlayList(SetParaGetDrive(),
                                             SetParaGetRecMode()) );
         app_SetMPMode();
         SysMdSetStatus(m_tbl, APP_STS_STOP, TRUE_T);
         break;
   }

   return( 0 );
}
```

Figure 7 shows a part of a source code in the step(#4), which is the “app_Stop_Init” routine in the “app_stop.c” file. This routine is called at first whenever the standalone music playback mode works, and it initializes some functions for music stop state, because the standalone music playback mode always starts from the music stop state. The “app_lcd_com” routine in red sets the LCD device to display characters which mean the music stop state. If the customers don’t need to use the LCD display in their system, they should inactivate the routine by adding the comment-out mark.

After the initialization of the music stop state, the continuing task waits for any available keys in the music stop state to be pushed. If the key which means “Playback” is pushed, the state of the standalone music playback mode will change from the music stop state to the music play state. If the key which means “Menu” is pushed, the sample application will get away from the standalone music playback mode to move the menu.

**Figure 7. Stop state in the music playback mode**

Step(#4) : app_stop.c

```
static SINT_T app_Stop_Init(t_SYS_MASTER_TBL *m_tbl)
{
        :
        :
   app_lcd_com(DISP_PART_TITLE, m_tbl);
        :
        :
   return (0);
}
```

Figure 8 shows a part of a source code in the step(#5), which is the “app_Stop_Play” routine in the “app_stop.c” file. This routine runs whenever the state becomes the music play state. The “Stop_play” routine instructs other tasks controlled in the multitask OS to play back music files as shown in the Music Play route in Figure 2. After the instruction, the continuing task waits for any available keys in the music play state to be pushed, and the other tasks for music playback simultaneously work to get the music file from the music storage and to push it to the audio buffers for LPDSP decode and to output the decoded music data to the headphone and so on.

**Figure 8. Play state in the music playback mode**

Step(#5) : app_stop.c

```
static BOOL_T app_Stop_Play(t_SYS_MASTER_TBL *m_tbl,
                            t_MSG_DATA_BASETYPE *pMsg)
{
   return   Stop_Play( m_tbl );
}
```