|  |
| --- |
| QT avec la STM32F4 Discovery |

|  |  |  |
| --- | --- | --- |
|  |  |  |

Table des matières

[I. Objectifs 3](#_Toc8233042)

[II. STM32F4 Discovery 3](#_Toc8233043)

[a. Présentation 3](#_Toc8233044)

[III. La HAL 4](#_Toc8233045)

[a. Qu’est-ce que la HAL 4](#_Toc8233046)

[b. Utilisation de la HAL 6](#_Toc8233047)

[IV. Partie STM32F4 Discovery 9](#_Toc8233048)

[1. Partie programmation 9](#_Toc8233049)

[2. Partie Test 14](#_Toc8233050)

[V. Partie QT Creator : affichage du signal 15](#_Toc8233051)

[VI. Code sur QT Creator 15](#_Toc8233052)

[3. Main.c 15](#_Toc8233053)

[4. Mainwidget.cpp 16](#_Toc8233054)

[5. mycanvas.cpp 17](#_Toc8233055)

[6. serialcapture.cpp 20](#_Toc8233056)

[7. mainwidget.h 20](#_Toc8233057)

[8. mycanvas.h 21](#_Toc8233058)

[9. serialcapture.h 22](#_Toc8233059)

[Figure 1 Schéma du projet à réalisé 3](#_Toc8233024)

[Figure 2 : STM32F4 Discovery 4](#_Toc8233025)

[Figure 3 : Configuration pour la HAL 5](#_Toc8233026)

[Figure 4 : Configuration à l'aide CubeMX 6](#_Toc8233027)

[Figure 5 Selection Horloge 7](#_Toc8233028)

[Figure 6 UART selection 7](file:///C:\Users\Ricod\Downloads\QT-STM.docx#_Toc8233029)

[Figure 7 Configuration de l'UART 7](#_Toc8233030)

[Figure 8 réglage UART 7](file:///C:\Users\Ricod\Downloads\QT-STM.docx#_Toc8233031)

[Figure 9 ADC configuration 8](#_Toc8233032)

[Figure 10relevés ADC n°1 14](#_Toc8233033)

[Figure 11 relevés ADC n°2 14](#_Toc8233034)

[Figure 12 relevés UART 14](#_Toc8233035)

[Figure 13 : Affichage du signal 15](#_Toc8233036)

# Objectifs

L’objectif de ce TP est de réaliser un signal sinusoïdal à l’aide d’une sortie ADC venant de la STM32F4 Discovery. Ce signal sera retranscrit sur un graphique à l’aide de QT Creator. Ce TP nous permettra de lier 2 langages ainsi deux environnements différents. D’un côté le C embarqué avec la STM32F4 Discovery et de l’autre le langage orienté objet : QT

Ci-dessous un schéma de ce que nous avons à faire.



Figure Schéma du projet à réalisé

# STM32F4 Discovery

## Présentation

Pour l’objectif de ce TP, nous avons utilisé une STM32F4 Discovery qui nous a permis de réaliser l’émission d’un signal sinusoïdal à l’aide de l’ADC.

Cette carte comporte :

* un Microcontrôleur stm32f407vgt6 32-bit ARM Cortex-M4F, 1 MB de Flash, 192 KB de RAM
* un connecteur ST-LINK/V2 ou la possibilité d’utiliser le connecteur USB d’alimentation pour la programmation et le debug
* un connecteur USB pour l’alimentation 5V
* des sorties 3V et 5V pour les applications externes
* un accéléromètre LIS302DL ou LIS3DSH ST MEMS 3-axis
* un micro MP45DT02 pour enregistrer des sons
* un  audio DAC CS43L22, avec un pilote de haut-parleur class D
* 4 LEDs, LD3 (orange), LD4 (vert), LD5 (rouge) and LD6 (bleu)
* 2 boutons poussoir (utilisateur et reset)
* un connecteur USB OTG FS (micro-AB) pour s’interfacer avec des systèmes externes (clé USB, PC, …)

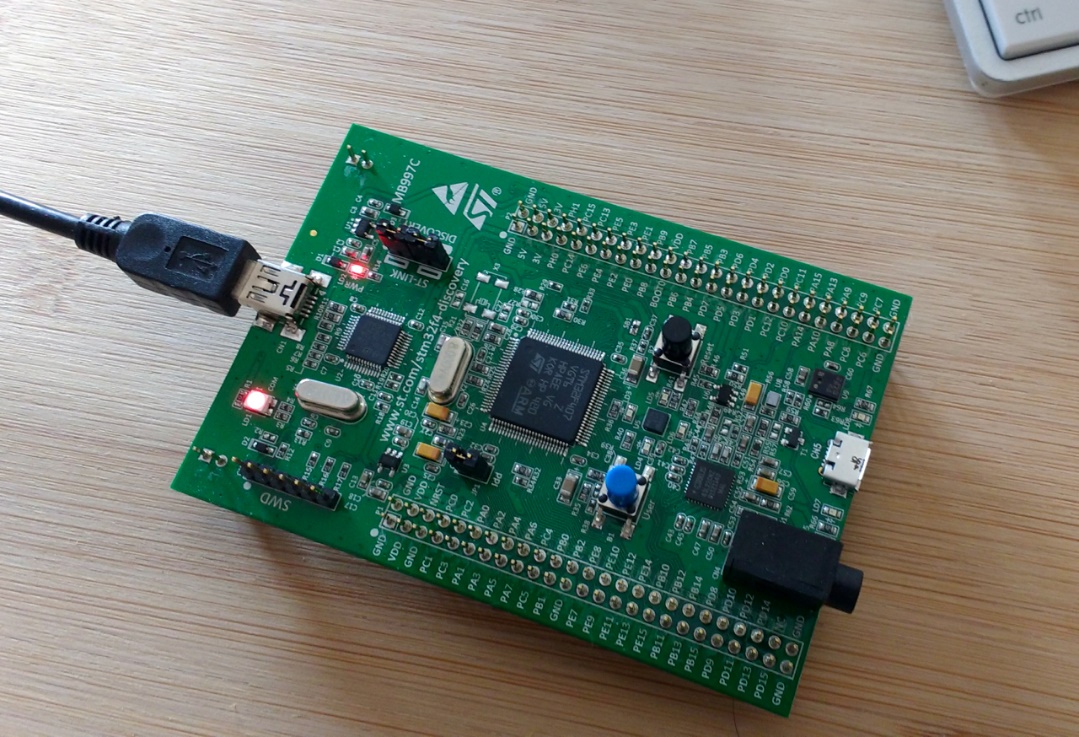


Figure : STM32F4 Discovery

# La HAL

## Qu’est-ce que la HAL

La HAL (Hardware Abstraction Layer) est une couche d’abstraction matériel permettant de faciliter la programmation à l’utilisateur, la HAL est constituée d’une multitude de fonctions et de variables réalisant les interactions avec les registres du microcontrôleur ou du processeur.

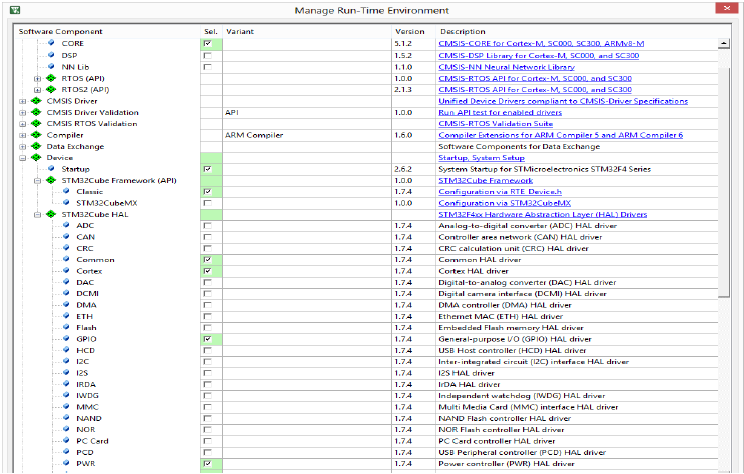


Figure : Configuration pour la HAL

Pour générer les différentes initialisations, et utilisé directement les différents composants présents sur la carte, nous utilisons STM32CUBEMX logiciel qui simplifie la programmation et réalise toutes les initialisations en fonction de ce que l’on a besoin.

## Utilisation de la HAL

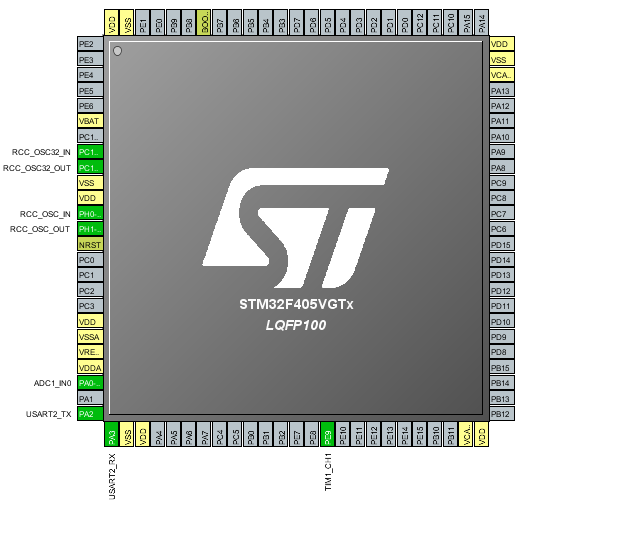


Figure : Configuration à l'aide CubeMX

Ci-dessus la configuration de la STM32F4 Discovery pour générer un signal ADC.

La programmation est facilitée grâce à la HAL, en effet si nous n’avions pas d’HAL nous devrions réaliser une étude de registre des fonctions nécessaires pour le projet. Et les fonctions HAL nous permette d’avoir une portabilité sur différentes STM32.

D’abord avant la sélection des différents composants nous sélectionnons la RCC, autrement dit ce qui gère l’horloge, et nous sélectionnons « CRYSTALCERAMIC RESONATOR.

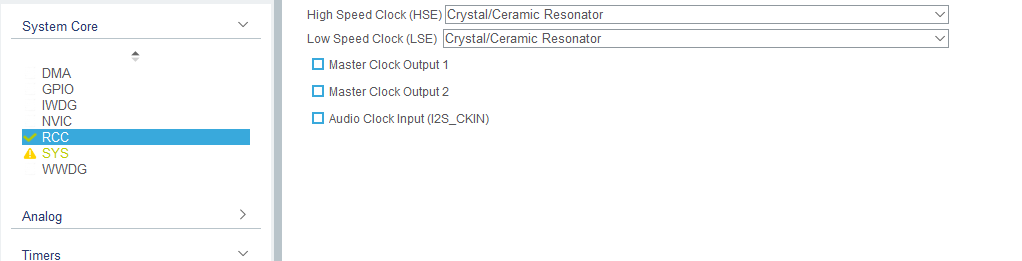


Figure Selection Horloge

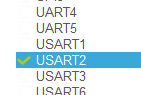
Ensuite , ayant besoin d’envoyer les différentes tension de la courbe, nous aurons besoin de l’UART pour envoyer ces position et l’afficher, donc on selectionne l’UART 2 (pas la premiere car l’ADC 1 utilisera une des broches utilisé par l’UART1).

Figure UART selection

Ensuite, nous mettons l’UART en asynchrone.

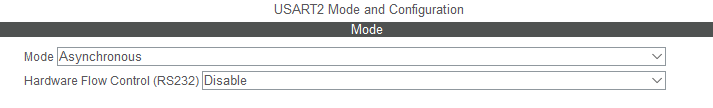


Figure Configuration de l'UART

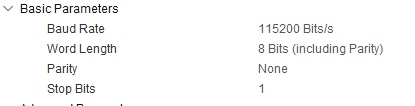
Dans la caractéristique de l’UART, qui seront utile de connaître lors de la lecture série du côté affichage.

Figure réglage UART

Dans ce projet, nous aurons aussi besoin d’un convertisseur analogique/ numérique pour avoir différents points de la courbe généré par le GBF et ainsi pouvoir la représenté.

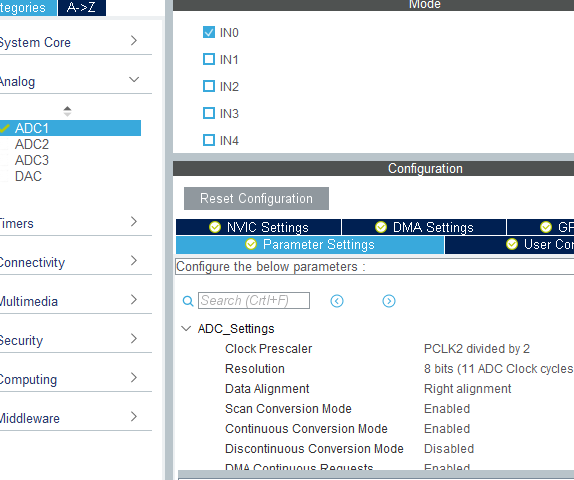


Figure ADC configuration

ADC1 sélectionné avec comme configuration 8 bits de résolution, le mode de génération de valeur en continue les autres paramètres seront laissés par défaut.

Si nous voulons réaliser un mode discontinu et à chaque évènement, nous envoyons une quantité limitée de valeur on peut ajouter un timer et mettre discontinue au lieu de la génération continue.

Une fois toute la configuration sélectionne nous gênerons le code et nous commencerons à coder L’ADC et l’UART.

# Partie STM32F4 Discovery

## Partie programmation

Le but de cette partie est de généré un signal sinusoïdal à l’aide de l’ADC présent sur la STM32. Pour ce faire, nous avons utilisé la HAL. Voici le code qui permet la génération de ce signal :

**#include** "main.h"

/\* Private variables ---------------------------------------------------------\*/

ADC\_HandleTypeDef hadc1;

DMA\_HandleTypeDef hdma\_adc1;

UART\_HandleTypeDef huart2;

DMA\_HandleTypeDef hdma\_usart2\_tx;

/\* Private function prototypes -----------------------------------------------\*/

**void** **SystemClock\_Config**(**void**);

**static** **void** **MX\_GPIO\_Init**(**void**);

**static** **void** **MX\_DMA\_Init**(**void**);

**static** **void** **MX\_ADC1\_Init**(**void**);

**static** **void** **MX\_USART2\_UART\_Init**(**void**);

CubeMX réalise ces 5 fonctions, et les appelles automatiquement dans le main, ce qui nous permet de passer directement à l’autre partie et nous permet d’avancer plus vites avec moins d’erreur possible.

**uint8t** data; //signed short pour avoir une précision suffisante pour l'affichage

**float** data\_convertit;// une fois la valeur convertit le short et inutil

/\* USER CODE END PFP \*/

\*/

**int** **main**(**void**)

{

HAL\_Init();

/\* Configure the system clock \*/

SystemClock\_Config();

/\* Initialize all configured peripherals \*/

MX\_GPIO\_Init();

MX\_DMA\_Init();

MX\_ADC1\_Init();

MX\_USART2\_UART\_Init();

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)"\033c",6,10); //reintialise la vision du terminal à chaque reset

Appelle des différentes fonctions d’initialisation permettant la bonne configuration de chaque composant utilisé, ici la communication USB UART et une ADC.

Pour la conversion du signal GBF vers un signal dit numérique, il faut une variable 8 bit non signé car la fonction permettant de contrôler l’ADC nous retournera une valeur sur 8 bits, ainsi on la stockera dans data.

Pour l’affichage il nous faut cette valeur en tension donc un float sera utile pour stocker cette valeur en float car elle sera obligatoirement à virgules. Dans le main, nous réinitialisons l’affichage de ce qui est envoyé sur l’USB UART.

**while** (1)

{

//conversion en continue

HAL\_ADC\_Start (&hadc1);//demarrag adc

HAL\_ADC\_PollForConversion(&hadc1,1000);//initialisation adc en reglant lequel ADC et combien de timout

data = HAL\_ADC\_GetValue(&hadc1);//récuperation de valeur

data\_convertit =(**float**) (((3.3)\* data)/(256-1));//conversion de la valeur en tension

HAL\_UART\_Transmit(&huart2,(&data\_convertit),2,10);//envoie de la valeur sur 2 octets car il faut 11 Clock d’horloge si la résolution est de 8 bits

HAL\_ADC\_Stop (&hadc1);

HAL\_Delay(10);

Pour réaliser une génération continue de valeur, il faut d’abord démarrer l’ADC en mettent bien en référence l’ADC que nous utilisons.

Après l’avoir démarré on utilise une fonction qui initialise l’ADC et qui demande le « TIMOUT », ensuite on récupérer la valeur on la convertit grâce à un produits en croix simple : tension =(VMAX) \* donnée reçues) /(2^N-1) et le résultat obtenu est envoyée par l’UART, et on arrête l’ADC, le tout réalisé dans une boucle infinie.

Cette fonction ci-dessous réalise la même chose mais en interruption, la différence c’est qui faut la mettre dans une fonction d’interruption prévue par le système. Et pour que l’interruption il faut indique dans ces paramètres le bouton qui réalisera l’interruption et donc l’évènements.

**void** **HAL\_ADC\_ConvCpltCallback**(ADC\_HandleTypeDef\* hadc){

**if**(HAL\_ADC\_PollForConversion(&hadc1,1000) == *HAL\_OK*)//

{

data = (**signed** **short**)HAL\_ADC\_GetValue(&hadc1);

//cast int vers char pour le stockage de la donnée reçue //en caractere

//permet la conversion de la valeur en tension pour l'affiché est l'afficher directement sur l'afficheur cpp

**if**(data<0)

data\_convertit = (((3.3)\*data)/255);

**else** **if** (data == 0 )

**return** data\_convertit=0;//si la valeur est 0 la conversion est inutile

}

HAL\_UART\_Transmit(&huart2,(&data\_convertit),2,100);

//envoie de 8 octet pour la transmission d'un double ,

//génération d'un warning car j'envoie un double au lieu d'un unsigned char

//mais si je realise ce cast la valeur ne conviendra plus

data = 0;//reinitialisation de la valeur

}

}

**void** **SystemClock\_Config**(**void**)

{

RCC\_OscInitTypeDef RCC\_OscInitStruct = {0};

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

/\*\*Configure the main internal regulator output voltage

\*/

\_\_HAL\_RCC\_PWR\_CLK\_ENABLE();

\_\_HAL\_PWR\_VOLTAGESCALING\_CONFIG(PWR\_REGULATOR\_VOLTAGE\_SCALE1);

/\*\*Initializes the CPU, AHB and APB busses clocks

\*/

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSI;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.HSICalibrationValue = RCC\_HSICALIBRATION\_DEFAULT;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_NONE;

**if** (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != *HAL\_OK*)

{

Error\_Handler();

}

/\*\*Initializes the CPU, AHB and APB busses clocks

\*/

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK|RCC\_CLOCKTYPE\_SYSCLK

|RCC\_CLOCKTYPE\_PCLK1|RCC\_CLOCKTYPE\_PCLK2;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_HSI;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV1;

RCC\_ClkInitStruct.APB2CLKDivider = RCC\_HCLK\_DIV1;

**if** (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct, FLASH\_LATENCY\_0) != *HAL\_OK*)

{

Error\_Handler();

}

}

/\*\*

\* @brief ADC1 Initialization Function

\* @param None

\* @retval None

\*/

**static** **void** **MX\_ADC1\_Init**(**void**)

{

ADC\_ChannelConfTypeDef sConfig = {0};

ADC\_InjectionConfTypeDef sConfigInjected = {0};

/\*\*Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)

\*/

hadc1.Instance = ADC1;

hadc1.Init.ClockPrescaler = ADC\_CLOCK\_SYNC\_PCLK\_DIV2;

hadc1.Init.Resolution = ADC\_RESOLUTION\_8B;

hadc1.Init.ScanConvMode = *DISABLE*;

hadc1.Init.ContinuousConvMode = *ENABLE*;

hadc1.Init.DiscontinuousConvMode = *DISABLE*;

hadc1.Init.ExternalTrigConvEdge = ADC\_EXTERNALTRIGCONVEDGE\_NONE;

hadc1.Init.ExternalTrigConv = ADC\_SOFTWARE\_START;

hadc1.Init.DataAlign = ADC\_DATAALIGN\_RIGHT;

hadc1.Init.NbrOfConversion = 1;

hadc1.Init.DMAContinuousRequests = *DISABLE*;

hadc1.Init.EOCSelection = ADC\_EOC\_SINGLE\_CONV;

**if** (HAL\_ADC\_Init(&hadc1) != *HAL\_OK*)

{

Error\_Handler();

}

/\*\*Configure for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.

\*/

sConfig.Channel = ADC\_CHANNEL\_0;

sConfig.Rank = 1;

sConfig.SamplingTime = ADC\_SAMPLETIME\_480CYCLES;

**if** (HAL\_ADC\_ConfigChannel(&hadc1, &sConfig) != *HAL\_OK*)

{

Error\_Handler();

}

/\*\*Configures for the selected ADC injected channel its corresponding rank in the sequencer and its sample time

\*/

sConfigInjected.InjectedChannel = ADC\_CHANNEL\_0;

sConfigInjected.InjectedRank = 1;

sConfigInjected.InjectedNbrOfConversion = 1;

sConfigInjected.InjectedSamplingTime = ADC\_SAMPLETIME\_3CYCLES;

sConfigInjected.ExternalTrigInjecConvEdge = ADC\_EXTERNALTRIGINJECCONVEDGE\_NONE;

sConfigInjected.ExternalTrigInjecConv = ADC\_INJECTED\_SOFTWARE\_START;

sConfigInjected.AutoInjectedConv = *DISABLE*;

sConfigInjected.InjectedDiscontinuousConvMode = *DISABLE*;

sConfigInjected.InjectedOffset = 0;

**if** (HAL\_ADCEx\_InjectedConfigChannel(&hadc1, &sConfigInjected) != *HAL\_OK*)

{

Error\_Handler();

}

}

**static** **void** **MX\_USART2\_UART\_Init**(**void**)

{

huart2.Instance = USART2;

huart2.Init.BaudRate = 115200;

huart2.Init.WordLength = UART\_WORDLENGTH\_8B;

huart2.Init.StopBits = UART\_STOPBITS\_1;

huart2.Init.Parity = UART\_PARITY\_NONE;

huart2.Init.Mode = UART\_MODE\_TX\_RX;

huart2.Init.HwFlowCtl = UART\_HWCONTROL\_NONE;

huart2.Init.OverSampling = UART\_OVERSAMPLING\_16;

**if** (HAL\_UART\_Init(&huart2) != *HAL\_OK*)

{

Error\_Handler();

}

}

/\*\*

\* Enable DMA controller clock

\*/

**static** **void** **MX\_DMA\_Init**(**void**)

{

/\* DMA controller clock enable \*/

\_\_HAL\_RCC\_DMA2\_CLK\_ENABLE();

\_\_HAL\_RCC\_DMA1\_CLK\_ENABLE();

/\* DMA interrupt init \*/

/\* DMA1\_Stream6\_IRQn interrupt configuration \*/

HAL\_NVIC\_SetPriority(*DMA1\_Stream6\_IRQn*, 0, 0);

HAL\_NVIC\_EnableIRQ(*DMA1\_Stream6\_IRQn*);

/\* DMA2\_Stream0\_IRQn interrupt configuration \*/

HAL\_NVIC\_SetPriority(*DMA2\_Stream0\_IRQn*, 0, 0);

HAL\_NVIC\_EnableIRQ(*DMA2\_Stream0\_IRQn*);

}

**static** **void** **MX\_GPIO\_Init**(**void**)

{

/\* GPIO Ports Clock Enable \*/

\_\_HAL\_RCC\_GPIOC\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOH\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

}

Toutes ces fonctions générer par CUBEMX nous évites de réaliser l’initialisation de chaque composant manuellement. Elles sont générées avec les paramètres qui nous intéresser.



## Partie Test

Une fois la partie programmation réaliser nous passons au test et nous remarquons que la valeur reçue sur data et bien convertit en tension. Ces deux relevés nous indiquent, que la partie ADC fonctionne parfaitement.

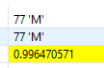


Figure relevés ADC n°1

Sur ce premier relever 0.99V = (77)10



Figure relevés ADC n°2

Sur ce relevés 0.54V = (42)10

En ce qui concerne de l’envoie UART , les valeurs afficher ne sont pas les valeurs que l’on a observés sur les relevés mais cela est tout à fait normal car la fonction hal envoie les données grâce à une variables en chaîne de caractères , or nous utilisons un float donc il envoie la valeur en chaîne de caractères sans la changer pour éviter les problèmes nous récuperons la valeur du côté affichage et on la remmetra en float pour avoir la bonne valeur.

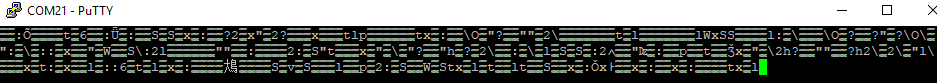


Figure relevés UART

Ce relevé ci-dessus nous indique que les données sont belles et bien envoyés.

Donc la partie STM32 est validée donc nous pouvons passer à la partie affichage.

# Partie QT Creator : affichage du signal

Ce relevés et la courbe représenté par le programme cpp.

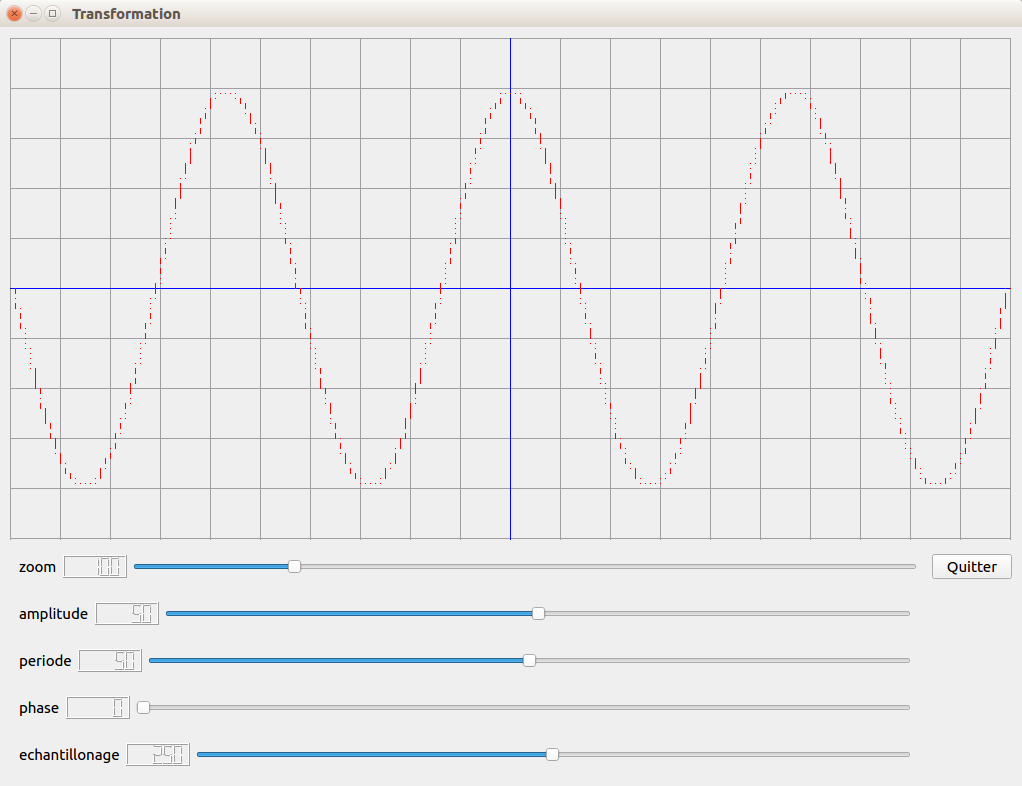


Figure : Affichage du signal

# Code sur QT Creator

## Main.c

#include "mainwidget.h"

#include <QApplication>

int **main**(int argc, char \*argv[])

{

QApplication a(argc, argv);

MainWidget w;

w.show();

return a.exec();

}

## Mainwidget.cpp

Partie qui rassemble tout les objet graphique utilisé pour l’affichage

#include "mainwidget.h"

#include "mycanvas.h"

#include "../LCDSlider/lcdslider.h"

#include <QPushButton>

#include <QGridLayout>

#include <QApplication>

#include <QSlider>

#include <QHBoxLayout>

#include "serialcapture.h"

#include <QLabel>

#include <QtSerialPort/QSerialPort>

MainWidget::**MainWidget**(QWidget \*parent)

: QWidget(parent)

{

QLabel\*lab0= new QLabel("data");

LcdSliderWidget \*lcd= new LcdSliderWidget("zoom",0,500,100);

//sliders réalisant le zoom et dézoome de la courbe

LcdSliderWidget \*lcd1= new LcdSliderWidget("amplitude",0,100,50);

//modifiera la courbe en amplitude si elle rentre pas le cadrillage représenté

LcdSliderWidget \*lcd2= new LcdSliderWidget("periode",0,100,50);

LcdSliderWidget \*lcd3= new LcdSliderWidget("phase",0,180,0);

LcdSliderWidget \*lcd4= new LcdSliderWidget("echantillonage",1,500,250);

// instanciation MyCanvas + QPushButton (pointeurs);

MyCanvas \*mc=new MyCanvas();

QPushButton \*qp1 = new QPushButton("Quitter");

// instanciation QGridLayout + ajout widgets

QGridLayout \*qgl=new QGridLayout();

// le QGridLayout devient le layout courant qgl->addWidget(qp1);

qgl->addWidget(mc,0,0,1,3);

qgl->addWidget(lcd,1,0,1,2);

qgl->addWidget(qp1,1,2);

qgl->addWidget(lcd1,2,0,1,1);

qgl->addWidget(lcd2,3,0,1,1);

qgl->addWidget(lcd3,4,0,1,1);

qgl->addWidget(lcd4,5,0,1,1);

//position de chaque objet graphique dans la fenêtre qui sera généré

setLayout(qgl);

// Ajouter des connexions SIGNAL -> SLOT

QObject::connect(qp1,SIGNAL(clicked()),qApp,SLOT(quit()));

connect(lcd,SIGNAL(valueChanged(int)),mc,SLOT(zoomchanged(int)));

connect(lcd1,SIGNAL(valueChanged(int)),mc,SLOT(amplitudeChanged(int)));

connect(lcd2,SIGNAL(valueChanged(int)),mc,SLOT(periodeChanged(int)));

connect(lcd3,SIGNAL(valueChanged(int)),mc,SLOT(phaseChanged(int)));

connect(lcd4,SIGNAL(valueChanged(int)),mc,SLOT(echantillonageChanged(int)));

ces connect permet de relier la variable qui genere et modufie le signal au sliders qui lui affichera juste la valeur que l’on envoie pour modifier la courbe.

emit lcd->valueChanged(100);//on genere une valeur par defauts du zoom qui sera en gros l’echellle 1 et en modifiant cette valeur on modifie l’echelle.

}

MainWidget::~***MainWidget***()

{}

## mycanvas.cpp

**#include "mycanvas.h"**

#include <QPainter>

#include <QColor>

#include <QtMath>

#include <cmath>

#include <QPushButton>

#include <QGridLayout>

#include <QApplication>

#include <QSlider>

#include <QPixmap>

#include <QFile>

#include "serialcapture.h"

// genere l'unite du graphique et possibilité de zoom

QPointF MyCanvas::**coo**(int x , int y)

{

return QPointF((x\*m\_unite\*m\_zoom)/100,(y\*m\_unite\*m\_zoom)/100);

}

// fonction test qui genere une fonction cosinus

double MyCanvas::**f**(double x)

{

return m\_amplitude\*(cos(11\*x\*m\_periode+m\_phase));

}

// fonction test qui genere une fonction sinus

double MyCanvas::**g**(double x)

{

return m\_amplitude\*(sin(x\*m\_periode+m\_phase));

}

// la fenetre genere pour le graphique sera fixe en hauteur et largeur

MyCanvas::**MyCanvas**(QWidget \*parent)

{

setFixedSize(m\_largeur+2,m\_hauteur+2);

}

// fonction qui dessinnera la courbe et le graphique et les axes

void MyCanvas::***paintEvent***(QPaintEvent \*)

{

SerialCapture m\_data; //reference à la donnée que l’on reçoit dans la classe Serial capture

signed i;

int y=10;

QPainter feuille(this);

// 1) Sauvegarde du système de coordonnées

feuille.save();

// 2) Translation, rotation dou changement de coordonnées

feuille.translate(m\_largeur/2,m\_hauteur/2);

feuille.scale(1,-1);

//dessin

// Écriture des différentes lignes dans le fichier

//grille horizontale

feuille.setPen(Qt::gray);

for (i=-m\_largeur/10;i<m\_largeur/10;i+=y)

{

feuille.drawLine(coo(i,m\_largeur/10),coo(i,-m\_largeur/10));

feuille.drawLine(coo(m\_largeur/10,i),coo(-m\_largeur/10,i));

}

feuille.drawLine(coo(-m\_largeur/10,-m\_largeur/10),coo(m\_largeur/10,-m\_largeur/10));

feuille.drawLine(coo(m\_largeur/10,-m\_largeur/10),coo(m\_largeur/10,m\_largeur/10));

feuille.drawLine(coo(-m\_largeur/10,m\_largeur/10),coo(m\_largeur/10,m\_largeur/10));

/\*feuille.drawLine(coo(-49,-49),coo(49,-49));

feuille.drawLine(coo(49,49),coo(49,-49));\*/

// dessiner dans le pixmap off-screen

//restauration du systeme de coo

// ordonnées et abscisse

feuille.setPen(Qt::blue);

feuille.drawLine(coo(0,-m\_largeur/10),coo(0,m\_largeur/10));

feuille.drawLine(coo(-m\_largeur/10,0),coo(m\_largeur/10,0));

//représentation des points envoyer par la stm32

feuille.setPen(Qt::red);

for (double var = -100; var < 100; var+=0.005)

{

for (unsigned long i=0 ;i<sizeof (m\_data);i++)

{

feuille.drawLine(coo((double)var,m\_amplitude\*(m\_data[i]\*m\_periode+m\_phase)),

coo((double)var++,m\_amplitude\*(m\_data[i++]\*m\_periode+m\_phase)));

}

}

feuille.restore();

}

//ajout de ce que l'on reçoit dans le QList qui sera utilisé pour la représentation

void MyCanvas::**receivedata**(QList<double>a)

{

m\_datarx.append(a);

}

// Sliders permettant l'adaptation du zoom de la fenêtre

void MyCanvas::**zoomchanged**(int z)

{

m\_zoom=z;

repaint();

}

//Sliders, Modifie l'amplitude du signal

void MyCanvas::**amplitudeChanged**(int amp)

{

m\_amplitude=amp;

repaint();

}

//Sliders,Modifie la période du signal

void MyCanvas::**periodeChanged**(int t)

{

m\_periode=t;

repaint();

}

// slider, modifie la phase du signal

void MyCanvas::**phaseChanged**(int alpha)

{

m\_phase=alpha;

repaint();

}

// ajout ou suppressiond'echantillon au signal

void MyCanvas::**echantillonageChanged**(int echantillon)

{

m\_echantillonage=echantillon;

repaint();

}

## serialcapture.cpp

**#include "serialcapture.h"**

#include <QDebug>

#include <QSerialPortInfo>

#include <QSerialPort>

#include <QFile>

SerialCapture::**SerialCapture**()

{

// configuration du port serie

m\_serial=new QSerialPort(this);

m\_serial->setPortName("/dev/ttyACM0");

m\_serial->setBaudRate( QSerialPort::Baud115200);

m\_serial->setDataBits( QSerialPort::Data8);

m\_serial->setStopBits(QSerialPort::OneStop);

m\_serial->setParity(QSerialPort::NoParity);

m\_serial->setFlowControl( QSerialPort::NoFlowControl);

if(!m\_serial->*open*(QIODevice::ReadOnly))

{

qDebug() <<QObject::tr("Je ne peux pas ouvrir le port série %1...").arg(m\_serial->portName());

}

connect(m\_serial,static\_cast<void (QSerialPort::\*)(QSerialPort::SerialPortError)>(&QSerialPort::error), this,&SerialCapture::handleError); connect(m\_serial,&QSerialPort::readyRead,this,&SerialCapture::readData);

}

// fonction retournant une erreur, si il a un probleme avec le port serie

void SerialCapture::**handleError**(QSerialPort::SerialPortError error)

{

if(error==QSerialPort::ResourceError)

{

qDebug() <<QObject::tr("Erreur critique : %1").arg(m\_serial->errorString());

}

}

// envoie des donnée dans le QList

void SerialCapture::**readData**(QList<double>b)

{

QByteArray buf=m\_serial->readAll();

m\_data.append(buf.toDouble());

return b.append(m\_data);;

}

## mainwidget.h

#ifndef MAINWIDGET\_H

#define MAINWIDGET\_H

#include <QWidget>

#include <QtSerialPort/QSerialPort>

class **MainWidget** : public QWidget

{

Q\_OBJECT

public:

**MainWidget**(QWidget \*parent = 0);

~***MainWidget***();

};

#endif // MAINWIDGET\_H

## mycanvas.h

#ifndef MYCANVAS\_H

#define MYCANVAS\_H

#include <QWidget>

#include <QTextStream>

#include <QtSerialPort/QSerialPort>

#include"serialcapture.h"

#include "QList"

#include <QObject>

class **MyCanvas**: public QWidget

{

Q\_OBJECT

private:

static const int m\_largeur=1000,m\_hauteur=500;

static const int m\_unite=5;

QPointF **coo**(int x, int y);

QList<double> m\_datarx;

int m\_zoom;

QTextStream flux;

float m\_echantillonage=250;

int m\_amplitude=40, m\_periode=10, m\_phase=0;

double **f**(double x);

double **g**(double x);

public:

explicit **MyCanvas**(QWidget \*parent = 0);

protected:

void ***paintEvent***(QPaintEvent \*);

public slots:

void **receivedata**(QList<double>);

void **zoomchanged**(int);

void **amplitudeChanged**(int);

void **periodeChanged**(int);

void **phaseChanged**(int);

void **echantillonageChanged**(int echantillon);

};

#endif // MYCANVAS\_H

## serialcapture.h

#ifndef SERIALCAPTURE\_H

#define SERIALCAPTURE\_H

#include <QWidget>

#include <QList>

#include <QtSerialPort/QSerialPort>

class **SerialCapture**: public QWidget

{

Q\_OBJECT

private:

QList<double> m\_data;

QSerialPort \*m\_serial;

public:

**SerialCapture**();

public slots:

void **handleError**(QSerialPort::SerialPortError error);

signals:

void **readData**(QList<double>);

};

#endif // SERIALCAPTURE\_H