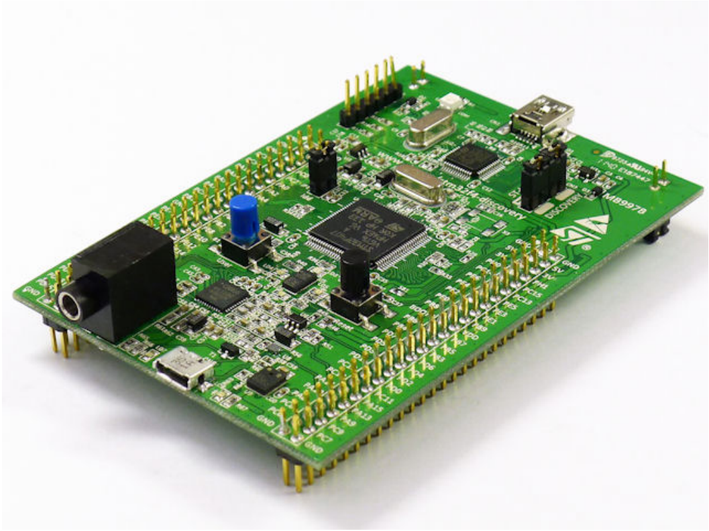
MINI PROJECT:

SERIAL COMMUNICATION



Work presented by:
Salsabil JABALLAH

ENIT Tutor :
M. JELASI KHALED

2ème Année GE2.
Année universitaire : 2023/2024

PLAN:

I INTRODUCTION

1 STM32F407VGTX MICROCONTROLLER

2 TOPIC OF THE PROJECT

II THE ALGORITHM AND LIBRARIES

III HARDWARE AND SOFTWARE TOOLS

1 CLOCK SETUP

2 UART SETUP

3 ADC SETUP

4 ASCII AND MAIN FUNCTION

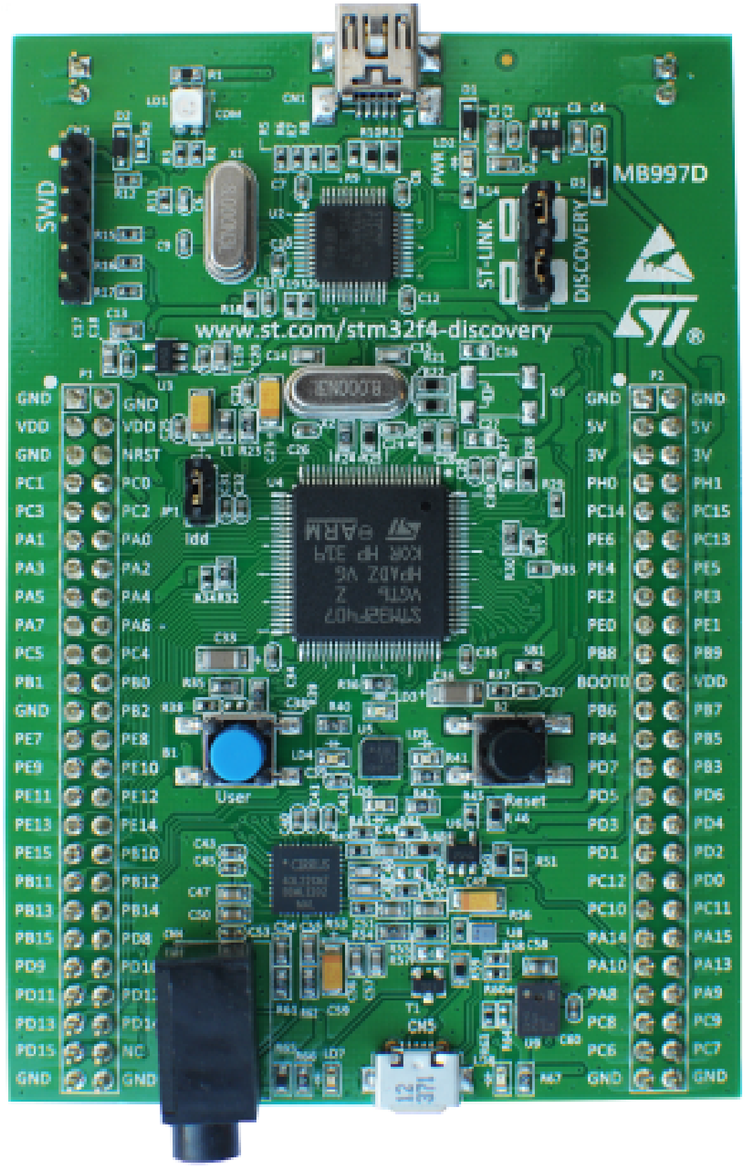
IV OUTPUT OF MY CODE

1 HARDWARE SETUP

2 KEIL / PUTTY ENVIROMENT

V CONCLUSION

I



STM32F407VGTX

Features

•STM32F407VGT6 microcontroller featuring 32-bit Arm®(a) Cortex®-M4 with FPU core, 1-Mbyte Flash memory, 192-Kbyte RAM in an LQFP100 package
• USB OTG FS
• ST MEMS 3-axis accelerometer
•ST-MEMS audio sensor omni-directional digital microphone
• Audio DAC with integrated class D speaker driver
• User and reset push-buttons

• Eight LEDs:

– LD1 (red/green) for USB communication
– LD2 (red) for 3.3 V power on
– Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
– Two USB OTG LEDs, LD7 (green) VBUS and LD8 (red) over-current

• Board connectors:

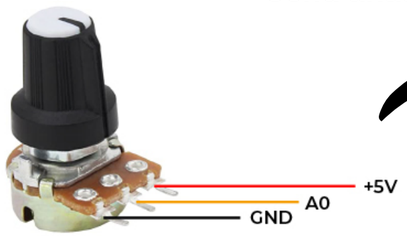
– USB with Micro-AB
–Stereo headphone output jack
– 2.54 mm pitch extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing

• Flexible power-supply options: ST-LINK, USB VBUS, or external sources
• External application power supply: 3 V and 5 V
• Comprehensive free software including a variety of examples, part of STM32CubeF4 MCU Package, or STSW-STM32068 for using legacy standard libraries
•On-board ST-LINK/V2-A debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port

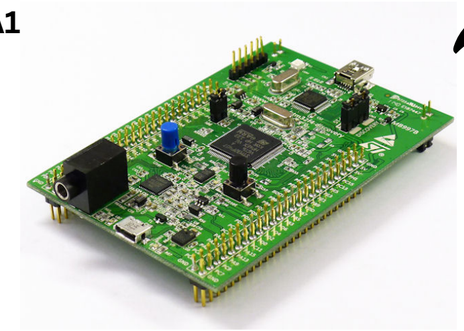
I

TOPIC OF THE PROJECT

"Exploitation of the serial link to send conversion data on channel 1 ADC1 in ASCII format. It is necessary to consider a scenario for verifying the proper functioning of the program."



Send analog signal to PA1



Connect the RX and TX

Send data to be visualized

11.1

CLOCK SETUP

HOW TO SETUP CLOCK USING REGISTERS

- TIMER CONFIGURATION
  - ENABLE THE TIMER CLOCK
  - SET PRESCALAR AND ARR
  - ENABLE THE TIMER AND WAIT FOR CONFIRMATION
- DELAY US / MS

TIMER CONFIGURATION

```
#include "Delay.h"
#include "RccConfig.h"

void TIM6Config (void)
{
  /************** STEPS TO FOLLOW *****************
  1. Enable Timer clock
  2. Set the prescalar and the ARR
  3. Enable the Timer, and wait for the update Flag to set
  ************************************************/
```

ENABLE THE TIMER CLOCK

CODE

```
// 1. Enable Timer clock
  RCC->APB1ENR |= (1<<4);  // Enable the timer6 clock
```

ENABLE THE TIMER CLOCK

6.3.13 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8 EN	UART7 EN	DAC EN	PWR EN	Reser-ved	CAN2 EN	CAN1 EN	Reser-ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser-ved
rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 4 **TIM6EN:** TIM6 clock enable

This bit is set and cleared by software.

0: TIM6 clock disabled

1: TIM6 clock enabled

CODE

```
// 1. Enable Timer clock
  RCC->APB1ENR |= (1<<4);  // Enable the timer6 clock
```

SET PRESCALAR AND ARR

18.4.11 TIMx prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value

The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).
PSC contains the value to be loaded in the active prescaler register at each update event (including when the counter is cleared through UG bit of TIMx_EGR register or through trigger controller when configured in “reset mode”).

CODE

```
/ 2. Set the prescalar and the ARR
 TIM6->PSC = 90-1;  // 90MHz/90 = 1 MHz ~~ 1 uS delay
 TIM6->ARR = 0xffff;  // MAX ARR value
```

ENABLE THE TIMER AND WAIT FOR CONFIRMATION

18.4.1 TIMx control register 1 (TIMx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **CEN**: Counter enable

0: Counter disabled

1: Counter enabled

Note: External clock, gated mode and encoder mode can work only if the CEN bit has been previously set by software. However trigger mode can set the CEN bit automatically by hardware.

CEN is cleared automatically in one-pulse mode, when an update event occurs.

18.4.5 TIMx status register (TIMx_SR)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bit 0 **UIF**: Update interrupt flag

" This bit is set by hardware on an update event. It is cleared by software.

0: No update occurred.

1: Update interrupt pending. This bit is set by hardware when the registers are updated:

" At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx_CR1 register.

" When CNT is reinitialized by software using the UG bit in TIMx_EGR register, if URS=0 and UDIS=0 in the TIMx_CR1 register.

When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx_CR1 register.

CODE

```
// 3. Enable the Timer, and wait for the update Flag to set
  TIM6->CR1 |= (1<<0); // Enable the Counter
  while (!(TIM6->SR & (1<<0)));  // UIF: Update interrupt flag..
}
```

DELAY US / MS

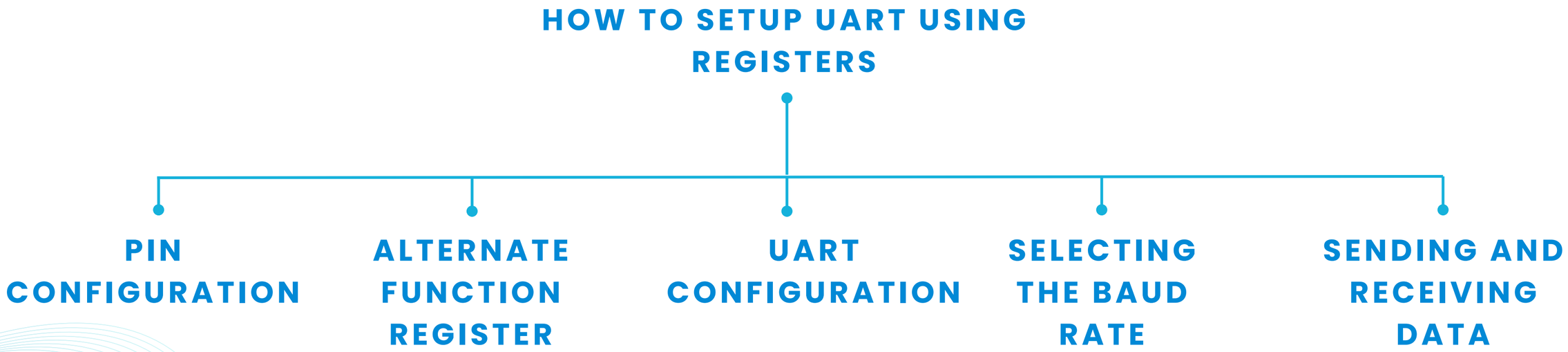
The function we'll be using in our main

CODE

```
void Delay_us (uint16_t us)
{
  /************** STEPS TO FOLLOW *****************
  1. RESET the Counter
  2. Wait for the Counter to reach the entered value. As each count will take 1 us,
     the total waiting time will be the required us delay
  ************************************************/
  TIM6->CNT = 0;
  while (TIM6->CNT < us);
}

void Delay_ms (uint16_t ms)
{
  for (uint16_t i=0; i<ms; i++)
  {
    Delay_us (1000); // delay of 1 ms
  }
}
```

11.2

UART SETUP



PIN CONFIGURATION

- Here we will enable the Clock for the GPIOA Port
- Then select the Pins PA2 and PA3 in the Alternate Function mode
- This is done by writing ‘1 0’ (2) in the 4th and 6th positions.

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

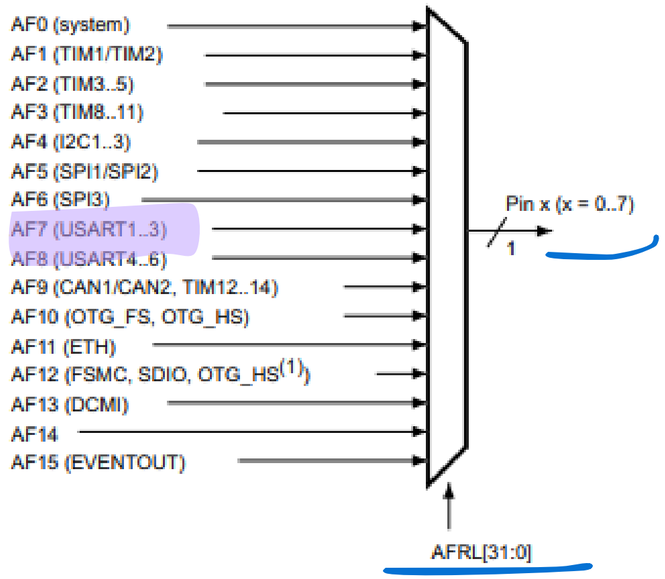
CODE

```
// 1. Enable the UART CLOCK and GPIO CLOCK
RCC->APB1ENR |= (1<<17);  // Enable UART2 CLOCK
RCC->AHB1ENR |= (1<<0); // Enable GPIOA CLOCK

// 2. Configure the UART PINs for ALternate Functions
GPIOA->MODER |= (2<<4);  // Bits (5:4)= 1:0 --> Alternate Function for Pin PA2
GPIOA->MODER |= (2<<6);  // Bits (7:6)= 1:0 --> Alternate Function for Pin PA3
```

ALTERNATE FUNCTION REGISTER

Figure 26. Selecting an alternate function on STM32F405xx/07xx and STM32F415xx/17xx

For pins 0 to 7, the GPIOx_AFRL[31:0] register selects the dedicated alternate function



For pins 8 to 15, the GPIOx_AFRH[31:0] register selects the dedicated alternate function

8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..I/J/K)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRLy:** Alternate function selection for port x bit y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

CODE

```
GPIOA->OSPEEDR |= (3<<4) | (3<<6);  // Bits (5:4)= 1:1 and Bits (7:6)= 1:1 --> High Speed for PIN PA2 and PA3

GPIOA->AFR[0] |= (7<<8);  // Bytes (11:10:9:8) = 0:1:1:1  --> AF7 Alternate function for USART2 at Pin PA2
GPIOA->AFR[0] |= (7<<12); // Bytes (15:14:13:12) = 0:1:1:1  --> AF7 Alternate function for USART2 at Pin PA3
```

UART CONFIGURATION

30.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 13 **UE**: USART enable

When this bit is cleared, the USART prescalers and outputs are stopped and the end of the current byte transfer in order to reduce power consumption. This bit is set and cleared by software.

0: USART prescaler and outputs disabled

1: USART enabled

Bit 12 **M**: Word length

This bit determines the word length. It is set or cleared by software.

0: 1 Start bit, 8 Data bits, n Stop bit

1: 1 Start bit, 9 Data bits, n Stop bit

Note: The M bit must not be modified during a data transfer (both transmission and reception)

Bit 3 **TE**: Transmitter enable

This bit enables the transmitter. It is set and cleared by software.

0: Transmitter is disabled

1: Transmitter is enabled

Note: During transmission, a "0" pulse on the TE bit ("0" followed by "1") sends a preamble (idle line) after the current word, except in smartcard mode.

When TE is set, there is a 1 bit-time delay before the transmission starts.

Bit 2 **RE**: Receiver enable

This bit enables the receiver. It is set and cleared by software.

0: Receiver is disabled

1: Receiver is enabled and begins searching for a start bit

CODE

```
// 3. Enable the USART by writing the UE bit in USART_CR1 register to 1.
USART2->CR1 = 0x00;  // clear all
USART2->CR1 |= (1<<13);  // UE = 1... Enable USART

// 4. Program the M bit in USART_CR1 to define the word length.
USART2->CR1 &= ~(1<<12);  // M =0; 8 bit word length

// 5. Select the desired baud rate using the USART_BRR register.
USART2->BRR = (7<<0) | (24<<4);   // Baud rate of 115200, PCLK1 at 45MHz

// 6. Enable the Transmitter/Receiver by Setting the TE and RE bits in USART_CR1 Register
USART2->CR1 |= (1<<2); // RE=1.. Enable the Receiver
USART2->CR1 |= (1<<3);  // TE=1.. Enable Transmitter
```

SELECTING THE BAUD RATE

30.6.3 Baud rate register (USART_BRR)

Note: The baud counters stop counting if the TE or RE bits are disabled respectively.

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value

Bits 15:4 **DIV_Mantissa[11:0]**: mantissa of USARTDIV
These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV_Fraction[3:0]**: fraction of USARTDIV
These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV_Fraction3 bit is not considered and must be kept cleared.

$$USARTDIV = \frac{F_{CK}}{16.baudrate}$$

- Here the UART2 is connected to the APB1 Clock, and therefore the Fck = 45 MHz (APB1 Peripheral Frequency)
- Now If we want the Baud Rate of 115200, the MENSTISSA will come equal to 24, and the FRACTION = 7
- We will program these values in the 0th Position (for FRACTION) and in the 4th position (for MENTISSA) in the BRR Register.

CODE

```
// 5. Select the desired baud rate using the USART_BRR register.
USART2->BRR = (7<<0) | (24<<4);    // Baud rate of 115200, PCLK1 at 45MHz
```

SENDING AND RECEIVING DATA

- TXE bit will be set once the content of the Data Register has been transferred to the Shift Register. Now the Shift Register will start sending the data to the Tx Line.
- In the means time we can put another data byte into the Data Register, and later that byte will also be transferred to the Shift register.
- Once all the data bytes have been transferred (after the stop bit is sent), the TC flag will be set.
- Similarly during the receive, we will wait for the RXNE bit to set.
- This bit is set if there is data in he DATA Register, and once it is confirmed, we can read the data.

30.6.1 Status register (USART_SR)

Address offset: 0x00

Reset value: 0x0000 00C0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

Bit 7 **TXE**: Transmit data register empty

This bit is set by hardware when the content of the TDR register has been transferred into the shift register. An interrupt is generated if the TXEIE bit =1 in the USART_CR1 register. It is cleared by a write to the USART_DR register.

0: Data is not transferred to the shift register

1: Data is transferred to the shift register)

Note: This bit is used during single buffer transmission.

Bit 6 **TC**: Transmission complete

This bit is set by hardware if the transmission of a frame containing data is complete and if TXE is set. An interrupt is generated if TCIE=1 in the USART_CR1 register. It is cleared by a software sequence (a read from the USART_SR register followed by a write to the USART_DR register). The TC bit can also be cleared by writing a '0' to it. This clearing sequence is recommended only for multibuffer communication.

0: Transmission is not complete

1: Transmission is complete

Bit 5 **RXNE**: Read data register not empty

This bit is set by hardware when the content of the RDR shift register has been transferred to the USART_DR register. An interrupt is generated if RXNEIE=1 in the USART_CR1 register. It is cleared by a read to the USART_DR register. The RXNE flag can also be cleared by writing a zero to it. This clearing sequence is recommended only for multibuffer communication.

0: Data is not received

1: Received data is ready to be read.

SENDING AND RECEIVING DATA

CODE

```
void UART2_SendChar (uint8_t c)
{
    /************ STEPS FOLLOWED *************

  1. Write the data to send in the USART_DR register (this clears the TXE bit). Repeat this
     for each data to be transmitted in case of single buffer.
  2. After writing the last data into the USART_DR register, wait until TC=1. This indicates
     that the transmission of the last frame is complete. This is required for instance when
     the USART is disabled or enters the Halt mode to avoid corrupting the last transmission.

  ***************************************/

  USART2->DR = c; // load the data into DR register
  while (!(USART2->SR & (1<<6)));  // Wait for TC to SET.. This indicates that the data has been transmitted
}
```

```
uint8_t UART2_GetChar (void)
{
      /************ STEPS FOLLOWED *************

  1. Wait for the RXNE bit to set. It indicates that the data has been received and can be read.
  2. Read the data from USART_DR  Register. This also clears the RXNE bit

  ***************************************/

  uint8_t temp;

  while (!(USART2->SR & (1<<5)));  // wait for RXNE bit to set
  temp = USART2->DR;  // Read the data. This clears the RXNE also
  return temp;
}
```

SENDING AND RECEIVING DATA

The function we'll be using in our main

CODE

```
void UART2_SendString(USART_TypeDef *USARTx, uint8_t *string, uint32_t length, uint32_t timeout) {
    // Send a string via UART2
    while (*string && length--) {
        // Send character
        UART2_SendChar(*string++);
    }
}
```

UART.h

```
#ifndef UART_H
#define UART_H

#include "stm32f4xx.h"

void Uart2Config(void);
void UART2_SendChar(uint8_t c);
void UART2_SendString(USART_TypeDef *USARTx, uint8_t *string, uint32_t length, uint32_t timeout);
uint8_t UART2_GetChar(void);

#endif /* UART_H */
```

11.3

ADC SETUP

HOW TO SETUP ADC USING REGISTERS

- ADC CONFIGURATION
  - ENABLE ADC AND GPIO CLOCK
  - SET THE PRESCALAR IN (CCR)
  - SET THE SCAN MODE AND RESOLUTION IN (CR1)
  - SET THE CONTINUOUS CONVERSION, EOC, AND DATA ALIGNMENT IN (CR2)
  - SET THE SAMPLING TIME FOR THE CHANNELS
  - SET THE REGULAR CHANNEL SEQUENCE LENGTH
  - SET THE RESPECTIVE GPIO PINS IN THE ANALOG MODE
- ENABLE THE ADC
- START THE ADC

ADC CONFIGURATION

```
void ADC_Init (void)
{
  /************** STEPS TO FOLLOW *****************
  1. Enable ADC and GPIO clock
  2. Set the prescalar in the Common Control Register (CCR)
  3. Set the Scan Mode and Resolution in the Control Register 1 (CR1)
  4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
  5. Set the Sampling Time for the channels in ADC_SMPRx
  6. Set the Regular channel sequence length in ADC_SQR1
  7. Set the Respective GPIO PINs in the Analog Mode
  ************************************************/
```

ENABLE ADC AND GPIO CLOCK

CODE

```
//1. Enable ADC and GPIO clock
  RCC->APB2ENR |= (1<<8);  // enable ADC1 clock
  RCC->AHB1ENR |= (1<<0);  // enable GPIOA clock
```

SET THE PRESCALAR IN (CCR)

- In this case, the ADC1 is connected to the APB2 Peripheral clock, which is running at it's maximum speed of 90 MHz
- We will use the prescalar to bring the ADC1 clock down.
- The Prescalar selection can be done in the CCR Register. The important point to note here is that we can only choose amongst the predefined Prescalar values. This is shown in the figure below

13.13.16 ADC common control register (ADC_CCR)

Address offset: 0x04 (this offset address is relative to ADC1 base address + 0x300)

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
								rw	rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]		DDS	Res.	DELAY[3:0]				Reserved			MULTI[4:0]				
rw	rw	rw		rw	rw	rw	rw				rw	rw	rw	rw	rw

Bits 17:16 **ADCPRE:** ADC prescaler

Set and cleared by software to select the frequency of the clock to the ADC. The clock is common for all the ADCs.

Note: 00: PCLK2 divided by 2
01: PCLK2 divided by 4
10: PCLK2 divided by 6
11: PCLK2 divided by 8

CODE

```
//2. Set the prescalar in the Common Control Register (CCR)
  ADC->CCR |= 1<<16;          // PCLK2 divide by 4
```

SET THE SCAN MODE AND RESOLUTION IN (CR1)

- Now we will modify the Control Register 1 (CR1). Here we will set up the scan mode and the Resolution for the ADC1.
- Scan mode must be set, if you are using more than 1 channel for the ADC.
- Resolution defines the Resolution of the ADC. In STM32F4, this can vary between 6-Bit, 8-Bit, 10-Bit or 12-Bit.
- Here choosing the Resolution of 12 bit means, the ADC values will vary between 0 to 4095.

13.13.2 ADC control register 1 (ADC_CR1)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES		AWDEN	JAWDEN	Reserved					
					rw	rw	rw	rw	rw						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 25:24 **RES[1:0]:** Resolution

These bits are written by software to select the resolution of the conversion.

00: 12-bit (15 ADCCLK cycles)

01: 10-bit (13 ADCCLK cycles)

10: 8-bit (11 ADCCLK cycles)

11: 6-bit (9 ADCCLK cycles)

Bit 8 **SCAN:** Scan mode

This bit is set and cleared by software to enable/disable the Scan mode. In Scan mode, the inputs selected through the ADC_SQRx or ADC_JSQRx registers are converted.

0: Scan mode disabled

1: Scan mode enabled

Note: An EOC interrupt is generated if the EOCIE bit is set:

- *At the end of each regular group sequence if the EOCS bit is cleared to 0*
- *At the end of each regular channel conversion if the EOCS bit is set to 1*

Note: A JEOC interrupt is generated only on the end of conversion of the last channel if the JEOCIE bit is set.

CODE

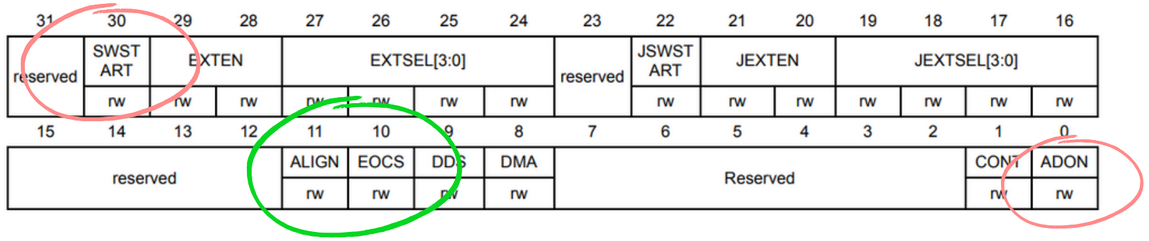
```
//3. Set the Scan Mode and Resolution in the Control Register 1 (CR1)
  ADC1->CR1 = (1<<8);     // SCAN mode enabled
  ADC1->CR1 &= ~(1<<24);    // 12 bit RES
```

SET THE CONTINUOUS CONVERSION, EOC, AND DATA ALIGNMENT IN (CR2)

13.13.3 ADC control register 2 (ADC_CR2)

Address offset: 0x08

Reset value: 0x0000 0000



- Continuous Conversion specifies whether we want to convert the ADC values continuously, or should it stop after one conversion only.
- EOC is End Of Conversion specifies whether the EOC Flag should set after each conversion, or at the end of all the conversions.
- Data Alignment specifies whether the 12 bit data should be Right Aligned or Left Aligned in a 16 bit Register.

Bit 11 **ALIGN:** Data alignment

This bit is set and cleared by software. Refer to *Figure 48* and *Figure 49*.

0: Right alignment

1: Left alignment

Bit 10 **EOCS:** End of conversion selection

This bit is set and cleared by software.

0: The EOC bit is set at the end of each sequence of regular conversions. Overrun detection is enabled only if DMA=1.

1: The EOC bit is set at the end of each regular conversion. Overrun detection is enabled.

Bits 7:2 Reserved, must be kept at reset value.

Bit 1 **CONT:** Continuous conversion

This bit is set and cleared by software. If it is set, conversion takes place continuously until it is cleared.

0: Single conversion mode

1: Continuous conversion mode

CODE

```
//4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
  ADC1->CR2 |= (1<<1);     // enable continuous conversion mode
  ADC1->CR2 |= (1<<10);    // EOC after each conversion
  ADC1->CR2 &= ~(1<<11);   // Data Alignment RIGHT
```

SET THE SAMPLING TIME FOR THE CHANNELS

- There are 2 sample Registers SMPR1 and SMPR2. Since I am using channel 1 and channel 4(for testing) , I have to use SMPR2.
- For this demonstration, we don't need any specific timing for the ADC, and that's why we can choose any sampling cycle from above.
- I am going to choose the 3 cycles from here.

Note: channel 4 is(for testing)

13.13.5 ADC sample time register 2 (ADC_SMPR2)

Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30 Reserved, must be kept at reset value.

Bits 29:0 **SMPx[2:0]:** Channel x sampling time selection

These bits are written by software to select the sampling time individually for each channel. During sample cycles, the channel selection bits must remain unchanged.

Note: 000: 3 cycles
001: 15 cycles
010: 28 cycles
011: 56 cycles
100: 84 cycles
101: 112 cycles
110: 144 cycles
111: 480 cycles

CODE

```
//5. Set the Sampling Time for the channels
  ADC1->SMPR2 &= ~((1<<3) | (1<<12));  // Sampling time of 3 cycles for channel 1 and channel 4
```

SET THE REGULAR CHANNEL SEQUENCE LENGTH

- Here L can be used to set the number of channels. As you can see in the picture above, we can set the number of channels between 1 to 16. A single ADC is capable of converting 16 channels at once.
- Since we are converting 2 channels, we would write 1 in the 20th position.

13.13.9 ADC regular sequence register 1 (ADC_SQR1)

Address offset: 0x2C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

Bits 31:24 Reserved, must be kept at reset value.

Bits 23:20 **L[3:0]:** Regular channel sequence length

These bits are written by software to define the total number of conversions in the regular channel conversion sequence.

0000: 1 conversion

0001: 2 conversions

...

1111: 16 conversions

CODE

```
//6. Set the Regular channel sequence length in ADC_SQR1
  ADC1->SQR1 |= (1<<20);    // SQR1_L =1 for 2 conversions
```

SET THE RESPECTIVE GPIO PINS IN THE ANALOG MODE

- GPIO Mode Register can be used to modify the Pin Modes.
- Here we need to set the Analog mode to the Pins PA1 and PA4, and that's why we will modify the MODER1 (Bits 2 and 3) and MODER4 (Bits 8 and 9).

Note: channel 4 is(for testing)

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

CODE

```
//7. Set the Respective GPIO PINs in the Analog Mode
  GPIOA->MODER |= (3<<2);  // analog mode for PA 1 (chennel 1)
  GPIOA->MODER |= (3<<8);  // analog mode for PA 4 (channel 4)
}
```

ENABLE ADC

- Once the ADC configuration is complete, we will enable the ADC. This is important to enable it in the end because we can not configure certain things while the ADC is enabled.
- We will again modify the Control Register 2 for this

Bit 0 **ADON:** A/D Converter ON / OFF
This bit is set and cleared by software.
Note: 0: Disable ADC conversion and go to power down mode
1: Enable ADC

CODE

```
void ADC_Enable (void)
{
  /************** STEPS TO FOLLOW *****************
  1. Enable the ADC by setting ADON bit in CR2
  2. Wait for ADC to stabilize (approx 10us)
  ************************************************/
  ADC1->CR2 |= 1<<0;    // ADON =1 enable ADC1

  uint32_t delay = 10000;
  while (delay--);
}
```

START ADC

Bit 31 Reserved, must be kept at reset value.

Bit 30 **SWSTART:** Start conversion of regular channels
This bit is set by software to start conversion and cleared by hardware as soon as the conversion starts.
0: Reset state
1: Starts conversion of regular channels
Note: This bit can be set only when ADON = 1 otherwise no conversion is launched.

CODE

```
void ADC_Start (int channel)
{
  /************** STEPS TO FOLLOW *****************
  1. Set the channel Sequence in the SQR Register
  2. Clear the Status register
  3. Start the Conversion by Setting the SWSTART bit in CR2
  ************************************************/


/** Since we will be polling for each channel, here we will keep or
    ADC1->SQR3 |= (channel<<0); will just keep the respective chann

  ADC1->SQR3 = 0;
  ADC1->SQR3 |= (channel<<0);    // conversion in regular sequence

  ADC1->SR = 0;          // clear the status register

  ADC1->CR2 |= (1<<30);  // start the conversion
}
```

The function we'll be using in our main

CODE

```
void ADC_WaitForConv (void)
{
  /*****************************************************
  EOC Flag will be set, once the conversion is finished
  *****************************************************/
  while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to set
}

uint16_t ADC_GetVal (void)
{
  return ADC1->DR;  // Read the Data Register
}

void ADC_Disable (void)
{
  /*************** STEPS TO FOLLOW *****************
  1. Disable the ADC by Clearing ADON bit in CR2
  ************************************************/
  ADC1->CR2 &= ~(1<<0);  // Disable ADC
}
```

ADC.h

```
#ifndef ADC_H
#define ADC_H
#include <stdint.h>
void ADC_Init(void);
void ADC_Enable(void);
void ADC_Start(int channel);
void ADC_WaitForConv(void);
uint16_t ADC_GetVal(void);
void ADC_Disable(void);

#endif /* ADC_H */
```

11.4

ASCII CONVERSION

```c
#include "ASCII.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* shift_digits(int input_integer) {
    // Convert the integer to a string
    char input_str[20];
    sprintf(input_str, "%d", input_integer);

    // Allocate memory for the result string
    char* result_str = malloc(3 * strlen(input_str));

    // Shift each digit by +48 and join them with spaces
    int index = 0;
    for (int i = 0; i < strlen(input_str); i++) {
        int shifted_digit = input_str[i];
        index += sprintf(result_str + index, "%d ", shifted_digit);
    }

    // Remove the trailing space and add the null terminator
    result_str[index - 1] = '\0';

    return result_str;
}
```

11.4

MAIN.C

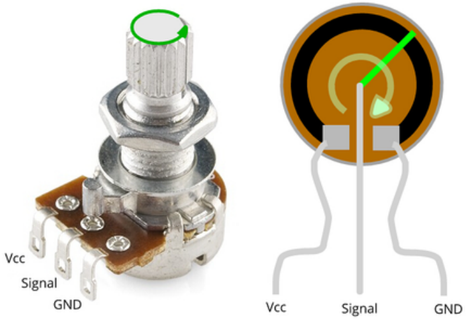
```c
#include "RccConfig.h"
#include "Delay.h"
#include "ADC.h"
#include "UART.h"
#include "ASCII.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    /* Configure the system clock */
    SysClockConfig();

    /* Initialize all configured peripherals */
    // UART configuration
    TIM6Config ();
    Uart2Config();

    // ADC initialization
    ADC_Init();
    ADC_Enable();
```

```c
while (1) {
    // ADC code
    ADC_Start(1); // Start ADC conversion for channel 1
    ADC_WaitForConv();
    uint16_t raw = ADC_GetVal();

    // Calculate voltage
    float vin = raw * (3.3 / 4096);

    // Convert raw value to ASCII string
    char* msg2 = shift_digits(raw);

    // Create message string
    char msg[50];
    sprintf(msg, "ASCII Code: %s, Voltage: %.2f V\r\n", msg2, vin);

    // Send message via UART
    UART2_SendString(USART2, (uint8_t*)msg, strlen(msg), 300);

    // Delay for 1 second
    Delay_ms(1000);

    // Free memory allocated for ASCII string
    free(msg2);
```

III.1

HARDWARE SETUP

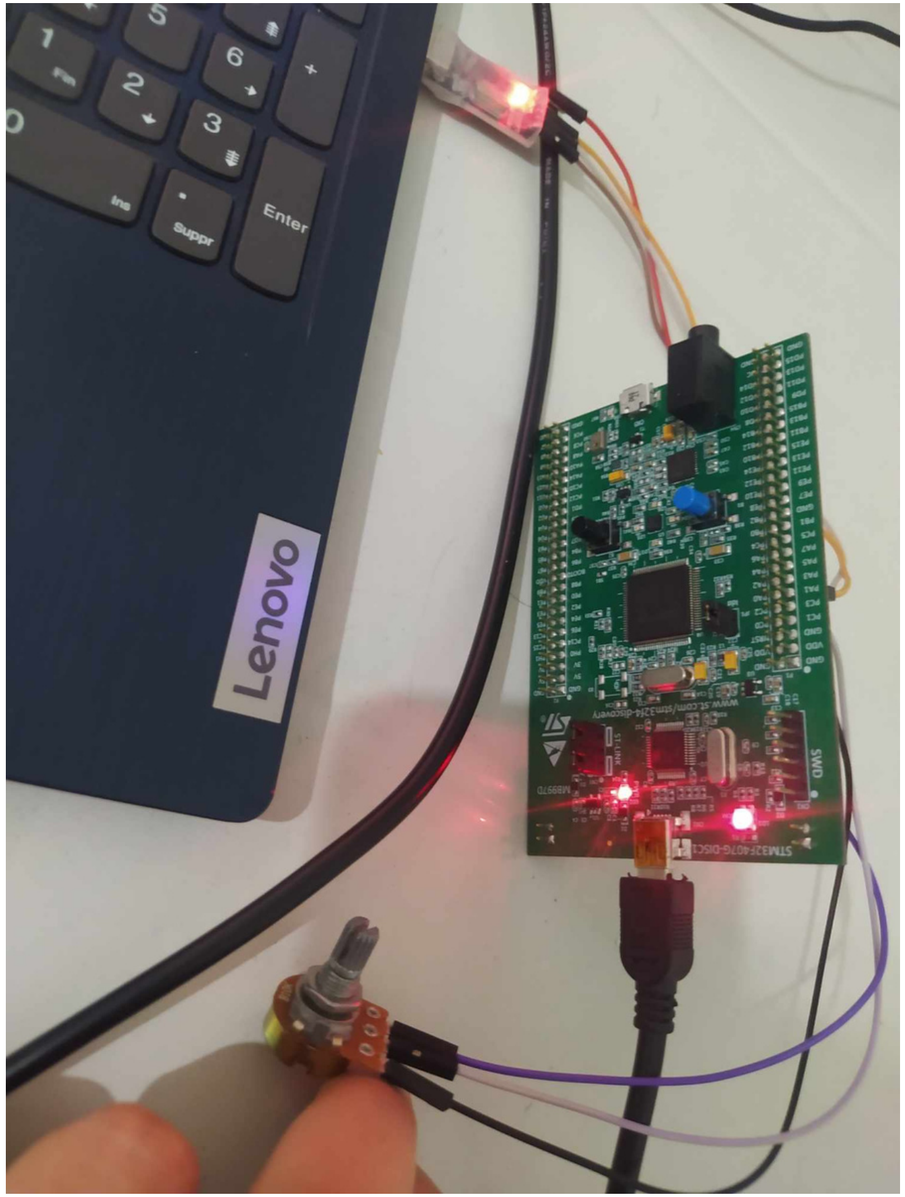
Potentiometer



- GND---GND
- VCC---3V
- Signal----PA1(ADC1)

USB to TTL

- GND---GND
- 5V---5V
- RXD---PA2(TX-USART2)
- TXD---PA3(RX-USART2)

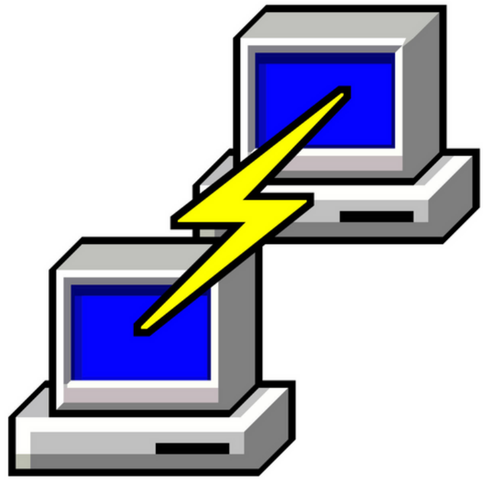


III.2

KEIL

Keil Vision 5 is a development environment that provides users with easy access to ARM® Cortex®-M microcontrollers. It includes a C/C++ compiler and a debugger. Keil enables real-time simulation of tasks performed by the microcontroller, allowing users to observe signals at the output.

PUTYY



PuTTY is a free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator.

IV

OUTPUT FROM_SCRATCH_VERSION

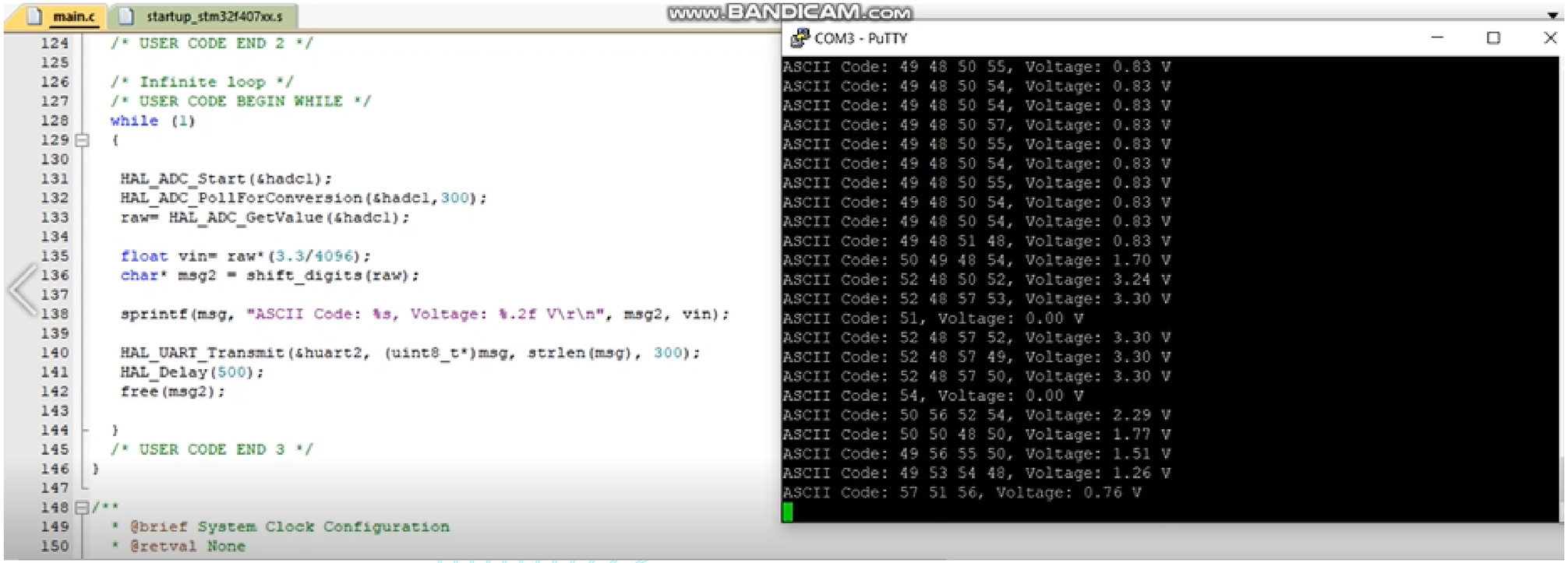
```c
#include "ADC.h"
#include "UART.h"
#include "ASCII.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    /* Configure the system clock */
    SysClockConfig();

    /* Initialize all configured peripherals */
    // UART configuration
    TIM6Config ();
    Uart2Config();

    // ADC initialization
    ADC_Init();
    ADC_Enable();

    while (1) {
        // ADC code
        ADC_Start(1); // Start ADC conversion for
        ADC_WaitForConv();
        uint16_t raw = ADC_GetVal();

        // Calculate voltage
        float vin = raw * (3.3 / 4096);
```

COM3 - PuTTY

```
ASCII Code: 51 57 53 53, Voltage: 3.19 V
ASCII Code: 51 57 53 54, Voltage: 3.19 V
ASCII Code: 51 57 53 52, Voltage: 3.19 V
ASCII Code: 51 57 53 52, Voltage: 3.19 V
ASCII Code: 51 57 53 50, Voltage: 3.18 V
ASCII Code: 51 57 53 53, Voltage: 3.19 V
ASCII Code: 51 57 53 52, Voltage: 3.19 V
ASCII Code: 51 57 53 53, Voltage: 3.19 V
ASCII Code: 51 57 53 51, Voltage: 3.18 V
ASCII Code: 51 57 53 54, Voltage: 3.19 V
ASCII Code: 51 57 53 54, Voltage: 3.19 V
ASCII Code: 51 57 53 52, Voltage: 3.19 V
ASCII Code: 51 57 53 53, Voltage: 3.19 V
ASCII Code: 52 48 57 53, Voltage: 3.30 V
ASCII Code: 52 48 57 53, Voltage: 3.30 V
ASCII Code: 52 48 57 53, Voltage: 3.30 V
ASCII Code: 52 48 57 53, Voltage: 3.30 V
ASCII Code: 51 48 51 57, Voltage: 2.45 V
ASCII Code: 50 49 55 56, Voltage: 1.75 V
ASCII Code: 50 48 50 53, Voltage: 1.63 V
ASCII Code: 50 48 50 54, Voltage: 1.63 V
ASCII Code: 49 57 54 49, Voltage: 1.58 V
ASCII Code: 57 53 53, Voltage: 0.77 V
```

IV

OUTPUT
HAL_VERSION



V

CONCLUSION

This mini-project aims to demonstrate a practical application of STM32F4 microcontrollers in the context of serial transmission and analog-to-digital conversion. Throughout the project, we learned how to navigate the manual and datasheets effectively, enabling us to code for all the peripherals utilized in the system.

THANK YOU FOR YOUR ATTENTION