STM32-Nucleo Shield for real-time audio acquisition and processing

Daidone Luca and Dedor Alessio

July 21, 2024

Contents

1 Project data **2**

2 Project description **2**
2.1 TLV320AIC3101 Codec . . . 2
2.2 VU-meter . . . 3
2.3 I2S Peripheral . . . 3

3 Design and implementation **4**
3.1 Hardware . . . 4
3.2 Software . . . 4
3.2.1 VU-meter . . . 4
3.2.2 TLV320AIC3101 . . . 4
3.2.3 Codec I2C communication . . . 5
3.2.4 Codec I2S communication . . . 6
3.2.5 DMA and ISR handling . . . 7
3.3 Demo . . . 7

4 Project outcomes **7**
4.1 Concrete outcomes . . . 7
4.2 Learning outcomes . . . 7
4.3 Existing knowledge . . . 8
4.4 Problems encountered . . . 8

5 Honor Pledge **8**

1 Project data

- Project supervisor: prof. Terraneo Federico
- Describe in this table the group that is delivering this project:

Last and first name	Person code	Email address
Daidone Luca	10723194	luca.daidone@mail.polimi.it
Dedor Alessio Walter	10654363	alessiowalter.dedor@mail.polimi.it

- Describe here how development tasks have been subdivided among members of the group:
  The members of the group worked together on all the tasks.
- Links to the project source code: **STM32Nucleo Codec Shield**

2 Project description

The project consists in designing a shield based on the **TLV320AIC3101** codec for **STM32F401RE Nucleo** boards and developing a driver to interface the shield with the Miosix RTOS.

The two main goals are:

1. **Design** and **assemble** a custom "shield" to add audio processing capabilities to the STM32 Nucleo boards.
2. Implement a **driver** to interface the shield with the **embedded RTOS "Miosix"** and exploit the advantages of multi-threaded programming.

2.1 TLV320AIC3101 Codec

The core of the project is the codec, a chip from Texas Instruments capable of sampling audio signals with its ADC functionality, processing the digitized data, and transmitting it to the microcontroller via the I2S protocol. Additionally, it can receive a digital audio stream and convert it back to an analog signal using its DAC capability. Apart from the I2S interface, the codec comes equipped with an I2C interface, used to modify the internal settings (e.g. which input channel to use, ADC gain factor, sampling frequency, internal PLL settings etc.)

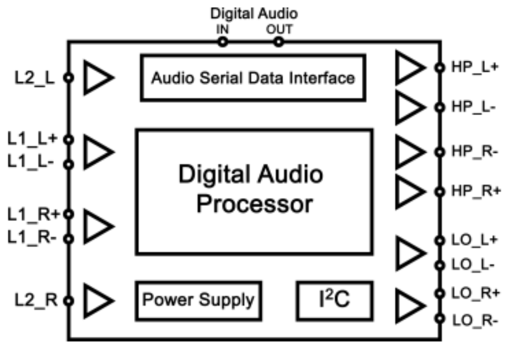

Figure 1: Codec and main functional blocks

To get the device working, implementing an appropriate library was necessary. The library's roles are:

- Setting up codec registers via the I2C protocol
- Managing I2S transmission and reception

2.2 VU-meter

The received audio can be displayed on a VU-meter consisting of 6 LEDs. To accurately represent the audio data value, which can range from 0 to 65535, it must first be converted to a logarithmic scale. Therefore, creating a "VU-meter" class was necessary. This class provides the following functionalities:

- Instantiating a VU-meter object with the corresponding GPIOs as parameters.
- Displaying the sound level by turning on the appropriate number of LEDs.

2.3 I2S Peripheral

Lastly, the STM32 micro controller should be able to communicate with the codec via I2S (Inter-IC Sound) protocol.

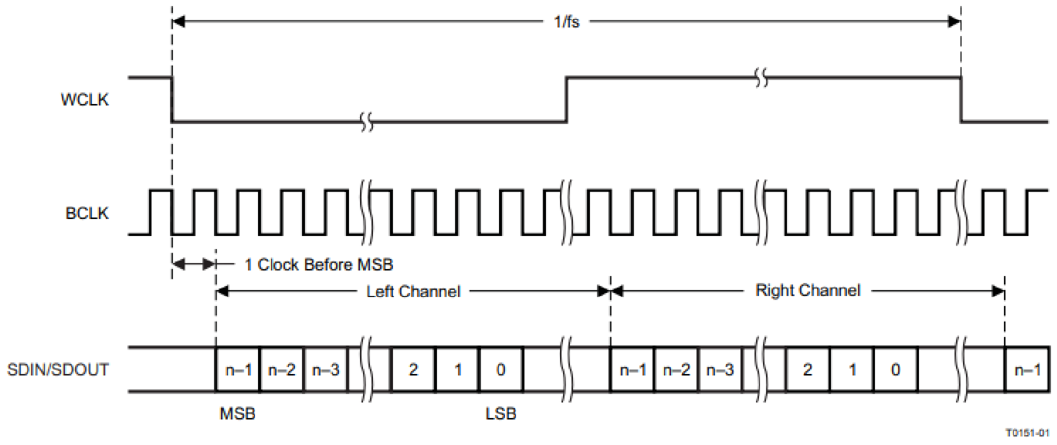

Figure 14. I^{2}S Serial Data Bus Mode Operation

Figure 2: Codec and main functional blocks

I2S is a Master-Slave type protocol in which the Master generates the Master Clock (MCLK) that is shared by both the Master and Slave devices to enable synchronous communication. The other important signals in the I2S protocol include:

- **Word Select (WS) or Left-Right Clock (LRCLK)**: Indicates whether the current word is for the left or right channel.
- **Bit Clock (BCLK)**: Synchronizes the data transmission by indicating when a bit should be read or written.
- **Serial Data I/O (SDIO)**: Carries the actual audio data being transferred.

Since the aim of the project is to process the sound signal in real-time, it is therefore crucial to avoid keeping busy the micro controller. For this reason we chose to employ the I2S in **full-duplex mode**, also enabling both **DMA** and its **interrupts**.

3 Design and implementation

3.1 Hardware

A quick note on the hardware implementation: the shield board for the Nucleo STM32F401RE is equipped with analog front-end, the codec, an LDO to supply the 1.8V, the VU-meter LEDs and input/output 3.5mm jack connectors.

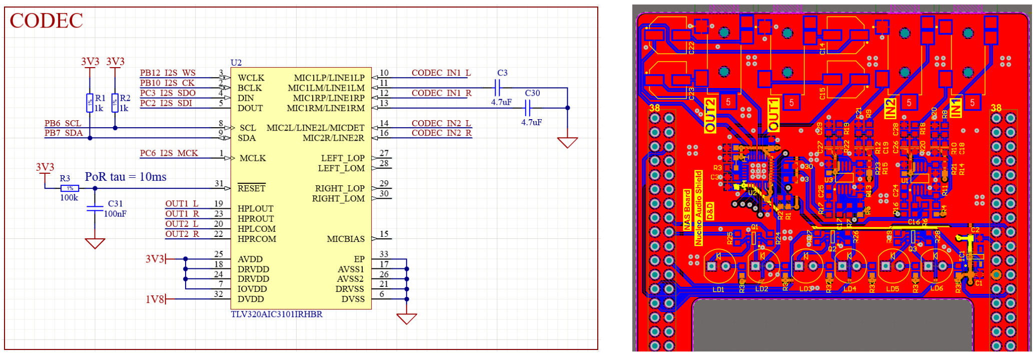

Figure 3: Part of the schematic and layout design

Both the schematic and the layout were produced using Altium Designer, whereas the PCB was produced by JLCPCB.

Links: Schematic.

3.2 Software

Libraries to easily manage all the HW present on the board must be provided. Finally, a simple loopback and a VU-meter showcase should be realized.

3.2.1 VU-meter

The **class** Vumeter is defined to manage the relative hardware, it's implementation can be found inside the vumeter.h file. Some key aspects include:

- The class' **constructor** Vumeter::Vumeter(GpioPin Rled1, GpioPin Rled2, GpioPin Yled1, GpioPin Yled2, GpioPin Gled1) takes the 6 Gpios directly connected to the leds as parameters and set them in output mode.
- The class' **public methods** **void** Vumeter::clear(), **void** Vumeter::setHigh(), **void** Vumeter::showVal(**unsigned int** sound_val) are respectively used to turn off/on all the leds and display the sound level in a log scale.

3.2.2 TLV320AIC3101

The **class** TLV320AIC3101 contains all the methods used to manage the codec, its definition can be found inside TLV320AIC3101.h while the implementation is in the respective .cpp file. Some key aspects include:

- The class can be instanced only once thanks to the **singleton** pattern.
- The I2C communication with the codec is achieved thanks to the **unsigned char** I2C_Receive(**unsigned char** regAddress) and **bool** I2C_Send(**unsigned char** regAddress, **char** data) class' methods.
- The I2S communication with the codec is achieved thanks to the **static bool** I2S_startRx() and **static void** I2S_startTx(**const unsigned short** *buffer_tx) class' methods.

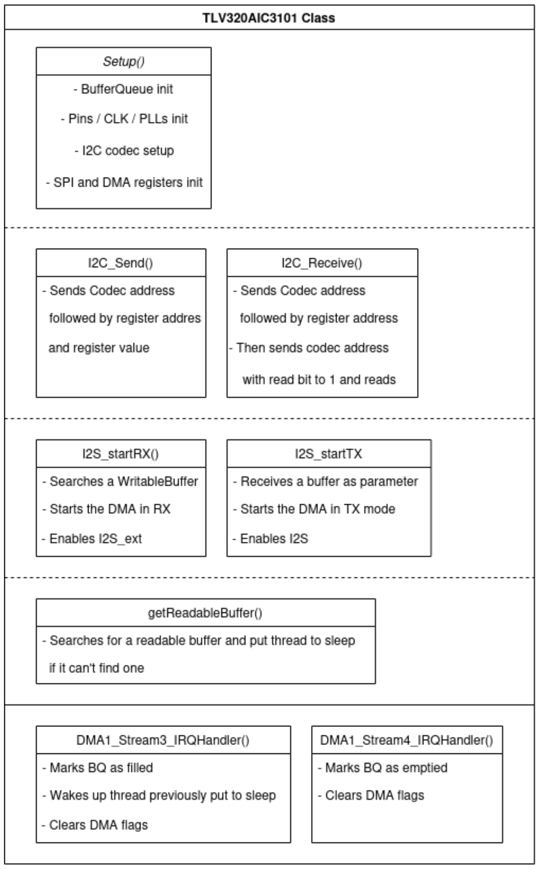
TLV320AIC3101 Class
Setup()
- BufferQueue init
- Pins / CLK / PLLs init
- I2C codec setup
- SPI and DMA registers init
I2C_Send()
- Sends Codec address
followed by register addres
and register value
I2C_Receive()
- Sends Codec address
followed by register address
- Then sends codec address
with read bit to 1 and reads
I2S_startRX()
- Searches a WritableBuffer
- Starts the DMA in RX
- Enables I2S_ext
I2S_startTX
- Receives a buffer as parameter
- Starts the DMA in TX mode
- Enables I2S
getReadableBuffer()
- Searches for a readable buffer and put thread to sleep
if it can't find one
DMA1_Stream3_IRQHandler()
- Marks BQ as filled
- Wakes up thread previously put to sleep
- Clears DMA flags
DMA1_Stream4_IRQHandler()
- Marks BQ as emptied
- Clears DMA flags

Figure 4: TLV320AIC3101 class overview

- The class contains the interrupt service routines called after DMA transfer completion.

The main idea behind the driver is to utilize a BufferQueue structure to both implement double buffering and synchronize between threads and interrupt routines. The driver is composed of different methods, let's see the main functionalities.

3.2.3 Codec I2C communication

To implement the I2C communication we decided to rely on the Miosix's native driver SoftwareI2C<sda,scl>. The codec datasheet specifies how a correct communications should be carried out:

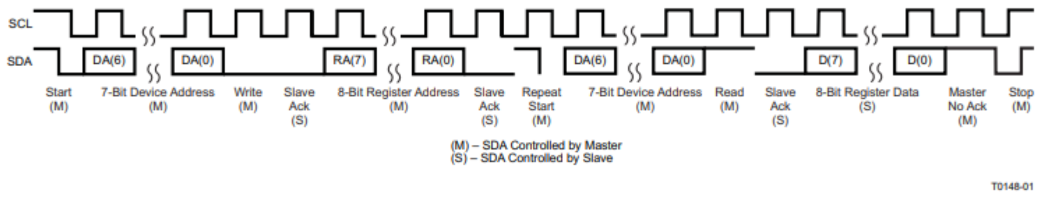

Figure 32. I^{2}C Read

Figure 5: Reading operation with I2C protocol

In order to receive data from the codec, the STM32 should send a start, send codec and register addresses, send another start and then wait for the data. The I2C receive functionality is used only for debug purposes in order to check that the codec's registers are set up correctly.

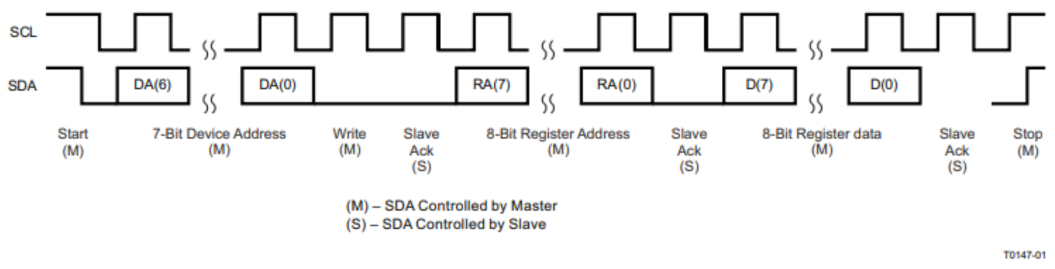

Figure 31. I²C Write

Figure 6: Writing operation with I2C protocol

In order to modify the content of a register of the codec, the write operation must be carried out. The STM32 should send a start, send codec and register addresses and finally the data.

The code can be found at the following link to the repo: TLV320AIC3101.cpp

3.2.4 Codec I2S communication

To setup the I2S peripheral to work in **full-duplex mode** with **DMA and interrupts** enabled three different methods are employed:

1. **void** setup(), is meant to be called only once, it enables peripherals clock (DMA1/SPI2), sets up I2S pins (with correct AF number), configure I2S PLL, configure I2S2 and I2S2 extended working mode (Master/Slave), sets up relative DMAx streams and enables the relative DMA1 (Stream3/Stream4) interrupts.
2. **static bool** I2S_startRx(), it searches a free and writable buffer among the ones of the BufferQueue, sets DMA1 (Stream 3) to transfer from peripheral (I2S) to memory and then starts it. Enables I2S2 extended peripheral.
3. **static void** I2S_startTx(**const unsigned short** *buffer_tx), takes the buffer filled with audio data as a parameter, sets DMA1 (Stream 4) to transfer from memory to peripheral and then starts it. Enables SPI2 (I2S2) peripheral.

The relative code can be found at the following link to the repo: TLV320AIC3101.cpp

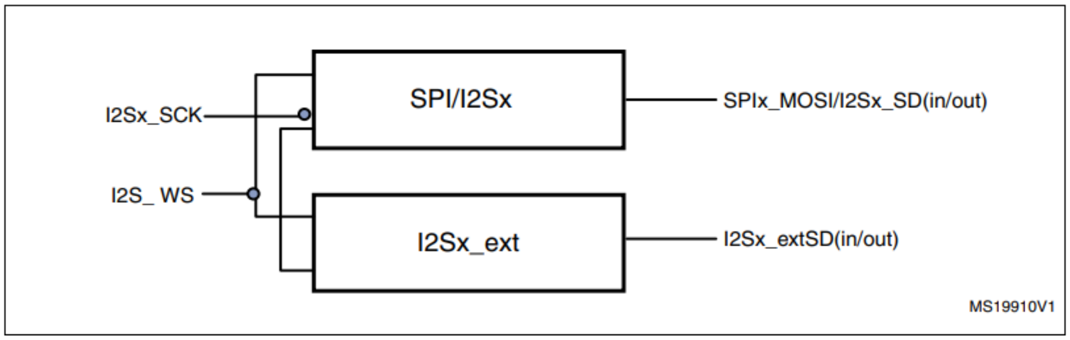

Figure 7: I2S peripherals for full-duplex mode

3.2.5 DMA and ISR handling

The DMA (Direct Memory Access) enables data transactions between memory and peripheral without the need for the CPU to continuously intervene. Enabling interrupts also allows us to provide a fast response when a transaction is completed. Once the interrupts and the DMA are enabled, we just need to define what is the critical work that needs to be done in the respective ISRs.

- **void** __attribute__((naked)) DMA1_Stream3_IRQHandler(), is the ISR called when a reception event is completed. It clears all the interrupt flags and marks the BufferQueue's buffer as "filled", meaning that valid set of audio data is available (it is thus a "readable" buffer).
- **void** __attribute__((weak)) DMA1_Stream4_IRQHandler(), is the ISR called when a transmission event is completed. It clears all the interrupt flags and marks the buffer as "emptied", meaning that the buffer is "writable".

Another crucial method used to properly synchronize the buffers among threads accessing them and ISRs is the getReadableBuff(). This method is called by the thread that needs access to the received audio data. The function looks for a readable buffer, and in case one is not available, it puts the caller thread to sleep. Only once the DMA RX transfer is complete and the respective ISR is executed, then the thread is woken up with IRQwakeup(). This mechanism implements synchronization between thread and ISR.

3.3 Demo

Finally, inside the main.cpp file, we implemented a simple demo application. The main idea is to create a thread that manages the VU-meter (Thread *led_thread) and another thread (**int** main()) that sequentially starts I2S in RX and TX modes to implement a simple loopback using the methods described earlier.

A potential issue could arise if the DMA tries to transmit a buffer that is simultaneously being read by the VU-meter thread. To prevent this, a Lock<Mutex> lock(mutex) is used and shared between both threads. In this way, if the main thread attempts to start data transmission while the mutex is already locked by the VU-meter thread, it must wait until the mutex is released.

4 Project outcomes

4.1 Concrete outcomes

The demo receives the audio signal from the input jack, displays the sound level on the VU-meter and performs a playback toward the output jack. However, some improvements can still be made: the VU-meter class could display the signal envelope or a filtered value of it, since for simplicity now only the first element of the audio buffer is showed on the LEDs. Also the total latency could be lowered by exploiting a better thread synchronization, modifying the buffer size and maybe reaching a total of 4 buffers in the BufferQueue, such as double buffering can be implemented for both reception and transmission.

4.2 Learning outcomes

On the software side, we learned: the basics of C++, we became familiar with the Miosix toolchain, to debug with GDB, to write basic multi-threaded code, to interface with the ARM libraries to write our own driver and we became more confident with git.

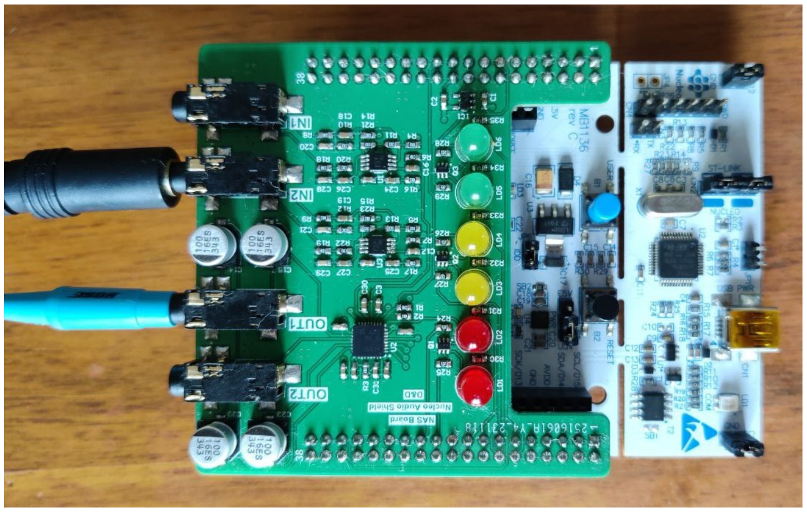

Figure 8: Completed Project

4.3 Existing knowledge

Attending the course of "Sensor System" was helpful since it introduced us to the STM32 world and especially the CubeIDE. In fact, we first used the STM32 IDE to develop a working demo to test the hardware. Obviously, having an electronic background was helpful to design the board but also to understand better how a micro controller works (pre existing knowledge of I2C, I2S, registers, PLLs and CLOCK...).

4.4 Problems encountered

We encountered different problems on both hardware/software level:

- (Hardware) The QFN package of the codec is not easy to solder and this led us to partially damage the output stage. One of the VU-meter's Leds can't be used since is directly connected to the SWD port (luckily it was the one displaying the minimum level of sound).
- (Software) The biggest problem we encountered was to get the I2S peripheral work in Full-duplex mode. In fact, to do so, both I2S and I2S extended must be configured. This crucial information was not mentioned in the STM33 manual.
- (Software) We understood late that to make everything work we needed to pay attention to synchronization and timing. This led us to eliminate all the delayUs() we still had inside the code and also unnecessary for-loops.

5 Honor Pledge

We pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents our original production, unless otherwise cited.
We also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2022.

Group Students' signatures:
Luca Daidone
Alessio Dedor