AVAS architecture based on AutoDevKit

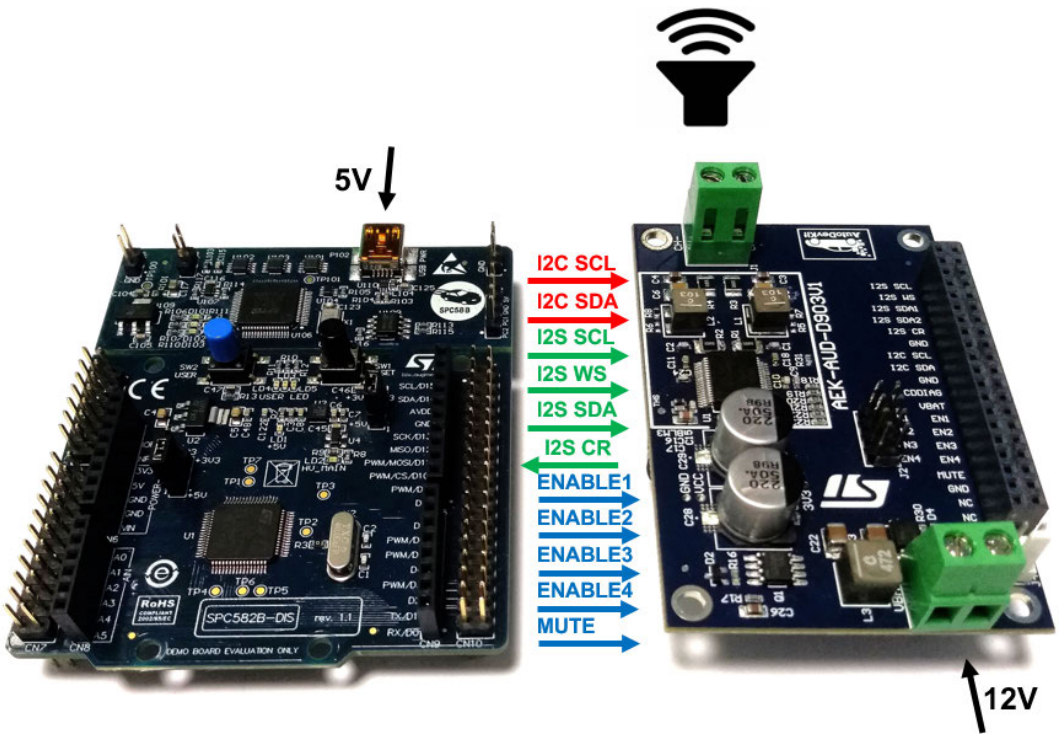
Introduction

The AutoDevKit Acoustic Vehicle Alerting System (AVAS) consists of an AEK-MCU-C1MLIT1 Discovery board, an AEK-AUD-D903V1 evaluation board, and appropriate speakers. The AEK-MCU-C1MLIT1 board MCU monitors and controls the FDA903D power amplifier on the AEK-AUD-D903V1 board via I²C and I²S serial interfaces and GPIOs.

The MCU board and the audio board can be wired together directly or via a connector board designed to simplify the process.

The AEK-MCU-C1MLIT1 board is supplied 5 V through its mini-USB connector, while the AEK-AUD-D903V1 can either be supplied low voltage (from 3.3 V to 18 V) or standard voltage (from 5 V to 18 V).

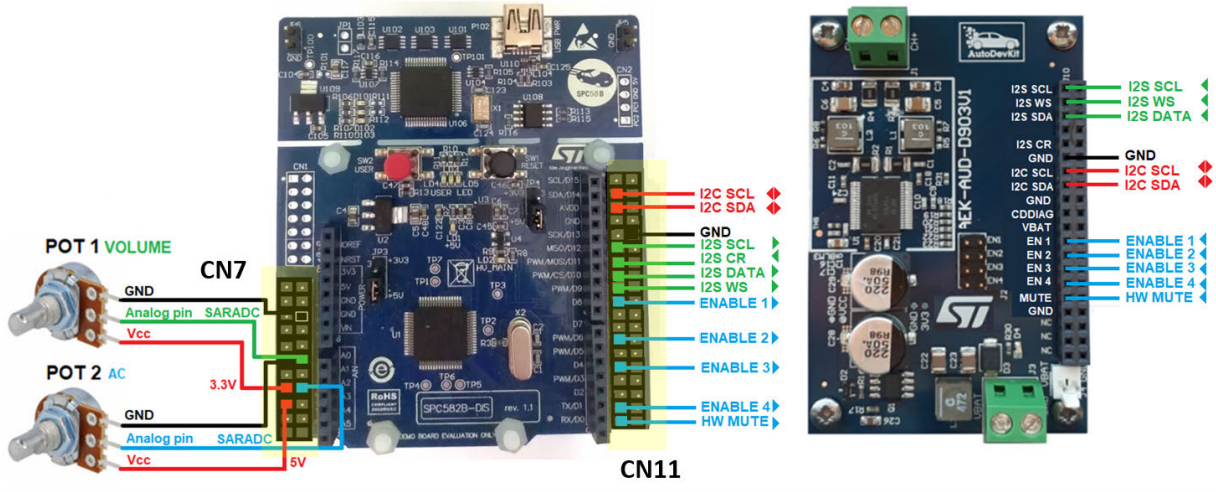
Figure 1. AVAS system AutoDevKit control board and audio board



The hardware is fully supported by a software ecosystem, which includes SPC5-STUDIO development environment, SPC5-UDESTK-SW software for debugging and STSW-AUTODEVKIT Eclipse plugin containing AEK-AUD-D903V1 driver and sample application codes.

1 AVAS system hardware

Figure 2. **AVAS Demo hardware and connections**



2 AEK-MCU-C1MLIT1 Discovery board audio support

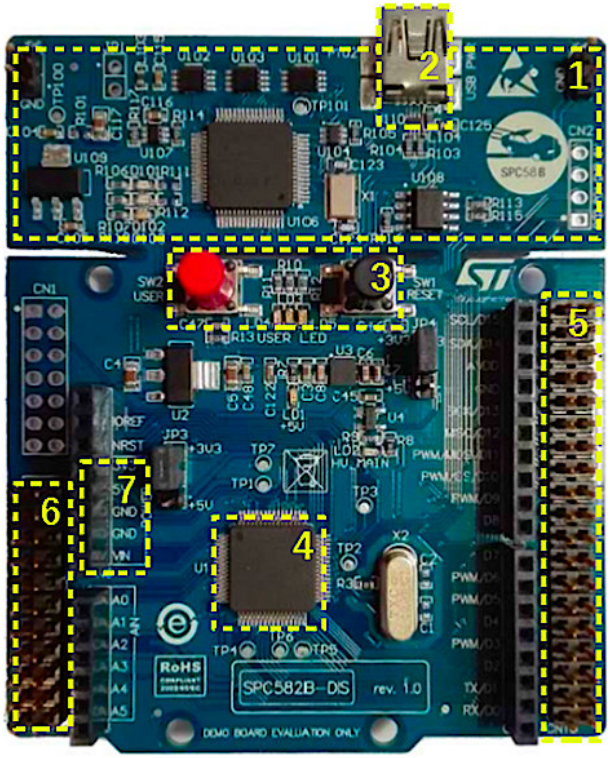
The AEK-MCU-C1MLIT1 Discovery evaluation board features the SPC582B60E1 automotive microcontroller with high performance e200z2 single core 32-bit CPU with 80MHz clock, 1088 KB Flash and 96 KB SRAM in an eTQFP64 package. The I²S (simulated by an SPI port), I²C port and GPIOs provide the necessary signal and communication lines to control a class D power amplifier.

The board also integrates a programmer/debugger interface based on the UDE PLS software, allowing the user to program the microcontroller and debug software applications. The integrated debugger software is available through ST's free integrated development environment, SPC5-STUDIO. To download the debugger software and to activate the license, refer to the PLS website.

Note: Arduino connectors are not mounted on this board and are not required for the audio application.

Figure 3. AEK-MCU-C1MLIT1 Discovery board components

1. PLS programmer/debugger
2. USB power connector to supply 5V and load firmware
3. User interface with three LEDs and two buttons
4. 32-bit SPC582B60E1 MCU
5. CN10 19x2 connector for access to I²C and I²S ports and GPIOs
6. CN7 11x2 connector for access to I²S ports and GPIOs
7. CN6 connector allows supplying the board with different external voltage (3.3 V, 5 V or 12 V)



The SPC582B60E1 microcontroller includes the following additional features:

- 1088 KB (1024 KB code flash + 64 KB data flash) on-chip flash memory: supports read during program and erase operations, and multiple blocks allowing EEPROM emulation
- Comprehensive new generation ASIL-B safety concept:
  - ASIL-B of ISO 26262 – FCCU for collection and reaction to failure notifications
  - Memory Error Management Unit (MEMU) for collection and reporting of error events in memories.

- 1 enhanced 12-bit SAR analog-to-digital converter:
  - Up to 27 channels (two channels are used in the AVAS application for sound volume and acceleration)
  - enhanced diagnostic feature.
- I²C interface
- 4 serial peripheral interface (DSPI) modules (a DSPI is used in the AVAS Demo to simulate the I²S bus interface).

2.1 I²S bus interface on the SPC582B60E1 microcontroller

The FDA903D audio amp receives the audio signal from the flash blocks of the SPC582B60E1 via the I²S bus. This interface can transmit two different audio channels on the same data line. As SPC5 microcontrollers do not have a native I²S interface, an emulation through the DSPI protocol is implemented.

2.1.1 I²S protocol details

The I²S bus consists of the following lines:

I2S SCL The clock signal frequency is the product of the sampling frequency and the number of bits transmitted.

I2S DATA The transmitted data are coded in two's complement, and the MSB (Most Significant Bit) is therefore in the first position of each word. The data word is composed of 32 bits.

Note: The device only processes the first 24 most significant bits and disregards the least significant 8 bits.

I2S WS The Word Select signal is synchronized with the sampling frequency. Its digital value identifies the transmission channel (0 = right channel, 1 = left channel).

2.1.2 I²S emulation on DSPI for SPC5 MCU control of FDA903D amplifier

The FDA903D power amplifier allows audio playback at the following sampling frequencies:

- 44.1 kHz
- 48 kHz
- 96 kHz
- 192 kHz

The maximum DSPI clock limit can only support the lowest frequency (f_s = 44.1 kHz).

DSPI is a synchronous serial communication interface primarily used for short-distance communication in embedded systems. This interface is based on four signals:

SCLK: the serial clock signal from the master (the microcontroller in our application)

MOSI: the serial data from the master to the slave (the FDA903D in our case)

MISO: the serial data from the slave to the master

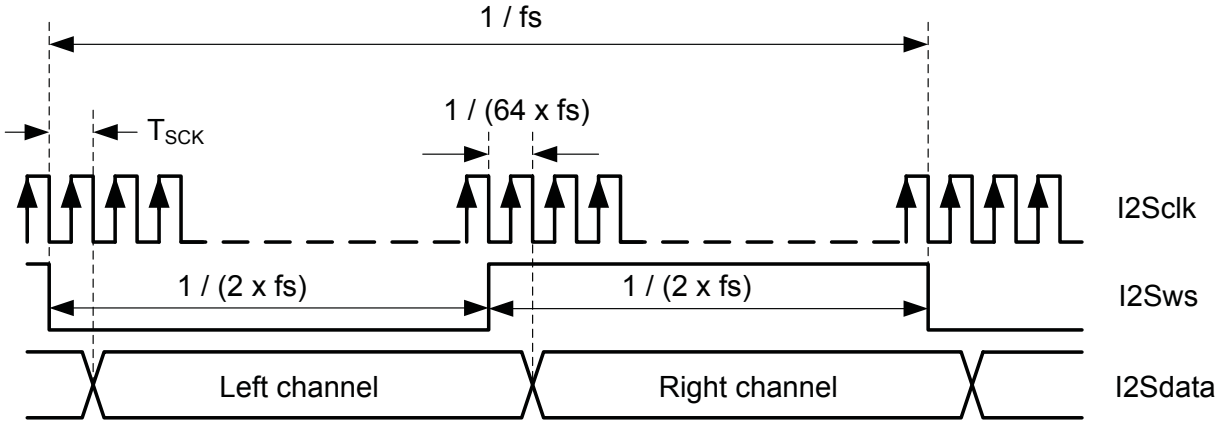
CS: selects which slave chip receives the message from the master

DSPI emulation of the I²S interface is therefore obtained through the following associations and parameter values:

- I2S DATA → DSPI MOSI
  - 32-bit data word
- I2S WS → DSPI CS
  - varies the channel (right or left) according to the f_s (sampling frequency).
- I2S SCLK → DPSI SCLK
  - $Frequency = number of channels \times number of bits in a word \times sampling frequency = 2x32x44.1kHz = 2.822\ MHz$.

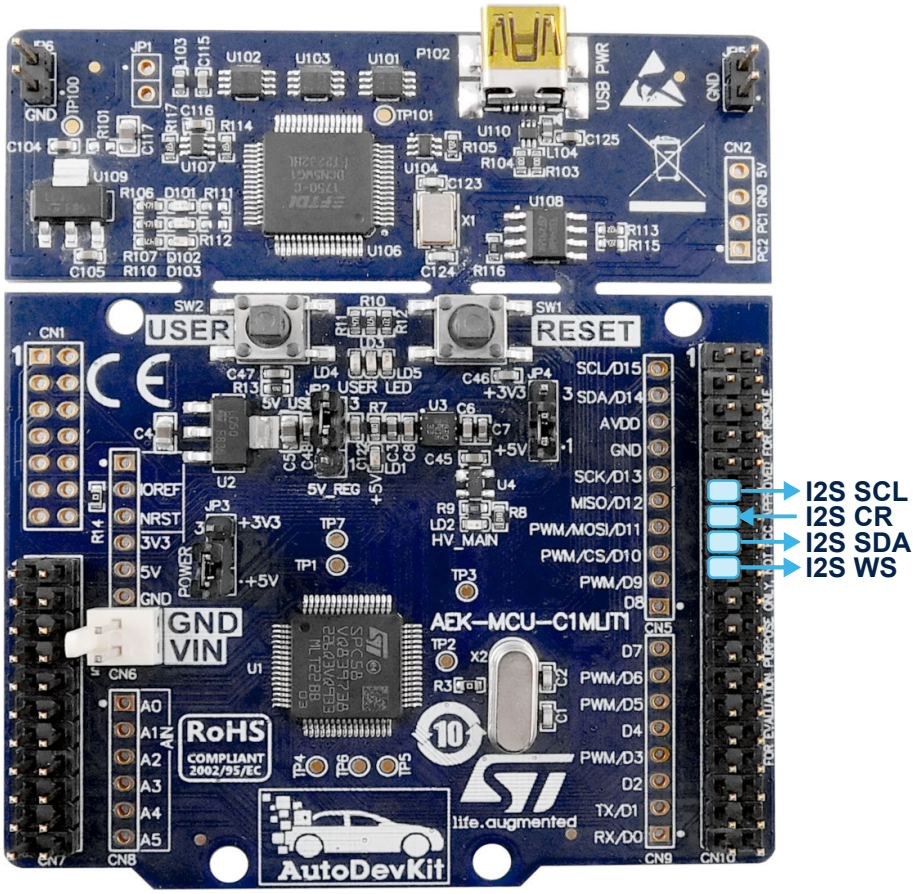
- I2S TEST → DSPI MISO
  - This additional signal allows the FDA903D to send real-time current sensing information to the microcontroller and to a DSP for sound processing.

Figure 4. Standard I²S data format



- MCU DSPI0 port access via four CN10 connector pins
- MCU has four DSPI ports

Figure 5. Connector CN10 pins for DSPI0



RELATED LINKS

Refer to TN1296: "I²S emulation on DSPI" for more information about emulating the I²S protocol

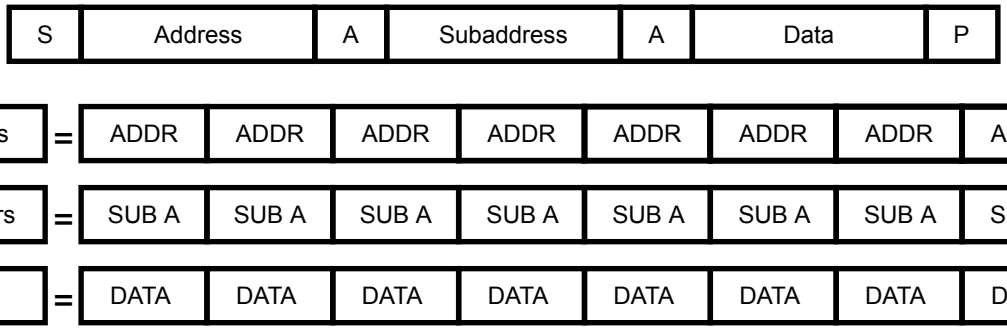
2.2 I²C bus interface on the SPC582B60E1 microcontroller

The I²C interface is used to control, program and request information from the audio amp. Data transmission from SPC582B60E1 to the FDA903D and vice versa takes place through the two-wire I²C bus interface for the SDA and SCL lines.

Note: According to the I²C protocol, it is mandatory to insert pull-up resistors to positive supply voltage on the SDA and SCL lines.

Figure 6. I²C typical data format

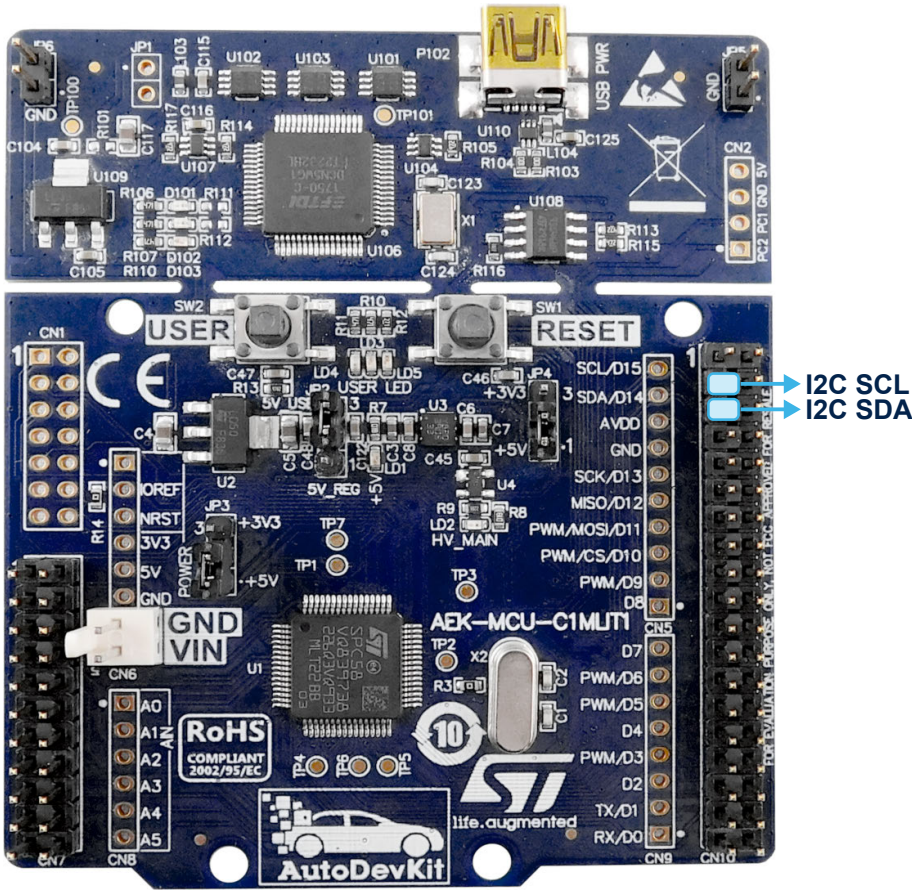
[S] Start bit
Chip address byte
Sub-address byte
[data] n-byte + Acknowledge bit
[P] Stop bit



The AEK-MCU-C1MLIT1 provides I²C port access through two pins on the CN10 connector shown in the figure below.

The discovery board has a single dedicated I²C port. Additional ports can be added by emulating the I²C protocol via software to configure a GPIO pin for I²C SCL and another pin for I²C SDA.

Figure 7. **Connector CN10 pins dedicated to I²C**

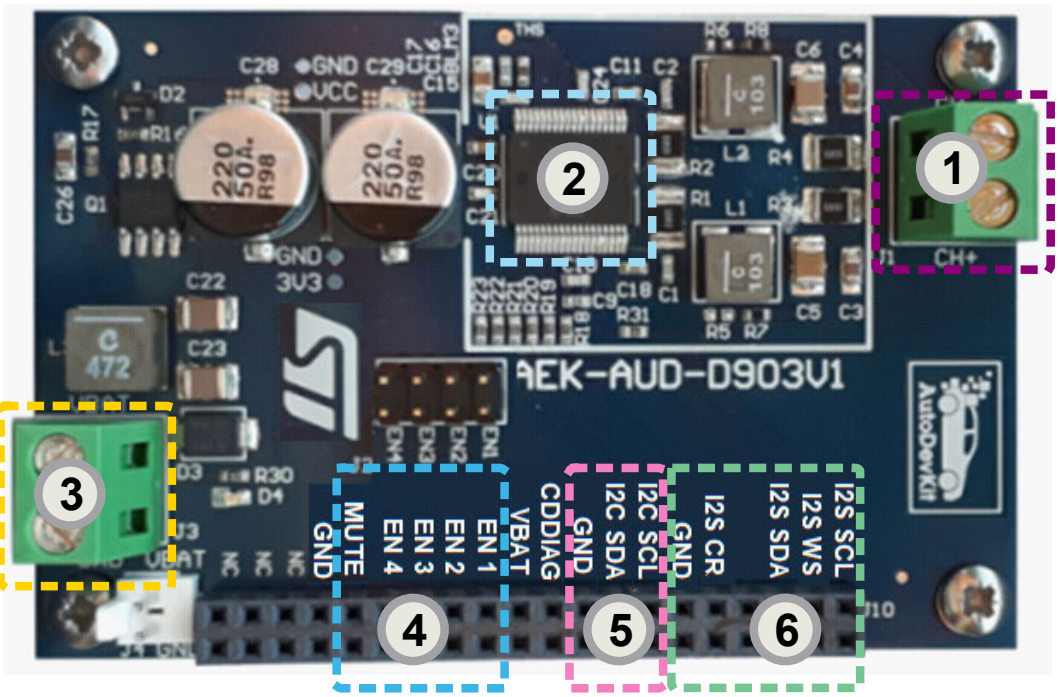


3 AEK-AUD-D903V1 evaluation board for automotive power amplifier

The AEK-AUD-D903V1 is designed to allow evaluation and application development based on the embedded FDA903D automotive digital class D power amplifier in a PowerSSO-36 slug-down package.

Figure 8. AEK-AUD-D903V1 main components and interfaces

1. Output channel; 2. FDA903D power amplifier; 3. Power supply connector
4. Enable and HW mute pins: [EN1 to EN4]: 4 pins can be configured to switch on the amplifier and assign it on of 7 possible an I²C addresses, [MUTE]: allows MUTE setting control of the power amplifier through a GPIO
5. I²C interface: [I2C SCL]: I2C clock line, [I2C SDA]: I2C data line
6. I²S interface: [I2S SCL]: I2S clock line, [I2S WS]: I2S Word select line, [I2S SDA]: digital input, [I2S CR]: I2S Output test current, [GND]



The FDA903D power amplifier can be configured through its I²C bus interface and the device includes the following diagnostics suite designed for automotive applications:

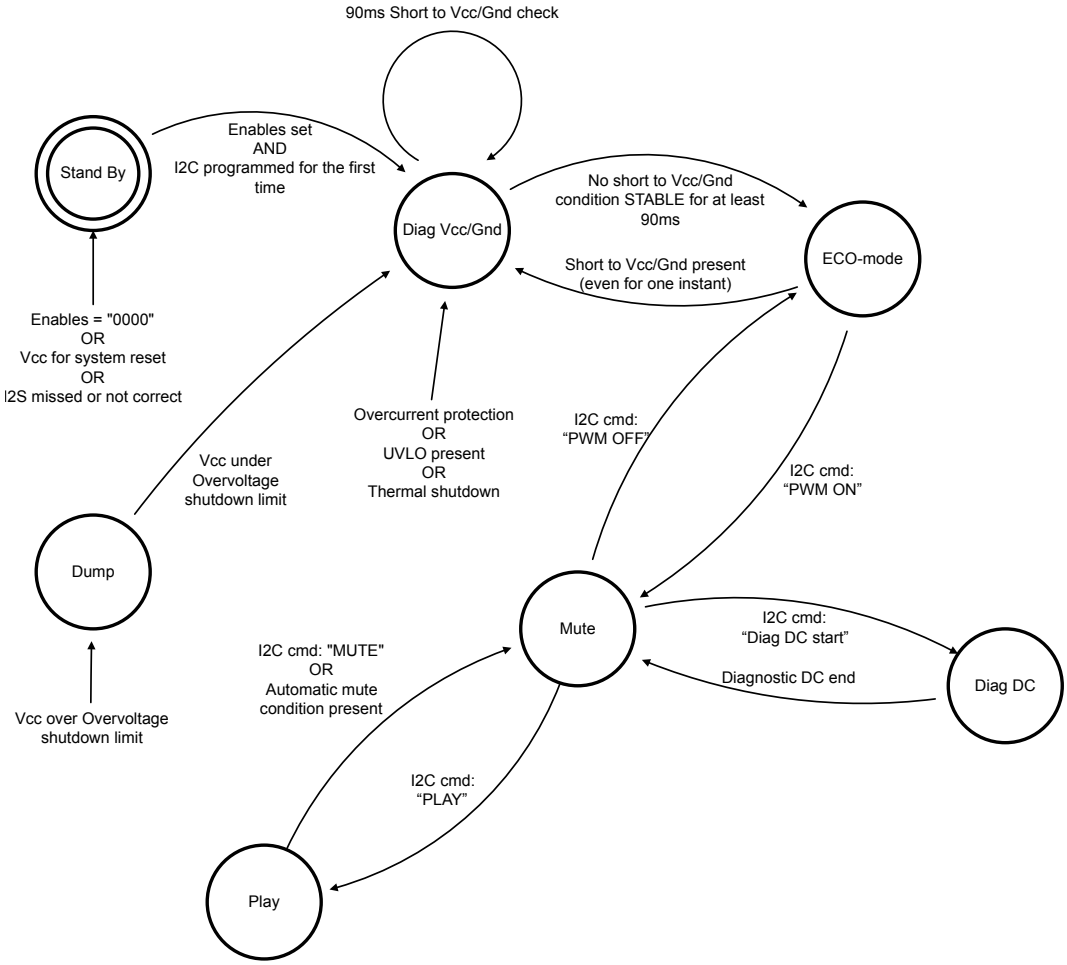
- open load in play detection
- DC Diagnostic in MUTE to monitor the load status
- short to V_{CC} / GND diagnostic
- digital Input Offset detection
- output Voltage Offset detection
- output Current Offset detection
- thermal protection

The FDA903D features a configurable power limiting function and can be optionally operated in legacy mode without I²C communication.

3.1 FDA903D finite state machine

The FDA903D finite state machine (FSM) describes how the device reacts to system and user inputs.

Figure 9. FDA903D state machine



The initial standby state of the device cannot be exited until the I²C interface has been correctly enabled by providing the correct supply voltage, the I²S clock, the I²S data and a valid combination of enable pins in order to determine the I²C device address.

Table 1. I²C device address combinations

Configuration	Pin			
	Enable 1	Enable 2	Enable 3	Enable 4
Standby	0	0	0	0
Amplifier ON address 1 = '1110000'	0	1	0	0
Amplifier ON address 2 = '1110001'	1	1	0	0
Amplifier ON address 3 = '1110010'	0	0	1	0
Amplifier ON address 4 = '1110011'	0	1	1	0
Amplifier ON address 5 = '1110100'	0	1	0	1
Amplifier ON address 6 = '1110101'	1	1	0	1
Amplifier ON address 7 = '1110110'	0	0	1	1
Amplifier ON address 8 = '1110111'	0	1	1	1

When a valid combination of Enable 1/2/3/4 is recognized, the device turns on all the internal supply voltages and outputs are biased to V_{CC} / 2. The internal I²C registers are preset in the default condition until the I²C next instruction. A return to the Standby condition (all the enable pins set to 0) resets of the amplifier. The finite state machine shows that a reset is also triggered if PLL is not locked, I²S is missing or not correct, or V_{CC} is removed.

There are also four possible legacy mode combinations for device operation without using the I²C interface.

Table 2. Legacy mode Enable configurations

Configuration	Pins			
	Enable 1	Enable 2	Enable 3	Enable 4
Legacy mode: low voltage mode; in-phase	1	1	1	0
Legacy mode: low voltage mode; out-phase	1	1	1	1
Legacy mode: standard voltage mode; in-phase	1	0	0	0
Legacy mode: standard voltage mode; out-phase	1	0	0	1

Note: FDA903D can only work in I²C slave mode; any combination other than those indicated are invalid.

3.1.1 FDA903D FSM state descriptions

Standby The ENABLEx pins set the I²C addresses and start up the system; if ENABLE1/2/3/4 are all low ("0000"), then the FDA903D is off, the outputs remain biased to ground and the current consumption is limited.

Diagnostic Vcc-Gnd This state checks the device is in a safe operating condition, with no short to ground (Gnd), short to V_{CC}, overcurrent, undervoltage ($UVLO_{Vcc}$), or thermal shutdown. The FDA903D moves to the next Eco-mode if none of these faults occur for at least 90 ms. A stable fault is communicated to the user via I²C messages after 90 ms. While in Diagnostic Vcc-Gnd state, FDA903D can receive all the I²C commands but will not turn the PWM on.

ECO-mode The amplifier is fully operational and can receive and execute any valid command. Output switching is disabled for low power consumption. The device can move from ECO-mode to the MUTE state in order to activate switching within about 1 ms and without experiencing POP-noise. This facilitates fast transition from ECO-mode to PLAY.

MUTE The FSM transitions from ECO-mode to the MUTE state through the I²C command to turn on PWM. The MUTE state allows quick transition to PLAY and diagnostic states.

PLAY The FSM transitions to this state from MUTE via the I²C “PLAY” command, and the same status register bit governs the return from PLAY back to MUTE. Certain external conditions such as low battery mute, high battery mute, hardware mute pin and thermal mute automatically return the amplifier to the MUTE state.

Diag DC This state starts the DC diagnostic routine to detect the load connection status and returns to the MUTE state when the routine has finished.

Note: I²C commands performed by the user are executed via the I²C protocol by modifying the device register settings.

RELATED LINKS

Refer to the FAD903 datasheet for more information regarding its state machine

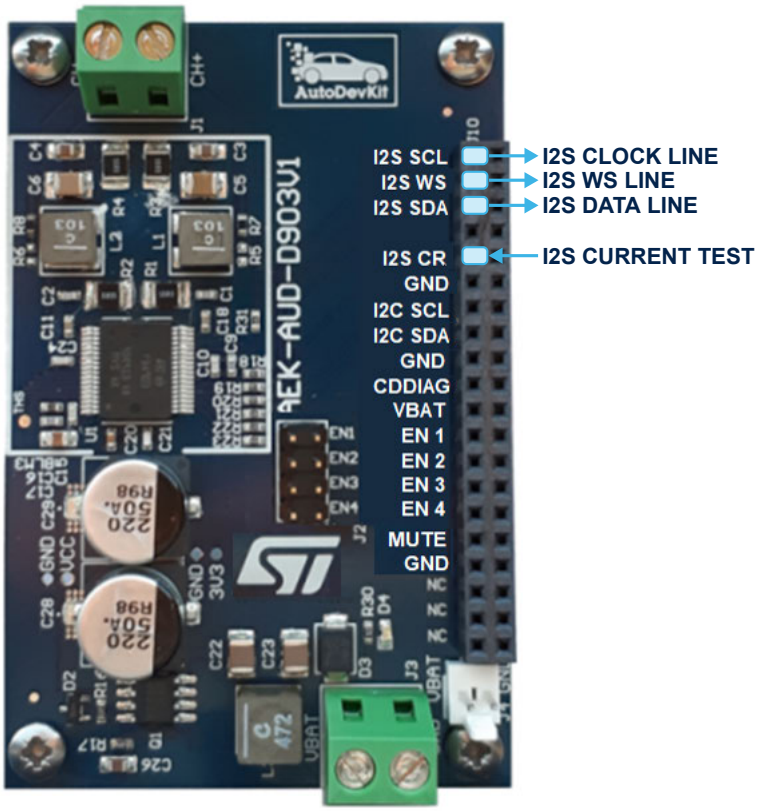
3.2 FDA903D I²S protocol

Audio data is transmitted to the power amplifier via the I²S protocol. The 32-bit data word is in two's complement representation starting from the MSB. The device only processes the first 24 most significant bits and disregards the 8 least significant bits.

Note: Besides the standard I²S used in our demo, the FDA903D device also supports Time Division Multiplexing mode (TDM).

The FDA903D internal PLL locks on the I²S clock line signal frequency, which is why it is important to configure the I²S bus appropriately. When the I²S clock is missing or corrupted, the PLL unlocks and the device is forced into a standby state.

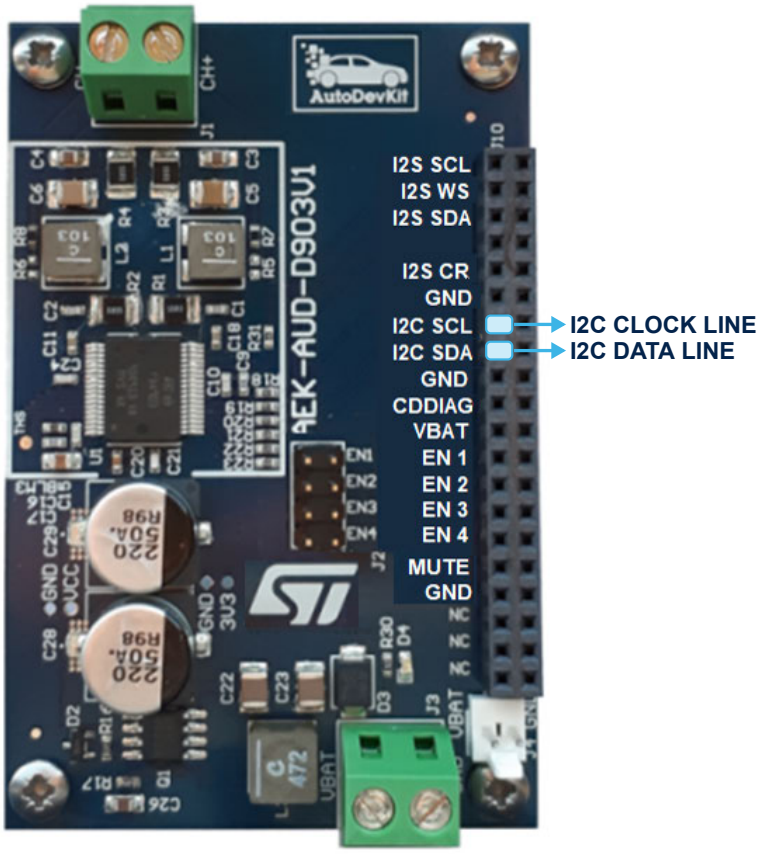
Figure 10. I²S (DSPI) connection in AEK-AUD-D903V1



3.3 FDA903D I²C protocol

The DATA and SCLK wires for the I²C protocol are used to communicate, control and manage the FDA903D. Connection between the I²C microcontroller port and I²C power amplifier pins on the AEK-AUD-D903V1 is provided by the pins on the connector shown below.

Figure 11. **I²C connection in AEK-AUD-D903V1**



The power amplifier FDA903D is controlled with appropriate read and write operations on Instruction Bytes registers (from IB0 to IB14) performed with the I2C protocol. Additional Data Bytes registers (from DB0 to DB6) in the device record the state of the amplifier.

Writing to the instruction registers and reading from the device status registers are the fundamental elements of device management.

3.3.1 I²C protocol writing procedure

Communication through the I²C protocol takes place via a well-defined sequence of bit packages: start bit → recipient address → acknowledge bit → sub-address → acknowledge bit → actual data → stop bit.

The amplifier address is chosen from eight possible enable pins combinations that represent eight corresponding addresses. For example, to assign I²C address1 = “1110000” to the device, enable pin 2 is set high (Enable 2 = “1”) and enable pins 1,3 and 4 are set low (Enable1 = “0”, Enable3 = “0”, Enable4 = “0”).

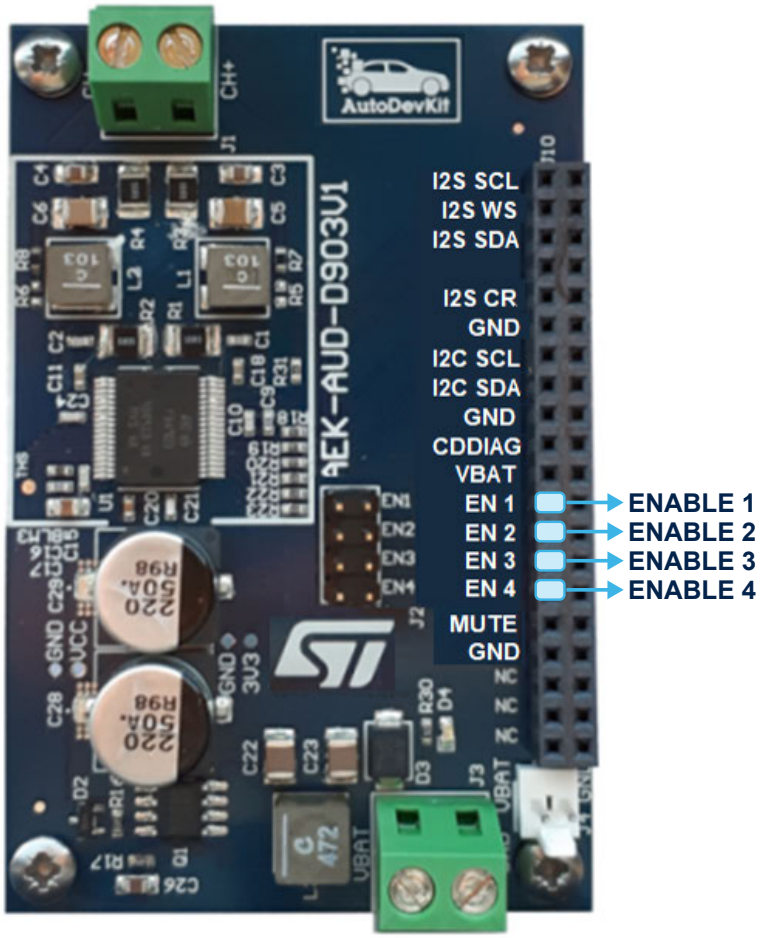
Table 3. **I²C address 1 selection**

Configuration	Pin			
	Enable 1	Enable 2	Enable 3	Enable 4
Standby	0	0	0	0
Amplifier ON address 1 = ‘1110000’	**0**	**1**	**0**	**0**
Amplifier ON address 2 = ‘1110001’	1	1	0	0
Amplifier ON address 3 = ‘1110010’	0	0	1	0
Amplifier ON address 4 = ‘1110011’	0	1	1	0
Amplifier ON address 5 = ‘1110100’	0	1	0	1

Configuration	Pin			
	Enable 1	Enable 2	Enable 3	Enable 4
Amplifier ON address 6 = ‘1110101’	1	1	0	1
Amplifier ON address 7 = ‘1110110’	0	0	1	1
Amplifier ON address 8 = ‘1110111’	0	1	1	1

The connector on the AEK-AUD-D903V1 provide access to the four enable pins by four corresponding GPIO pins on the microcontroller.

Figure 12. ENABLE pin locations on the connector



The subaddress is assigned according to the IB register to be written, as shown in the following table.

Table 4. Subaddress association

Register name	Corresponding Subaddress
IB0	I0000001
IB1	I0000010
IB2	I0000011
IB3	I0000100

Register name	Corresponding Subaddress
IB4	I0000101
IB5	I0000110
IB6	I0000111
IB7	I0001000
IB8	I0001001
IB9	I0001010
IB10	I0001011
IB11	I0001100
IB12	I0001101
IB13	I0001110
IB14	I0001111

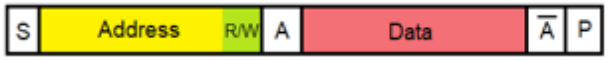
In the above table, bit 7 of the subaddress is the letter “I” to represent the possibility of having an incremental writing procedure. If the “I” bit is set to 1, the write operation is performed from the corresponding register and all consecutive ones with a unique flow of data from I²C. The process can involve all registers or can be interrupted by a stop bit received from I²C.

The data bits carry the actual information required to control the power amplifier.

3.3.2 I²C protocol: reading procedure

The reading procedure consists of the device address (sent by master) and the data (sent by slave).

Figure 13. Read operation packet



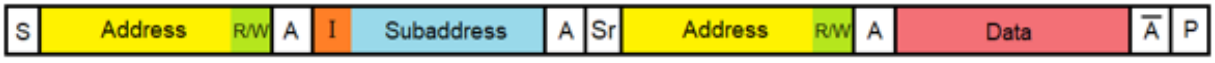
When a reading procedure is performed, the first register read is the last addressed in a previous access to I²C peripheral. Hence, the reading of a register is enabled by a write action (a write interrupted after the sub-address is sent) to specify which register must be read. The following figure shows the complete procedure to read a specific register where:

1. The master performs a write action by sending only the device address and the subaddress; the transmission must be interrupted with the stop condition after the subaddress.
2. The master starts a new communication by sending the device address and the FDA903D slave responds by sending the data bits.
3. The read communication is ended by the master which sends a stop condition preceded by a not-acknowledge.

Figure 14. Read operation required data

Alternatively, performing a start immediately after the stop condition can be used to generate the repeated start condition (Sr), which also keeps busy the I²C bus until the stop is reached.

Figure 15. Read operation with repeated start condition



3.3.3 IB registers in I²C

The microcontroller accesses all amplifier functionality through the IB registers.

Table 5. IB register map

IB register map							
D7	D6	D5	D4	D3	D2	D1	D0

IB0

D7: enable/disable writing on IB registers
D6-D5: enable/disable the I²S standard protocol for transmitting the digital input
D4-D1: choose between right or left channel
DO: select between low voltage and standard voltage modes

IB1

D7-D6: select the I²S WS
D4-D3: select the PWM switching frequency based on the I²S WS value
D2: select between PWM amplifier dithered or not dithered
DO: select between PWM in phase or out of phase

IB2

D7-D6: establish the short to supply diagnostic timing
D4: activate/deactivate the low radiation function
D3-D0: enable and configure the amplifier power limiter

IB3

D5: enable/disable output voltage offset detector
D4: enable/disable input offset detector
D3: enable/disable output current offset detector
D2: enable/disable high pass filter in DAC amplifier DAC
D1: enable/disable noise gating
D0: enable/disable open load in play detection

IB4

D7: enable CDDIAG to report presence of output voltage offset
D6-D4: enable CDDIAG to report temperature warnings
D3: enable CDDIAG to report overcurrent faults
D2: enable CDDIAG to report input offset
D1: enable CDDIAG to report short to V_{CC} or to Gnd fault
D0: enable CDDIAG to report high voltage Mute fault

IB5

D7: enable CDDIAG to report undervoltage fault
D6: enable CDDIAG to report thermal shutdown fault
D5-4: enable CDDIAG to report PWM pulse skipping

IB6

D7-D6: establish MUTE timing setup
D5: select audio signal gain control
D4: choose between standard gain or low gain

IB7

D7-D6: select the diagnostic ramp time

D5-D4: select the diagnostic hold time

D1: choose between data generated on I²S clock falling edge or rising edge

D0: select the current sensing protocol configuration

IB8

D7-D6: set the full current sensing scale

D5: turn on/off the PWM

D4: enable the DC Diagnostic

D3-D1: configure the I²S CR pin

D0: put the amplifier in MUTE/PLAY

IB9

D4: enable/disable the watchdog for word select management

IB10

D7: set short load impedance threshold for DC diagnostic

D6: set open load impedance threshold for DC diagnostic and open load in play

D4-D3: configure the output current offset detector threshold

IB11

D5-D4: select the overcurrent protection level

D3: select between default PWM or PWM Slow slope

IB12

D7: select between standard thermal warning or thermal warning shift - 15 °C

IB13

D6: select whether digital mute is enabled or disabled in PLAY when Start Analog Mute without thermal warnings occurs

IB14:

D4: set feedback on LC filter/Out

D3-D1: configure the LC filter setup

D0: select whether or not setup is programmed via 1²C

3.3.4 DB registers in I²C

DB registers allow the microcontroller to monitor the status and operation of the power amplifier.

Table 6. DB register map

DB register map							
D7	D6	D5	D4	D3	D2	D1	D0

DB0

D7: indicates whether an offset at input is present

D6: indicates whether the current offset test has ended and if it is valid

D5: indicates whether an offset at current offset is present

D3: indicates whether an offset at voltage offset is present

D2: indicates whether the open load in play test has ended

D1: indicates whether the open load in play test is valid

Do: indicates whether an open load is present or not

DB1

D7-D4: indicates whether the thermal warning is active

D3: indicates whether the PLL is locked

D2: indicates whether an undervoltage UVLOALL has been detected

D1: indicates whether an overvoltage shutdown has been detected

D0: indicates whether PWM pulse skipping has been detected

DB2

D7: indicates whether the DC diagnostic pulse has ended

D6: indicates whether the DC diagnostic is valid

D5: indicates whether the overcurrent protection has been activated

D4: indicates when a short load on channel occurs

D3: indicates when a short to V_{CC} on channel occurs

D2: indicates when a short to Gnd on channel occurs

D1: indicates when an open load on channel occurs

D0: indicates whether the channel is in MUTE or in PLAY

DB3

Reserved for DC Diagnostic Error codes

DB4

Register is reserved for Channel Current Sensing (1O - 8)

DB5

Register is reserved for Channel Current Sensing (7 - 0)

DB6

D7: indicates whether the high voltage mute has started

D6: indicates whether an undervoltage $UVLOV_{CC}$ has been detected

D5: indicates whether a thermal shutdown has been detected

D4: indicates whether the analog mute is started

D2: indicates whether the watchdog for word select occurs

D1: indicates whether an error frame occurs

3.3.5 Driver

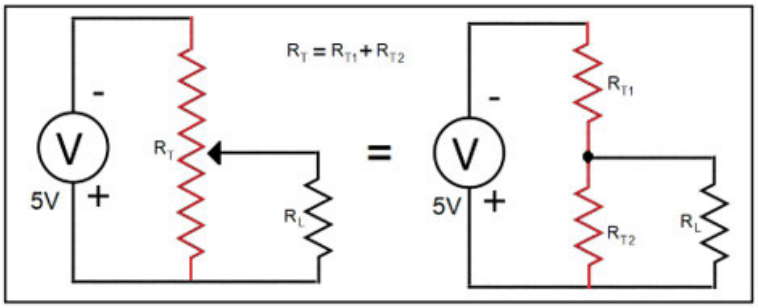
A driver has been developed to allow the user to monitor and control the amplifier without engaging in tedious IB and DB register read and write operations associated with a task.

3.4 Potentiometers

The AVAS system includes two potentiometers to help simulate the sound of a car engine: one to simulate the accelerator pedal and another to adjust the sound volume. The potentiometers are powered through two supply voltages (5V and 3.3V) from the AEK-MCU-C1MLIT1 control board via female connector CN6 or male connector CN7.

Our system uses the potentiometer as a voltage divider to obtain a manually adjustable output voltage from a fixed input voltage applied across the two ends of the potentiometer. It is formed by an insulating cylinder on which a metal wire is wound, and the two ends are connected to two terminals. One of these terminals is connected to a sliding contact that runs the length of the cylinder. The operation is equivalent to a pair of resistors in series whose total value is constant, but individually variable according to the position of the sliding contact.

Figure 16. **Linear potentiometer circuit**



Considering R_L an open, we have the voltage on R_{T2} equal to the power supply voltage of the potentiometer multiplied by $\frac{R_{T2}}{R_T}$, and since this ratio is equal to that of L_{T2} (R_{T2} resistor length) on L_T (total resistor length), we see that the output voltage of the potentiometer is a function of the cursor position.

$$V_2 = \frac{V^* L_{T2}}{L_T} \quad (1)$$

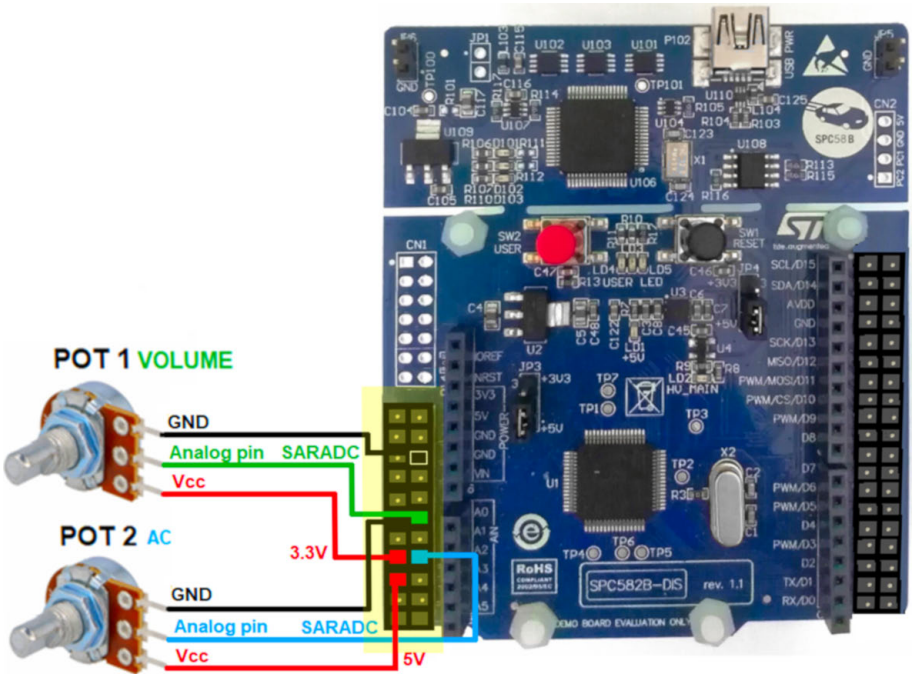
It is possible to implement speed and volume control by directly relating these variables to the output voltage of a potentiometer. The analog output of the potentiometers are converted into discrete values by the SPC582B60E1 microcontroller ADCs.

3.5 Successive approximation analog-to-digital converter (SARADC)

Two of the 27 SARADC channels on the AEK-MCU-C1MLIT1 control board microcontroller are used to convert the potentiometer speed and volume signals into digital quantities through the connector CN7.

Note: These signals can also be routed through CN11.

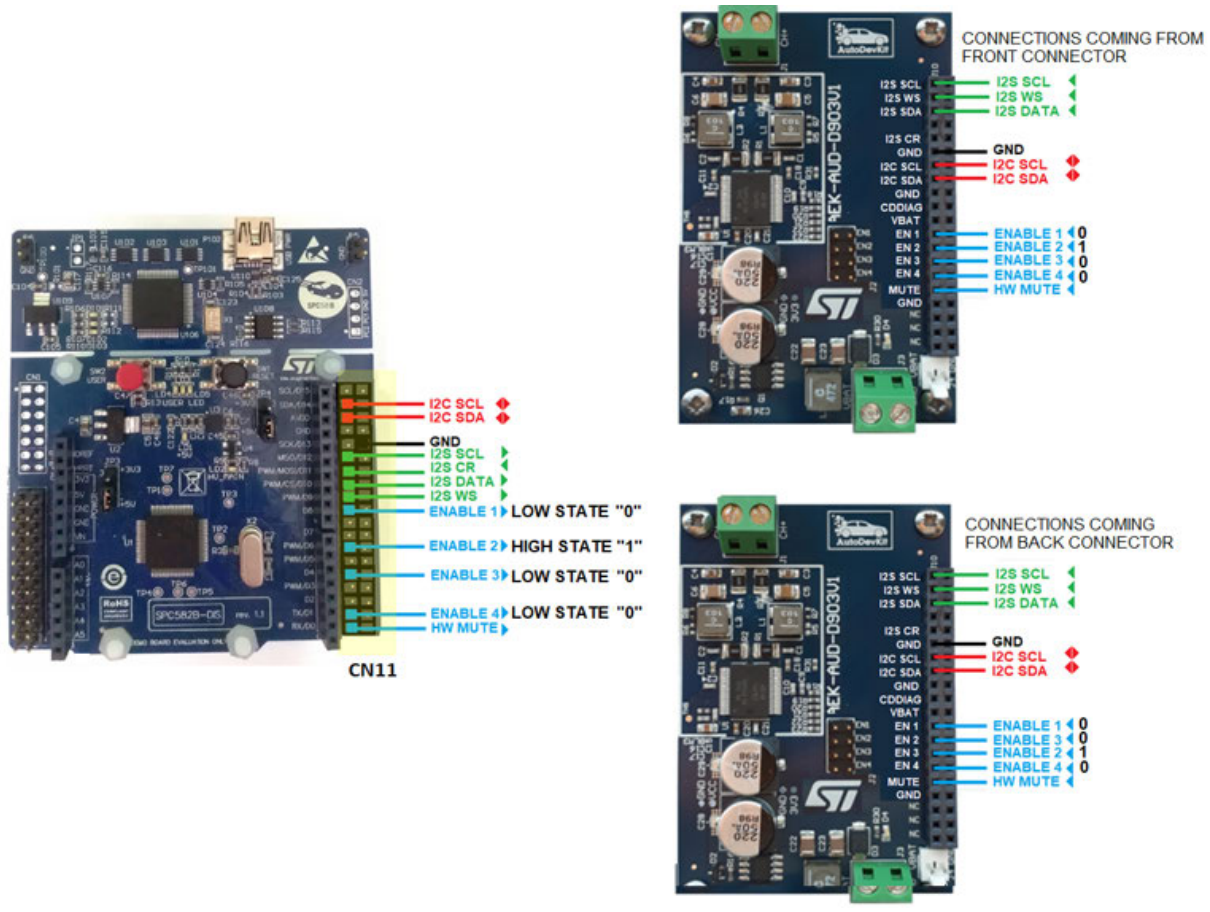
Figure 17. **Potentiometer connections**



3.6 Stereo mode

In order to produce stereo audio, the system requires a second AEK-AUD-D903V1 board to occupy both the left and right channels available on the I²S DATA line.

Figure 18. **AVAS system for two stereo sound**

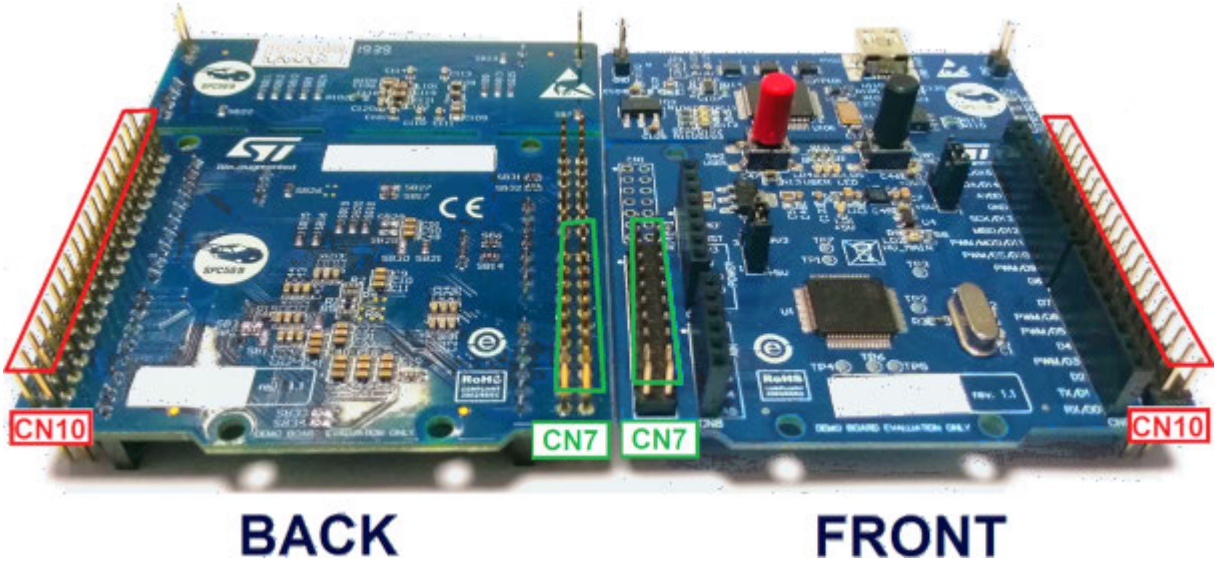


The connection of a second audio board will involve the following modifications to the AVAS system:

- The I²C interface is shared, so the I2C SCL clock line and the I2C DATA line are connected to both AEK-AUD-D903V1 audio boards.
- The I²S interface is also shared as the lines (I2S SCLK, I2S WS and I2S DATA) are also used by the second amp. The right channel and the left channel travel on the same line, and the I2S WS distinguishes the information for the right channel and the left channel.
- The I²C communication between the microcontroller (master) and the two amplifiers (slaves) are distinguished by the addresses that identify the two devices.
- The address that identifies each of the two amplifiers is obtained through a combination of the four enables, so eight GPIOs are required in total.

This AEK-MCU-C1MLIT1 control board has a copy of the male CN7 and CN10 connectors on the back of the board, which makes it relatively easy to split the connections for the I²C and I²S interfaces between the two amps.

Figure 19. **AEK-MCU-C1MLIT1 seen on both sides**



Even though it would normally take eight GPIOs to assign addresses to the two amplifiers, we can use the copy of the connectors on the back side of the control board to halve the number of GPIOs. To do this, we use two combinations that have the same number of pins to put high ("1").

Table 7. **Comparison of pin settings for addresses 1 and 3**

Configuration	Pin			
	Enable 1	Enable 2	Enable 3	Enable 4
Amplifier ON address 1 = ‘1110000’	0	1	0	0
Amplifier ON address 3 = ‘1110010’	0	0	1	0

In the above example, where address 1 and address 3 have the same number of high and low pins, it is evident that we can use the same GPIOs to create the two necessary combinations by connecting the GPIO high (“1”) to Enable 2 of the first board and Enable 3 of the second board, and the three low GPIOs (“0”) to the remaining three enable pins on each board.

4 AVAS system software

The AVAS demo system requires the following set of software tools to develop and load the microcontroller firmware to drive and monitor the power amplifier:

- SPC5-STUDIO and SPC5-UDESTK-SW debugger
- STSW-AUTODEVKIT
- AEK-AUD-D903V1 driver

RELATED LINKS

Refer to user manual UM2623 for more information regarding SPC5-STUDIO and STSW-AUTODEVKIT

4.1 SPC5-STUDIO

SPC5-STUDIO is an integrated development environment (IDE) based on Eclipse designed to assist the development of embedded applications based on SPC5 Power Architecture 32-bit microcontrollers.

The package includes an application wizard to initiate projects with all the relevant components and key elements required to generate the final application source code. It also contains straightforward software examples for each MCU peripheral.

Other advantages of SPC5-STUDIO include:

- ability to integrate other software products from the standard Eclipse marketplace
- free license GCC GNU C Compiler component
- support for industry-standard compilers
- support for multi-core microcontrollers
- PinMap editor to facilitate MCU pin configuration

RELATED LINKS

Download the SPC5-UDESTK-SW software to run and debug applications created with SPC5-STUDIO

4.2 STSW-AUTODEVKIT

The STSW-AUTODEVKIT plug-in for Eclipse extends SPC5-STUDIO for automotive applications.

The main advantages of STSW-AUTODEVKIT are:

- integrated hardware and software components, component compatibility checking and MCU and peripheral configuration tools
- allows creation of new system solutions from existing solutions by adding or removing compatible function boards
- Hardware abstraction means new code can be generated immediately for any compatible MCU
- High-level application APIs to control the AEK-AUD-D903V1 board.

The GUI helps configure interfaces, including I²C and I²S, and can automatically manage all relevant pin allocation and deallocation operations.

4.3 AEK-AUD-D903V1.c and sound.c drivers

The AEK-AUD-D903V1.c driver and sound.c library are provided with the STSW-AUTODEVKIT installation to facilitate the programming phase.

4.3.1 AEK-AUD-D903V1.c driver

This driver contains the functions to configure the IB and DB registers of the FDA903D audio amplifier for appropriate system management and control.

Consider the IB8 register below.

Table 8. FDA903D IB8 register description

Data bit	Default value	Definition	
		Bit value	Effect
D7-D6	11	Current Sensing Full scale setting:	
		0	1 A I_{max}
		1	2 A I_{max}
		10	4 A I_{max}
		11	8 A I_{max}
D5	0	0	Channel in TRISTATE (PWM OFF)
		1	Channel with PWM ON
D4	0	0	Channel DC Diag disable
		1	Channel DC Diag start
D3-D1	000	I²S test pin configuration	
		000	High impedance configuration
		001	Reserved
		010	Reserved
		011	Output: Current sensing enable
		100	Reserved
		101	Output: PWM synchronization signal
		110	Reserved
		111	Reserved
D0	0	0	Channel in MUTE
		1	Channel in PLAY

To put the amplifier in PLAY mode, we need to configure the register accordingly:

- turn on the PWM setting IB8[D5] = 1
- put the channel in play setting IB8[D0] = 1

The Initial state is the default 11000000. To reach the state PWM on state, we compute 11000000 OR 00100000 to obtain 11100000. To reach the PLAY mode state, we perform 11100000 OR 0001000 to obtain 11110000.

It takes several operations to modify the relevant bits in the IB register in order to transmit a simple instruction to the amplifier.

The AEK-AUD-D903V1.c simplifies these operations through a list of APIs that can configure the IB registers in a single command.

For example, the `AEK_903D_Play(AEK_AUD_D903V0)` function configures the IB8 register bits required to set the amplifier in PLAY mode.

Note: The parameter of the function indicates the name of the amplifier to control, so in a stereo system with two audio boards, we must distinguish between AEK-AUD-D903V0 and AEK-AUD-D903V1.

Figure 20. API AEK_903D_Play(AEK_AUD_D903V0)

1. The function saves the current state of the IB8 register to the variable `FDA903D_Status_IB`
2. The function changes the value of variable `FDA903D_Status_IB` to turn on PWM and activate PLAY mode.
3. The function writes the value of the `FDA903D_Status_IB` variable to the IB8 register, effectively setting the amplifier in PLAY mode.

```
/**
 * @brief       This function allows the audio amplifier to go in PLAY state.
 *
 * @param[in]   AEK_AUD_D903V1_DEVICE dev
 *
 * @return      i2c_result_t
 *
 * @api
 */
i2c_result_t AEK_903D_Play(AEK_AUD_D903V1_DEVICE dev)
{
    if(FDA_Status[dev]  != PLAY)
    {
        AEK_903D_Read_IB(dev, IB8,&FDA903D_Status_IB[dev][8],1);  (1)
    }

    FDA903D_Status_IB[dev][8] = (FDA903D_Status_IB[dev][8] & 0xFE) | IB8_PWM_ON | IB8_PLAY;  (2)
    return AEK_903D_Write_IB(dev, IB8, &FDA903D_Status_IB[dev][8], 1);  (3)
}
```

Some APIs in the AEK-AUD-D903V1.c driver require specific configuration parameters. In the following example, bits D3, D2 and D1 combine to define different configurations (000 = high impedance configuration, 011 = current sensing configuration, etc.). This API therefore requires indication of the desired configuration as well as the relevant device when it is invoked.

Figure 21. I2S Test Pin configuration API

1. The user must replace the description with the appropriate value field to indicate the desired configuration.

```
/**
 * @brief       This function configures the I2Stest pin.
 *
 * @param[in]   AEK_AUD_D903V1_DEVICE dev
 *
 * @param[in]   value (Choose one of these parameters and copy it into the 'value' field of the function):
 *
 *      -   IB8_HIGH_IMPEDENCE_CONFIG  (1)
 *      -   IB8_CURRENT_SENSING_ENABLED
 *      -   IB8_PWM_SYNCHRO_SIGNAL
 *
 * @return      i2c_result_t
 *
 * @api
 */
i2c_result_t AEK_903D_I2TestPinConfiguration(AEK_AUD_D903V1_DEVICE dev, uint8_t value)
{
    if(FDA_Status[dev]  != PLAY)
    {
        AEK_903D_Read_IB(dev, IB8,&FDA903D_Status_IB[dev][8],1);
    }

    FDA903D_Status_IB[dev][8] = (FDA903D_Status_IB[dev][8] & 0xF1) | value;
    return AEK_903D_Write_IB(dev, IB8, &FDA903D_Status_IB[dev][8], 1);
}
```

The following table shows all the available functions divided according to the register on which they act.

Table 9. list of API functions in AEK-AUD-D903V1.c

Register	API
IB0	AEK_903D_EnableWritingOnIBs
	AEK_903D_DisableWritingOnIBs
	AEK_903D_SetInputDataFormats
	AEK_903D_SelectChannelPosition
	AEK_903D_SetVoltageMode
IB1	AEK_903D_SetI2SWordSelect
	AEK_903D_SetPWMSwitchingFrequency
	AEK_903D_SetPwmAplifierDithered
	AEK_903D_SetPwmAplifierNotDithered
	AEK_903D_SetPwmInPhase
	AEK_903D_SetPwmOutOfPhase
IB2	AEK_903D_SetDiagShort2SupplyTiming
	AEK_903D_DisableLowRadiationFunction
	AEK_903D_EnableLowRadiationFunction
	AEK_903D_ConfigurePowerLimit
IB3	AEK_903D_DisableOutputVoltageOffsetDetector
	AEK_903D_EnableOutputVoltageOffsetDetector
	AEK_903D_DisableInputOffsetDetector
	AEK_903D_EnableInputOffsetDetector
	AEK_903D_DisableOutputOffsetCurrentDetector
	AEK_903D_TriggerOutputOffsetCurrentDetector
	AEK_903D_DisableHighPassInDAC
	AEK_903D_EnableHighPassInDAC
	AEK_903D_DisableNoiseGating
	AEK_903D_EnableNoiseGating
	AEK_903D_DisableOpenLoadInPlayDetection
	AEK_903D_TriggerOpenLoadInPlayDetection
IB4	AEK_903D_EnableOutputVoltageOffsetInfoOnCDDIAG
	AEK_903D_DisableOutputVoltageOffsetInfoOnCDDIAG
	AEK_903D_ConfigureThermalWarningInfoOnCDDIAG
	AEK_903D_EnableOvercurrentInfoOnCDDIAG
	AEK_903D_DisableOvercurrentInfoOnCDDIAG
	AEK_903D_EnableInputOffsetInfoOnCDDIAG
	AEK_903D_DisableInputOffsetInfoOnCDDIAG
	AEK_903D_EnableShortToVccOrGndInfoOnCDDIAG
	AEK_903D_DisableShortToVccOrGndInfoOnCDDIAG
	AEK_903D_EnableHighVoltageMuteInfoOnCDDIAG
	AEK_903D_DisableHighVoltageMuteInfoOnCDDIAG
IB5	AEK_903D_EnableUvlovccInfoOnCDDIAG

Register	API
IB5	AEK_903D_DisableUvlovccInfoOnCDDIAG
	AEK_903D_EnableThermalShutdownInfoOnCDDIAG
	AEK_903D_DisableThermalShutdownInfoOnCDDIAG
	AEK_903D_EnablePwmPulseSkippingInfoOnCDDIAG
	AEK_903D_DisablePwmPulseSkippingInfoOnCDDIAG
IB6	AEK_903D_SelectMuteTimingsetup
	AEK_903D_SelectAudioSignalGainControl
	AEK_903D_SelectGainSetting
IB7	AEK_903D_SelectDiagnosticRampTime
	AEK_903D_SelectDiagnosticHoldTime
	AEK_903D_SelectCurrentSensingCommunication
	AEK_903D_SelectCurrentSensingProtocolConfiguration
IB8	AEK_903D_SetCurrentSensingFullScale
	AEK_903D_SetChannelWithPWMoff
	AEK_903D_SetChannelWithPWMon
	AEK_903D_Eco_Mode
	AEK_903D_StartDCDiag
	AEK_903D_DisableDCDiag
	AEK_903D_I2TestPinConfiguration
	AEK_903D_Play
	AEK_903D_Mute
IB9	AEK_903D_EnableWatchDogForWordSelect
	AEK_903D_DisableWatchDogForWordSelect
IB10	AEK_903D_SetShortLoadImpedanceThreshold
	AEK_903D_SetOpenLoadImpedanceThreshold
	AEK_903D_SetCurrentOffsetThreshold
IB11	AEK_903D_SelectOverCurrentProtectionLevel
	AEK_903D_SetSlowSlopePWMConfiguration
	AEK_903D_SetDefaultPWMConfiguration
IB12	AEK_903D_SetThermalWarning
IB13	AEK_903D_EnableDigitalMuteInPlayForTW1
	AEK_903D_DisableDigitalMuteInPlayForTW1
IB14	AEK_903D_SetFeedbackOnLCFilter
	AEK_903D_SetFeedbackOnOutPin
	AEK_903D_SetupLCFilter
	AEK_903D_Enable903ToBeProgramVial2C
	AEK_9030_Disable903ToBeProgramVial2C

Other functions in AEK-AUD-D903V1.c that are not register specific are listed below:

AEK_903D_Write_IB: I²C write to IB registers

AEK_903D_Read_IB: I²C read of single IB register

AEK_903D_Read_All_IB: I²C read of all IB registers

AEK_903D_Read_DB: I²C read of single DB register

AEK_903D_Read_All_DB: I²C read of all DB registers

AEK_903D_SetDefaultRegisters: initializes I²C registers in AEK-AUD-D903V1 and sets first bit of IB14 to 1 (ready to work)

AEK_903D_SetEnables: used inside the AEK_903D_Init function to set/clear the Enable pins of the board as defined in the configuration

AEK_903D_Init: initializes the I²C and I²S protocol and to launch the AEK_903D_SetEnables function

AEK_903D_I2C_Init: initializes the I²C peripheral

AEK_903D_I2S_Init: initializes the I²S peripheral

AEK_903D_CheckOpenLoadInPlayDetection: returns the result of the Open Load in Play Detection test on the DB0 register in the `FDA903D_Errors` structure

AEK_903D_CheckOffsetCurrent: returns the result of the Current Offset detection test on the DB0 register in the `FDA903D_Errors` structure

AEK_903D_CheckInputOffsetDetector: returns the result of the Input Offset Detection test on the DB0 register in the `FDA903D_Errors` structure

AEK_903D_CheckOutputVoltageOffsetDetector: returns the result of the Output Voltage Offset Detection test on the DB0 register in the `FDA903D_Errors` structure

AEK_903D_CheckDCDiagnostic: returns the result of the DC Diagnostic on the DB1 register in the `FDA903D_Errors` structure

AEK_903D_Diagnostic: reads the DB register and signals whether a certain failure condition has occurred (SHORT2VCC, SHORT2GND, OVERCURRENT, UNDERVOLTAGE, OVERTEMPERATURE OVERVOLTAGE) in the `FDA903D_Errors` structure

4.3.2 sound.c description

This library contains APIs for the generation, reproduction and simulation of audio wave signals.

initWaveFile: This function takes as input the start address of the first WAV file and the number of files that you intend to load into memory (maximum number dim=10) and initializes the `sound_db` structure with all the necessary addresses to identify the beginning and end of each WAV file.

getStartWavFile: This function computes the address that points to the first audio sample of a given WAV file. You must provide the function with an integer that identifies the location of the WAV file within the `sound_db` structure previously initialized by the `initWaveFile` function.

getHalfWavFile: This function computes the address that points to the middle audio sample of a given WAV file. You must provide this function with an integer that identifies the location of the WAV file within the `sound_db` structure previously initialized by the `initWaveFile` function.

GetEndWavFile: This function computes the address that points to the last audio sample of a given WAV file. You must provide this function with an integer that identifies the location of the WAV file within the `sound_db` structure previously initialized by the `initWaveFile` function.

swapEndian32: This function swaps the order of the bits: from little (big) endian to big (little) endian.

validate_wav_file: This function validates the WAV file by checking the WAV file descriptor parameters.

checkWavFile: This function checks the WAV file and identifies the start, middle and end of each WAV file, removing the WAV file header.

load_channel_data: This function loads new data to the transmission buffer.

playSound: This function plays the sample provided in MONO mode by taking as input a pointer to function that generates the audio samples and an integer indicating the volume.

playSoundStereo: This function plays the sample provided in STEREO mode by taking as input a pointer to function that generates the audio samples and an integer indicating the volume.

The last two functions deal with actual sound reproduction. Since these two functions work in the same way, with the only difference being that one plays mono WAV files and the other stereo files, we will describe how the first one works.

Figure 22. playSound API

1. This input is a function pointer to the sample that playSound will run. The pointer must refer to the user function, which must return a `uint32_t` data type that is assigned to the variable to be transmitted.
2. In this MONO mode example, the same sample is transmitted to both channels. In STEREO mode, the samples for the left and right channels may differ.

```
/**
 * @brief       This function plays the samples provided in MONO mode.
 *
 * @param[in]   volume: integer which determines the sound volume
 *
 * @param[in]   (*sample_source)(void): function which provides the samples to be played.
 *
 * @api
 */
void playSound( int volume, uint32_t (*sample_source)(void))
{
    uint8_t *new_sample;

    if (load_new_sample == 1U)
    {
       new_sample = &txbuf[which_buffer * (sizeof(txbuf) >> 1)];
       for ( i = 0; i < ((sizeof(txbuf) >> 1)); i += 8U)
       {
          sample = (*sample_source)();          (1)
          sample = (sample*volume) << 16;

          /**********Load right channel***********/
          load_channel_data(sample , new_sample);
          new_sample += 4;

          /**********Load left channel**********/
          load_channel_data(sample, new_sample);
          new_sample += 4;
       }

       FDA_Status[0] = PLAY;
       AEK_903D_Read_All_DB(0);
       AEK_903D_Read_All_IB(0);

       load_new_sample = 0;
    }
}
```

The `sound.c` library allows storing samples generated by mathematical functions or writing a function able to take audio samples from any file WAV loaded in memory.

5 How to play an audio WAV file

5.1 SPC582B60E1 memory map

Excluding the 64 KB data flash, the remaining 1024 KB MCU memory is divided into the following blocks:

- 4 blocks of 16 KB (Low Flash Blocks)
- 2 blocks of 32 KB (Low Flash Blocks)
- 2 blocks of 64 KB (Mid Flash Blocks)
- 6 blocks of 128 KB (Large Flash Blocks)

We allocate the Low and Mid Flash Blocks to load and execute the source code of our application, and the remaining 6 Large Flash Blocks for the WAV audio files. This of course means that the WAV files cannot exceed 768 KB.

The following microcontroller Flash memory map shows the physical addresses used to identify and divide the different memory portions.

Table 10. Flash memory map of SPC582B

Start address	End address	Description	1 RWR partition
			RWW Partition ID
Data Flash: 64 KB			
0x00804000	0x00807FFF	16 KB EEPR0M block1	1
0x00808000	0x0080BFFF	16 KB EEPR0M block2	1
0x0080C000	0x0080FFFF	16 KB EEPR0M block3	1
0x00810000	0x00FBFFFF	Reserved	
Low & Mid Flash blocks: 256 KB for application code			
0x00FC0000	0x00FC3FFF	16 KB Code Flash block1	0
0x00FC4000	0x00FC7FFF	16 KB Code Flash block2	0
0x00FC8000	0x00FCBFFF	16 KB Code Flash block3	0
0x00FCC000	0x00FCFFFF	16 KB Code Flash block4	0
0x00FD0000	0x00FD7FFF	32 KB Code Flash block0	0
0x00FD8000	0x00FDFFFF	32 KB Code Flash block1	0
0x00F E0000	0x00FEFFFF	64 KB Code Flash block0	0
0x00FF0000	0x00FFFFFF	64 KB Code Flash block1	0
Large Flash Blocks: 768 KB for audio WAV files			
0x01000000	0x101FFFF	128 KB Code Flash block0	0
0x01020000	0x103FFFF	128 KB Code Flash block1	0
0x01040000	0x0105FFFF	128 KB Code Flash block2	0
0x01060000	0x0107FFFF	128 KB Code Flash block3	0
0x01080000	0x0109FFFF	128 KB Code Flash block4	0
0x010A0000	0x010BFFFF	128 KB Code Flash block5	0
0x010C0000	0x0FFFFFFF	Reserved	

5.2 Uploading audio WAV file

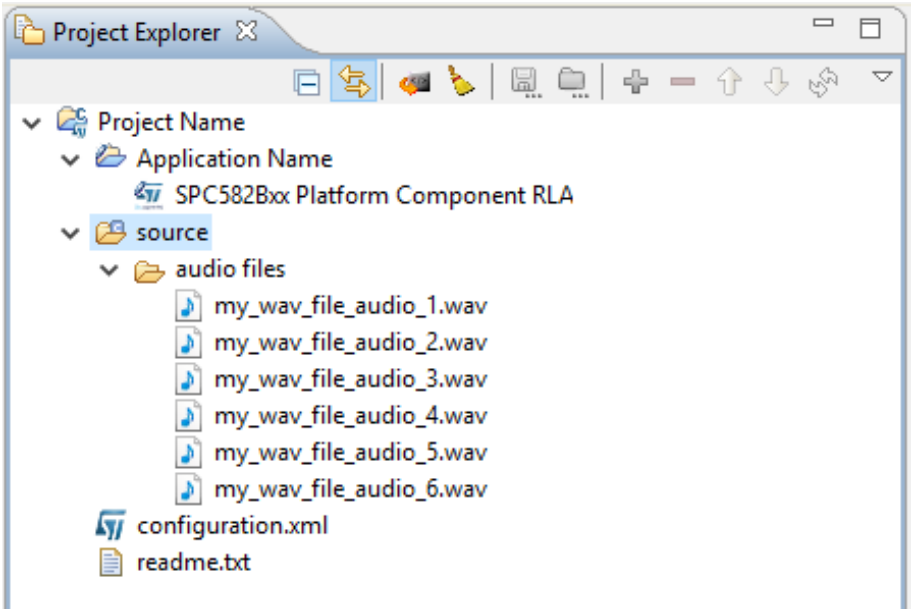
Use the procedure below to upload audio WAV files.

Step 1. Launch SPC5-STUDIO and Create a new SPC5-STUDIO application for Chorus 1M (SPC582B).

Step 2. Right-click the [**source**] folder to create another folder inside it.
You can name the folder “audio files”.

Step 3. Copy the desired WAV files and paste them inside the newly created folder.
Verify that all files have been inserted.

Figure 23. Project folder for audio files



Step 4. Compile your application.
This creates a file named `application.ld` in your project folder.

Step 5. In the same folder, make a copy of the `application.ld` file and rename the file according to the compiler you are using.

- Free GCC → `user_freegcc.ld`.
- Green Hills → `user_ghs.ld`.
- Hitech → `user_hightec.ld`.

Step 6. Double click on the `user_freegcc.ld` file to open it.

Step 7. Inside the file, modify the memory partition by splitting the flash block into flash and sound blocks.

– The new flash block is 256 KB where application source code is loaded and executed

– The new sound block is 768 KB where audio WAV files are saved

Figure 24. Old flash block memory allocation

1. Old 1 MB flash block

```
user.ld
22
23 __irq_stack_size__       = 0;
24 __process_stack_size__   = 4096;
25
26 MEMORY
27 {
28     dataflash : org = 0x00800000, len = 128k
29     flash     : org = 0x00FC0000, len = 1M   (1)
30     ram       : org = 0x400A8000, len = 96K
31 }
32
33 ENTRY(_reset_address)
34
```

Figure 25. New flash block memory allocation

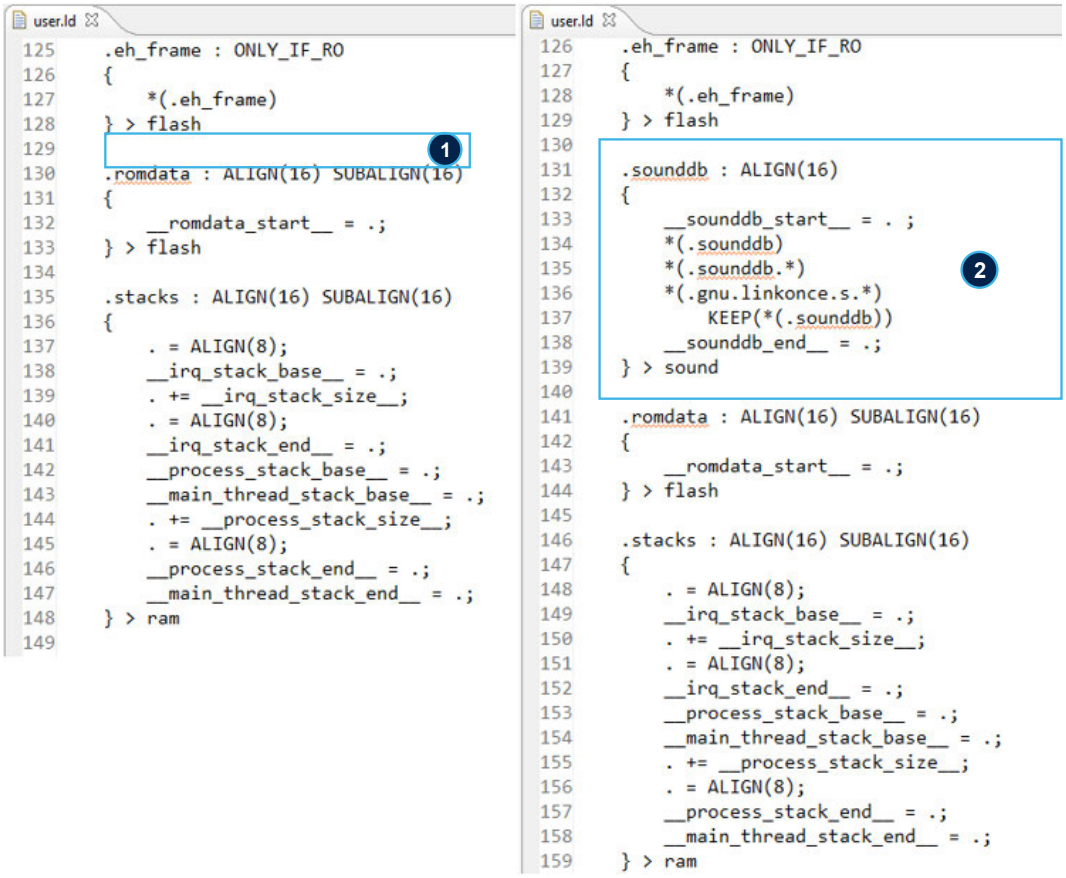
1. New 256 KB flash block
2. New 768 KB sound block

```
user.ld
22
23 __irq_stack_size__       = 0;
24 __process_stack_size__   = 4096;
25
26 MEMORY
27 {
28     dataflash : org = 0x00800000, len = 128k
29     flash     : org = 0x00FC0000, len = 256K  (1)
30     sound     : org = 0x01000000, len = 768K  (2)
31     ram       : org = 0x400A8000, len = 96K
32 }
33
34 ENTRY(_reset_address)
35
```

Step 8. Define a section called `sounddb` as indicated below.

Figure 26. sounddb definition

1. Location of new sounddb definition
2. sounddb definition code



```
user.ld
125     .eh_frame : ONLY_IF_RO
126     {
127         *(.eh_frame)
128     } > flash
129
130     .romdata : ALIGN(16) SUBALIGN(16)
131     {
132         __romdata_start__ = .;
133     } > flash
134
135     .stacks : ALIGN(16) SUBALIGN(16)
136     {
137         . = ALIGN(8);
138         __irq_stack_base__ = .;
139         . += __irq_stack_size__;
140         . = ALIGN(8);
141         __irq_stack_end__ = .;
142         __process_stack_base__ = .;
143         __main_thread_stack_base__ = .;
144         . += __process_stack_size__;
145         . = ALIGN(8);
146         __process_stack_end__ = .;
147         __main_thread_stack_end__ = .;
148     } > ram
149
```

```
user.ld
126     .eh_frame : ONLY_IF_RO
127     {
128         *(.eh_frame)
129     } > flash
130
131     .sounddb : ALIGN(16)
132     {
133         __sounddb_start__ = . ;
134         *(.sounddb)
135         *(.sounddb.*)
136         *(.gnu.linkonce.s.*)
137             KEEP(*(.sounddb))
138         __sounddb_end__ = .;
139     } > sound
140
141     .romdata : ALIGN(16) SUBALIGN(16)
142     {
143         __romdata_start__ = .;
144     } > flash
145
146     .stacks : ALIGN(16) SUBALIGN(16)
147     {
148         . = ALIGN(8);
149         __irq_stack_base__ = .;
150         . += __irq_stack_size__;
151         . = ALIGN(8);
152         __irq_stack_end__ = .;
153         __process_stack_base__ = .;
154         __main_thread_stack_base__ = .;
155         . += __process_stack_size__;
156         . = ALIGN(8);
157         __process_stack_end__ = .;
158         __main_thread_stack_end__ = .;
159     } > ram
```

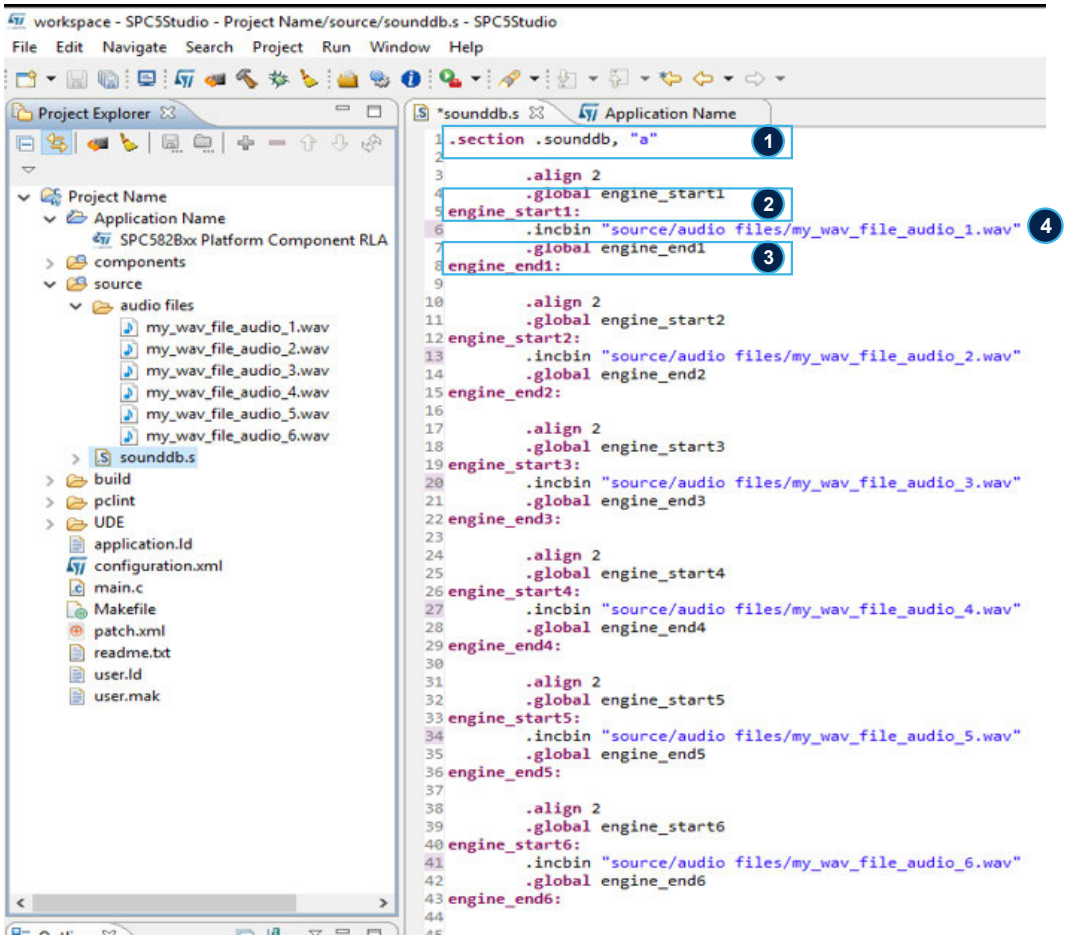
Step 9. Right-click the [**source**] folder to create a new file called `sounddb.s`.

Step 10. In the file, indicate the path of the WAV files to be loaded and declare the variables that identify the physical start and end addresses of the various WAV files.

Below is an example with paths and address variables for six WAV files.

Figure 27. sounddb.s audio file declarations

1. first line of code
2. start address variable for first audio file
3. end address variable for first audio file
4. path to first audio file



Step 11. Recompile the project.

The size of the build output should now include the audio files added in the `sounddb.s` file.

After compiling, the [**build**] folder will contain the output file of the application in BIN, DMP, ELF, HEX, MAP and MOT formats.

Figure 28. Build output before adding sounddb.s component

1. size of build output before adding sounddb.s component

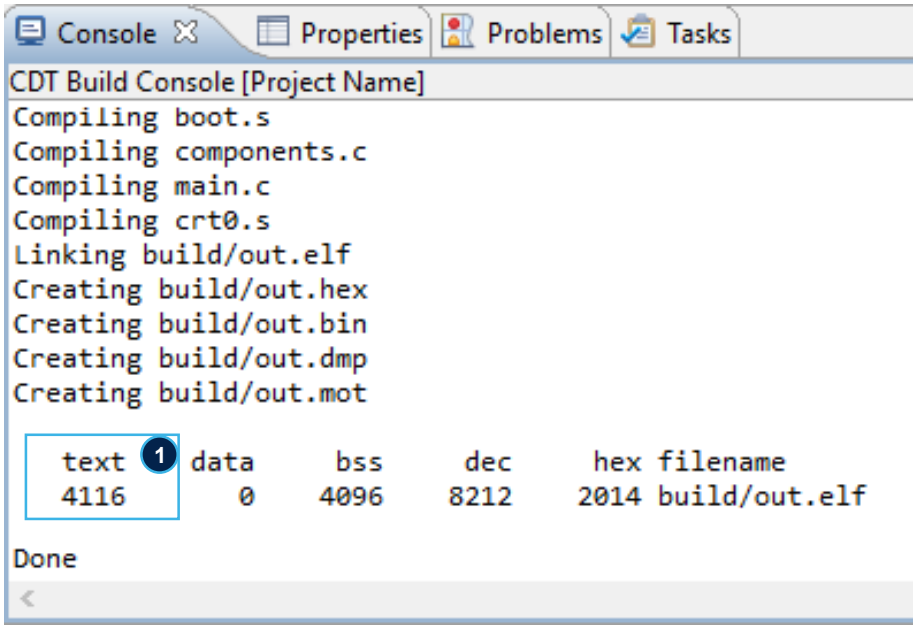
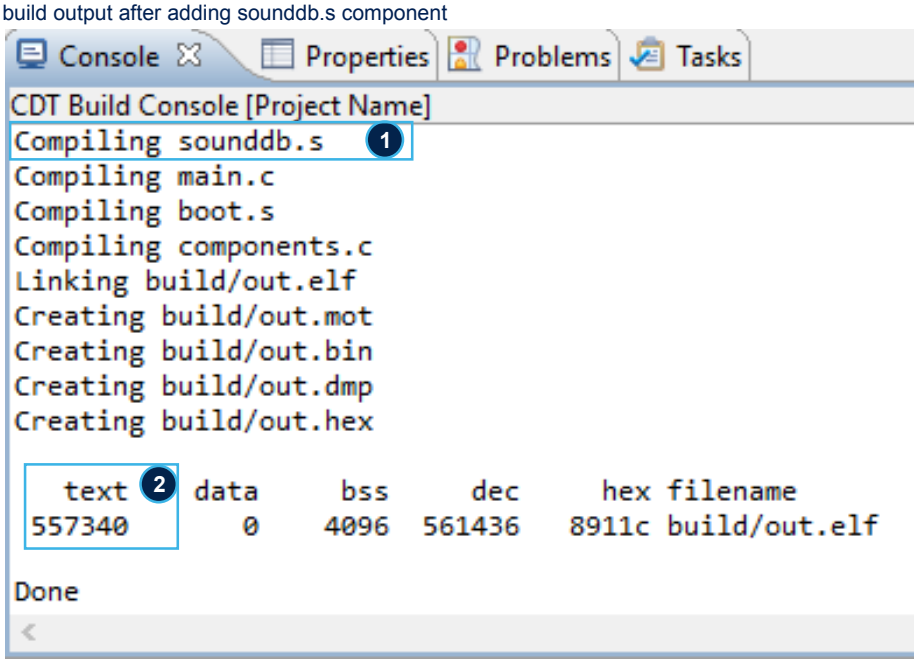


Figure 29. Build output after adding sounddb.s component

1. added sounddb.s component
2. size of build output after adding sounddb.s component



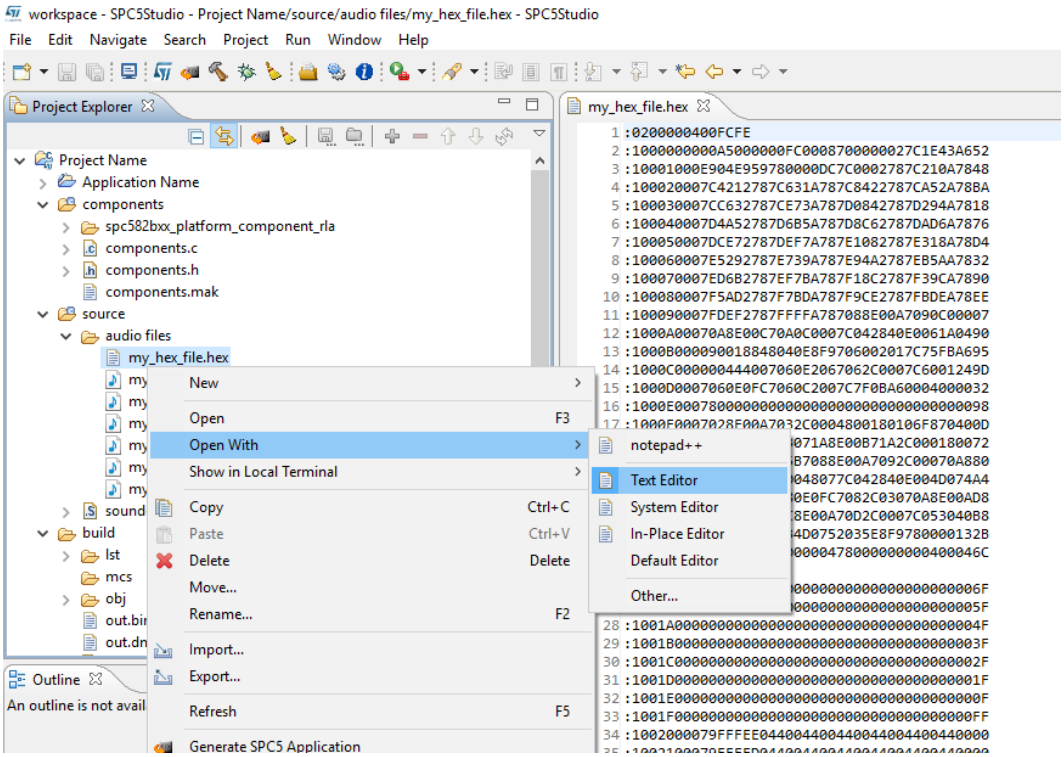
Step 12. Expand the [**build**] folder, select the `out.hex` file and move or copy it to the same folder where the audio files were saved.

The `.hex` file can actually be saved to any external folder.

Step 13. Rename the `.hex` file as you prefer.
This file contains all the application files.

Step 14. Open your .hex file using a text editor.
You can use the SPC5-STUDIO editor or an external one.

Figure 30. Open hex file with editor



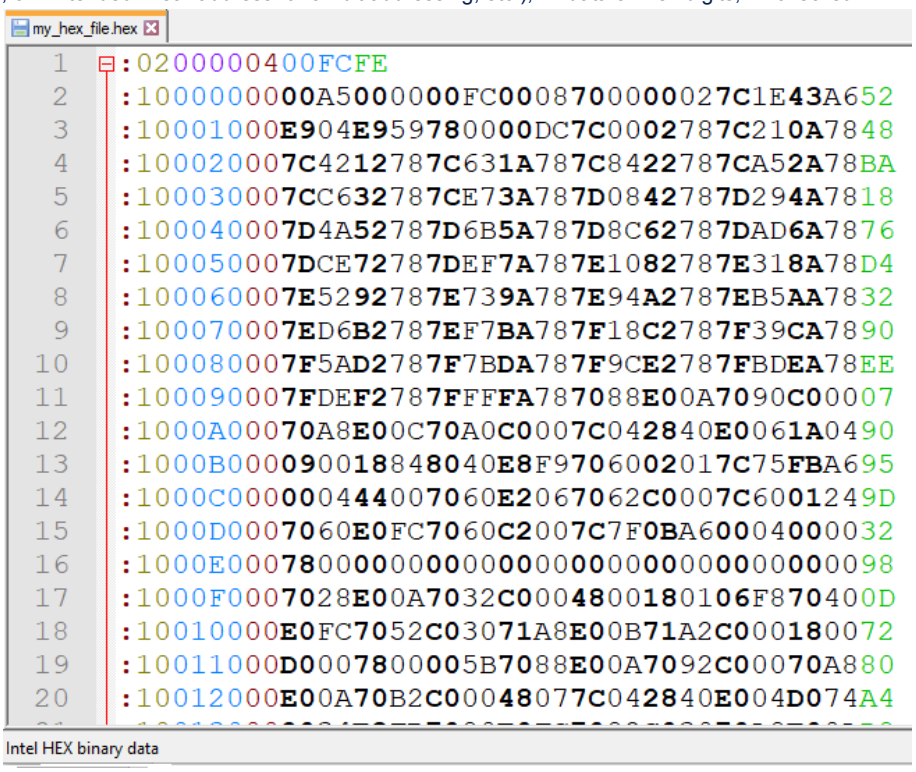
Step 15. Examine the file and verify that the address is 0200000400FCFE.

00FC is the base address (the first 16 bits) of the physical address of the data contained in each record.

Figure 31. hex file data

Legend:

: start code, number of bytes in data field , address of data added to base address, record type (00 data, 01 end of file, 04 Extended linear address for 32-bit addressing, etc.), data 32 hex digits, checksum

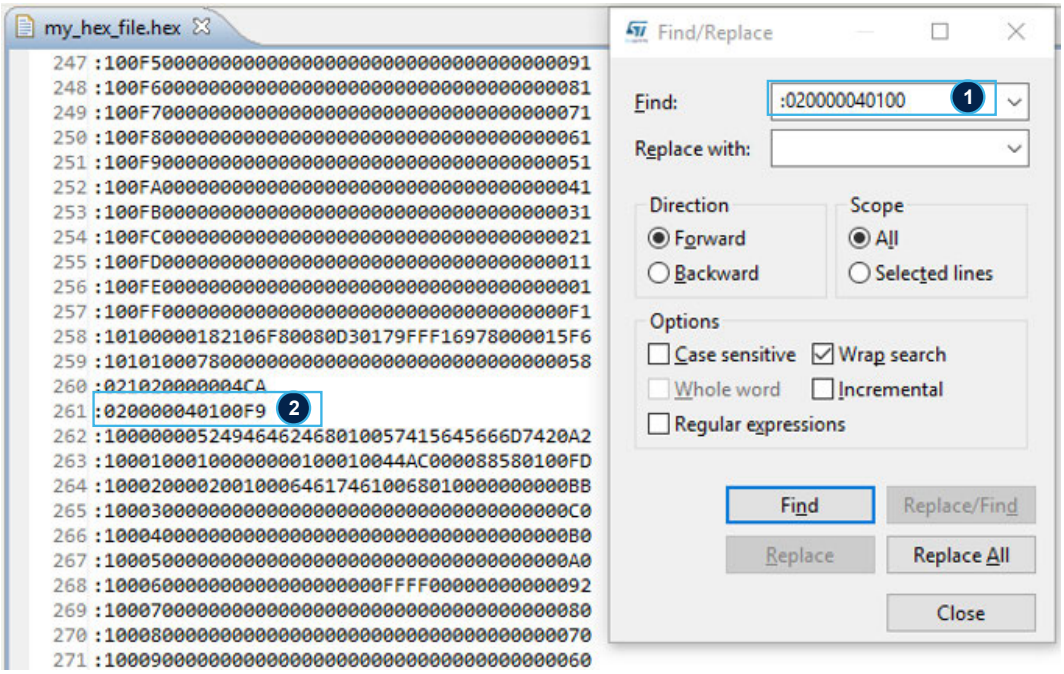


Step 16. In the text editor, search for the string following: 020000040100.

This string contains the base address of the sound partition in the memory where the audio WAV samples specified in sounddb.s are loaded. The sound partition was defined previously in the `user_freegcc.ld` file.

Figure 32. Starting point of audio content in hex file

1. search string to find
2. string found in hex file



Step 17. Delete all the lines in the hex file before the starting line of the audio content.

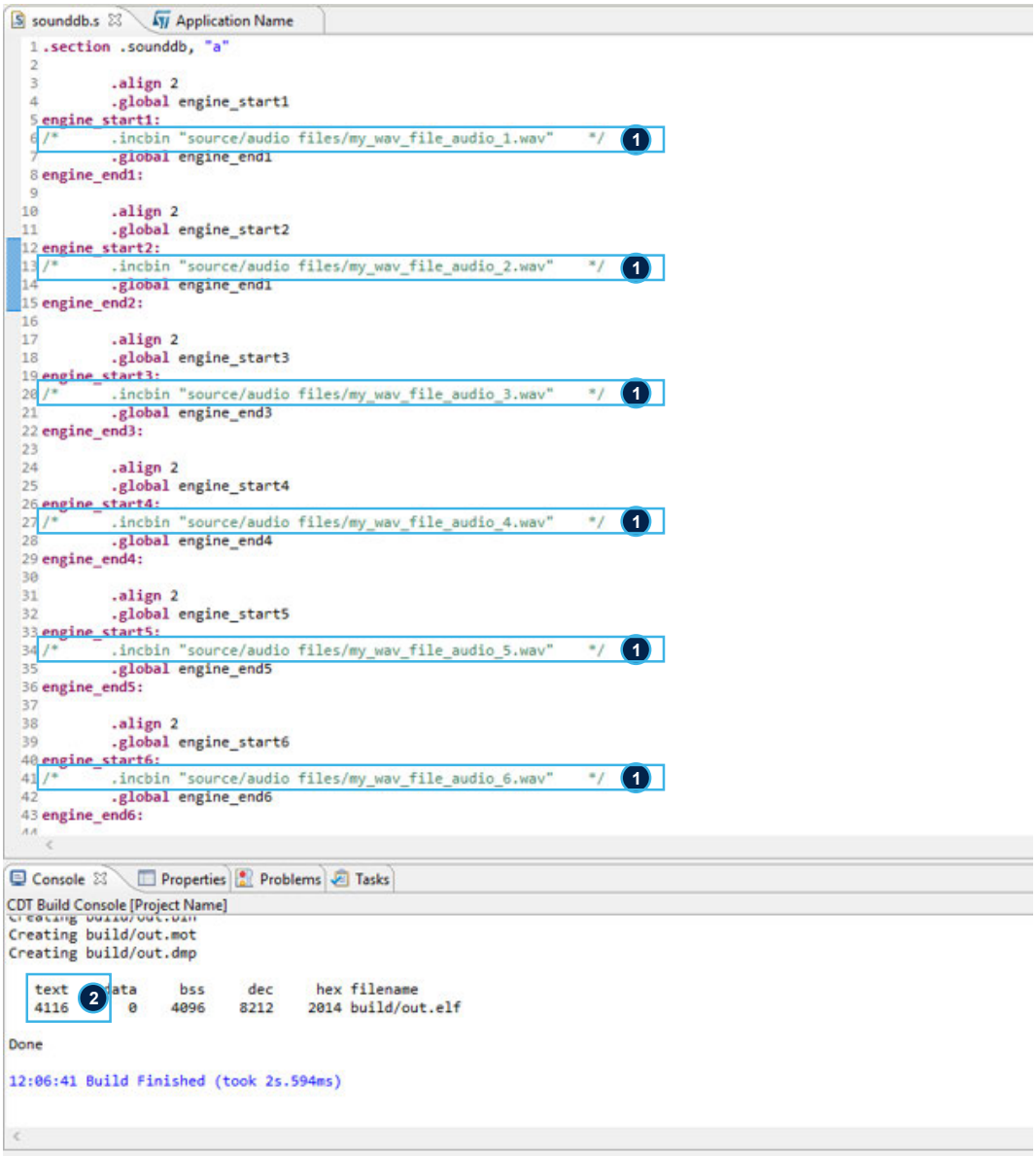
The hex file now only contains the sample audio data to be played by the AVAS system.

Step 18. Open the `soundb.s` file and remove or comment out all paths to the audio sample files.

This ensures that the application does not include the audio content the next time you compile, leaving only the information regarding the source code.

Figure 33. sounddb.s file with audio sample file paths removed

1. path statements to be commented or deleted
2. build file returns to its original size when the audio content is removed



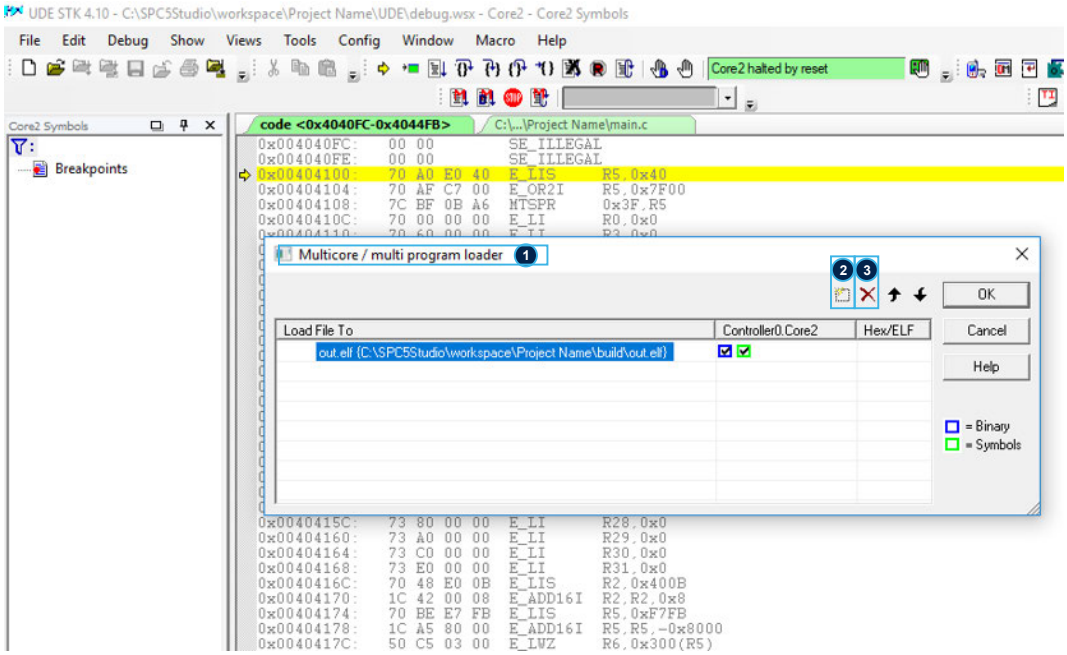
Step 19. Recompile the project.

Step 20. Connect the mini-B USB port on the Discovery AEK-MCU-C1MLIT1 control board to a USB port on your PC with an appropriate mini USB to USB cable.

Step 21. Launch SPC5-UDESTK-SW on your PC and run the `debug.wsx` file.
This opens a [**Multicore / multi program loader**] window.

Figure 34. SPC5-UDESTK-SW software with loader window

1. loader window
2. browse file icon
3. remove file icon



Step 22. Click on the [**delete**] icon in the menu bar to remove the current file from the [**Load File To**].

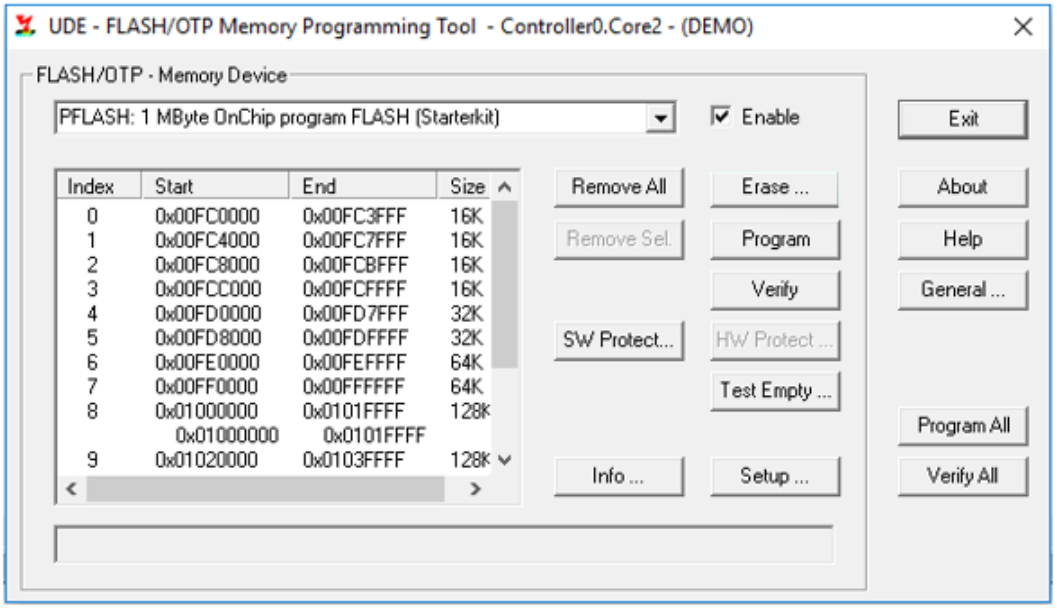
Step 23. Click on the [**browse**] icon to add a new file.

Step 24. Browse to you hex file and select [**open**].
The [**Multicore / multi program loader**] window reopens with the hex file in the [**Load File To**] list.

Step 25. Click [**OK**] to open the [**FLASH/OTP Memory Programming Tool**] window.

Note: The free UDE license allows you to upload a maximum file size of up to 256 KB.

Figure 35. FLASH/OTP Memory Programming Tool

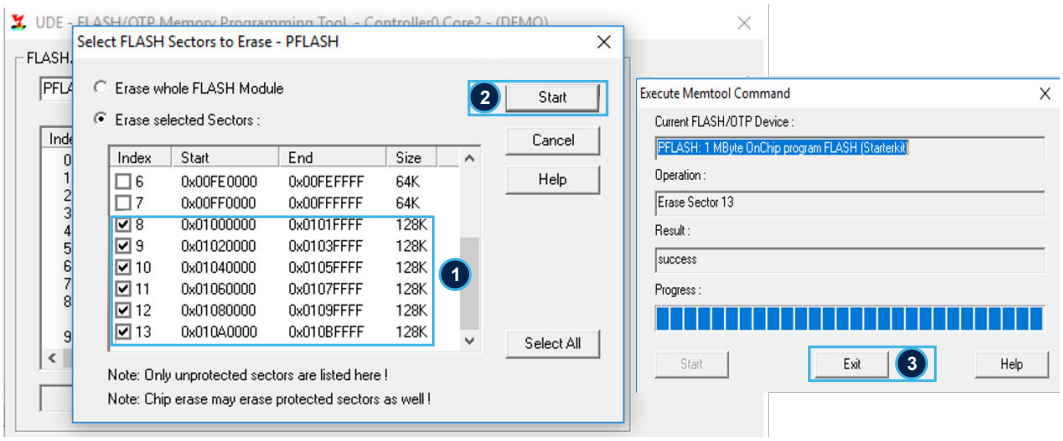


Step 26. Select the [**Erase**] button.
A Flash selection window appears.

Step 27. Select from memory block 8 (0x01000000) to block13 and click [**Start**].

Figure 36. UDE Flash erase tool

1. blocks to be erased
2. start erase procedure
3. exit when erase has completed



Step 28. Click [**Exit**] when the process has finished.

Step 29. In the FLASH/OTP Memory Programming Tool, click on [**Program All**].
This process loads the audio samples.

Step 30. Wait until the operation has completed and select [**Exit**].

Step 31. Close UDE-STK.

You have loaded your audio samples into the designated memory portion.

Step 32. Save your SPC5-STUDIO application and keep the `user.ld` and `sounddb.s` files in a folder.

These two files are important because they are the starting point for the creation of the AEK-AUD-D903V1 application.

6 AEK-AUD-D903V1 sample applications

6.1 How to create a simple AEK-AUD-D903V1 application

Before you start, ensure that the audio samples have been correctly loaded in the microcontroller memory.

This procedure shows you how to play an audio WAV file loaded in memory and perform some diagnostics.

Step 1. Create a new SPC5-STUDIO application for the SPC582B series microcontroller and add the following components:

- SPC582Bxx Init Package Component RLA
- SPC582Bxx Low Level Drivers Component RLA

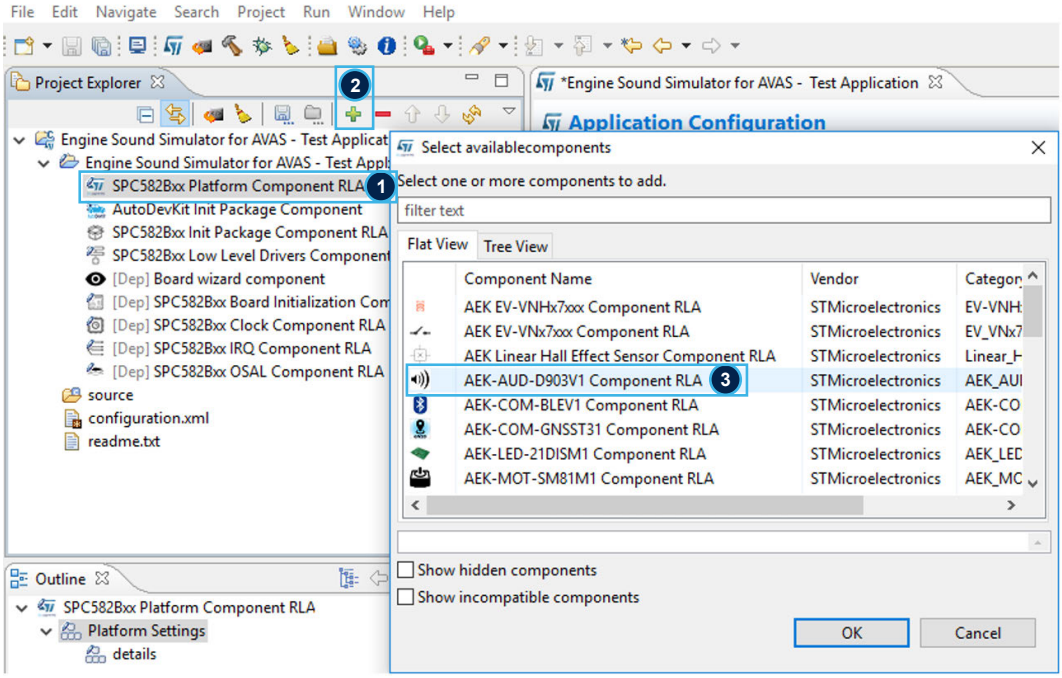
These components must be added immediately, or the remaining will not be visible.

Step 2. Add the following further components:

- AutoDevKit Init Package Component
- SPC582Bxx Platform Component RLA
- AEK-AUD-D903V1 Component RLA

Figure 37. SPC5-STUDIO adding audio project components

1. SPC582Bxx Platform Component RLA
2. Open available components
3. AEK-AUD-D903V1 Component RLA



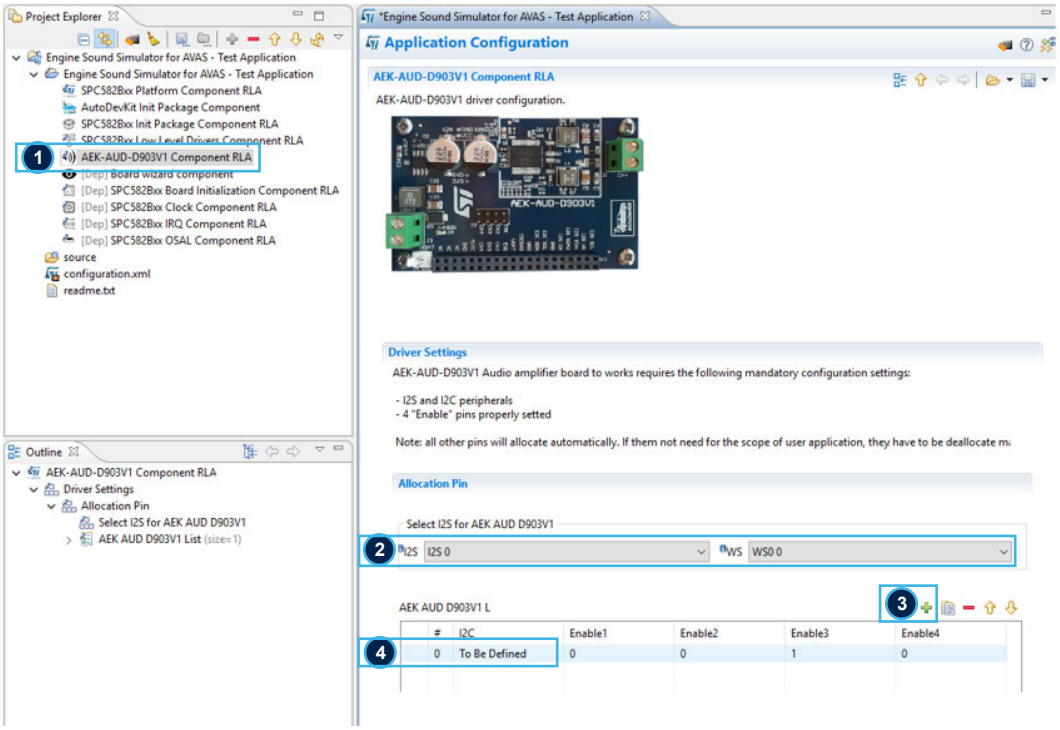
Step 3. Select [**AEK-AUD-D903V1 Component RLA**] to open the [**Application Configuration**] window.

Step 4. Select the I2S (DSPI) port and the I2S WS. Then, click on [**+**] to add a new element to the AEK AUD D903V1 list.

If you want to create a STEREO version, you will need to insert a second element in the list.

Figure 38. AEK-AUD-D903V1 component configuration

1. AEK-AUD-D903V1 component
2. Pin association
3. add new element icon
4. new entry



Step 5. Double click on the newly added element to configure the I²C interface.

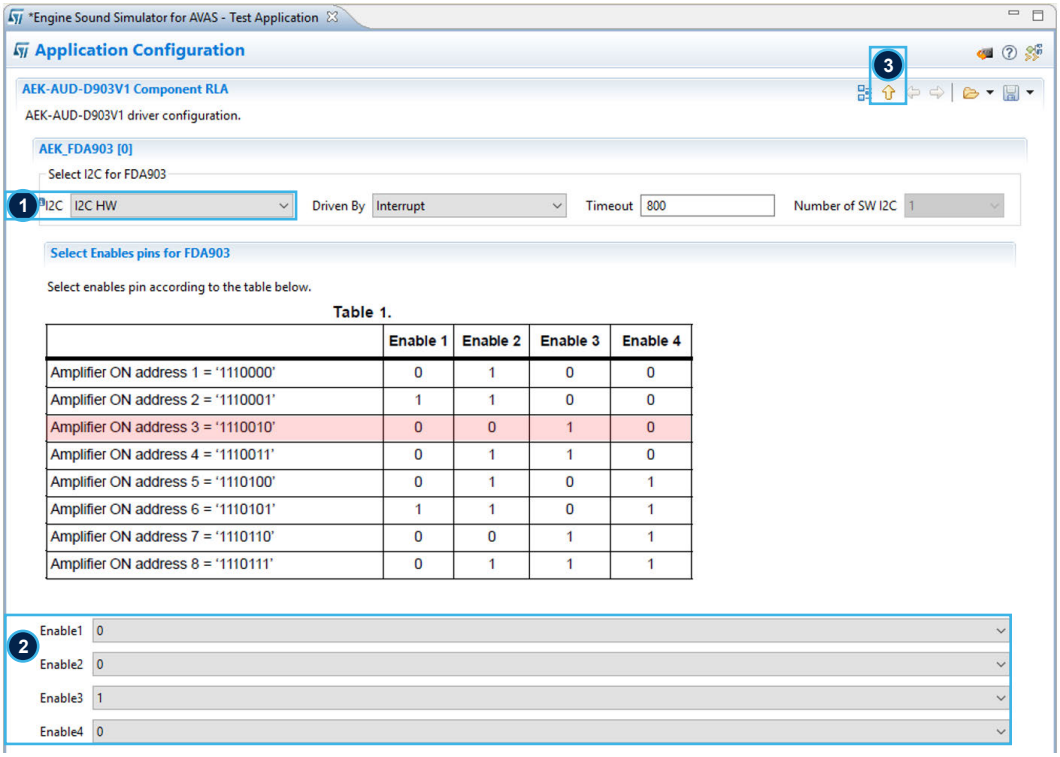
The I²C configuration window opens.

Step 6. Select the I²C HW and the address for the power amplifier derived from a combination of enable pins.

To create a STEREO version, you need to assign two different addresses for the two elements so that you can communicate with each power amplifier independently.

Figure 39. AEK-AUD-D903V1 I²C configuration

1. I2C HW selection
2. Enable pin configuration
3. Confirm configuration



	Enable 1	Enable 2	Enable 3	Enable 4
Amplifier ON address 1 = '1110000'	0	1	0	0
Amplifier ON address 2 = '1110001'	1	1	0	0
Amplifier ON address 3 = '1110010'	0	0	1	0
Amplifier ON address 4 = '1110011'	0	1	1	0
Amplifier ON address 5 = '1110100'	0	1	0	1
Amplifier ON address 6 = '1110101'	1	1	0	1
Amplifier ON address 7 = '1110110'	0	0	1	1
Amplifier ON address 8 = '1110111'	0	1	1	1

Step 7. Click the [**Allocation**] button below the AEK-AUD-D903V1 list and click [**OK**] in the confirmation window.

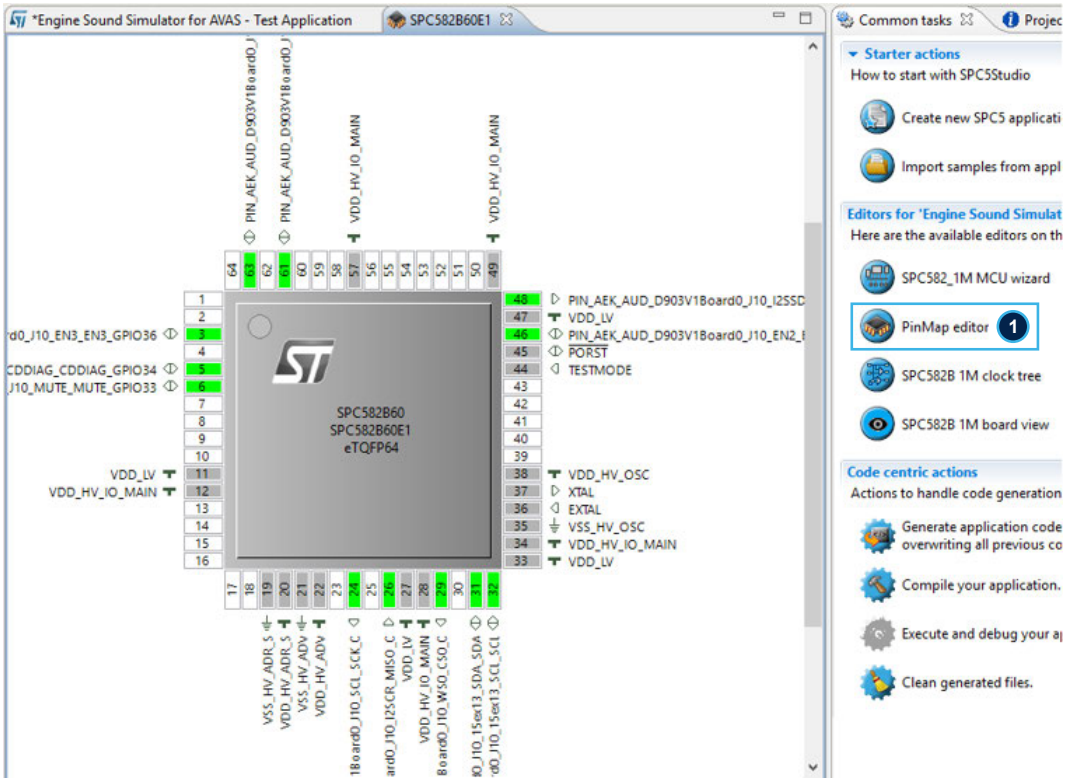
This operation delegates automatic pin allocation to STSW-AUTODEVKIT. If the system warns you that the selected I²S (DSPI) port is not available, restart from step 3 and select another I²S port or another I²S WS.

Step 8. Click on the PinMap editor icon to check that the twelve required pins have been allocated appropriately.

- I2S SCL (pin 24 → PG11)
- I2S SDA1 (pin 48 → PD5)
- I2S WS (pin 29 → PB11)
- I2S CR (pin 26 → PD11), not used in this application
- I²C SCL (pin 32 → PB8)
- I²C SDA (pin 31 → PB9)
- 4 GPIO pins for Enable
- 1 GPIO pin for Hardware Mute
- 1 GPIO pin for CDDIAG, not used in this application

Figure 40. PinMap editor

1. PinMap editor icon



Step 9. Close the PinMap Editor and save the application.

Step 10. Generate and build the application using the appropriate icons in SPC5-STUDIO.

The project folder will be populated with new files, including `main.c` and the components folder with AEK-AUD-D903V1 and sound drivers.

Step 11. Before starting coding, insert the `user_freegcc.ld` and `sounddb.s` files prepared previously.

Step 12. Open the `main.c` file and include AEK-AUD-D903V1 and sound files, and define the required constants and variables.

```
#include "components.h"
#include "sound.h"
#include "AEK-AUD-D903V1.h"

/*************** variables section ***************/
extern uint32_t* engine_start1;
/* These variables are defined in the file sounddb.s,
 thus it is necessary to declare them as extern */

int16_t* wavfileBeginPtr;   //pointer to wave file initial point
int16_t* wavfileEndPtr;     //pointer to wave file final point
int16_t* wavfilePtr;        //pointer to wave file current position
uint32_t volume = 1;
/* This variable is defined for the Playsound function.
There is a volume threshold under which the Open Load in Play Detection Test is not
valid.
Thus, if you perform this Test, make sure to increase the volume until you overcome
this threshold */
/************* end variables section *************/
```

Step 13. Place the following functions inside `main()`:

```
int main(void)
{
  componentsInit();
/* This function initializes all the imported components. It is present in the gener
ated file. */
  irqIsrEnable(); /* This function deals with interrupt management */
  initWaveFile(&engine_start1, 6);
/*This function takes in input the starting address of the first WAVE file and the n
umber of files yow want to upload */
  AEK_903D_Init(AEK_AUD_D903V1_DEV0);
/* This function initializes the I²C and I2S peripherals and sets the enable pins ch
osen during the configuration.
In the STEREO case, you would need to duplicate this function for the second device
called AEK_AUD_D903V1_DEV1. */
  AEK_903D_SetDefaultRegisters(AEK_AUD_D903V1_DEV0);
/* This function sets the register to the default state. In the STEREO case,
you would need to duplicate this function for the second device called AEK_AUD_D903V
1_DEV1.
  AEK_903D_SelectOverCurrentProtectionLevel(AEK_AUD_D903V1_DEV0, IB11_OVER_CURRENT
_PROTECTION_4A);
/* This function selects the current protection level from four possible values: 4A,
 6A, 8A, and 11A.
The protection circuit will trigger as soon as the chosen threshold is exceeded.
Increasing the volume, you could end up triggering the current protection circuit.
Clearly, lower the threshold (4A), higher the protection triggering probability.
In the STEREO case, you would need to duplicate this function for the second device
called AEK-AUD-D903V1_DEV1. */
  AEK_903D_Play(AEK_AUD_D903V1_DEV0);
/*This function turns on the PWM and puts in PLAY state the amplifier.
In the STEREO case, you would need to duplicate this function for the second device
called AEK_AUD_D903V1_DEV1. */
wavfilePtr = wavfileBeginPtr = getStartWavFile(0); // set the start address of the f
irst wave file
wavfileEndPtr = getEndWavFile(0); // set the end address of the first wave file/* Ap
plication main infinite loop. */for ( ; ; )
  {
    playSound(volume, userFunction);
/* This function allows to play the samples generated with the function pointed by '
userFunction'.
In the STEREO case, you would need to use the playSoundStereo().*/
    AEK_903D_Diagnostic(AEK_AUD_D903V1_DEV0);
/* This function updates the FDA903_Errors structure with the information contained
in the DB registers.
In the STEREO case, you would need to duplicate this function for the second device
called AEK_AUD_D903V1_DEV1. */
    AEK_903D_TriggerOpenLoadInPlayDetection(AEK_AUD_D903V1_DEV0);
/* This function triggers the Open Load in Play Detection.
In the STEREO case, you would need to duplicate this function for the second device
called AEK_AUD_D903V1_DEV1. */
    AEK_903D_CheckOpenLoadInPlayDetection(AEK_AUD_D903V1_DEV0);
/* This function updates the FDA903_Errors structure with the information inside the
 DB1 register.
In the STEREO case, you would need to duplicate this function for the second device
called AEK_AUD_D903V1_DEV1. */
  }
}
```

Step 14. Declare the function `uint32_t userFunction(void);` in the header file.
You can now write the following code between the variables section and the `main()` function.

```
/* In case of multiple wave files, these variables will be re-assigned with new
addresses using the same above functions with different parameters, e.g. getStartWav
File(1); */
uint32_t userFunction()
{
  uint16_t sampleWav; // sample from wave file to be processed
  int32_t sampleToPlay = 0;
  if(wavfilePtr > wavfileEndPtr)
// if we reach the end of the file we restart from the initial address
  {
wavfilePtr = wavfileBeginPtr;
  }
  sampleWav = ( *wavfilePtr << 8) | ((*wavfilePtr >> 8) & 0xFF);
// swap endianness
  sampleToPlay = (int32_t)sampleWav;
// place the sample in a 32-bit format
  wavfilePtr++;
//pointer to the next wave file sample
return sampleToPlay;
}
```

Step 15. Save, generate and compile the application.

Step 16. Open the BoardView Editor provided by STSW-AUTODEVKIT.

Step 17. This provides a graphical point-to-point guide on how to wire the boards.

Step 18. Connect the AEK-MCU-C1MLIT1 to a USB port on your PC using a mini-USB to USB cable.

Step 19. Launch SPC5-UDESTK-SW and open the `debug.wsx` file in the AEK_AUD_D903V1 – Application / UDE folder.

Step 20. Run and debug your code.

6.2 Available demos for AEK-AUD-D903V1

There are eight different demos with specific features provided with the audio component:

1. SPC582Bxx_RLA AEK-AUD-D903V1 - Test Application
2. SPC582Bxx_RLA AEK-AUD-D903V1 - Mono audio and Diagnostic - Test Application
3. SPC582Bxx_RLA AEK-AUD-D903V1 - Stereo audio and Diagnostic - Test Application
4. SPC582Bxx_RLA AEK-AUD-D903V1 - I²C Software Mono audio - Test Application
5. SPC582Bxx_RLA AEK-AUD-D903V1 - Engine Sound Simulator Test Application
6. SPC58ECxx_RLA AEK-AUD-D903V1 - I²C Software Mono audio - Test Application for SPC58EC-DISP
7. SPC58ECxx_RLA AEK-AUD-D903V1 - Mono audio - Test Application for SPC58EC-DISP
8. SPC584Bxx_RLA AEK-AUD-D903V1 - Mono audio - Test Application for SPC584B-DIS

Note: More demos may become available with new AutoDevKit releases.

6.2.1 How to upload the demos for AEK-AUD-D903V1

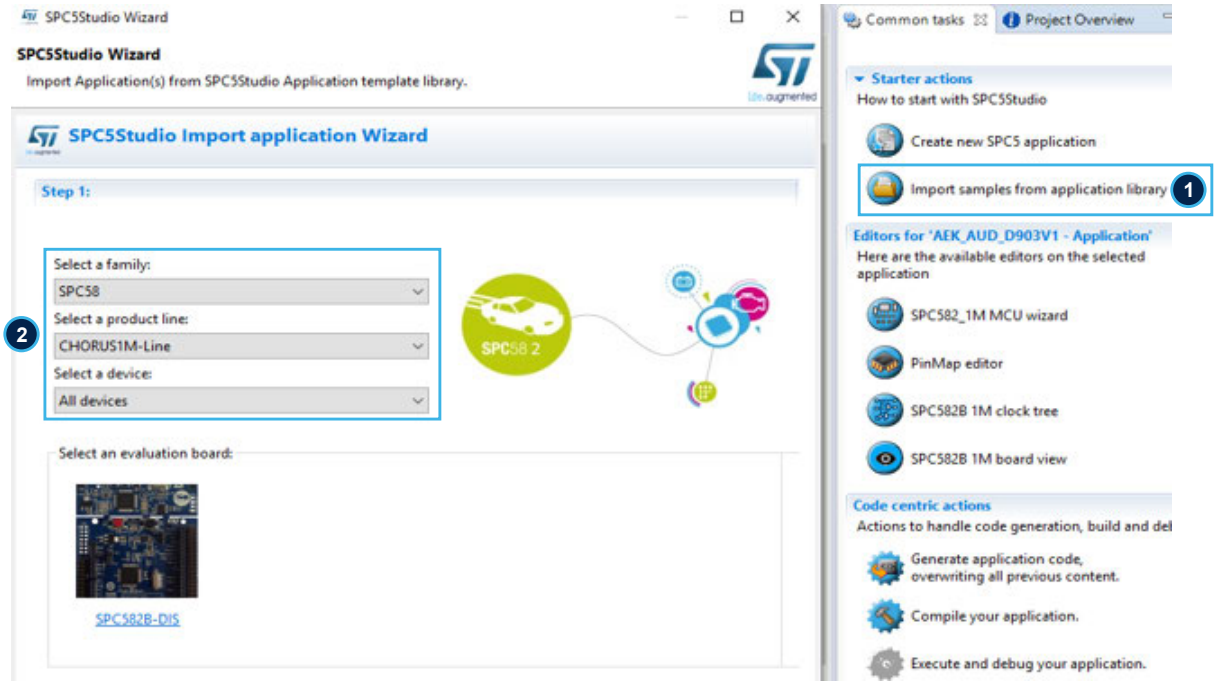
Follow the procedure below to import the demos into SPC5-STUDIO.

Step 1. Select [**Import samples from application library**] from the Common tasks pane.
An Import application Wizard appears.

Step 2. In the Import application Wizard, insert the appropriate product family details.

Figure 41. SPC5-STUDIO Import application Wizard

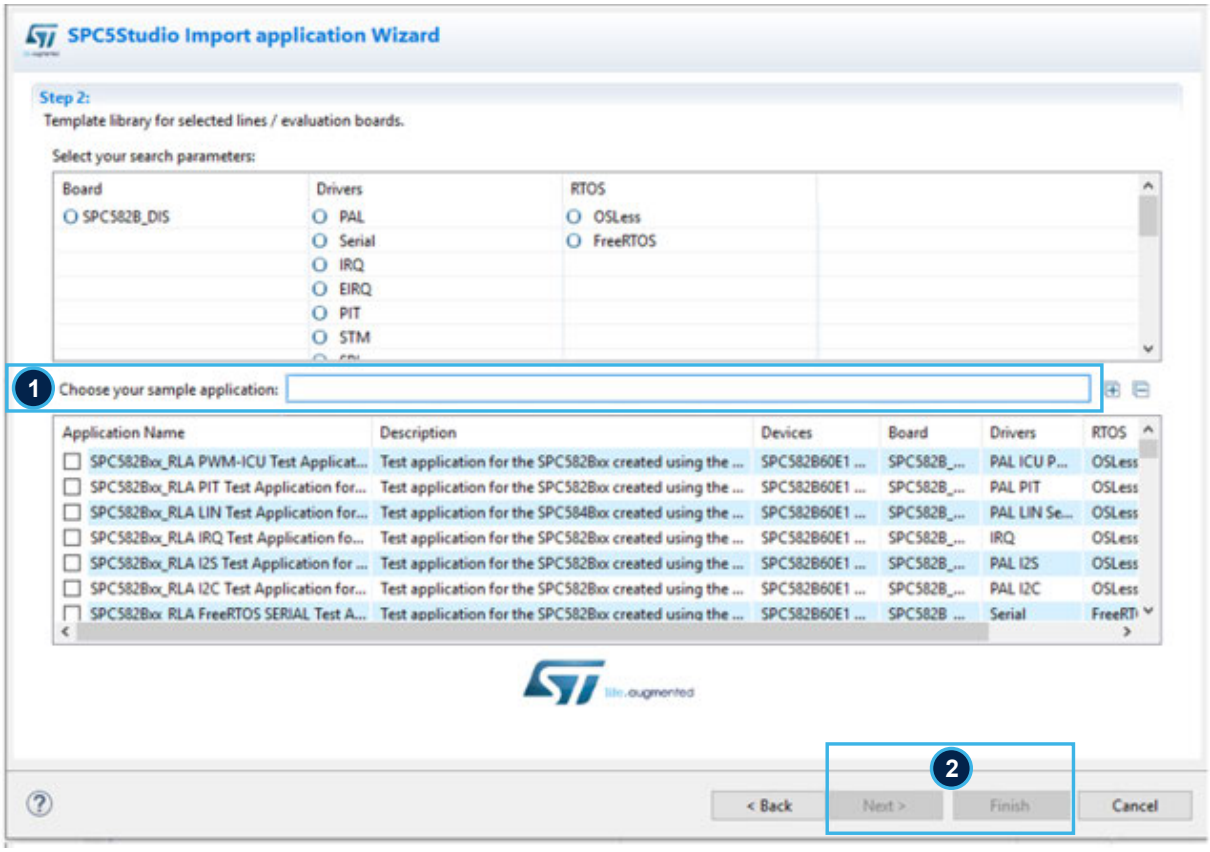
1. Import samples task button
2. Product family selection panel



Step 3. Select the desired application from the library.

Figure 42. SPC5-STUDIO application library

1. application selector
2. confirmation buttons



6.2.2 Mono audio – Test Application (SPC584Bxx and SPC58ECxx)

This demo plays an audio wave file stored in memory.

The main APIs in this demo are:

- `AEK_903D_Init`: initializes the I²C and I2S interface
- `initWaveFile`: initializes the structure that contains the addresses used to identify the beginning and end of each wave file
- `getStartWavFile`: computes the address pointing to the first audio sample of a given wave file
- `GetEndWavFile`: computes the address pointing to the last audio sample of a given WAV file
- `playSound`: plays the samples provided through a pointer to function able to generate audio samples

6.2.3 I²C Software Mono audio – Test Application (SPC582Bxx and SPC58ECxx)

This demo is like the Mono audio – Test Application, but the I²C protocol is implemented via software thanks to the allocation of two GPIO pins suitably configured by AutoDevKit itself.

6.2.4 Mono audio and Diagnostic - Test Application (SPC582Bxx)

This demo shows how to reproduce an audio wave file and how to perform system diagnostics during the PLAY and MUTE states, using the button on the microcontroller board to switch between the two states. The audio reproduction functions are the same as those in the Mono audio - Test Application, so the functions described below relate to diagnostics only.

- `AEK_903D_TriggerOpenLoadInPlayDetection`: triggers the Open Load in Play Detection test
- `AEK_903D_CheckOpenLoadInPlayDetection`: verifies the result of the Open Load in Play Detection test
- `AEK_903D_CheckOffsetCurrent`: verifies the result of the Output Current Offset Detection test

- `AEK_903D_CheckOutputVoltageOffsetDetector`: verifies the result of the Output Voltage Offset Detection test
- `AEK_903D_CheckInputOffsetDetector`: verifies the result of the Input Offset Detection test
- `AEK_903D_Diagnostic`: reads the DB register and reports if a failure condition has occurred (SHORT2VCC, SHORT2GND, OVERCURRENT, UNDERVOLTAGE, OVERTEMPERATURE OVERVOLTAGE)
- `AEK_903D_Mute`: changes from PLAY to MUTE state
- `AEK_903D_StartDCDiag`: changes from MUTE to DC Diag state and to perform the DC diagnostic
- `AEK_903D_CheckDCDiagnostic`: verifies the result of the DC diagnostic.

The red LED on the Discovery control board provides load (speaker) connection status resulting from the Open Load in Play Detection test during PLAY state operation and the DC diagnostic in MUTE:

- The red LED goes on when an open load fault is detected, and the tests performed are self-validated.
- The red LED stays off when the load is correctly connected, or the tests cannot be self-validated.

6.2.5 Stereo audio and Diagnostic - Test Application (SPC582Bxx)

This demo is the stereo version of the Mono audio and Diagnostic application, in which the functions used are duplicated to perform the same diagnostics on both boards (i.e., audio channels).

- The red LED communicate the load status of the first board
- The yellow LED communicates the load status of the second board.

6.2.6 Engine Sound Simulator - Test Application for AVAS purpose (SPC582Bxx)

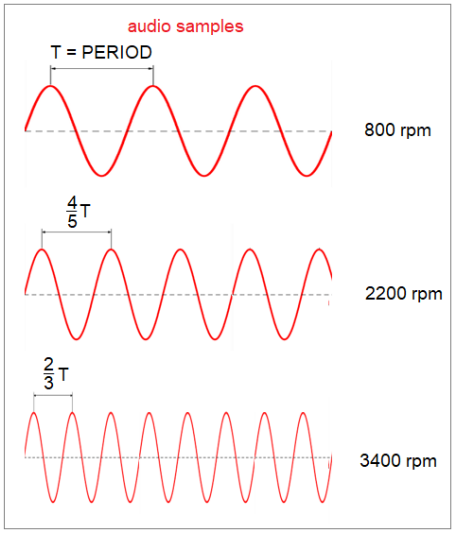
The demo offers an entry-level AVAS implementation using the AEK-AUD-D903V1 AutoDevKit component. The application uses an algorithm to simulate the sound of an internal combustion engine during acceleration and deceleration.

The application allows you to:

1. reproduce a WAV file for a car engine sound recorded in Neutral
2. simulate engine ignition and shutdown sounds through a user button on the control board
3. simulate engine noise during acceleration and deceleration
4. change the volume and rpm values through two potentiometers

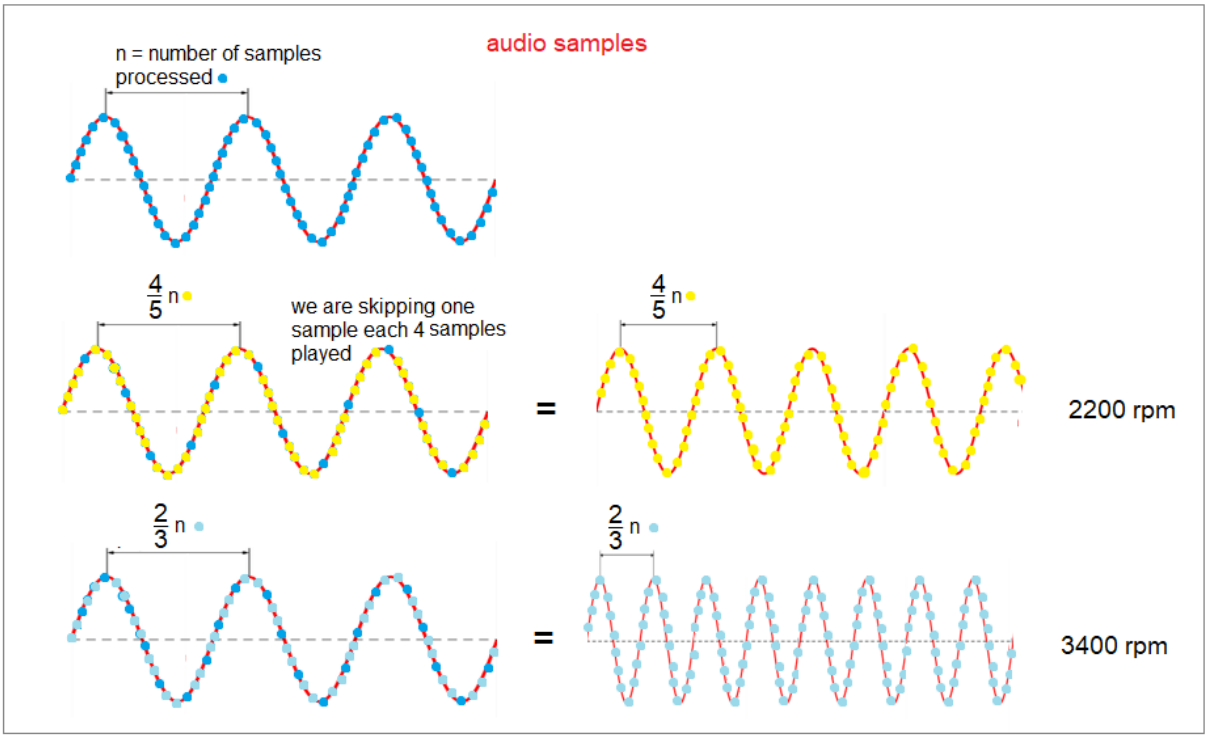
The acceleration and deceleration phases are simulated by increasing and decreasing the number of samples reproduced. For example, consider that for an engine idling in Neutral at 800rpm, its sound may be represented by a sinusoidal wave with period T. We simulate the acceleration phase by decreasing the period (increasing signal frequency), as shown in the following figure.

Figure 43. Audio sample frequency modulation



The samples in the MCU memory are sampled at 44,100 Hz frequency, and frequency modulation is simulated by varying the number of samples to be played. The following image shows the algorithm operating principle.

Figure 44. Varying sample numbers to simulate frequency modulation



The `userFunction()` used in this application has the following characteristics:

1. As soon as the application is started and the ignition command is given, the initial pointer is assigned to the first sample to be played, while the final address is assigned to the last sample.
2. At the end of the first cycle (i.e., when the pointer to the current sample is the same as the pointer of the last sample in the audio file for the first time), the algorithm no longer restarts from the first sample, but 90,000 samples after the first. This effectively simulates the effect of engine start and engine idle with a few seconds of recording.

```
uint32_t userFunction()
      {
         if(acc_first_time)
            {
             wavfilePtr = wavfileBeginPtr = getStartWavFile(0);
         wavfileEndPtr = getEndWavFile(0);
         acc_first_time = false;
      }
   if(wavfilePtr > wavfileEndPtr)
      wavfilePtr =  wavfileBeginPtr + 90000;
   int32_t amplitude = 0;
   uint16_t sampleWav;
   sampleWav = ( *wavfilePtr << 8) | ((*wavfilePtr >> 8) & 0xFF);
   //raw sample, change endianness
   amplitude = (int32_t)sampleWav;
```

The acceleration and deceleration is obtained through voltage variation from a potentiometer. This external input is converted by the microcontroller SARADC, and its value is stored in an `rpm` variable. The higher the value of `rpm`, the greater the number of samples to be skipped.

This method, shown in the code snippet below, can therefore simulate variations in engine rpm sounds in acceleration and deceleration.

```
if (rpm1 <= 2200)
   {
      if (rpm1 == 900)
      {
         if (j < 23)
            wavfilePtr++;
         else
            {
               wavfilePtr+ = 2;
                  j = 0;
            }
         }
         else if (rpm1 == 1000)
         {
            if (j < 20)
               wavfilePtr++
            else
            {
               wavfilePtr+ = 2;
                  j = 0;
            }
         }
         etc...
         …
         …
         …
else if (rpm1 == 2200)
         {
               if (j < 12)
               {
                     wavfilePtr++;
               }
               else
               {
                     wavfilePtr =+ 4;
                        j = 0;
               }
         }
         else//800 rpm
               wavfilePtr++;
```

Revision history

Table 11. **Document revision history**

Date	Version	Changes
19-May-2020	1	Initial release.

Contents

List of figures

List of tables