Programming industrial embedded systems
Laboratory exercise 2
Serial communication & Audio Codec

December 2, 2022

Contents

1 Required hardware, software and additional literature **3**

2 Setting up STM32CubeIDE **4**

2.1 Linux OS 4

2.2 Windows OS 4

2.3 Instructions for using STM32CubeIDE 4

3 Serial communication **7**

3.1 Configure USART1 peripheral and ISR 7

3.2 Implement high-level API functions for sending and receiving characters 13

3.3 Implement the loopback functionality in the main module 14

3.4 Lab outcomes 15

4 Audio interface **15**

4.1 I2C protocol 15

4.2 I2S protocol 15

4.3 Configuration of I2C and I2S peripherals 16

4.4 Audio interface configuration 17

4.5 Generating audio signals 21

4.6 Lab outcomes 22

1 Required hardware, software and additional literature

For this laboratory exercise, you will need the following:
Hardware:

– STM32F4DISCOVERY development kit,
– USB-mini cable for power supply and programming,
– UART/USB bridge.

Software:

– cross compiler for STM32F4 microcontroller,

– ST-link for flashing binary image to hardware,

– STM32CubeMX tool for generating HAL,

– STM32CubeIDE development environment [1].

Literature and materials:

– user guide covering the STM32F4 microcontroller family[6],

– datasheet for specific microcontroller part on the development board[7],

– programming manual[5],

– HAL library user manual[9].

Additional literature includes the following:

– STM32F4DISCOVERY board user manual[8],

– STM32CubeMX user manual[4],

– STM32CubeIDE documentation[3].

Additional materials for this lab exercise are packed conveniently in the archive *PIES_2020_Lab2_Pack.zip*, available from FER web page, under the section *Laboratory exercises.* This lab guideline will address above references at some points and it is advisable to read referenced parts of the literature whenever further clarifications and instructions are needed.

In the first part of this exercise, we will use USART peripheral to implement a serial interface that will receive characters from a computer terminal and echo back the received characters. In the second part, we will demonstrate how to configure on-board audio codec and generate audio signals. We will show how to write and debug our application in STM32CubeIDE.

[1]available from ST webpage

2 Setting up STM32CubeIDE

2.1 Linux OS

Download STM32CubeIDE installer from ST webpage. If you are using Debian or Ubuntu, choose STM32CubeIDE-DEB package. For Fedora/CentOS, choose STM32CubeIDE-RPM. For other distributions, choose generic Linux installer (STM32CubeIDE-Lnx).

After the installer is downloaded, change your working directory the location of the installer file. Unzip the installer archive. Enter the following command to start the installer:

```
$ sudo sh st-stm32cubeide_<VERSION>_<ARCHITECHURE>.<PACKAGE>
```

Follow the further instructions provided through the console window.

2.2 Windows OS

Download STM32CubeIDE Windows installer from ST webpage. Launch the product installer (`.exe` file):

```
st-stm32cubeide_VERSION_ARCHITECHURE.exe
```

where:

– `VERSION` is the actual product version and build date, e.g. `1.0.0_2026_20190221_1309`,

– `ARCHITECTURE` is the architecture of the target host computer to run STM32CubeIDE, e.g. `x86_64`.

In the installer, the *Selection of components* dialog will appear. Select *ST-Link drivers* and *ST-Link server*. Then click *Install* to start the installation.

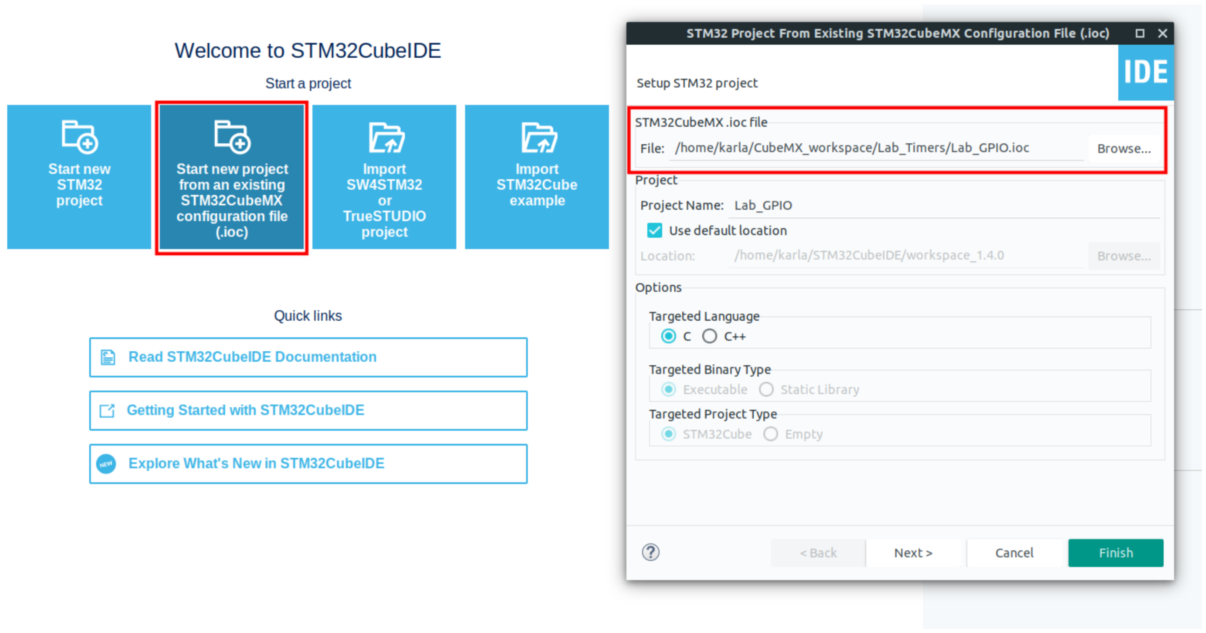
2.3 Instructions for using STM32CubeIDE

STM32CubeIDE is based on the Eclipse development environment which uses *perspectives*. A perspective is a set of windows dedicated to some purpose, e.g. the *C/C++ perspective* is used for writing and editing code, while the *Debug* perspective is used for testing and debugging.

When STM32CubeIDE is started, the workspace selection dialog appears. A workspace is a directory that includes information about projects and project directories. The selected workspace can be changed at any time through *File - Switch Workspace* option.

In this exercise, we will start our new project based on the configurations made in STM32CubeMX. This is done by clicking *Start new project from an existing STM32CubeMX configuration file (.ioc)* in IDE home screen. In this step, we will use CubeMX project created in previous laboratory exercise. Before importing the project, open STM32CubeMX and set STM32CubeIDE as toolchain under *Project Manager* tab. Click on *Generate Code*. When the code is generated, click on *Open project* in the pop-up window. This will launch STM32IDE.

Alternatively, you can create a new project based on the configurations defined in `.ioc` file (generated by STM32CubeMX). See Fig. 1 for instructions.


Welcome to STM32CubeIDE
Start a project
Start new STM32 project
Start new project from an existing STM32CubeMX configuration file (.ioc)
Import SW4STM32 or TrueSTUDIO project
Import STM32Cube example
Quick links
Read STM32CubeIDE Documentation
Getting Started with STM32CubeIDE
Explore What's New in STM32CubeIDE
STM32 Project From Existing STM32CubeMX Configuration File (.ioc)
IDE
< Back
Next >
Cancel
Finish

Setup STM32 project

STM32CubeMX .ioc file
File: /home/karla/CubeMX_workspace/Lab_Timers/Lab_GPIO.ioc Browse...

Project
Project Name: Lab_GPIO
✓ Use default location
Location: /home/karla/STM32CubeIDE/workspace_1.4.0 Browse...

Options
Targeted Language
[✓ C] C++
Targeted Binary Type
[✓ Executable] Static Library
Targeted Project Type
[✓ STM32Cube] Empty

Figure 1: Importing STM32CubeMX configuration file.

The project will open in *C/C++* perspective. Compile the project by clicking *Project - Build Project*. **Note:** running the compiler, starting the debug session etc. can also be done from the shortcuts menu (below the main menu bar), see Fig. 2.

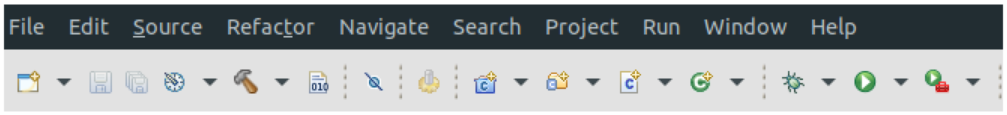


Figure 2: Shortcuts menu for quicker access to project actions.

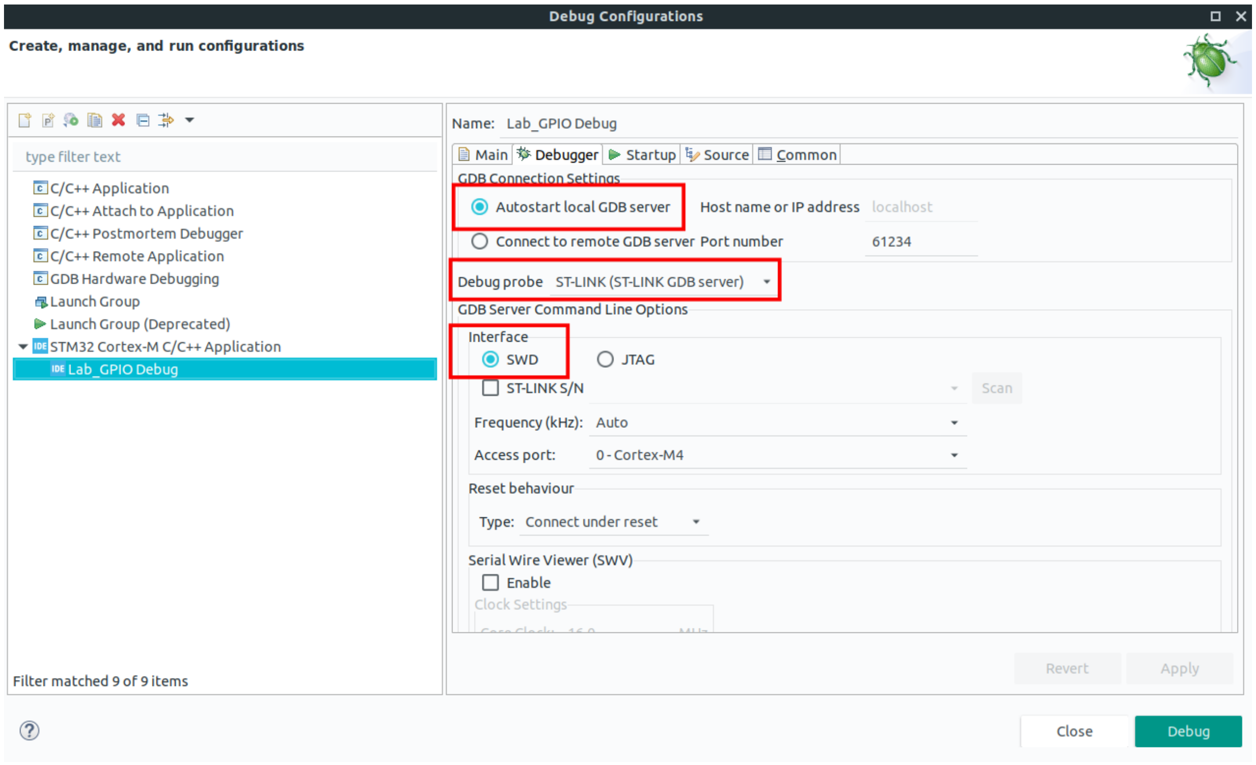
Once the code is compiled and linked with no errors or warnings, we can demonstrate the debugging process. The debugging environment is launched by clicking *Run - Debug*. Before launching our first debug session, we have to configure the debugging settings. This is done by clicking *Run - Debug Configurations*.

In the *Debug Configurations* window, right-click on *STM32 Cortex-M C/C++ Application* and select *New Configuration*. Make sure that the settings under *Debugger* panel are configured as shown in Fig. 3. Make sure that you selected SWD port as debugging interface, not JTAG. Next, click on *Debug*. This will switch the current perspective to *Debug*. If flashing the application has succeeded, you should see the following output:

```
File download complete
Time elapsed during download operation: 00:00:00.294

Verifying ...

Download verified successfully
```


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
- GDB Hardware Debugging
- Launch Group
- Launch Group (Deprecated)
- STM32 Cortex-M C/C++ Application
  - Lab_GPIO Debug

Filter matched 9 of 9 items

Name: Lab_GPIO Debug

Main | Debugger | Startup | Source | Common

GDB Connection Settings

[✓ Autostart local GDB server] Connect to remote GDB server

Host name or IP address: localhost

Port number: 61234

Debug probe: ST-LINK (ST-LINK GDB server)

GDB Server Command Line Options

Interface: [✓ SWD] JTAG

☐ ST-LINK S/N

Frequency (kHz): Auto

Access port: 0 - Cortex-M4

Reset behaviour

Type: Connect under reset

Serial Wire Viewer (SWV)

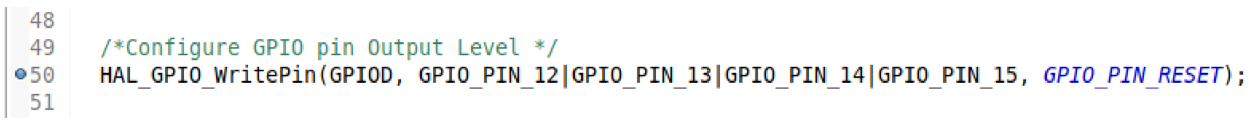
☐ Enable

Clock Settings

Figure 3: Debugger settings.

You should also see the LED LD1 COM blinking. This indicates that communication between ST-Link and PC is in progress.

The first breakpoint is automatically set on `main()` function. At this point everything works, but the program is halted. Breakpoints can be set by double-clicking a line number (e.g. line 50 in Fig. 4). The



```
48
49   /*Configure GPIO pin Output Level */
●50  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15, GPIO_PIN_RESET);
51
```

Figure 4: Adding a breakpoint.

code will stop execution when it reaches this line. You can then use the *Step Into*, *Step Over*, and *Step Return* buttons:

- *Step Into*: If the current line is a function call, step into that function's definition to execute its lines one at a time.
- *Step Over*: Execute the current line of code. If current line is a function call, execute all the code within that function.
- *Step Return*: Execute the rest of the code from the current function and return from the function.

To continue program execution until the next breakpoint, use *Resume*.

3 Serial communication

Assignment:
Make a loopback serial interface that will receive characters from a computer terminal (e.g. HyperTerm) and echo back all received characters. Use USART1 (Universal synchronous asynchronous receiver transmitter) interface for communication with computer. As STM32F4DISCOVERY board does not have RS-232 interface implemented on board, and most of modern computers do not have RS-232 port, it is advisable to use either UART/USB bridge interface or UART/RS232 + RS232/USB interfaces to connect with computer. Set the communication parameters to 115200, 8, N, 1. The solution must use interrupt ISR for buffering the incoming characters, while sending the characters does not have to be buffered.

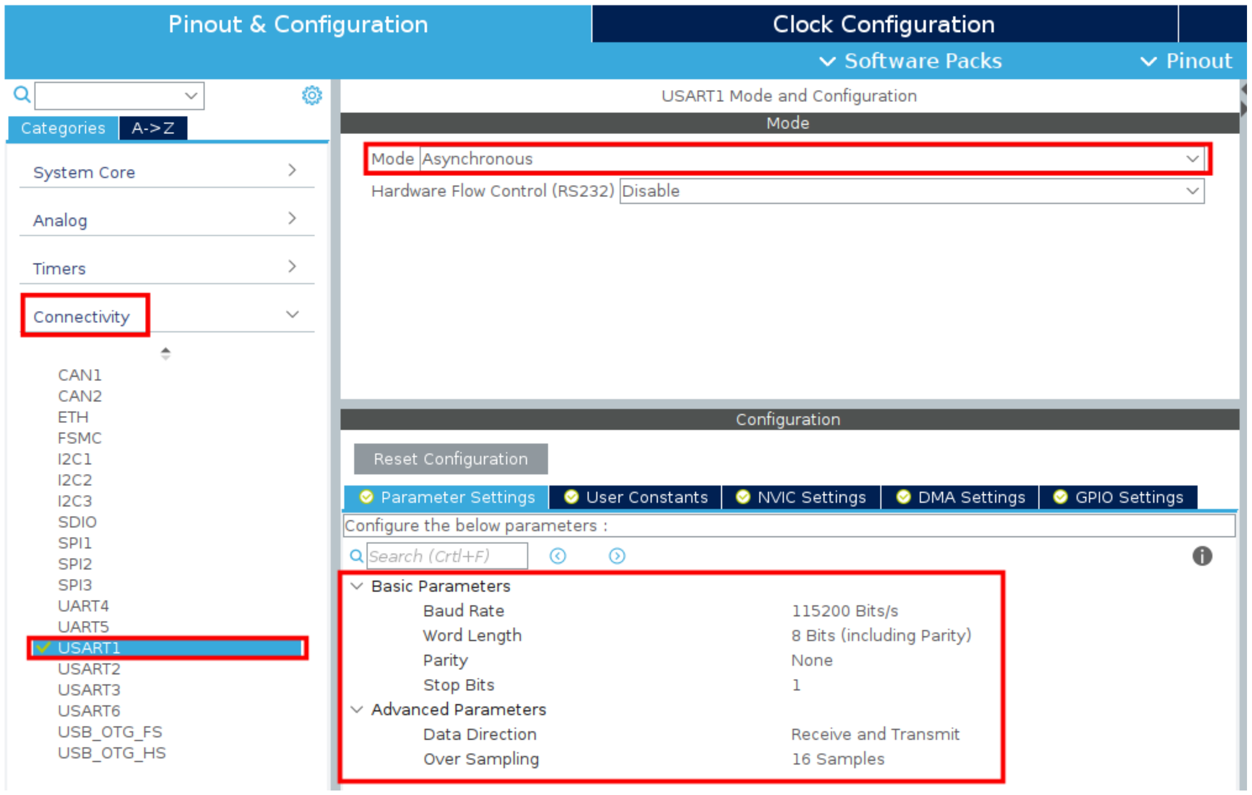
3.1 Configure USART1 peripheral and ISR

As described in the previous laboratory exercise, create a new STM32F407 project in STM32CubeMX. We can add USART1 peripheral from *Pinout & Configuration* window under *Connectivity* panel. Under *Mode* option, choose *Asynchronous*. We will set the serial communication parameters as shown in Table 1.

Parameter	Value
Baud Rate	115200
Word Length	8 bits
Parity	None
Stop Bits	1
Data Direction	receive and transmit
Over Sampling	16 samples

Table 1: Serial communication parameters.

The parameter settings are shown in Fig. 5.



Mode Asynchronous

Hardware Flow Control (RS232) Disable

Basic Parameters

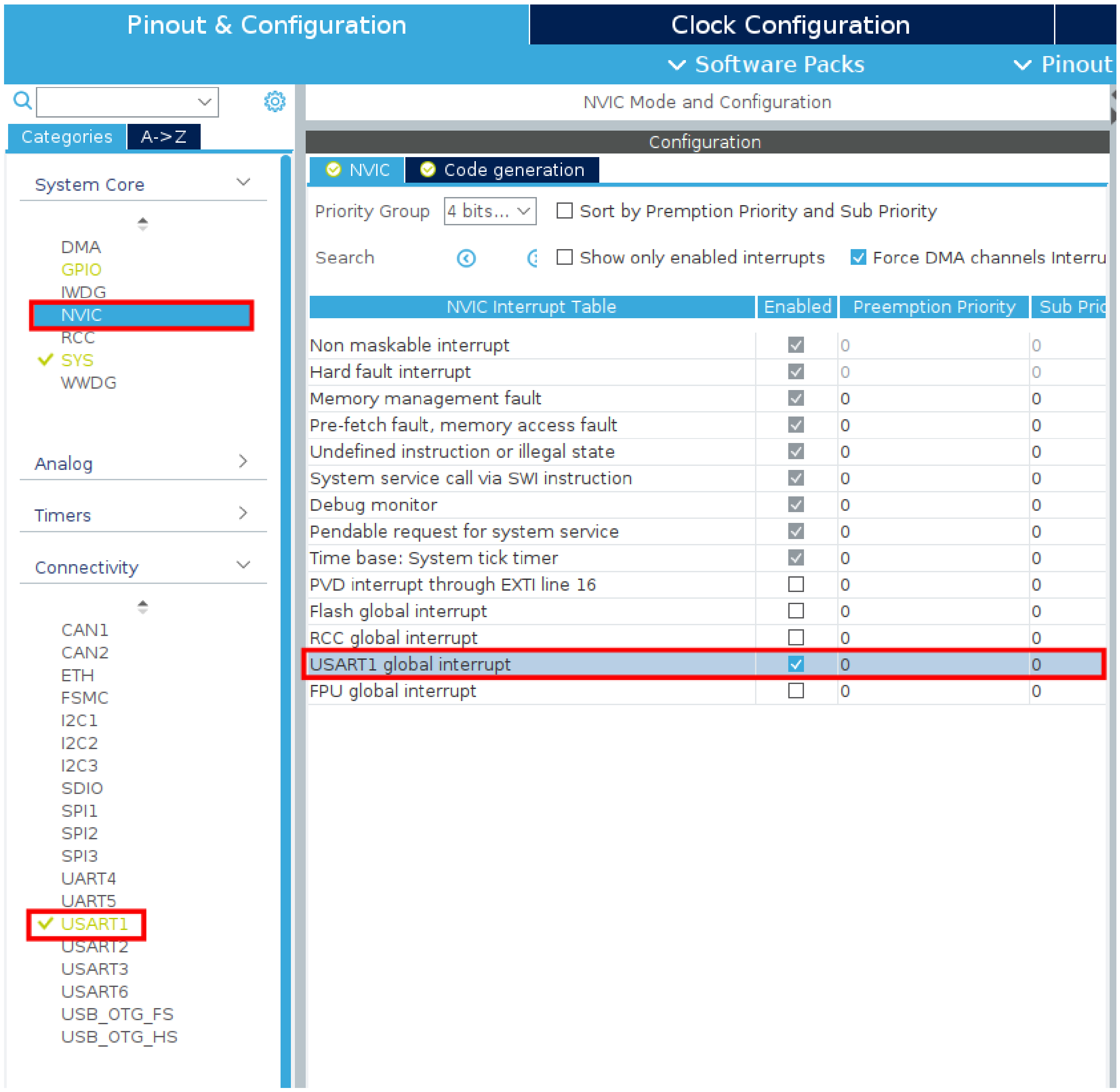
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Advanced Parameters

Data Direction	Receive and Transmit
Over Sampling	16 Samples

Figure 5: Configuring serial communication parameters.

Next, we must enable USART interrupt. Under *System Core*, click on *NVIC*. Enable the USART1 global interrupt as shown in Fig. 6.



Priority Group: 4 bits...
☐ Sort by Premption Priority and Sub Priority
☐ Show only enabled interrupts
✓ Force DMA channels Interru

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Pri
Non maskable interrupt	✓	0	0
Hard fault interrupt	✓	0	0
Memory management fault	✓	0	0
Pre-fetch fault, memory access fault	✓	0	0
Undefined instruction or illegal state	✓	0	0
System service call via SWI instruction	✓	0	0
Debug monitor	✓	0	0
Pendable request for system service	✓	0	0
Time base: System tick timer	✓	0	0
PVD interrupt through EXTI line 16	☐	0	0
Flash global interrupt	☐	0	0
RCC global interrupt	☐	0	0
USART1 global interrupt	✓	0	0
FPU global interrupt	☐	0	0

Figure 6: Enabling USART global interrupt.

Configure project settings as described in subsection 2.3. Click on *Generate Code*. STM32CubeIDE should launch automatically and import the new project into workspace.

We will add necessary definitions and API function prototypes in `uart.h` file:

```
/* USER CODE BEGIN Private defines */
#define     BUFSIZE 16
/* USER CODE END Private defines */

...

/* USER CODE BEGIN Prototypes */
extern char RX_BUFFER[BUFSIZE];
```

```c
extern int RX_BUFFER_HEAD, RX_BUFFER_TAIL;

void USART1_SendChar(uint8_t c);
int USART1_Dequeue(char* c);
/* USER CODE END Prototypes */
```

We define the size of first-in first-out (FIFO) buffer for received characters (`BUFSIZE = 16`) that can be easily changed by modifying `#define` values in this header file. For a minimum USART1 high-level API interface we define two functions:

– `USART1_SendChar()` - a high-level function that sends a single character through the USART1 interface; the function guarantees that character will be sent, but does not provide any guarantees about the timing,

– `USART1_Dequeue()` - the function fetches a single character from a queue of received characters; in case that there is more than one character waiting to be processed in a queue, the one that first arrived will be fetched; function is non-blocking, meaning that it will not block on empty queue (it will return 0 if there are no characters to be processed); otherwise, the function will return 1, and fetched character will be returned through the byref parameter (supplied as a pointer to char).

At the beginning of `main.c` module, we first add the internal global variables that will take care of received characters buffering:

```c
/* USER CODE BEGIN PV */
char RX_BUFFER[BUFSIZE];
int RX_BUFFER_HEAD, RX_BUFFER_TAIL;
uint8_t rx_data;
/* USER CODE END PV */
```

The character array `RX_BUFFER` holds all unprocessed incoming characters. The buffer is organized as a FIFO queue using static C character array. Therefore, we need auxiliary variables pointing to head (`RX_BUFFER_HEAD`) and tail (`RX_BUFFER_TAIL`) to implement queue functionality with static array. This global `RX_BUFFER` queue is filled with characters by USART1 ISR as they arrive. USART1 ISR is executed on each received character, and its only job is to place newly received character in `RX_BUFFER` FIFO (ISR must be very short, and provide only minimum necessary work with I/O registers). The main program will dequeue characters from `RX_BUFFER` FIFO in a main control loop. Buffering of incoming characters enables less strict timings in the main control loop, which can be even more relaxed by increasing the FIFO depth (`BUFSIZE` constant).

This scenario holds for all microcontrollers with hardware USART FIFO length of one character. Some microcontrollers have hardware FIFO depth of more than one character, and some (like STM32F4 family) provide hardware FIFO in RAM by means of DMA. In this example will shall use only software buffering of received characters, and no support for buffering of transmitted characters.

Now lets take a look at `MX_USART1_UART_Init()` function which initializes USART. USART is initialized via UART handle structure `UART_HandleTypeDef` which is defined in `stm32f4xx_hal_uart.h`:

```c
typedef struct __UART_HandleTypeDef
{
  USART_TypeDef            *Instance;   /*!< UART registers base address        */
  UART_InitTypeDef         Init;        /*!< UART communication parameters      */
  uint8_t                  *pTxBuffPtr; /*!< Pointer to UART Tx transfer Buffer */
  uint16_t                 TxXferSize;  /*!< UART Tx Transfer size              */
  __IO uint16_t            TxXferCount; /*!< UART Tx Transfer Counter           */
  uint8_t                  *pRxBuffPtr; /*!< Pointer to UART Rx transfer Buffer */
  uint16_t                 RxXferSize;  /*!< UART Rx Transfer size              */
  __IO uint16_t            RxXferCount; /*!< UART Rx Transfer Counter           */
  DMA_HandleTypeDef        *hdmatx;     /*!< UART Tx DMA Handle parameters      */
  DMA_HandleTypeDef           *hdmarx;     /*!< UART Rx DMA Handle parameters              */
  HAL_LockTypeDef             Lock;        /*!< Locking object                             */
  __IO HAL_UART_StateTypeDef gState;       /*!< UART state information related to global
                                                Handle management and also related to Tx
                                                operations. This parameter can be a value
                                                of @ref HAL_UART_StateTypeDef              */
  __IO HAL_UART_StateTypeDef RxState;      /*!< UART state information related to Rx
                                                operations. This parameter can be a value
                                                of @ref HAL_UART_StateTypeDef              */
  __IO uint32_t               ErrorCode;   /*!< UART Error code                            */
...
} UART_HandleTypeDef;
```

The `HAL_UART_Init()` function initializes UART module according to the parameters specified in `UART_HandleTypeDef` structure and creates the associated handle.

```
void MX_USART1_UART_Init(void) {
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK) {
    Error_Handler();
  }
}
```

In the initialization function, we should enable UART interrupt. There are various sources of UART interrupt (e.g. transmission complete, parity error etc.). In this example, UART interrupt should be triggered when data is stored in receive data register. To enable *Receive Data register not empty* interrupt, add the following line to initialization function:

```
__HAL_UART_ENABLE_IT(&huart1, UART_IT_RXNE);
```

The initialization of pins and peripheral clock corresponding to USART1 interface is done in `HAL_UART_MSPInit()` function:

```
...
__HAL_RCC_GPIOA_CLK_ENABLE();
/**USART1 GPIO Configuration
PB6     ------> USART1_TX
PB7     ------> USART1_RX
*/
GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
...
```

By default, PB6 and PB7 are plain GPIO pins without any corrspondance with USART1. How to know which GPIO pins we can be used as USART1 TX and RX pins? This cannot be answered just by reading STM32F4 family guide, because there are too many different microcontrollers sub-types and IC packages that differ in number of pins, exposed GPIO ports etc. Therefore, we need to open the datasheet of the microcontroller we use (e.g. STM32F407VG), and find the chapter *Pinouts and pin description.* In Table 7 *STM32F40x pin and ball definitions* there are all pin definitions, along with alternate functions, for all package types. For microcontroller on STM32F4DISCOVERY board (STM32F407VG) we need to refer to LQFP100 package. In the column *Alternate functions*, we can see all possible alternate functions that can be assigned to some pin.

Alternate functions for pins PB6 and PB7 for LQFP100 package are shown in Tbl. 2.

Pin number (LQFP100)	Pin name (after reset)	Pin type	I/O structure	Alternate functions
92	PB6	I/O	FT	I2C1_SCL / TIM4_CH1 / CAN2_TX / DCMI_D5 / USART1_TX / EVENTOUT
93	PB7	I/O	FT	I2C1_SDA / FSMC_NL / DCMI_VSYNC / USART1_RX / TIM4_CH2 / EVENTOUT

Table 2: Alternate functions for pins PB6 and PB7.

Since we decided to use PB6 and PB7, we must change their mode to alternate function:

```
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Alternate = GPIO_AF7_USART1;
```

The next step is to implement the ISR. Firstly, we will add a user-defined interrupt handler. Add the following lines to `usart.c`:

```
void USER_UART_IRQHandler(UART_HandleTypeDef *huart) {
  if( huart->Instance == USART1 ) {
    rx_data = __HAL_UART_FLUSH_DRREGISTER( huart );

    static char rx_head;
    rx_head = RX_BUFFER_HEAD + 1;
    if( rx_head == BUFSIZE ) {
      rx_head = 0;
    }
    if( rx_head != RX_BUFFER_TAIL ) {
      RX_BUFFER[RX_BUFFER_HEAD] = rx_data;
      RX_BUFFER_HEAD = rx_head;
    }
  }
}
```

This function shall be called every time a character is received through UART. It is needed to check whether the interrupt was caused by USART1. The `__HAL_UART_FLUSH_DRREGISTER()` macro function is used for storing the content of UART data register. It automatically clears the USART interrupt pending bit. When a character is received, it is placed into global received characters queue. In that way, ISR job is finished as fast as possible, doing minimum interaction with peripheral registers, without doing any actual application-specific job. The rest of the code above will check whether the newly received character will cause the buffer overrun, and put it into the buffer only if there is a space left.

The user-defined handler shall be called from global interrupt handler which is implemented in `stm32f4xx_it.c`:

```
void USART1_IRQHandler(void)
{
  HAL_UART_IRQHandler(&huart1);
}
```

Add a call to user-defined handler after the call to `HAL_UART_IRQHandler()`:

```
void USART1_IRQHandler(void)
{
  HAL_UART_IRQHandler(&huart1);
  USER_UART_IRQHandler(&huart1)
}
```

3.2 Implement high-level API functions for sending and receiving characters

After we properly initialized USART1 peripheral and prepared logic in ISR regarding the global buffers, we can finally implement high-level API functions for sending and receiving characters from the main program. Add the following functions to `usart.c`:

```c
void USART1_SendChar(uint8_t c) {
  HAL_UART_Transmit(&huart1, &c, sizeof(c), 10);
}

int USART1_Dequeue(char* c) {
  int ret;
  ret = 0;
  *c = 0;

  HAL_NVIC_DisableIRQ(USART1_IRQn);

  if (RX_BUFFER_HEAD != RX_BUFFER_TAIL) {
    *c = RX_BUFFER[RX_BUFFER_TAIL];
    RX_BUFFER_TAIL++;

    if (RX_BUFFER_TAIL == BUFSIZE) {
      RX_BUFFER_TAIL = 0;
    }

    ret = 1;
  }

  HAL_NVIC_EnableIRQ(USART1_IRQn);
  return ret;
}
```

Let us consider the function `USART1_SendChar()`. This function will try to send a character `c` by calling function `HAL_UART_Transmit()` function. Let's take a look at this function's prototype:

```c
HAL_StatusTypeDef HAL_UART_Transmit( UART_HandleTypeDef *huart, uint8_t *pData, uint16_t
    Size, uint32_t Timeout );
```

The first parameter to `HAL_UART_Transmit()` function is a pointer to a `UART_HandleTypeDef` structure. Parameter `pData` is a pointer to data buffer which will be sent, while `Size` parameter corresponds to the amount of data to be sent. Timeout duration is given as the amount of HAL ticks. Since the default frequency of HAL tick is 1 kHz, a timeout value of 10 means that the duration of timeout equals 10 ms.

The function `USART1_Dequeue()` fetches single character (the oldest one) from the receive characters FIFO buffer. Since the receive buffer `RX_BUFFER` is a global variable that can be accessed both from the main function and USART1 interrupt service routine, care must be taken because this resource is shared between two lines of code execution. Therefore, any attempt in the main function to access or alter the values in `RX_BUFFER` or `RX_BUFFER_HEAD` and `RX_BUFFER_TAIL`, should be done in atomic way within the critical section where USART1 ISR must be temporarily disabled. The critical section is realized by simply enabling and disabling the USART1 ISR:

```c
NVIC_DisableIRQ(USART1_IRQn);
...
NVIC_EnableIRQ(USART1_IRQn);
```

Within a critical section, we first check whether the buffer is not empty, and in that case we fetch the oldest value to byref parameter (`char *c`), adjust the value of the buffer tail, and set the `ret` flag to 1, indicating that receive buffer was not empty when `USART1_Dequeue()` function was called:

```
if (RX_BUFFER_HEAD != RX_BUFFER_TAIL) {
  *c = RX_BUFFER[RX_BUFFER_TAIL];
  RX_BUFFER_TAIL++;

  if (RX_BUFFER_TAIL == BUFSIZE) {
    RX_BUFFER_TAIL = 0;
  }

  ret = 1;
}
```

If the receive buffer is empty, the function returns 0, and the content of byref parameter `char* c` should be ignored.

3.3 Implement the loopback functionality in the main module

The `main.c` module source code implementing the loopback interface using high-level functions from `usart.c` is very simple:

```
...
int main(void) {
  char c;
...
  MX_USART1_UART_Init();

  while(1) {
    if (USART1_Dequeue(&c) != 0) {
      USART1_SendChar(c);
    }
  }
}
```

The `main.h` header contains all necessary common definitions and imports the `usart.c` module functionality. Calling the function `USART1_Init()` will initialize USART1 peripheral (communication settings), GPIO pins, interrupt service routine (IRQ triggers, ISR processing), and housekeeping variables internal to `usart.c` module (`TxReady` flag and `RX_BUFFER` queue). If we want to change anything in the current USART1 setup, we can do that in source code of `usart.h` and `usart.c`.

The main function contains an infinite loop that continously checks for incoming characters. Characters are buffered to `RX_BUFFER` in USART1 ISR, and the main loop does not directly poll hardware registers, neither it must finish the current character processing before arrival of the new character. Timing restrictions are relaxed because ISR takes care of fast characters buffering into temporary memory storage. Since the `RX_BUFFER` is a memory resource shared between the main program and the USART1 ISR, reading and writing variables and data related to `RX_BUFFER` must be done in atomic way. Therefore, the `usart.h` header does not expose `RX_BUFFER` related variables outside of the module to prevent direct access in non-atomic way from external modules. The atomicity of `RX_BUFFER` access is achieved by implementing the critical section in `USART1_Dequeue()` function that wraps all internal operations on `RX_BUFFER`, and returns the result of operation and read character, if any.

If the character was successfully read, it is sent through UART1 interface right away. This is done by calling `USART1_SendChar()` function. `HAL_UART_Transmit()` function transmits data in blocking mode, which means that it will prevent new character transmission for a given amount of time (timeout) to make sure that USART1 interface is ready for a new transmission (i.e. last character is completely transmitted).

3.4 Lab outcomes

Students must individually accomplish the following outcomes:

– working STM32CubeIDE project, following the guidelines for this lab exercise, with the same remarks as in the previous chapters regarding the project building and transfering to STM32F4DISCOVERY development board,

– program that demonstrates loopback functionality,

– extend the code example in a way that the program exits the loopback functionality when it receives 'x' character from PC, and sends back the message "now exiting loopback mode"; write your own `printf()` function to send message (i.e. null-terminated string) through the USART1 interface, based on functions presented in this part of exercise.

4 Audio interface

The STM32F4 Discovery board comes with an audio digital-to-analog (DAC) converter, the Cirrus Logic CS43L22[1]. The STM32F407VG microcontroller controls the audio DAC through Inter-Integrated Circuit (I2C) interface and processes digital audio data through Inter-IC-Sound (I2S) interface.

In this exercise, we will generate audio signals and configure the audio interface to output the signals through audio mini-jack connector.

4.1 I2C protocol

I2C is a syncronous serial communication protocol used for communication between a master device (or multiple masters) and a single or multiple slave devices. I2C uses two wires for data transmission: Serial Data (SDA) and Serial Clock (SCK). Since I2C does not have a slave select line (like SPI), it uses addressing: the master sends the address of the slave it wants to communicate with to every slave connected to the I2C bus. The addressed slave responds by sending acknowledge (ACK) bit. I2C data frame is always 8 bits long and sent with MSB first. Each data frame is followed by an ACK bit to verify that the frame has been received successfully. After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

4.2 I2S protocol

I2S is a standard interface for transmission of digital data among analog-to-digital converters (ADCs), DACs, digital filters, digital signal processors and other components of digital audio systems. I2S interface consists of three signals: Serial Data (SD), Word Select (WS) and Serial Clock (SCK). Data is driven on the SD line, MSB first. The state of WS line corresponds to the audio channel (right or left) that is currently being transmitted. Depending on which component generates WS and SCK, I2S supports three configurations shown in Fig. 7.

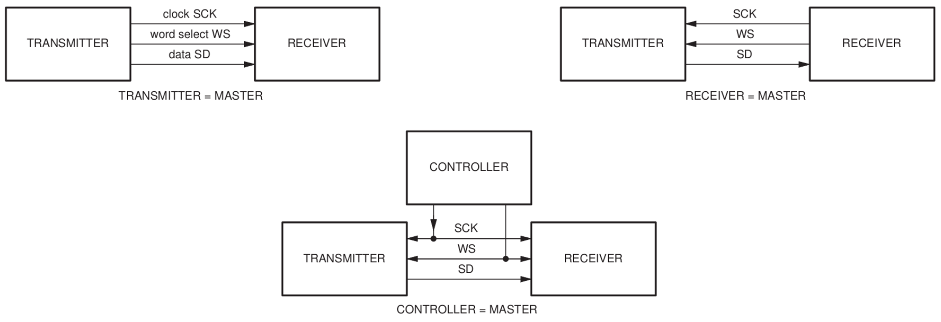


Figure 7: Supported I2S interface configurations. Diagram taken from I2S specification[2].

4.3 Configuration of I2C and I2S peripherals

Firstly, we need to configure I2C and I2S interface on STM32F4 side. Launch *STM32CubeMX* and start a new project.

Under *System Core* settings, select *RCC* and set both *High Speed Clock* and *Low Speed Clock* to *Crystal/Ceramic Resonator*.

We will now configure the I2C interface. Select *Connectivity* panel. Configure *I2C1* peripheral as shown in Fig. 8. We will use clock frequency of 100 kHz and 7-bit address length. This configuration will automatically initialize pins PB9 (SDA) and PB6 (SCL).

The next step is to configure I2S interface. On STM32F4-Discovery, the I2S lines are connected to pins of the SPI3 peripheral. The STM32F4 pins used for I2S interface are listed in Table 3.

ST32F4 pin	Function
PA4	Write Select (WS)
PC7	Master Clock (MCK)
PC10	Serial Clock (SCK)
PC12	Seral Data (SD)

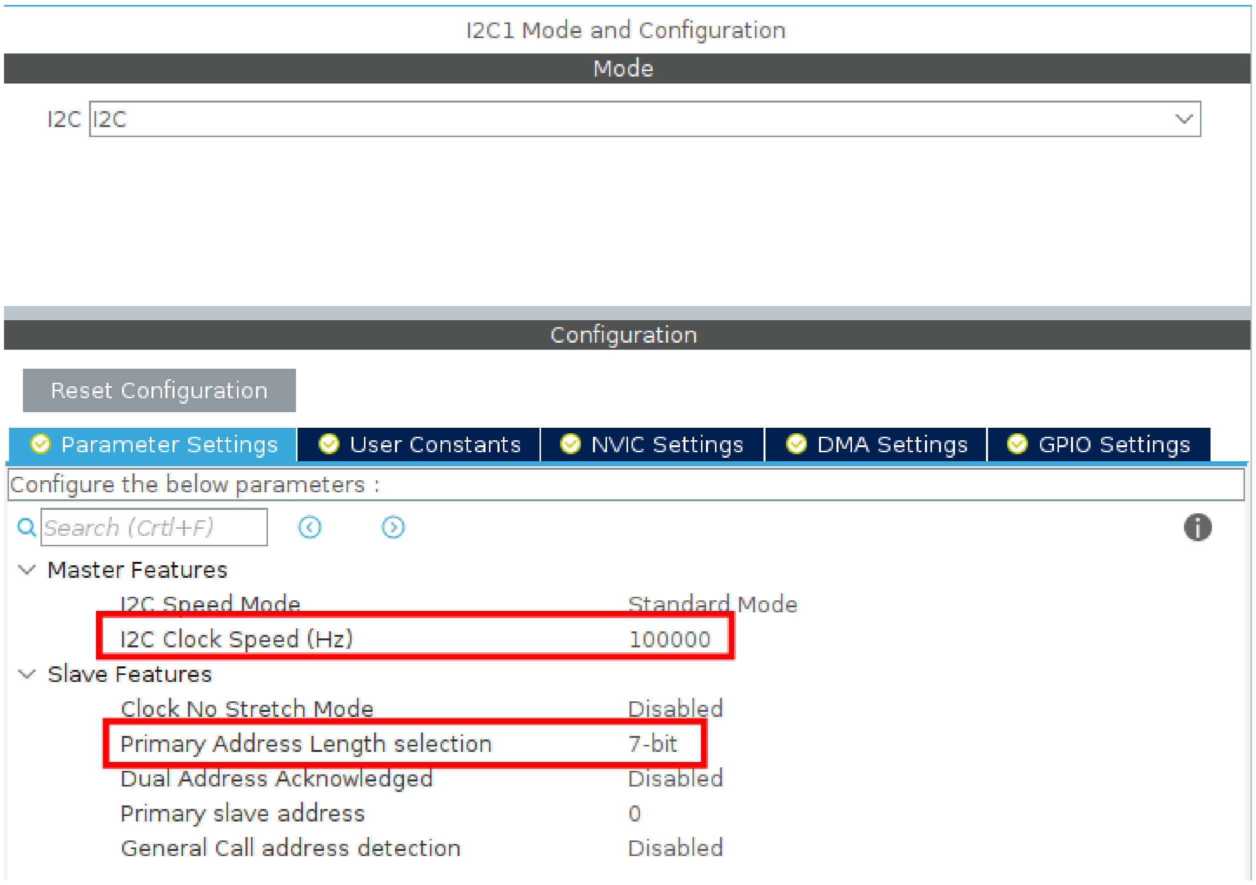
Table 3: I2S interface pins on STM32F4.

Initialize I2S interface by selecting *Multimedia - I2S3*. The MCU will be in *Master Transmit* mode. We will use I2S Philips communication standard and an audio frequency of 48 kHz. Configure the settings as shown in Fig. 9.

The reset signal for CS43L22 chip is connected to GPIO pin PD4 [1]. We will initialize this pin as GPIO output. Activate the *Pull-Down* option.

The final step is to configure system clock. Open *Clock Configuration* window and select the parameters as shown in Fig. 10.

Now click on *Generate Code* and open the project in *STM32CubeIDE*.


I2C1 Mode and Configuration

Mode

I2C: I2C

Configuration

Configure the below parameters :

Master Features	
I2C Speed Mode	Standard Mode
I2C Clock Speed (Hz)	100000
Slave Features	
Clock No Stretch Mode	Disabled
Primary Address Length selection	7-bit
Dual Address Acknowledged	Disabled
Primary slave address	0
General Call address detection	Disabled

Figure 8: Configuring I2C peripheral.

4.4 Audio interface configuration

We will implement the functions for audio interface configuration. Create a new source file `audio.c` and header file `audio.h`.

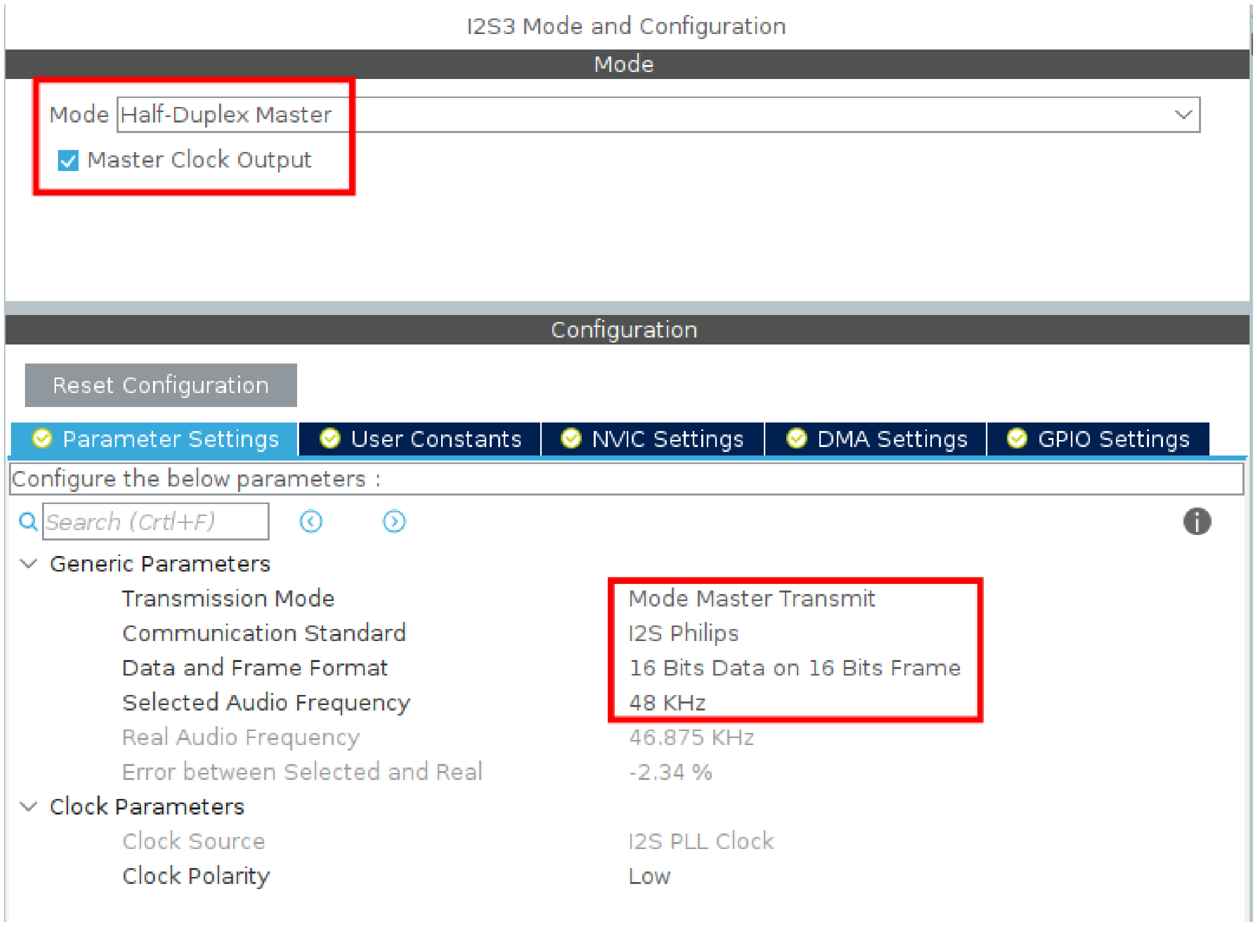
Firstly, we will add a function which initializes the pin connected to CS43L22 reset signal and drives the signal low. This will turn the CS43L22 chip on. Define the PD4 pin as audio reset pin in `audio.h`:

```
#define AUDIO_RESET_PIN     GPIO_PIN_4
```

Add the `init_AudioReset()` function to `audio.c`:

```
void init_AudioReset() {
  HAL_GPIO_WritePin(GPIOD, AUDIO_RESET_PIN, GPIO_PIN_SET);
}
```

The next step is to initialize the CS43L22 chip. Its operation is controlled through various registers which are configured using I2C interface. Section 5 of the CS43L22 datasheet [1] describes the procedures for reading and writing to control registers. Each access to register consists of sending the register address followed by the value that shall be written. Basics of I2C protocol can be found in chapter 23.3 of STM32F4 Reference Manual [6].


I2S3 Mode and Configuration

Mode

Mode Half-Duplex Master

✓ Master Clock Output

Configuration

Reset Configuration

Parameter Settings | User Constants | NVIC Settings | DMA Settings | GPIO Settings

Configure the below parameters :

Generic Parameters

Transmission Mode	Mode Master Transmit
Communication Standard	I2S Philips
Data and Frame Format	16 Bits Data on 16 Bits Frame
Selected Audio Frequency	48 KHz
Real Audio Frequency	46.875 KHz
Error between Selected and Real	-2.34 %

Clock Parameters

Clock Source	I2S PLL Clock
Clock Polarity	Low

Figure 9: Configuring I2S peripheral.

We will use HAL function `HAL_I2C_Master_Transmit()` for writing to control registers:

```
HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData,
    uint16_t Size, uint32_t Timeout);
```

The `hi2c` parameter is a pointer to a `I2C_HandleTypeDef` structure that contains the configuration information for the specific I2C. The address of the device is defined by `DevAddress` parameter. In this case, we will use the address of CS43L22 chip. The `pData` parameter is a pointer to buffer containing the data which will be sent. Since `HAL_I2C_Master_Transmit()` is a blocking function, we must define timeout duration. This is set via the `Timeout` parameter.

The I2C addresses for writing and reading are defined on page 36 of the datasheet. Address `0x94` is used for writing and `0x95` is used for reading. Define the CS43L22 read address in `audio.h`:

```
#define AUDIO_I2C_ADDRESS     0x94
```

We will now implement the function for initializing the CS43L22 audio chip. Add the function prototype to `audio.h`:

```
void init_AudioReset();
```

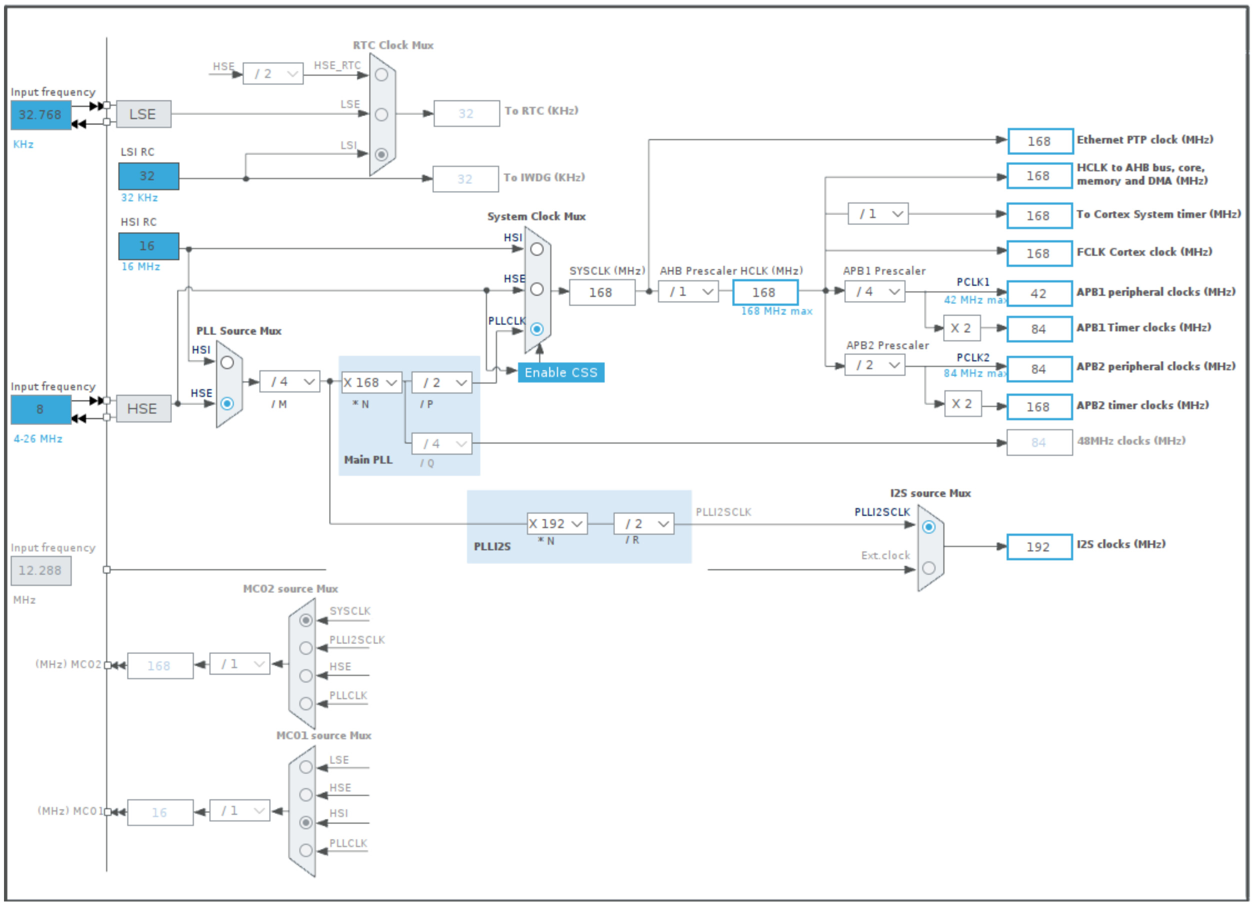


Figure 10: Clock configuration settings.

The recommended power-up sequence for CS43L22 audio chip can be found on page 32 of the datasheet. Firstly, we must set the Power Control 1 Register (`0x02`) to“off” state by writing `0x01`. According to the datasheet, it is recommended not to change the state of this register until all the other registers are configured. Next, we must load the required initialization sequence (page 33 in the datasheet). The final step is to configure the following control registers:

- Power Control 2 (`0x04`),
- Playback Control 1 (`0x0D`),
- Clocking Control (`0x05`),
- Interface Control (`0x06`),
- Analog and ZC Settings (`0x0A`),
- Limit Control 1 (`0x27`),
- Tone Control (`0x1F`),
- Headphone A Volume (`0x22`),
- Headphone B Volume (`0x23`),
- Passthrough A Volume (`0x14`),

– Passthrough B Volume (`0x15`),

– Master A Volume (`0x20`),

– Master B Volume (`0x21`).

After all the registers are configured, we must set the Power Control 1 register to value `0x9E`.

Add the CS43L22 initialization code to `audio.c`:

```c
void configAudio() {
  uint8_t bytes[2];
  init_AudioReset();

  /** Power sequence **/
  // Set Power Control Register to``on" state
  bytes[0] = 0x02;
  bytes[1] = 0x01;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  /** Initialization sequence **/
  bytes[0] = 0x00;
  bytes[1] = 0x99;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x47;
  bytes[1] = 0x80;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x32;
  bytes[1] = 0x80;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x32;
  bytes[1] = 0x0;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x00;
  bytes[1] = 0x00;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  /** Ctl registers configuration **/
  bytes[0] = 0x04;
  bytes[1] = 0xAF;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x0D;
  bytes[1] = 0x70;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x05;
  bytes[1] = 0x81;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x06;
  bytes[1] = 0x07;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x0A;
  bytes[1] = 0x00;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x27;
  bytes[1] = 0x00;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x1F;
  bytes[1] = 0x0F;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x22;
  bytes[1] = 0xC0;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x14;
  bytes[1] = 2;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x15;
  bytes[1] = 2;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x20;
  bytes[1] = 24;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  bytes[0] = 0x21;
  bytes[1] = 24;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

  /** Power up **/
  bytes[0] = 0x02;
  bytes[1] = 0x9E;
  HAL_I2C_Master_Transmit(&hi2c1, AUDIO_I2C_ADDRESS, bytes, 2, 100);

}
```

4.5 Generating audio signals

Both devices are now ready to send or receive audio data over I2S. Let's take a look at `HAL_I2S_Transmit()` function:

```
HAL_I2S_Transmit(I2S_HandleTypeDef *hi2s, uint16_t *pData, uint16_t Size, uint32_t Timeout)
```

Function parameters are similar as the parameters of `HAL_I2C_Transmit()` function. This function uses SPI3 peripheral configured in I2S mode. If repeated writes to the transmit buffer are required, the function checks whether the transmit buffer is empty (`TXE` flag) before sending new data. SPI peripheral will automatically switch the WS signal to indicate left or right audio channel.

To make sure that the interface is properly initialized, we will first generate noise and send it to CS43L22 audio chip. Add the following code to `main.c`:

```
#include <stdlib.h>
#include "main.h"
#include "i2c.h"
#include "i2s.h"
#include "gpio.h"
#include "audio.h"

int main(void) {

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_I2C1_Init();
  MX_I2S3_Init();

  configAudio();

  uint16_t signal;

  while (1) {
    signal = rand();
    HAL_I2S_Transmit(&hi2s3, &signal, 1, 10);
  }
}
```

We should now be able to hear noise on the headphones output of the STM32F4-Discovery.

4.6 Lab outcomes

Students must individually accomplish the following outcomes:

- working STM32CubeIDE project, following the guidelines for this lab exercise, with the same remarks as in the previous chapters regarding the project building and transfering to STM32F4DISCOVERY development board,
- program that demonstrates functional audio interface,
- extend the code example in a way that the program generates a sine wave with a frequency of 480 Hz and play the generated signal on headphone output.

References

[1] *CS43L22 Datasheet.* URL: `https://hr.mouser.com/datasheet/2/76/CS43L22%5C_F2-1142121.pdf`.

[2] *I2S specification.* URL: `http://www.nxp.com/acrobat_download/various/I2SBUS.pdf`.

[3] *STM32CubeIDE quick start guide.* URL: `https://www.st.com/resource/en/user_manual/dm00598966-stm32cubeide-quick-start-guide-stmicroelectronics.pdf`.

[4] *STM32CubeMX User manual.* URL: `https://www.st.com/resource/en/user_manual/dm00104712-stm32cubemx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf`.

[5] *STM32F3 and STM32F4 Series Cortex-M4 programming manual.* URL: `https://www.st.com/resource/en/programming_manual/dm00046982-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf`.

[6] *STM32F4 Reference manual.* URL: `https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[7] *STM32F407VG Datasheet.* URL: `https://www.st.com/resource/en/datasheet/stm32f407vg.pdf`.

[8] *UM1472 - Discovery kit for STM32F407.* URL: `https://www.st.com/resource/en/data_brief/stm32f4discovery.pdf`.

[9] *UM1725 - STM32F4 HAL Manual.* URL: `https://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf`.