AN2739
Application note

How to use the high-density STM32F103xx microcontroller to play audio files with an external I²S audio codec

Introduction

This application note describes how to use the high-density STM32F103xx I^2S feature to play audio files using an external codec.

The I^2S protocol is widely used to transfer audio data from a microcontroller/DSP to an audio codec in order to play melodies (stored in a memory) or, to capture analog sound (from a microphone).

The high-density STM32F103xx allows I^2S audio communications using the SPI peripheral, and implements specific functionalities for this communications mode.

The first, preliminary section of this application note may be skipped by advanced users.

Note: Throughout this document, and unless otherwise specified, the term of I^2S will be used to refer to the I^2S feature of the SPI peripheral that is implemented in high-density STM32F103xx microcontrollers.

Contents

1 **I²S general description 5**

1.1 I²S protocol 5

1.2 STM32F103xx I²S feature presentation 6

2 **Implementation example 8**

2.1 General overview 8

2.2 Hardware description 8

2.2.1 Audio codec 9

2.2.2 STM32F103xx and board configuration 10

2.3 Firmware description 11

2.3.1 `I2S_CODEC` driver firmware description 11

2.3.2 Demo firmware description 18

2.3.3 Timing considerations 22

2.4 General, allowed parameters 23

3 **Conclusion 24**

4 **Revision history 25**

List of tables

Table 1. Driver library description . . . 11
Table 2. I2S_CODEC driver high-level functions . . . 11
Table 3. I2S_CODEC_Init function . . . 12
Table 4. Configuration parameters list . . . 13
Table 5. I2S_CODEC_ReplayConfig function . . . 13
Table 6. I2S_CODEC_Play function . . . 14
Table 7. I2S_CODEC_ControlVolume function . . . 14
Table 8. I2S_CODEC medium-level driver functions . . . 15
Table 9. I2S_CODEC low-level driver functions . . . 16
Table 10. Joystick_Config function . . . 18
Table 11. Description of the Joystick functionalities . . . 18
Table 12. LCD_Update function . . . 19
Table 13. Document revision history . . . 25

List of figures

Figure 1. I^2S Phillips protocol waveforms 16/32-bit 5
Figure 2. I^2S protocol signal description and configuration 6
Figure 3. Typical implementation design description 8
Figure 4. Audio codec hardware implementation. 9
Figure 5. Driver's functional flowchart 1 16
Figure 6. Driver's functional flowchart 2 17
Figure 7. Demo's functional flowchart 1 20
Figure 8. Demo's functional flowchart 2 21

1 I²S general description

1.1 I²S protocol

I²S (IC-to-IC sound) is an audio data transfer standard using a three-line bus for serial and synchronous data transmission.

Data are transmitted on the SD line (Serial Data) in Little Endian format (MSB first). Data length is not limited (usually 16/20/24/32/64 bits). Data are synchronized by the rising or falling edge of SCK (Serial Clock) for the transmitter, and by the falling edge of SCK for the receiver. Refer to *Figure 1*.

Data represent stereo digital sound, so each sample contains two words, the right-channel sample and the left-channel sample. Instead of using two data channels, multiplexing is performed by transmitting each word over half a sampling period, which doubles the sampling rate, and makes it possible to transmit two words per period.

A control signal WS (Word Select) is then used to determine if the word being sent is the right or left one. This signal also determines the beginning and the end of the data: there is no need to fix the data length. Receiver and transmitter data lengths can therefore be different, as well as the right and left data lengths.

WS is synchronized to either the rising or the falling edge of SCK and precedes the MSB by one SCK period in order to have enough time to store and shift operations.

As in most communication protocols, there must be a master and a slave. The master provides and controls the SCK clock and the WS signal, while the slave only sends or receives data. The master can be the receiver, the transmitter or a third element (Controller). Refer to *Figure 2*.

Figure 1. I²S Phillips protocol waveforms 16/32-bit

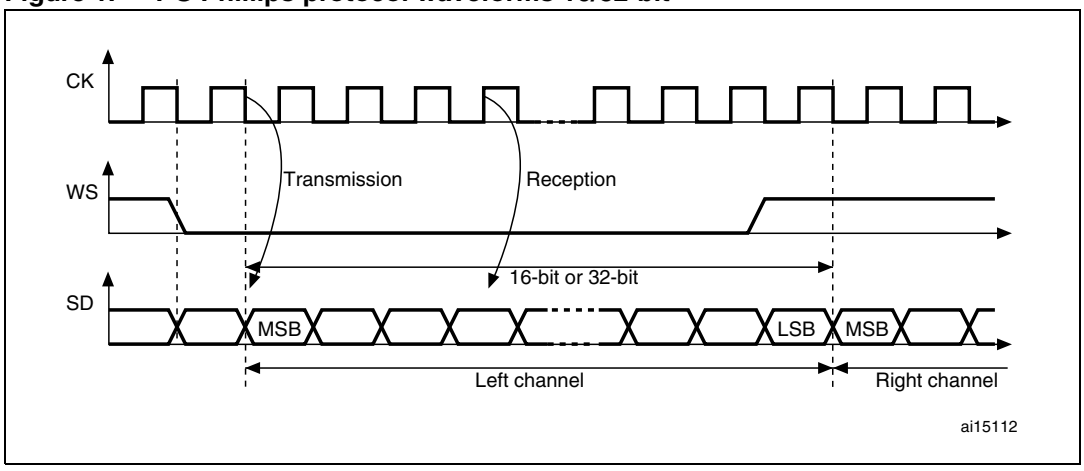
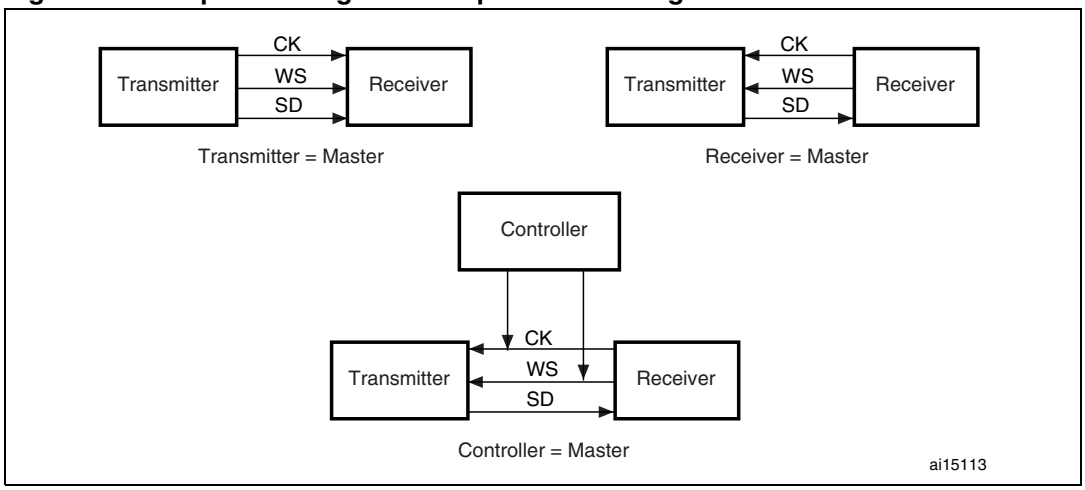


Figure 2. I²S protocol signal description and configuration



1.2 STM32F103xx I²S feature presentation

The STM32F103xx implements the I²S feature as a mode included in the SPI peripheral. The user has to choose either the SPI mode or the I²S mode (software configuration).

The STM32F103xx I²S is available in simplex mode only (receive-only or transmit-only), the communication direction is configured by software.

The I²S peripheral supports four audio protocols (configurable by software):

- I²S Phillips protocol
- MSB protocol
- LSB protocol
- PCM protocol (including PCM Short and PCM Long)

It also supports most audio frequencies (8 kHz, 16 kHz, 22.05 kHz, 44.1 kHz, 48 kHz, etc.)

The data format is programmable to 16-, 24- or 32-bit data length (for each channel), MSB first, and to 16- or 32-bit packet length (for each channel).

The WS signal assignment is managed by hardware and a relative flag (CHSIDE) is available to monitor the channel side (for Phillips, MSB and LSB standards).

The I²S peripheral can be configured as the master or the slave in the audio communication. The I²S generates its own clock (independent of the SPI clock used to interface registers to the APB bus) using a 9-bit prescaler and designed to reach accurate audio frequencies (8 kHz, 16 kHz, 22.05 kHz, 44.1 kHz, 48 kHz, etc.)[(a)]. When configured in master mode, the peripheral is able to output an additional master clock (MCLK) at a fixed rate: 256 × F_S (where F_S is the audio frequency).

a. The sampling frequency is the bit clock frequency (CK) and is equal to:
F_S × number of bits per channel × number of channels, where F_S is the WS frequency in the Phillips, MSB and LSB standards and, the WS/2 frequency in PCM mode.

To decide if MCLK should be generated or not, the following facts have to be taken into consideration:

- The external I²S device requirements (codec/DAC).
  In general these devices need a master clock (usually at the rate 256 × F_S) to perform internal and sampling operations.
- The audio frequency accuracy is, in some cases, compromised by enabling the MCLK output (that is when a low-frequency clock is driving the STM32F103xx system (SYSCLK less than 36 MHz)).

The audio communication can be controlled in one of the following ways:

- by polling on the TXE/RXNE flag (bits 1/0 in SPI_CR2 register): wait until TXE/RXNE flag is set then write/read the channel wave data to/from the SPI_DR register. (Suitable for tests/small applications, etc.)
- Interrupt on TXE/RXNE: configure and enable the transmit/receive interrupt. And in the interrupt subroutine, write/read the channel wave data to/from the SPI_DR register. (Suitable for most applications/RT software, etc.)
- DMA transfer: configure the DMA to load/unload the data from/to the SPI_DR register on each Rx/Tx request. (Suitable for high-performance requirements.)

Note: In I²S mode, the DMA is used in exactly the same way as the SPI mode (with respect to the supported audio transmission protocols, the CRC feature is not available in I²S mode).

The choice of the SYSCLK frequency directly impacts the I²S transmission quality (in master mode): the sampling clock (CK) and WS clock are derived directly from SYSCLK divided by a 9-bit prescaler in order to obtain the most accurate F_S frequency. For maximum accuracy, the prescaler allows odd division by two (using the ODD bit in the SPI_I2SPR register).

But when the SYSCLK frequency is too low (it is typically greater than 36 MHz), the division results in a low accuracy factor leading to audio quality degradation.

2 Implementation example

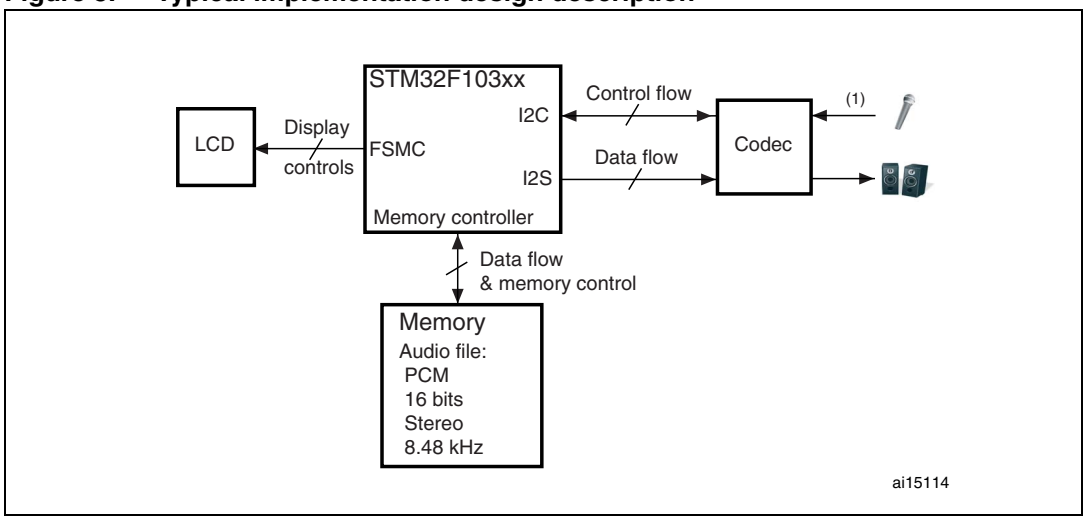
2.1 General overview

The example presented in this application note aims at providing typical hardware and software implementation basics for an audio application like portable audio players, sound synthesis systems, speech recorders, cell phones or interactive control boards.

Typically, the system embeds:

- a microcontroller (STM32F103xx device)
- an audio codec
- a speaker
- a memory support (where the audio file is stored).

Figure 3. Typical implementation design description



1. The audio input functionality (microphone) is not discussed in this application note.

The audio file format supported by the application is PCM, 16-bit data length, stereo/mono channels, 8 to 48 kHz audio frequency.

2.2 Hardware description

General requirements

The developed example is mainly based on the STM3210E-EVAL evaluation board but the functional and structural description is similar for most applications and platforms.

The memory in which the audio file is stored is the NOR Flash memory implemented on the board. A different memory/source may be configured as the audio file support (like the SPI Flash memory).

Other board resources are used to interface the application:

- Audio codec: AK4343 implemented on the STM3210E-EVAL and connected to the I2S2 interface (and to relative passive components).
- Stereo audio speaker and audio jack connected to the audio codec and implemented on the STM3210E-EVAL.
- Joystick and key push-buttons: connected to the PG7, PD3, PG13, PG14, PG15 and PG8 pins on the board. These push-buttons are used to control the audio stream.
- LCD screen: implemented on the STM3210E-EVAL evaluation board and controlled by the FSMC interface.

2.2.1 Audio codec

The audio codec implemented on the STM3210E-EVAL is the AK4343 from AKM. This codec allows digital (PCM raw data transmitted with I^2S protocol) to analog conversion. The audio parameterization and the codec configuration are performed through an I^2C interface. The codec has 36 configuration registers mainly used to:

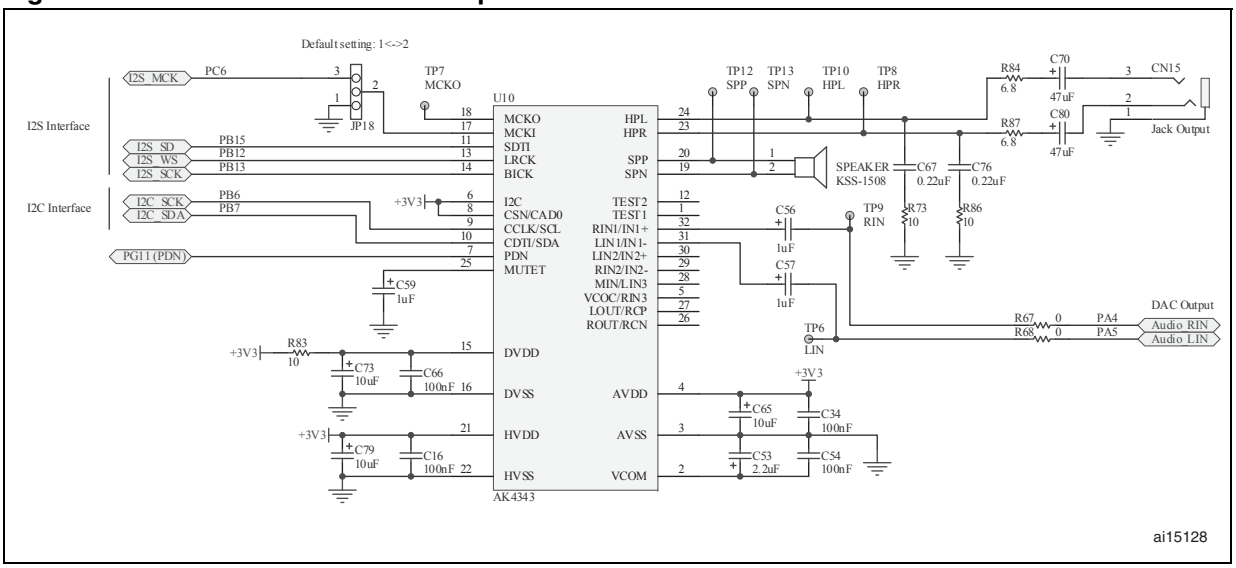
- program the audio output (speaker or headphone) and input (analog input or digital I^2S data, etc.).
- enable or disable the master clock feature and, set the reference clock for internal and sampling operations.
- set the digital volume level, the mute status and the digital filter coefficients.

The codec can operate in different modes. The modes allowed by the hardware board implementation are listed below:

- PLL Slave mode: the internal codec clock is derived, with an internal PLL, from an external clock. The external clock can be either the bit clock (SCK) or the channel clock (WS) (this mode requires a high clock accuracy on SCK/WS clock).
- EXT slave mode: no PLL is used and the internal clock is derived from the MCLK input clock (at 256 × F_S frequency rate).

Figure 4 illustrates the hardware implementation schematic and how the codec is connected to the STM32F103xx and the board components.

Figure 4. Audio codec hardware implementation



When the MCLK clock is enabled (by configuring the relative configuration register), MCKI jumper (JP18) should be configured in the 2<>3 position in order to drive the MCLK signal from the STM32F103xx I^2S interface to the codec MCLKI pin.

The default codec configuration used in this application note is:

- I^2S standard: I^2S Phillips (can be changed to MSB or LSB standards)
- MCLK clock enabled at 256 × F_S frequency rate and PLL disabled. (Disabling the MCLK clock might lead to a reduced audio quality due to the high accuracy needed on the alternative clocks: WS or CK)
- The default output is the headphone but the configuration can be changed before or during the application execution (using the up/down joystick push-buttons).

Codec configuration steps

- Power up the codec (supply voltage) and activate it (pull the PDN pin high). This operation causes all the codec registers to reset.
- Select the audio protocol (Phillips, MSB or LSB) by writing to Mode Control 1 register at address 0x04.
- Select the MCLK rate (256 × F_S) if the MCLK mode is configured, by writing to the Mode Control 2 register at address 0x05.
- Power up the internal modules of the codec by writing to the Power Management 1 register at address 0x00.
- To set the PLL mode, write to the Power Management 2 register at address 0x01.
- Supply the main clock to the codec (either the MCLK or the SCK/WS clocks) by sending dummy data (this clock must be supplied through all subsequent operations).
- Choose the audio output device:
  - select headphone by writing to the Mode Control 4 register at address 0x0F
  - select speaker by writing to the Signal Select register at address 0x02
- Configure the digital volume by writing to the Lch Digital Volume Control and Rch Digital Volume Control registers at addresses 0x0A and 0x0D, respectively.
- Power up:
  - the speaker by writing to the Power Management 1 register at address 0x00
  - the headphone by writing to the Power Management 2 register at address 0x01
- Exit the Mute mode by writing to the Power Management 1 register at address 0x00.
- Send the audio data through the I^2S interface and stop I^2C communications.

In the `I2S_CODEC` driver file, a single function performs the codec configuration:

```
CODEC_Config(u16 OutputDevice, u16 I2S_Standard, u16 I2S_MCLKOutput, u8 Volume)
```

2.2.2 STM32F103xx and board configuration

The STM32F103xx peripherals used for this application are: I2S2 for audio communication, I2C1 for codec configuration and the memory interface (could be FSMC for NOR Flash memory or SPI1 for SPI_Flash memory, etc.).

- Since the MCLK feature is enabled, configure MCKI Jumper (JP18) in the 2<>3 position to connect the I2S2 MCLK signal to the MCLKI codec pin.
- If the audio file to be played has a big size, it should be previously loaded into the memory source (NOR Flash memory or SPI Flash) using an independent application (IAP, DFU, etc.). It may also be included as a table file.

For more details, refer to the STM3210E-EVAL evaluation board user manual on www.st.com.

2.3 Firmware description

This application note is based on:

- the STM32F10xxx firmware library
- the `I2S_CODEC` driver firmware (offering the main functions required to control the codec and I^2S environment for an audio application)
- a specific firmware to call the `I2S_CODEC` driver functions, as well as other functions required for control and display (`main.c` and `stm32f10x_it.c files`).

The user may build any similar application using the same library and driver, and different interfacing firmware/hardware.

2.3.1 `I2S_CODEC` driver firmware description

The user may interface the audio codec directly through the driver application layer. The driver functions are summarized in the following sections. *Table 1* presents the general driver file organization.

Table 1. Driver library description

File	Description
`i2s_codec.h`, `i2s_codec.c`	– I^2S and codec definitions, type definitions and function prototypes – Basic functions (init, read, write, play, pause, stop, etc.).

High-level functions

These are the functions that can simply be called by the final application to execute all needed configurations and perform high-end functionalities (like playing a wave sound, pausing playing, configuring all the hardware components, etc.).

These functions are presented in *Table 2*

Table 2. `I2S_CODEC` driver high-level functions

Function name	Description
`I2S_CODEC_Init`	Initializes the entire application environment (I^2S, I^2C, codec, memory)
`I2S_CODEC_ReplayConfig`	Sets the number of replays (number of stream playing loops ≥ 1 or 0 for infinite loop)
`I2S_CODEC_Play`	Causes the audio file to start playing (or to resume from Paused state)
`I2S_CODEC_Pause`	Pauses the audio stream playing and saves the current position
`I2S_CODEC_Stop`	Causes the audio file to stop playing and resets all local pointers
`I2S_CODEC_ControlVolume`	Increases/Decreases/Sets the digital volume
`I2S_CODEC_Mute`	Causes the codec to mute the released sound.

Table 2. I2S_CODEC driver high-level functions (continued)

Function name	Description
I2S_CODEC_ForwardPlay	Increments the audio file pointer by a fixed step (percentage of the file length) then continues playing.
I2S_CODEC_RewindPlay	Decrements the audio file pointer by a fixed step (percentage of the file length) then continues playing.

Only the most relevant functions will be detailed in the following sections.

- **I2S_CODEC_Init function**

  This function implements all the needed initialization for the I^2S, the codec and the memory interfaces and peripherals.

Table 3. I2S_CODEC_Init function

Function name	I2S_CODEC_Init
Prototype	u32 I2S_CODEC_Init(u32 OutputDevice, u32 Address)
Behavior description	Initializes the I^2S and the I^2C peripherals, the codec, the memory and the audio file.
Input parameters	OutputDevice: used to set the output device, can be: – OutputDevice_SPEAKER – OutputDevice_HEADPHONE Address: specifies the address of the audio file.
Output parameter	None
Return value	– 0: if all initializations were successful – 1: Memory initialization failure – 2: Audio file failure (wrong file or wrong format) – 3: I^2C communication failure
Required preconditions	None
Called functions	*NVIC_Config, GPIO_Config, AudioFile_Init, I2S_Config, CODEC_Config*

This function calls some subfunctions related to each component:

– NVIC_Config: Configures the I^2S interrupt channel

– GPIO_Config: Configures the I^2S, I^2C, memory- and codec-related GPIOs.

– AudioFile_Init: Initializes the memory then reads the audio file header, checks if it is compliant with the supported audio format and sets the related parameters (audio file length and frequency) used to configure the other peripherals.

– I2S_Config: Configures the I^2S peripheral according to the previous parameters. Standard and MCLK output are configured using two constants: *I2S_STANDARD* and *I2S_MCLKOUTPUT*, defined in the i2s_codec.c file.

– CODEC_Config: Configures the codec using the same configuration parameters as I2S_Config, and using the OutputDevice input parameter (from I2S_CODEC_Init), that is either OutputDevice_SPEAKER or OutputDevice_HEADPHONE. The digital volume is set at a default level defined in the i2s_codec.h file as DEFAULT_VOL.

Table 4 lists all the parameters that have to be set in order to correctly configure the application (some are automatically set by the `I2S_CODEC_Init` function, others are constants defined in the driver files).

Table 4. Configuration parameters list

Parameters	Values	Location	Default value
`OutputDevice`	– OutputDevice_SPEAKER – OutputDevice_HEADPHONE	`I2S_CODEC_Init` function Input	OutputDevice_HEADPHONE
`Address`	Any value in respect to the memory size.	`I2S_CODEC_Init` function Input	AudioFileAddress = 0x6406 0000 (variable in `i2s_codec.c` file)
`I2S_STANDARD`	– I2S_Standard_Phillips – I2S_Standard_MSB – I2S_Standard_LSB	Constant in `i2s_codec.h` file:	I2S_Standard_Phillips
`I2S_MCLKOUTPUT`	– I2S_MCLKOUTPUT_Enable – I2S_MCLKOUTPUT_Disable	Constant in `i2s_codec.h` file	I2S_MCLKOUTPUT_Enable
`DEFAULT_VOL`	– Any value from 0xFF(Mute) and 0x00 (Max volume).	Constant in `i2s_codec.h` file.	0x48
`I2S_AudioFreq`	– 8000, 16000, 22050, 44100 or 48000.	Automatically detected: Set by `AudioFile_Init` function	--
`AudioDataLength`	– Total Length of the current audio file stream.	Automatically detected: Set by `AudioFile_Init` function.	--
`DataStartAddr`	– First audio data offset into the current audio file (corresponds to the header length).	Automatically detected: Set by `AudioFile_Init` function.	--

- **`I2S_CODEC_ReplayConfig` function**

  This function sets the number of times the audio file stream will be repeated (replayed) each time its end is reached.

Table 5. `I2S_CODEC_ReplayConfig` function

Function name	`I2S_CODEC_ReplayConfig`
Prototype	`void I2S_CODEC_ReplayConfig(u32 Repetions)`
Behavior description	Sets the number of current audio stream repetitions to Repetitions value (input).
Input parameter	Repetitions: any number ≥ 1 or 0 for infinite replay.
Output parameter	None
Required preconditions	None
Called functions	None

- **`I2S_CODEC_Play` function**

  This function starts playing the audio file from a programmable position.

Table 6. `I2S_CODEC_Play` function

Function name	`I2S_CODEC_Play`
Prototype	`u32 I2S_CODEC_Play(u32 AudioStartPosition)`
Behavior description	Sets the memory read address to AudioStartPosition and enables the I^2S interrupt to begin sending audio data to the codec. (Update the status of the current audio file to playing).
Input parameter	AudioStartPosition: first address of the audio data to be played in the current stream (relatively to the audio file, after the end of the audio file header, and not relatively to the memory).
Output parameter	None
Required preconditions	None
Called functions	None.

- **`I2S_CODEC_ControlVolume` function**

  This function controls the digital volume level in accordance with the input parameters (increases, decreases or sets a volume level).

Table 7. `I2S_CODEC_ControlVolume` function

Function name	`I2S_CODEC_ControlVolume`
Prototype	`u32 I2S_CODEC_ControlVolume(u32 direction, u8 volume)`
Behavior description	Depending on the direction value, increases, decreases or sets a volume level defined by the volume variable.
Input parameter	– direction: `VolumeDirection_HIGH` or `VolumeDirection_LOW`, respectively to increment/decrement the current volume by a step: “volume” value. Or `VolumeDirection_LEVEL` to set the volume value to “volume“. – volume: any step value (from 0xFF and 0x01) or any level value from 0xFF (Mute) and 0x00 (Max volume)
Output parameter	None
Required preconditions	None
Called functions	None.

Medium-level functions

Medium-level functions are used to better control some basic functionality and the audio stream. *Table 8* presents these functions.

Table 8. `I2S_CODEC` medium-level driver functions

Function name	Description
`I2S_CODEC_DataTransfer`	Sends the audio data using the SPI2 peripheral and checks the audio play status (if a (Pause/Stop) command is pending) by calling the `I2S_CODEC_UpdateStatus` function. This function should be called in the SPI2 TXE interrupt subroutine when the interrupt mode is used.
`I2S_CODEC_UpdateStatus`	Checks if the current status (through a local variable: AudioPlayStatus) is Playing, Paused or Stopped. If the commanded status is Stopped or Paused the function performs the requested command and updates the local variables.
`GetVar_DataStartAddr`	Returns the current audio Data Start Address value for the current stream (depends on the audio file type and the file header length).
`GetVar_AudioDataIndex`	AudioDataIndex is the virtual audio data pointer for the audio file being played. This function returns the current value of the AudioDataIndex.
`ResetVar_AudioDataIndex`	Resets the AudioDataIndex to the current audio file Data Start Address value.
`IncrementVar_AudioDataIndex`	Increments the AudioDataIndex of the current audio file by a programmable value step (input of the function).
`GetVar_AudioPlayStatus`	Returns the current AudioPlayStatus variable value (Playing, Stopped, Paused).

Low-level functions

Some low-level functions provide flexible management of the memory (SPI Flash/NOR, etc.). Other types of memories can be supported simply by replacing the body of these low-level functions by the corresponding functions for the memory.

Example: the `Media_Init` function should contain the entire memory initialization procedure. The application can just call to corresponding memory driver function (that is the `NOR_Init()` function).

Table 9. `I2S_CODEC` low-level driver functions

Function name	Description
`Media_Init`	Calls the initialization procedure for the used memory medium.
`Media_StartReadSequence`	Enables reading from the memory medium.
`Media_ReadByte`	Reads and returns a byte from the memory medium.
`Media_ReadHalfWord`	Reads and returns a half word from the memory medium.
`Media_BufferRead`	Reads a buffer of bytes from the memory medium (used to read the wave file header).

Driver firmware flowcharts

Figure 5 and *Figure 6* illustrate the driver functionality.

Figure 5. Driver's functional flowchart 1

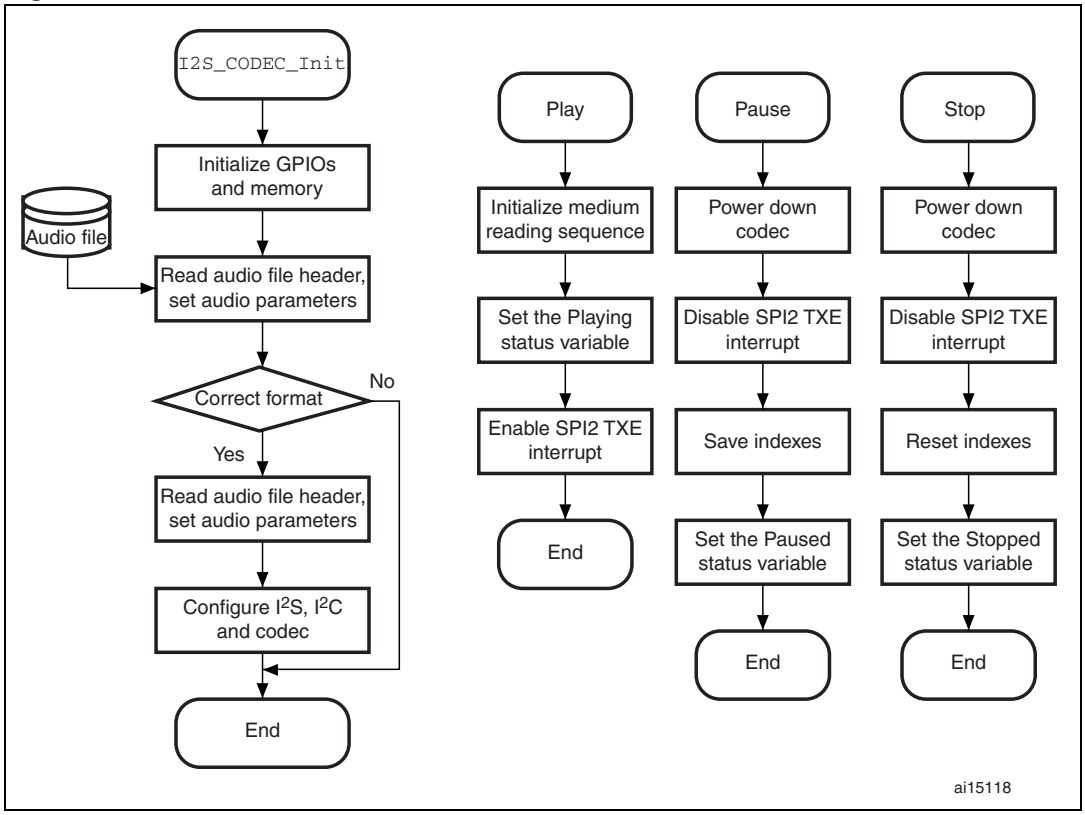
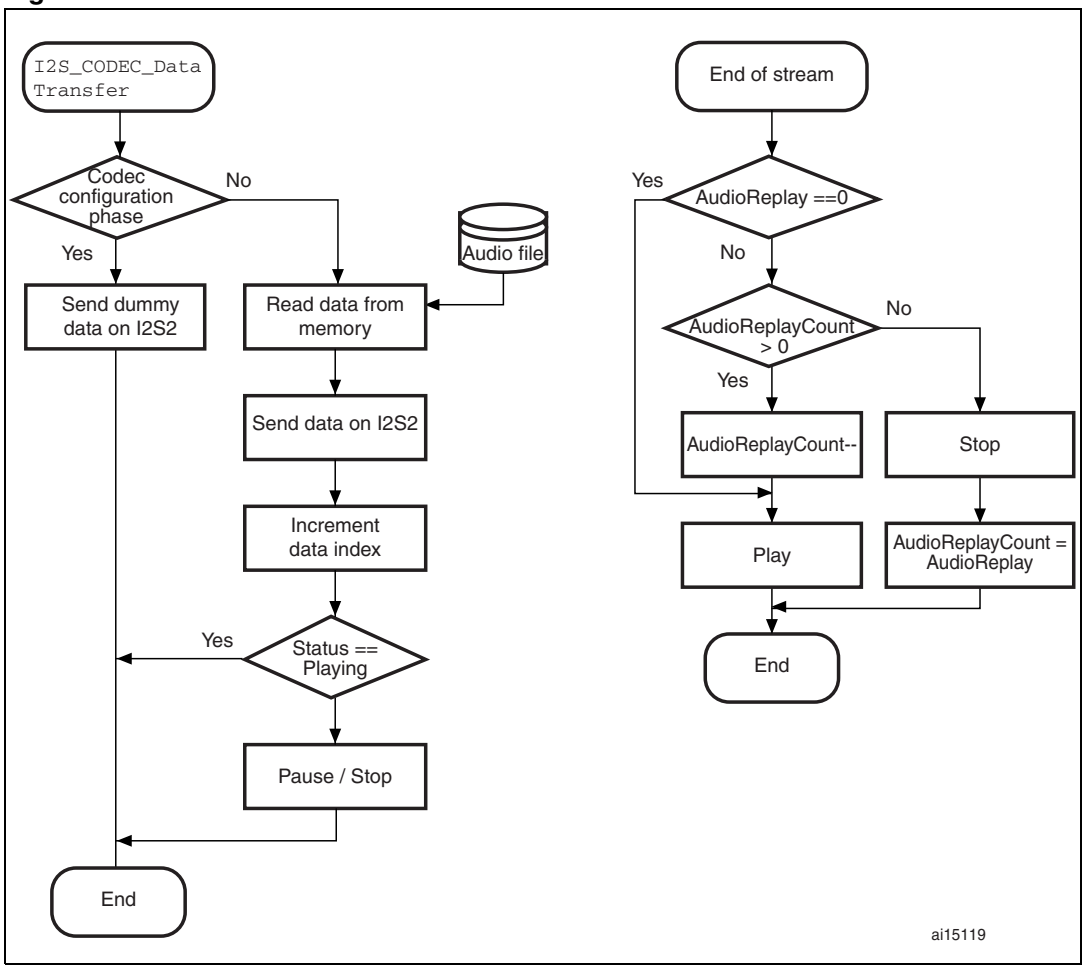


Figure 6. Driver's functional flowchart 2



2.3.2 Demo firmware description

The demo uses the `i2s_codec` driver functions to control the audio stream. Other functions are implemented to make the application interactive.

Two main functions are handled by the application:

- Joystick and push-button input control
- LCD display output (all LCD-related functionalities are enabled only when the `USE_LCD` constant is defined in the `main.h` file by uncommenting the following line: `#define USE_LCD)`.

`Joystick_Config` function

The push-button-related GPIO and EXTI configuration is performed by the `Joystick_Config` function (instantiated in the `main.c` file).

Table 10. `Joystick_Config` function

Function name	`Joystick_Config`
Prototype	`u32 Joystick_Config(void)`
Behavior description	Configures and initializes the EXTI, NVIC and GPIOs related to the Joystick and the Key push-buttons.
Input parameter	None
Output parameter	None
Required preconditions	None
Called functions	None.

The functionalities allowed by this configuration are explained in *Table 11.*

Table 11. Description of the Joystick functionalities

Button	Function	Pin location	EXTI line
Key button	– Play: if the stream is paused. – Pause: if the stream is Playing.	PG8	EXTI8
Joystick Up button	– Set Speaker as audio output device and begin playing the file: if the stream is Paused or Stopped. – Increase volume: if the stream is Playing.	PG15	EXTI15
Joystick Down button	– Set Headphone as audio output device and begin playing the stream: if the stream is Paused or Stopped. – Decrease volume: if the stream is Playing.	PD3	EXTI3
Joystick Left button	Forward and play: if the stream is Playing.	PG14	EXTI14
Joystick Right button	Rewind and play: if the stream is Playing.	PG13	EXTI13
Joystick Select button	Stop: if the stream is Playing.	PG7	EXTI7

`LCD_Update` function

The LCD configuration and the current audio stream information display operations are performed by the `LCD_Update` function as detailed in *Table 12*.

Table 12. `LCD_Update` function

Function name	`LCD_Update`
Prototype	`void LCDUpdate(u32 Status)`
Behavior description	Updates the information displayed on the LCD according to the parameter Status. The information is gathered from the audio stream status and the codec driver variables.
Input parameter	Status: determines which information has to be updated (all other information remains unchanged). It can be: – STOP, PLAY, PAUSE to update the control button menu lines and the status line menu. – FRWRWD to update the progress bar in case of forward or rewind. – VOL to update the volume bar. – PROGRESS to update the progress bar periodically. – ALL to initialize the LCD and update all the information at once. These values are defined in the main.h file.
Output parameter	None
Required preconditions	None
Called functions	`LCD_DisplayStringLine, GetVar_CurrentVolume, GetVar_AudioDataIndex, I2S_CODEC_LCDConfig.`

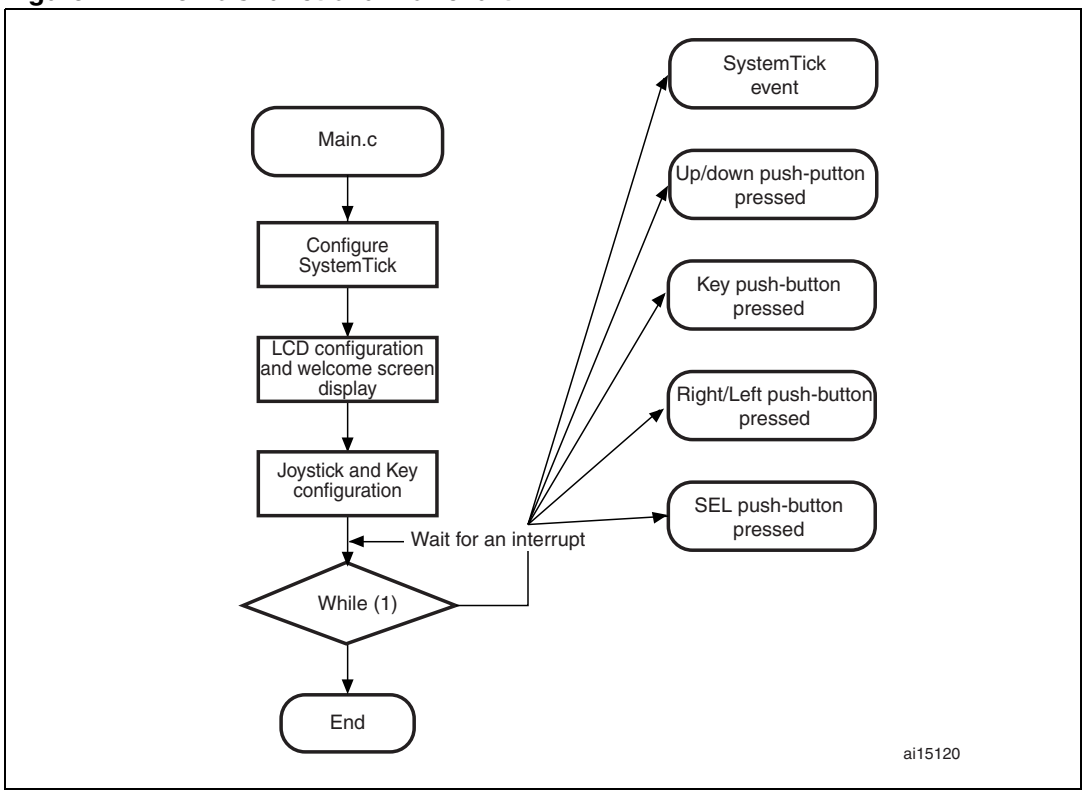
The example program implements the i2s_codec driver in a basic application that consists in the following:

- Initialize the LCD and the displayed information
- Configure the Joystick and Key push-button GPIOs and related EXTI.
- Wait for an interrupt to execute one of the actions described in *Table 11*.
- Every time an interrupt is generated by pushing one of the buttons, the dedicated operation is executed and the LCD display is updated according to the changed parameter.
- Every time ~1% of the playing stream elapses an interrupt is generated by the SystemTick timer and the Progress bar is computed. The LCD-related information is then updated.

Demo firmware flowcharts

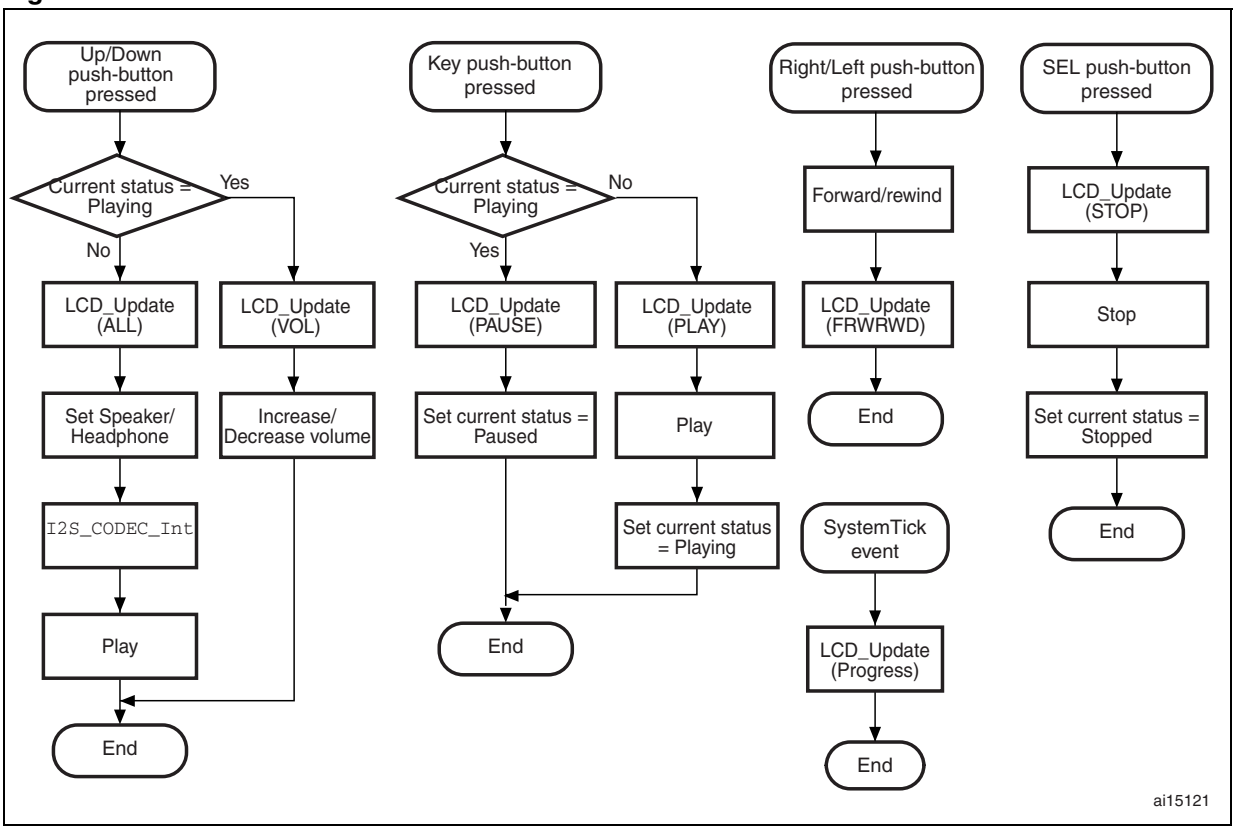
Figure 7 presents the overall function flowchart.

Figure 7. Demo's functional flowchart 1



After performing the configuration steps, the demo program enters an infinite loop, and only interrupt subroutines are executed (the program can perform other tasks in the mean time). Each interrupt subroutine performs an action depending on the current stream status (Playing, Paused or Stopped) as detailed in *Figure 8*.

Figure 8. Demo's functional flowchart 2



Error messages

If an error occurs during the initialization phase, the `LCD_DisplayError` function is called (instantiate in `main.c` file). A message may be displayed on the LCD to determine the error source:

- **Memory error**: the *"ERROR: Memory ->RST"* message is displayed, meaning that the memory initialization failed and a system reset is needed to recover.
- **Audio file error**: the *"ERROR: File ->RST"* message is displayed, meaning that the audio file initialization failed or the audio file format is not supported. A system reset is needed to recover.
- **I²C communication error**: the *"ERROR:I2C com. ->RST"* message is displayed, meaning that communication with the codec through the I²C interface failed. A system reset is needed to recover.

Audio file loading

In order to use the application note firmware, an audio file with the allowed specifications (refer to *Section 2.4*), has to be loaded into the used memory medium (the default memory medium is the NOR Flash memory interfaced through the FSMC peripheral).

For this purpose, one of the three following procedures may be performed:

- Use the audio file from the demo delivered with the STM3210E-EVAL STMicroelectronics evaluation board. The audio file is included in the DFU image of the demonstration firmware and the file address to be used is the default address (AUDIO_FILE_ADDRESS = 0x6406 0000 constant in `main.h` file, corresponding to the NOR memory).

- Use the STMicroelectronics DFU demonstration firmware to compile any audio file and load it into the memory (for more details, refer to the DFU section in the STM3210E-EVAL evaluation board demonstration user manual). Once this DFU image is loaded, the file address has to be set (using the AUDIO_FILE_ADDRESS constant in the `main.h` file) and the application note firmware can then run correctly.
- Use a different application to load a wave file into the appropriate memory. The file address has to be stored and used as input when calling the `I2S_CODEC_Init` function. If the file is in Mono format, make sure that the first address is an even number.

2.3.3 Timing considerations

Except during the configuration phase, all the programs run through interrupt subroutines. This may be convenient to implement other tasks or to add some options (additional displays, computing, audio encoding/decoding, etc.). Thus, some timing considerations are to be respected.

The audio stream sampling frequency is the major timing factor. But audio frequencies are many times slower than the frequencies of other communication IPs (for instance I^2C runs at up to 400 kHz, SPI at up to 18 MHz, etc.), which means that timing constraints are not very tough.

The program must be able to release an audio data (16 bits) every 8/16/22.05/44.1/48 kHz. This timeout includes the reading of data from the memory, the update of the stream status, the increment of indexes and the information display update. If other tasks are implemented (such as decoding operations), the period of time taken by all these operations has to be inferior to the period between two data-sending operations (two SPI TXE interrupts in I^2S mode).

Interrupt priority

- Set the SPI interrupt (TXE in I^2S mode) as the highest-priority NVIC channel (if another interrupt takes a very short and deterministic time, it can have higher priority. Example: the systick interrupt).

Memory accesses

Mainly, two operations should be as short as possible:

- Memory read operation: depending on the memory type, and the memory interface configuration, reading the audio data from the file located in the memory could be a long operation. In some cases, it may be necessary to optimize this operation (add buffers, increase interface frequency, use DMA, etc.).
- Display update: this operation consists in writing data into the LCD RAM. Write access to this RAM may also be long when many information are to be updated. This operation is optimized in the demo application (group/ungroup information, create a separate event for each information update, etc.).

Function calls

- In the interrupt file (`stm32f10x_it.c`), most handlers perform calls to `i2s_codec` driver functions or `main.c` functions. In order to optimize time, some calls can be replaced by the body of the function itself (with specific modifications, depending on the application requirements).

2.4 General, allowed parameters

Audio file

Supported audio configuration is:

- **Format**: PCM
- **Channel number**: Stereo/Mono[b]
- **Data length**: 16 bits
- **Frequency**: 8 kHz, 16 kHz, 22.05 kHz, 44.1 kHz, 48 kHz (depending on the system clock frequency)
- **Length**: depends on the memory used and the program memory usage (16 MB available on NOR Flash memory and8 MB available on SPI Flash memory).

SPI peripheral

The SPI peripheral used for I^2S communication is SPI2 and the relative allowed parameters are:

- **Mode**: I^2S master.
- **Standard**: I^2S Phillips, MSB, LSB[c]
- **Master Clock**: Enabled or disabled (depends on the system clock, the audio frequency and the audio file type (speech, music, etc.)).
- **Frequencies**: 8 kHz, 16 kHz, 22.05 kHz, 44.1 kHz, 48 kHz.
- **Data format**: 16-bit data length in 16-bit packets.

b. This configuration only depends on the codec. If a different codec is used, the Mono format may not be available.

c. This configuration only depends on the codec. If another codec type is used, this parameter may be different.

3 Conclusion

This application note explains the basics to build an interactive audio playing application using the STMicroelectronics evaluation board with an audio codec, a speaker, a headphone jack and an optional LCD. The application may be enhanced by adding/removing some options depending on the requirements.

4 Revision history

Table 13. Document revision history

Date	Revision	Changes
30-May-2008	1	Initial release.