Table des matières

Introduction 2

Chapitre 1 : TrustZone 3

1.1 Décomposition de la solution du sujet…….…….3

1.1.1 Activation de la TrustZone……...……..…3

1.1.2 Activation de la TrustZone……...……..…4

1.1.3 Configuration logiciel (Software)……...5

Chapitre 2 : Création d'une interface graphique 8

2.1 Introduction……...…….....….8

2.2Création de l'interface graphique…….……..8

2.2.1 Importation des objets à partir de PyQt5……...……10

2.2.2 Création des deux fenêtres ……....10

2.2.2.1 Création de la première fenêtre…….……10

2.2.2.2 Passage de la première a la deuxième fenêtre……12

2.2.2.3 Création du clavier dans la deuxième fenêtre……..…13

2.2.2.4 Mise à jour des informations affichées à l'écran……..…..16

2.2.2.5 Création du contrôleur……....17

2.2.2.6 Transmission série des données entre l'interface et la carteSTM32……....19

2.2.2.6.1 Configuration nécessaire dans le STM32 pour recevoir les données ……...20

Chapitre 3 : Exécutable de l'interface graphique 21

3.1 Création d'un exécutable de l'interface graphique ……21

Conclusion 25

Introduction :

De nos jours, la sécurité des dispositifs est de plus en plus primordiale, Cela est dû au fait que l'Internet des objets fait partie intégrante de notre utilisation quotidienne, ce qui augmente le risque de faire face à des attaques potentielles.

Dans le cadre de notre formation, nous avons réalisé notre stage au sein du laboratoire XLIM répondant à ces enjeux du futur en matière de sécurité des systèmes des IOT. La mission de l'évaluation de la technologie TrustZone dans le cadre d'applications spécifiques à l'IoT, nous a attiré particulièrement, puisqu'elle éveille notre esprit créatif tout en se basant sur nos compétences acquis au cours de notre formation, ce qui nous a donner une bonne pratique qui illustre le monde professionnel.

Notre projet sera une bonne application des bonnes fonctionnalités de la technologie TrustZone, tout en l'appliquant sur la sécurité d'un portail électrique, qui sera ouvrable que pour les personnes qui saisissent le même code secret qui existe dans la mémoire sécurisée du système, et cette comparaison sera effectuer tout en respectant les différents accès possible et impossible entre les trois parties de la mémoire Secure, non Secure et la partie Secure Callable.

Dans un premier temps nous décrirons les différentes étapes de configuration de l'activation et la désactivation de la TrustZone sur la carte STM32L5, à la seconde partie nous allons aborder la configuration Software tout en détaillant les différentes parties du programme avec une explication détaillés des différents passages d'une zone mémoire a une autre. A la fin nous allons présenter une interface graphique qui fera l'intermédiaire entre l'utilisateur du matériel et du système Software.

Chapitre 1

TrustZone

1.1Décomposition de la solution du sujet

Pour résoudre notre problématique nous avons procéder de la façon suivante :

1.1.1 Activation de la TrustZone

Pour résoudre notre problématique nous avons procéder de la façon suivante :
Au tout début, nous avons commencé par installer les logiciels nécessaires (STM32 CubeMX, STM32CubeIDE et STM32CubeProgrammer). Une fois c'est fait nous avons commencé la configuration de la TrustZone en suivant les étapes suivantes :

1. Connecter la carte STM32L552ZE-Q au STM32CubeProgrammer
2. Configurer la TrustZone dans l'ongle OB « Option Bytes », comme suit :

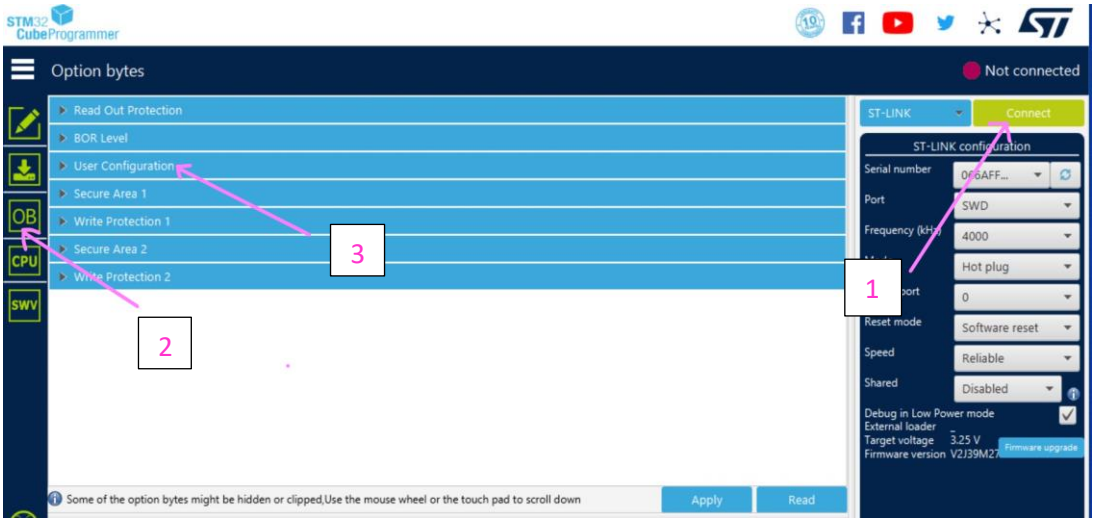

Figure 1.1 Les étapes d'activation de la TrustZone dans STM32CubeProgrammer

Pour commencer, nous avons sélectionner le TZEN et confirmer que le DBANK est à 1 pour activer le mode double banque du flash, comme suit :

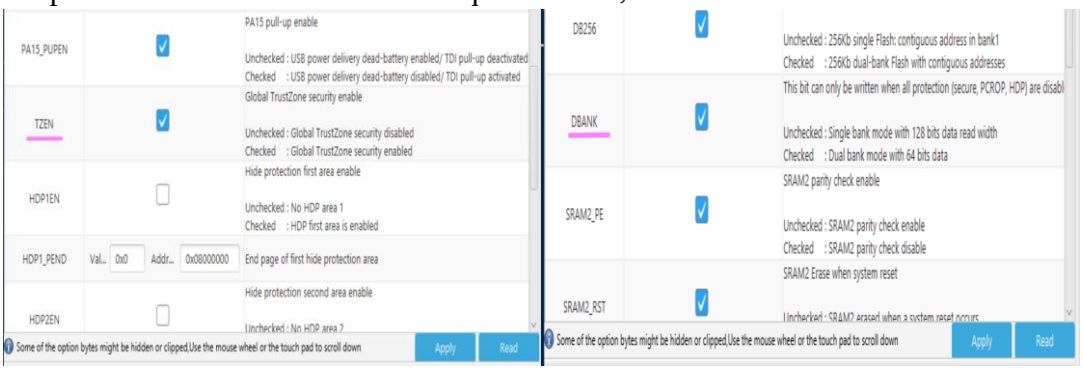

Figure 1.2 Activation de TZEN et DBANK dans User Configuration

Il faut s'assurer que ces valeurs soient en unités de pages Flash (d'où le nom PSTRT/Page-Start et PEND/Page-end) qui doivent être ajoutées à l'adresse de base de l'adresse de base Flash Bank correspondante. Comme le montre la figure ci-dessous.

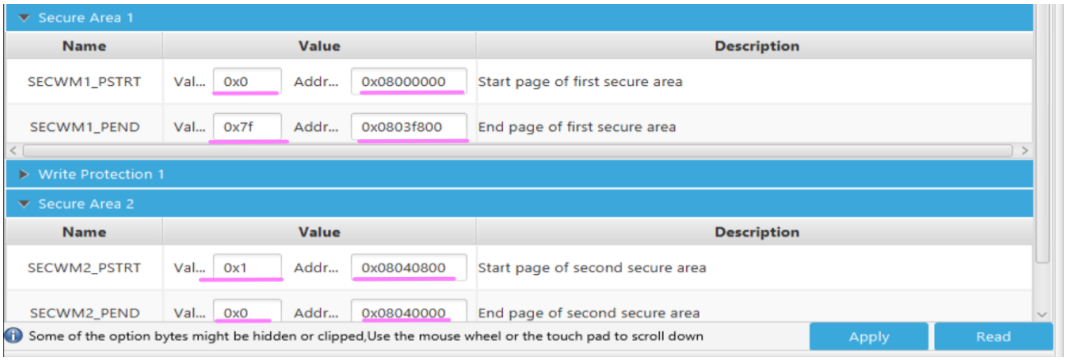

Figure 1.3 Définir les zones sécurisée et non sécurisée dans Secure Area1 et 2

Pour résumer les étapes :

- Nous avons activé la TrustZone. (TZEN)
- Puis régler le flash pour fonctionner en mode banque. (DBANQUE)
- Enfin, nous avons défini le Bank1 comme sécurisé (SECWM1) et Bank2 comme non sécurisé (SECWM2).

1.1.2 Configuration matérielle (Hardware)

Une fois l'activation de la Trust Zone est effectuée avec succès, nous allons procéder dans cette partie aux configurations matérielle au niveau du STM32CubeMX afin d'initialiser le programme qui nous permettra de coder notre système de Trust Zone. Après avoir définie tous nos besoins, nous avons décidé d'utiliser :

- Le LPUART1 pour établir une communication série sécurisé car nous allons l'utiliser pour envoyer les données de la carte STM32L552ZE-Q a l'interface PuTTY.

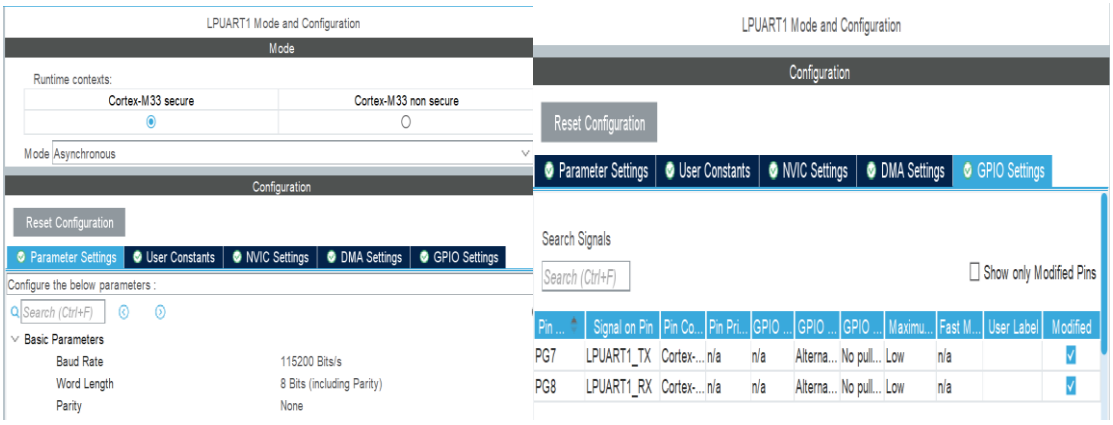

Figure 1.4 Configuration du LPUART1dans le STM32CubeMX

- **Le USART3** pour établir une communication série non sécurisé car nous allons l'utiliser pour recevoir le code saisit par l'utilisateur a la carte STM32L552ZE-Q et cela ne nécessite pas d'êtres sécurisé.

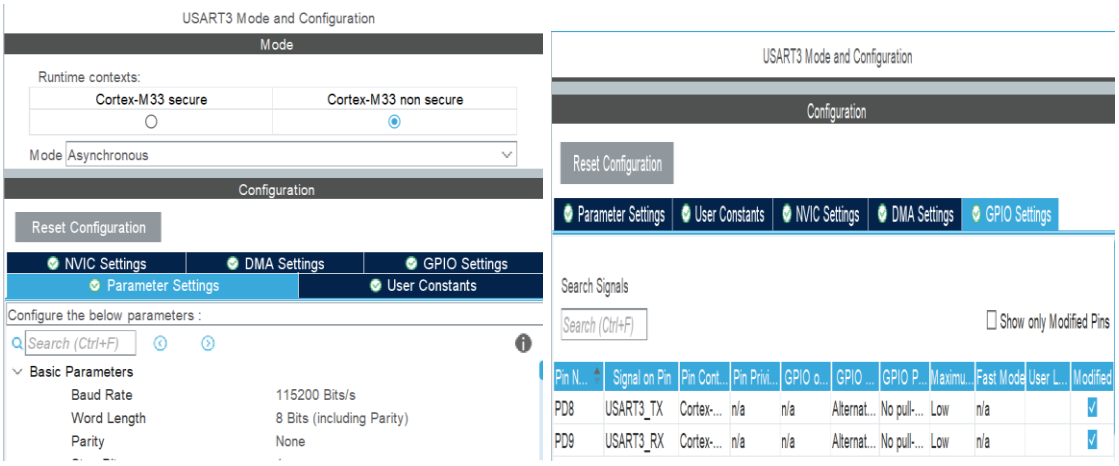

Figure 1.5 Configuration du USART3 dans le STM32CubeMX

- **Les LEDs PA9 et PC7** ont été déclarées dans un contexte d'exécution sécurisé, pour qu'elles ne peuvent pas être contrôlés dans le contexte non-sécurisé, vu qu'ils simulent l'ouverture du portail.

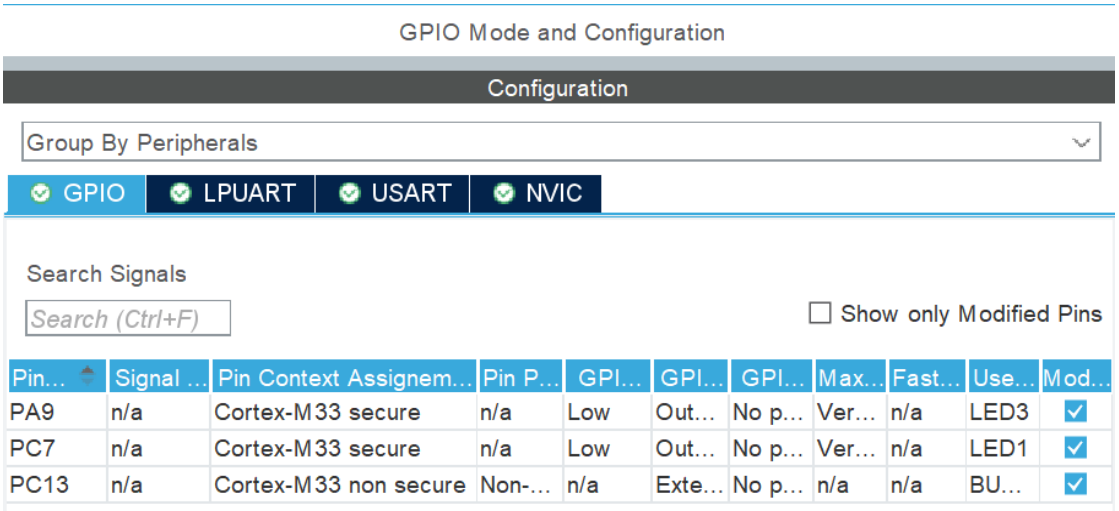

Figure 1.6 Configuration des LEDs dans le STM32CubeMX

1.1.3 Configuration logiciel (Software)

Dans cette partie nous allons parler de la création du programme qui fait la vérification de code secret tout en respectant les étapes de la TrustZone, qui assure la sécurisation du code secret.

Le programme que nous allons présenter consiste à exécuter 3 codes séparés ou chaque code il représente une partie de la mémoire (Secure, Non Secure et Non Secure Callable), mais chaque code est relié avec l'autre pour donner le fonctionnement voulu tout en respectant le fait que le mode non sécurisé ne doit accéder directement à la partie sécurisée, en revanche il doit passer par la partie non Secure callable.

Le programme Non Secure commence par récupérer le code secret **num** envoyer pas l'utilisateur en utilisant la fonction **scanf ().**

```
135
136   /* USER CODE BEGIN 2 */
137
138   PRINTF_NSE("Please Enter the Access Code: \r\n");
139     scanf("%d", &num);
140
```

Figure 1.7 Fonction **scanf ()** qui récupère le code secret

Ensuite il fait appel à une fonction de comparaison que nous avons nommé **PinCompare_NSE()**, qui se situe dans la partie Non Secure Callable, comme suit :

```
148 // Appeler la fonction de comparaison
149
150   result = PinCompare_NSE(num);
```

Figure 1.8 Appel de la fonction de comparaison

Cette dernière fait appel à une fonction **GetTestCaseNumber_NSE ()** qui est une fonction d'entrée à la partie sécurisée, comme suit :

```
147 /*-------------------------------------------------------------------------------------------------------*/
148 /*-----------Non-secure callable (entry) function, calling a non-secure callback function-----------*/
149 CMSE_NS_ENTRY int PinCompare_NSE(uint32_t s1)
150 {
151     if(s1 == GetTestCaseNumber_NSE())
152     {
153         return 1;
154     }
155     else
156     {
157         return 0;
158     }
159 }
```

Figure 1.9 Fonction de comparaison définie dans la partie Non Secure Callable

Sachant que **GetTestCaseNumber_NSE ()** est définie dans la partie Non Secure Callable et utiliser pour récupère le code secret **testCaseNumber** du portail qui est dans la partie sécurisée, en appelant la fonction **GetTestCaseNumber ():**

```
140 /*-------------------------------------------------------------------------------
141 /*----------------------------------Non-secure callable (entry) function---------
142 CMSE_NS_ENTRY uint32_t GetTestCaseNumber_NSE(void)
143 {
144     return GetTestCaseNumber();
145 }
146
```

Figure 1.10 Fonction de récupération du code secret de la zone sécurisée

La fonction **GetTestCaseNumber ()** est définie dans la partie sécurisée comme suit **:**

```
63 uint32_t testCaseNumber=15012000;
64
65 /* USER CODE END PV */
66
67 /* Private function prototypes ---
68 static void NonSecure_Init(void);
69 void SystemClock_Config(void);
70 /* USER CODE BEGIN PFP */
71
72 /* USER CODE END PFP */
73
74 /* Private user code -------------
75 /* USER CODE BEGIN 0 */
76
77 uint32_t GetTestCaseNumber()
78 {
79     return testCaseNumber;
80 }
```

Figure 1.11 Fonction de récupération du code secret caché dans la partie sécurisée

Comme vous le voyer sur la figure ci-dessus le code secret **testCaseNumber** et bien dans la partie sécurisée et il y a que la fonction **GetTestCaseNumber ()** qui peut le récupérer.

Une fois le code secret est récupéré, **PinCompare_NSE ()** compare les deux codes et fait un retour vers le mode non sécurisé pour ouvrir la porte si les code sont identique ou n'accepte pas d'ouvrir s'ils ne sont pas identiques.

Dans notre cas nous avons choisi de représenter l'ouverture du portail par le clignotement de la LED verte pendant 5 secondes avec un message « Correct Code », dès que les 5 secondes se termine le portail se fermera ce qui veux dire la LED verte s'éteint. En revanche l'accès refusé nous l'avons simulé par la LED rouge, avec un message « Wrong Code ». Comme le montre la fonction ci-dessous

```
151 if (result == 1)
152 {
153     PRINTF_NSE("Correct Code!\r\n");
154     Access_ok_toggle();
155     HAL_Delay(5000);
156     Access_ok_toggle();
157 }
158 else
159 {
160     PRINTF_NSE("Wrong Code!\r\n");
161     Access_Nok_toggle();
162 }
163
```

Figure 1.12 Appel des fonctions pour clignoter les LEDs et affichage des messages sur PuTTY

Pour allumer l'une des LEDs nous avons créés les deux fonction Acces_ok_toggle () et Access_Nok_toggle () et pour afficher les messages nous avons définis une fonction PRINTF_NSE () qui sont définies dans la partie Non Secure Callable. Ce dernier ne

peut être accessible que par le LPUART1 qui est déclaré dans la partie sécurisée. Voir la figure ci-dessous :

```
/*-------------------------------------------------------------------------------
/*-----------------------------------------LED CONTROL----------------------
CMSE_NS_ENTRY void Access_ok_toggle(void)
{
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
}

CMSE_NS_ENTRY void Access_Nok_toggle(void)
{
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_9);
}

```

Figure 1.13 Fonction pour clignoter les LEDs

Chapitre 2

Création d'une interface graphique

2.1 Introduction

L'interface graphique est le langage d'échange entre l'humain (l'utilisateur) et la machine (notre système d'exploitation). Chaque système doit se disposer de sa propre interface afin de facilité les différentes interactions avec le produit informatique.

Dans cette partie nous allons essayer de décrire d'une façon très simple les différentes étapes nécessaires à suivre pour la création d'une interface graphique.
Nous allons décomposer l'interface graphique en élément simple

2.2 Création de l'interface graphique

Il existe plusieurs outils pour créer une interface graphique : langage de programmation C, C++, Python, Java, etc…
Nous avons choisi de travailler avec le langage de programmation Python, et pour cela nous avons utiliser le logiciel Anaconda, pour cela il suffit juste de voir le lien suivant : https://www.anaconda.com/products/distribution

Figure 2.1 La page officiel pour télécharger ANACONDA

Nous avons choisi Jupiter afin de développer l'interface :

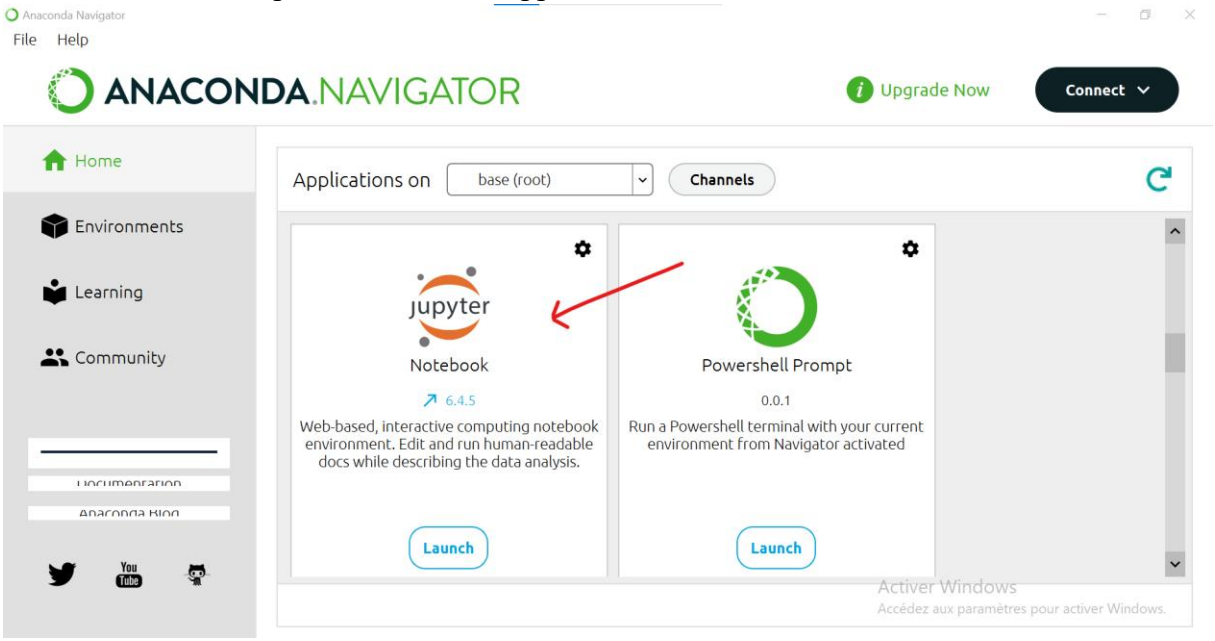

Figure 2.1 Les éditeurs utilisés

Pour commencer nous avons choisi la liaison Python pyqt5 de la boîte à outils d' interface graphique multiplateforme Qt , implémentée en tant que plug-in Python.

Nous avons utilisé la commande **conda install -c anaconda pyqt** et **pip installer PyQt5** pour installer la bibliothèque PyQt5.

Notre interface aura la structure et les fonctionnalités suivantes :

- Première fenêtre contient un bouton « log in »
- Deuxième fenêtre contient un clavier
- Transmission du code secret de l'interface a la carte STM32

2.2.1 Importation des objets à partir de PyQt5

A ce niveau nous pouvons fournir toutes les importations nécessaires à partir de **PyQt5** pour utiliser les différents objets qui existe dans les modules, comme suit :

```
"""PyClavier est un simple clavier qui utilise Python et PyQt5."""

import sys
from PyQt5 import QtCore, QtWidgets

# Importez QApplication et les widgets requis depuis PyQt5.QtWidgets
from PyQt5.QtWidgets import QApplication
from PyQt5.QtWidgets import QMainWindow
from PyQt5.QtWidgets import QWidget

#Importations pour la mise en forme du clavier
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QGridLayout
from PyQt5.QtWidgets import QLineEdit
from PyQt5.QtWidgets import QPushButton
from PyQt5.QtWidgets import QVBoxLayout
```

Figure 2.3 Importation des objets nécessaires

Tout d'abord, nous avons importez **sys**, ce qui va nous permettre de gérer le statut de sortie de l'application. Ensuite, nous avons importé QApplication, **QWidget** depuis **QTWidgets**, qui font partie du package appeler **PyQt5** que nous allons les utilisez plus tard pour avoir un squelette pour construire le clavier et l'interface d'accueil.

L'import ation de **QVBoxLayout** a pour but la disposition générale de l'interface. Nous utilisant également un **QGridLayout** objet pour organiser les boutons, et en fin nous importons pour l'affichage QLineEdit et **QPushButton** pour les boutons.

2.2.2 Création des deux fenêtres

2.2.2.1 Création de la première fenêtre

Pour commencer nous avons besoin d'une fenêtre principale qui fera la page d'accueil, pour cela nous allons créer une classe Principale comme suit :

```
##########################################################################

class Principal(QtWidgets.QMainWindow):

    def __init__(self, parent=None):

        """Initialise la fenêtre"""
        super(Principal, self).__init__()
        self.setWindowTitle(u"TrustZone log in")
        self.setFixedSize(400, 400)

        # mettre un fond (nécessaire avec un QMainWindow)
        self.setCentralWidget(QtWidgets.QFrame())

        # créer un bouton
        self.bouton = QtWidgets.QPushButton(u"Log in", self.centralWidget())
        self.bouton.clicked.connect(self.quelclient_m)

        # positionner les widgets sur le fond de la fenêtre
        posit = QtWidgets.QGridLayout()
        posit.addWidget(self.bouton, 1, 0)
        self.centralWidget().setLayout(posit)
```

Figure 2.4 initialisation de la fenêtre principale

Cette première partie de définition __init__ va permettre de crée une fenêtre qui comporte un bouton log in, qui a était créé en utilisant **QtWidgets.QPushButton**. En cliquant sur le bouton la deuxième fenêtre s'affiche grâce à la fonction **self.bouton.clicked.connect(self.quelclient_m).**

La taille de fenêtre sera fixé par la commande **self.setFixedSize**.
La ligne ou nous avons utilisé la commande **self.setWidowTitle** définit le titre de la fenêtre.

A la fin nous avons positionner les widgets sur le fond de la fenêtre.

Remarque très importante :
Jusqu'ici le travail n'est pas complet, car pour afficher la fenêtre il faut créer un main () ou nous ajoutons tous les opérations nécessaires et très importantes suivant :

- Crée un objet QApplication (sys.argv) : ce objet nous devons le créer avant tout autre objet lié à l'interface graphique car il effectue de nombreuses initialisations.
  L'objet QApplication () traite également des arguments de ligne de commande courants, donc nous devons également les transmettre en sys.argv qu'argument lors de la création d'app.
- Il faut faire appel à Principale.show () : . show () planifie un évènement de peinture des widgets qui composent une interface graphique)
- A la fin il faut Exécutez la boucle d'événements de notre application (ou boucle principale)

Ci-dessous le main () finale de notre interface graphique :

```
#############################################################

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = Principal()
    main.show()
    sys.exit(app.exec_())
```

Figure 2.5 Déclaration du main ()

D'après ce script, la fenêtre suivante apparait sur notre écran :

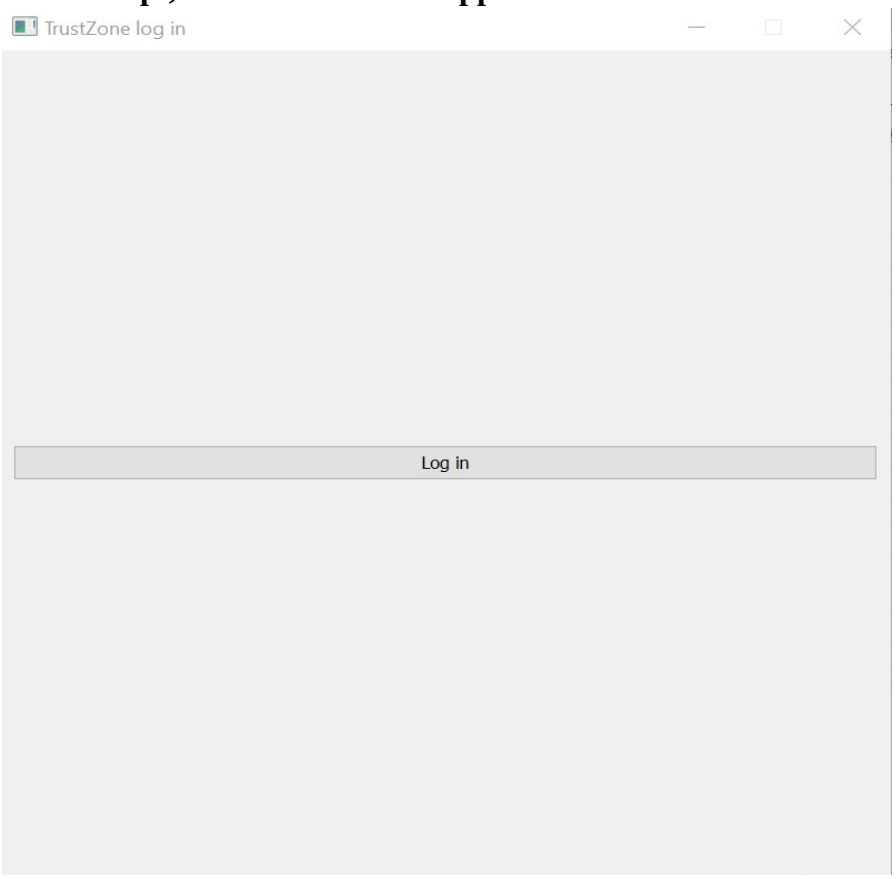

Figure 2.6 Première fenêtre de l'interface graphique

2.2.2.2 Passage de la première a la deuxième fenêtre

Pour passer à la deuxième fenêtre il faut faire appel à la classe du clavier en créant l'instance de classe et demandons l'affichage à l'aide de ces deux lignes de code :
quelclient = PyClavierUi()
quelclient.show()

```
def quelclient_m(self):
    """Lance la deuxième fenêtre"""
    quelclient = PyClavierUi()
    test=PyClavierCtrl(quelclient)

    # la deuxième fenêtre sera 'modale' (la première fenêtre sera inactive)
    quelclient.setWindowModality(QtCore.Qt.ApplicationModal)

    # appel de la deuxième fenêtre
    quelclient.show()
```

Figure 2.7 Appel à la deuxième fenêtre en appelant **PyClavierUi**

Pour que la deuxième fenêtre soit modale par défaut au lancement, nous avons ajouté une ligne de commande avant show () comme il est montre dans l'image ci-dessus.

2.2.2.3 Création du clavier dans la deuxième fenêtre

Après avoir terminé la création de la première fenêtre et l'appelle de la deuxième fenêtre et établi, nous devons crée la classe **PyClavier**, comme suit :

```
class PyClavierUi(QMainWindow):
    """PyClavier's View (GUI)."""
    def __init__(self):
        """View initializer."""
        super().__init__()
        # Set some main window's properties
        self.setWindowTitle('PyClavier')
        self.setFixedSize(235, 235)

        # creation du gestionnaire de mise en forme de type QVBoxLayout
         # Set the central widget and the general layout
        self.generalLayout = QVBoxLayout()

        # Set the central widget
        self._centralWidget = QWidget(self)
        self.setCentralWidget(self._centralWidget)

        self._centralWidget.setLayout(self.generalLayout)
        # Create the display and the buttons
        self._createDisplay()
        self._createButtons()

    def _createLogin(self):
        self.Login=QPushButton("Log in")
        LoginLayout=QGridLayout()
        Loginlayout.addWidget(self.Login[btnText])
```

Figure 2.8 Initialisation de la deuxième fenêtre

Au début nous avons met quelque propriété pour la fenêtre comme a était déjà fait pour la première fenêtre, puis nous avons utilisé un **QVBoxLayout** pour placer l'affichage en haut et les boutons en forme de grille en bas.

Mais les appels a **._createDisplay ()** et **._createButtons ()** ne fonctionnent pas, car nous n'avons pas encore implémenté ces méthodes. Pour corriger cela nous avons coder **._createDisplay ()** :

```
##############################################################################
# Create a subclass of QMainWindow to setup the calculator's GUI
class PyClavierUi(QMainWindow):

    # Snip
    def _createDisplay(self):
        """Create the display."""
        # Create the display widget
        self.display = QLineEdit()
        # Set some display's properties
        self.display.setFixedHeight(35)
        self.display.setAlignment(Qt.AlignRight)
        self.display.setReadOnly(True)
        # Add the display to the general layout
        self.generalLayout.addWidget(self.display)
```

Figure 2.9 Activation des fonctions **._createDisplay ()** et **._createButtons ()**

Pour créer le widget d'affichage, nous avons utilisé l'objet **QLineEdit**. Ensuite, nous avons défini les propriétés d'affichage suivantes :

- L'affichage a une hauteur fixe de 35pixels.
- L'écran affiche le texte aligné à gauche.
- L'affichage est défini en lecture seule pour éviter l'édition directe.

La dernière ligne ajoute l'affichage à la disposition générale du clavier avec **generalLayout.addWidget ().**

Ensuite, nous avons implémenter **._createButtons ()** pour créer les boutons de notre clavier, comme suit :

```
    def _createButtons(self):
        """Create the buttons."""
        self.buttons = {}
        buttonsLayout = QGridLayout()
        # Button text | position on the QGridLayout
        buttons = {'7': (0, 0),
                   '8': (0, 1),
                   '9': (0, 2),
                   'C': (3, 2),
                   '4': (1, 0),
                   '5': (1, 1),
                   '6': (1, 2),
                   '1': (2, 0),
                   '2': (2, 1),
                   '3': (2, 2),
                   '0': (3, 0),
                   '00': (3, 1),
                   'Entrer': (2, 4)
                  }

    # Create the buttons and add them to the grid layout
        for btnText, pos in buttons.items():
            self.buttons[btnText] = QPushButton(btnText)
            self.buttons[btnText].setFixedSize(40, 40)
            buttonsLayout.addWidget(self.buttons[btnText], pos[0], pos[1])
        # Add buttonsLayout to the general layout
        self.generalLayout.addLayout(buttonsLayout)
```

Figure 2.10 Création des boutons du pavé numérique

Comme vous pouvez la remarquer sur le code présenter ci-dessus, nous avons utilisé un dictionnaire pour conserver le texte et la position de chaque bouton sur la grille. Nous avons utilisé également **QGridLayout** pour organiser les boutons sur la fenêtre de la calculatrice.

Petite explication du fonctionnement du dictionnaire :

Nous avons d'abord créé un dictionnaire vide **self.buttons** pour contenir les boutons du clavier. Ensuite, Nous avons créez un dictionnaire temporaire pour stocker leurs étiquettes et leurs positions relatives sur la disposition de la grille (buttonsLayout).

A l'intérieur de la boucle for, nous avons créé les boutons et les ajouter à la fois à **self.buttons** et **buttonsLayout**. Chaque bouton a une taille fixe de 40x40 pixels, que nous avons défini avec **.setFixedSize(40,40)**.

Résultat de la deuxième fenêtre :

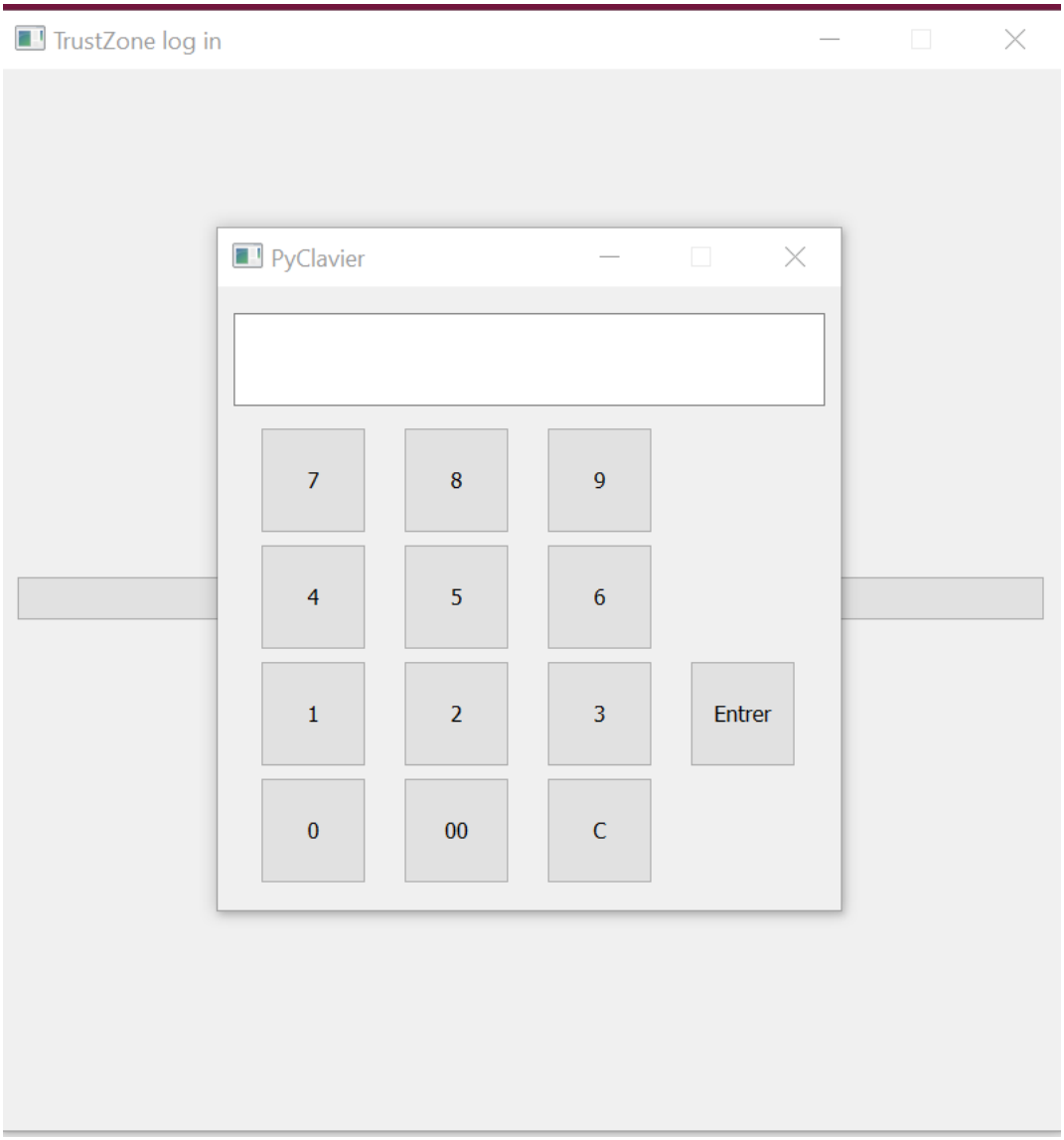

Figure 2.11 Affichage de la deuxième fenêtre

2.2.2.4 Mise à jour des informations affichées à l'écran

L'interface graphique contient un écran rectangulaire qui affiche les données saisies et le clavier. Pour que les informations affichées à l'écran soient identiques à celles que l'utilisateur va saisir, nous avons ajouté 3 méthodes supplémentaires à la classe PyClavierUi.

- **.setDisplayText ()** pour définir et mettre à jour le texte de l'affichage
- **.displayText ()** pour obtenir le texte de l'affichage actuel
- **.clearDisplay ()** pour effacer le texte de l'affichage

```
    # Snip
def setDisplayText(self, text):
    """Set display's text."""
    self.display.setText(text)
    self.display.setFocus()

def displayText(self):
    """Get display's text."""
    return self.display.text()

def clearDisplay(self):
    """Clear the display."""
    self.setDisplayText('')
```

Figure 2.12 Fonction de mise à jour des informations saisis

Voici ce que chaque fonction fait :

- **.setDisplayTexte ()** permet **.setTexte ()** de définir et de mettre à jour le texte de l'affichage et **.setFocus ()** de définir la focalisation du curseur (ou l'appuie sur l'écran digitale) sur l'affichage .
- **.displatText ()** est une methode getter qui renvoie le texte actuel de l'affichage. Lorsque l'utilisateur clique sur le signe Entrer, le programme utilisera la valeur de retour de **.displayText ()** comme un système de transmission que nous allons détaille dans la suite de du codage.
- **.clearDisplay ()** définit le texte de l'affichage sur une chaine vide (' ') afin que l'utilisateur puisse introduire une nouvelle composition de chiffre.

2.2.2.5 Création du contrôleur

Pour le moment nous avons terminé l'interface graphique du clavier. Ce pendant si nous essayons de composer notre code, nous remarquons que le clavier ne fait rien pour l'instant. C'est parce que nous n'avons pas implémenté le modèle ou le contrôleur.

Pour cela nous avons ajouté une classe de contrôleur de base pour commencer à donner une vie à notre clavier.

Le but de cette classe de contrôleur est de connecter la vue au modèle. Donc nous utilisant cette classe pour que le clavier effectue des actions en réponse aux évènements de l'utilisateur.

Pour commencer nous avons fait l'importation suivante :

```
from functools import partial
```

Figure 2.13 Importation de partial

Partiel () va permettre la connexion des signaux avec des méthodes qui doivent prendre des arguments supplémentaires.

Notre classe de contrôleur va effectuer trois taches principales :

1. Accéder à l'interface publique de l'interface graphique.
2. Gérer la transmission des données de l'interface vers la carte STM32.
3. Gérer la connexion des boutons, et la création des expressions.
4. Connecter les signaux cliqués sur le bouton avec les emplacements approprie.

Nous avons commencé par donner à **PyClavierCtrl** une instance de la vue **PyClavierUi**, pour obtenir un accès complet à l'interface publique de la vue, comme suit :

```
from functools import partial
import serial
#############################################################################################
# Create a Controller class to connect the GUI and the model
class PyClavierCtrl:
    """PyCalc Controller class."""
    def __init__(self, view):
        """Controller initializer."""
        #model=self._evaluate
        self._view = view
        # Connect signals and slots
        self._connectSignals()
```

Figure 2.14 initialisation du controleur

Ensuite, nous avons créé **._buildExpression ()** pour gérer la création des expression, cette méthode met également à jour l'affichage du clavier en repose à l'entrée de l'utilisateur. En fin, nous avons utilisé **._connectSignals ()** pour connecter les boutons imprimables avec **._buildExpression ()**, ce qui permet aux utilisateurs de taper leurs code en cliquant sur les boutons du clavier.

Dans la dernière ligne, nous avons connecter le bouton d'effacement (C) à **._view.clearDisplay ()**, cette méthode effacera le texte à l'écran. Le code suivant montrera le développement de tout ce qui précède comme explication :

```
    def _buildExpression(self, sub_exp):
        """Build expression."""
        expression = self._view.displayText() + sub_exp
        self._view.setDisplayText(expression)

    def _connectSignals(self):
        """Connect signals and slots."""
        for btnText, btn in self._view.buttons.items():
            if btnText not in {'Entrer', 'C'}:
                btn.clicked.connect(partial(self._buildExpression, btnText))

        self._view.buttons['Entrer'].clicked.connect(self._Transmission)
        self._view.buttons['C'].clicked.connect(self._view.clearDisplay)
```

Figure 2.15 Connection de chaque bouton avec la fonction qui lui correspond

Résultat de la mise à jour des informations affichés et du contrôleur

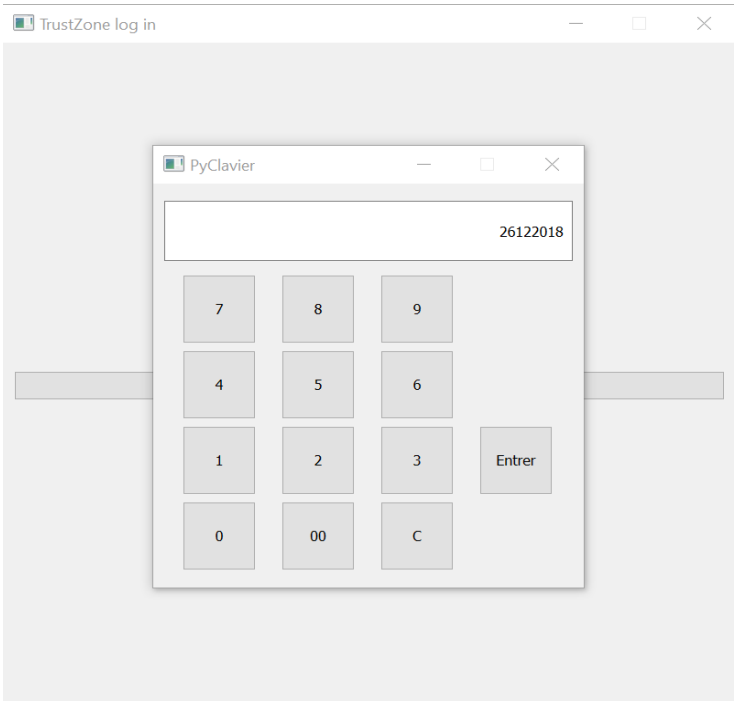

Figure 2.16 Affichage du pavé numériques fonctionnel

2.2.2.6 Transmission série des données entre l'interface et la carteSTM32

Le but de cette interface est d'établir un lien direct entre l'utilisateur et la partie Software du système. Mais pour le moment le code secret saisi est toujours pas transmis à la carte STM32 pour pouvoir commencer le processus de la technologie TrustZone. Pour cela une communication série est établi entre l'interface graphique et la carte STM32 en utilisant la bibliothèque **pySerial** du langage de programmation Python une fois l'utilisateur clique sur le bouton « Entrer » de l'interface.

Alors nous commençons par se connecter avec le port de la carte STM32, une fois le port est bien ouvert nous recevons un message qui dit que le port est ouvert puis nous récupérons ce qui est écris sur le display avec la fonction **displayText ().** Une fois arrivé à la ligne **se.write ()** la data sera envoyée et nous aurons un message qui informe l'utilisateur que le code a été envoyer avec succès. En fin dès que la carte est retirée le **ser.close ()** arrête la communication, voir figure ci-dessous

```
    def _Transmission(self):
        """Evaluate expressions."""
    #configure the serial connections (the parameters differs on the device you are connecting to)
        ser=serial.Serial("com5", 115200, timeout=10)
        if ser.isOpen ():

            print ("Serial port is open")

            data=self._view.displayText()#data sent


#             ser.write(str.encode(data))
            ser.write(str(data).encode("UTF-8"))
            ser.write(str('\n').encode("UTF-8"))

            print ("you send data:", data)

            ser.close ()
```

Figure 2.17 Fonction qui fait la transmission du code secret saisi par l'utilisateur

2.2.2.6.1 Configuration nécessaire dans le STM32 pour recevoir les données :

Pour récupérer le code secret envoyer par la fonction définie dans l'interface graphique (Voir la figure ci-dessus) nous avons utilisé le USART3 comme périphérique de communication série entre la carte STM32L5 et l'interface graphique (Pour plus de taille voire le chapitre 1). En ce qui concerne la configuration logicielle, nous avons configurer la réception et le stockage du code secret de la façon suivant :

```
92 int main(void)
93 {
94   /* USER CODE BEGIN 1 */
95       int num;
```

Figure 2.18 Initialisation de la variable ou nous allons stocker le code secret

```
94   setvbuf(stdin, NULL, _IONBF, 0);
95   printf("Program has started!\r\n");
96   /* USER CODE END 2 */
```

Figure 2.19 Déclaration de setvbuf () pour désactiver la mise en mémoire tampon interne pour le flux d'entrée, avant tout appel à scanf ()

```
136  /* USER CODE BEGIN 2 */
137
138  PRINTF_NSE("Please Enter the Access Code: \r\n");
139    scanf("%d", &num);
```

Figure 2.20 Déclaration de scanf ()

```
184
185 /* USER CODE BEGIN 4 */
186
187 int __io_putchar(int ch)
188 {
189     if(HAL_UART_Transmit(&huart3, (uint8_t *)&ch, 1, 10) != HAL_OK)
190         return -1;
191     return ch;
192 }
193
194 int __io_getchar(void)
195 {
196     char data[4];
197     uint8_t ch, len = 1;
198
199
200     while(HAL_UART_Receive(&huart3, &ch, 1, 10) != HAL_OK)
201     {
202     }
203
204     memset(data, 0x00, 4);
205     switch(ch)
206     {
207         case '\r':
208         case '\n':
209             len = 2;
210             sprintf(data, "\r\n");
211             break;
212
213         case '\b':
214         case 0x7F:
215             len = 3;
216             sprintf(data, "\b \b");
217             break;
218
219         default:
220             data[0] = ch;
221             break;
222     }
223     HAL_UART_Transmit(&huart3, (uint8_t *)data, len, 10);
224     return ch;
225 }
```

Figure 2.21 Ajout du code de redirection de scanf ()

Chapitre 3

Exécutable de l'interface graphique

3.1 Création d'un exécutable de l'interface graphique

Pour faciliter l'exécution du programme de l'interface nous avons créé un exécutable, c'est à dire une application à partir du code python de l'interface. Pour cela nous avons utilisé **Anaconda Prompt (Anaconda3).**

Figure Anaconda Prompt

Nous avons choisi d'utiliser le script python **PyInstaller** qui a l'avantage de bien fonctionner sur tous les version python. Ce script nous a permis de finaliser notre interface en compilant un exécutable, ce qui va nous permettre d'utiliser notre interface sur n'importe quel ordinateur, même si Python n'est pas installé (sans problème de librairie non installée ni conflit de version).

Pour commencer nous avons installé Py installer dans un nouvel environnement conda que nous avons créé, en utilisant la commande suite :

```
(base) C:\Users\ines>conda create --name pyinstaller-demo
```

Figure 3.1 Creation du fichier ou nous allons installer pyinstaller

Puis nous avons activer cet environnement, et afficher les autres environnements qui existe dans conda, afin de voir que nous somme dans l'environement pyinstaller, comme le montre la figure ci-dessous :

Anaconda Prompt (anaconda3) - conda install pyinstaller

```
(base) C:\Users\ines>conda activate pyinstaller-demo

(pyinstaller-demo) C:\Users\ines>conda env list
# conda environments:
#
base                  C:\Users\ines\anaconda3
pyinstaller-demo      *  C:\Users\ines\anaconda3\envs\pyinstaller-demo


(pyinstaller-demo) C:\Users\ines>conda install pyinstaller
Collecting package metadata (current_repodata.json): done
Solving environment: done

# All requested packages already installed.
```

Figure 3.2 Installation de pyinstaller

A ce niveau nous pouvons installer pyinstaller, en utilisant la commande suivante :

```
(pyinstaller-demo) C:\Users\ines>conda install pyinstaller
```

Figure 3.3 Installation de pyinstaller

Une fois nous executons cette commande nous aurons une demande pour installer d'autres bibliothéque dependantes dont nous avons besoin, donc il faut accepter cette installation, voir la figure ci dessous:

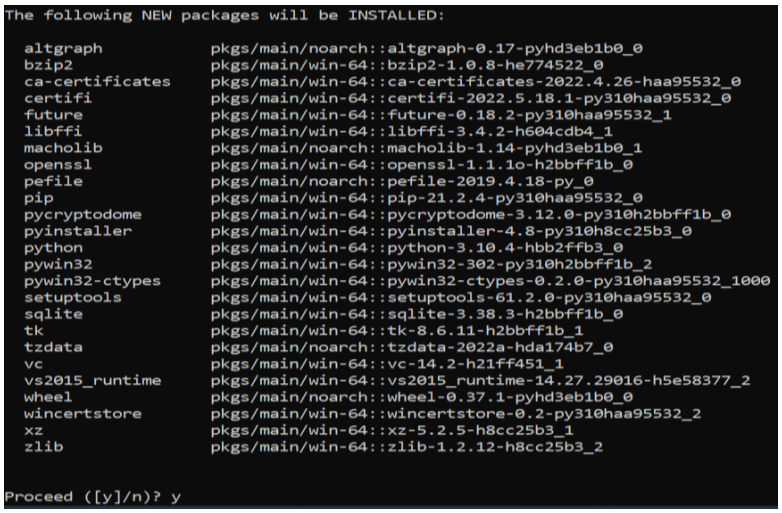

Figure 3.4 Acceptation de la demande d'instalation des bibliotheque dependantes

Pour verifier que la bibliotheque pyinstaller est installer avec succés, il faut taper la commande **conda list** :

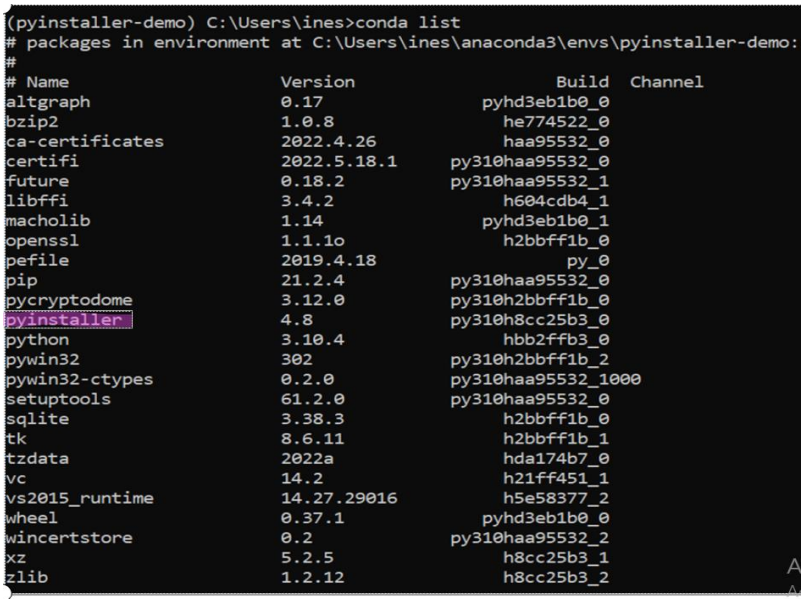

Figure 3.5 Confirmation del'installation de pyinstaller

L'étape suivante est de trouver l'emplacement de python executable, car c'est la ou python vit réellement pour cet environnement. Pour cela il suffit de tapé les commande suivantes :

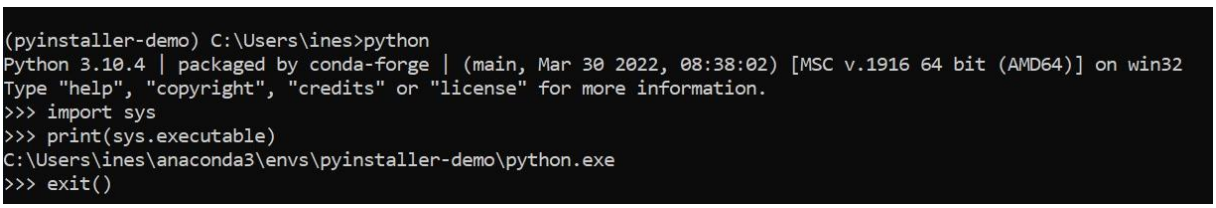

Figure 3.6 Emplacemant de l'executable Python

L'executable python est dans C:\Users\ines\anaconda3\envs\pyinstaller-demo, ce chemain va nous servira pour configurer l'interpréteur dans l'editeur de code que nous allons utiliser pour crée notre interface (Exemple : PyCharm), comme le montre les figure ci-dessous :

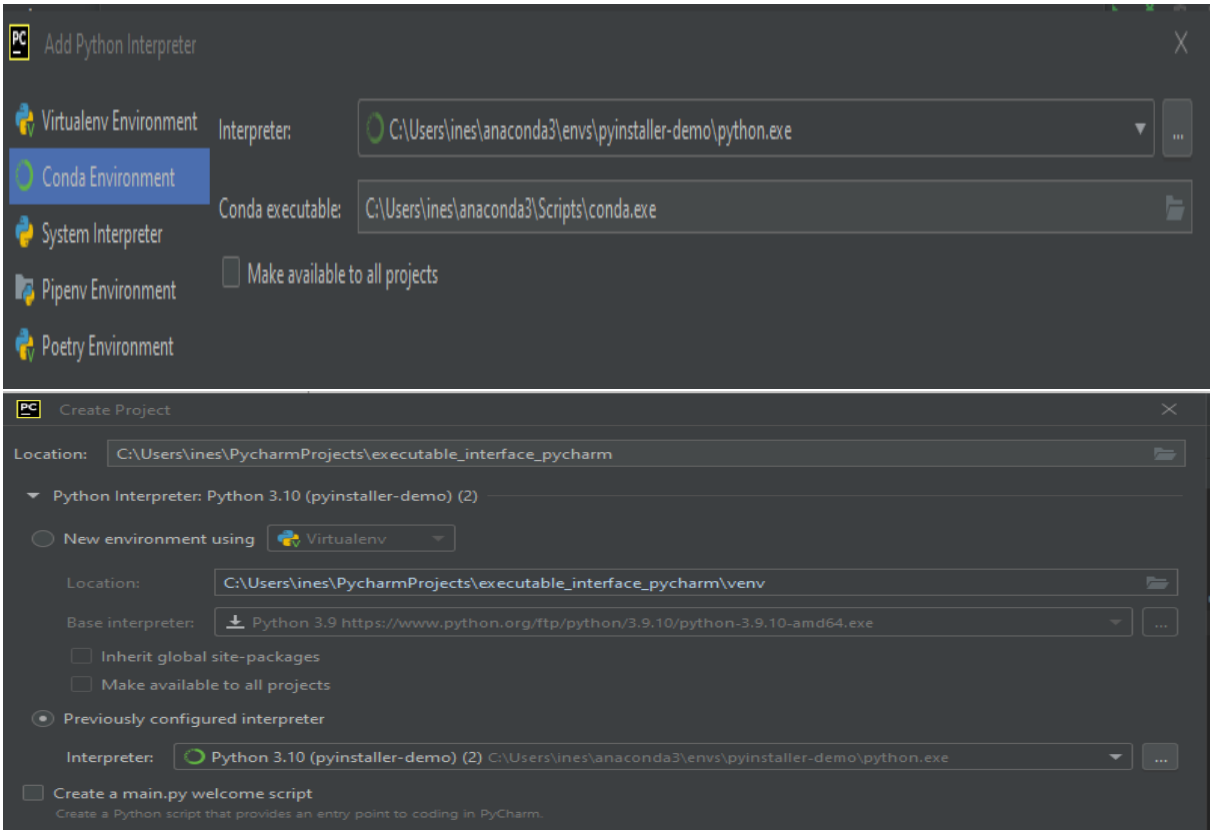

Figure 3.7 Configuration de l'interpreteur dans l'editeur de code PyCharm

A ce niveau il faut confirmer que tout fonctionne parfaitement pour pouvoir continuer, pour cela il faut utiliser la commande pyinstaller et afficher le message suivant :

```
(pyinstaller-demo) C:\Users\ines>pyinstaller
usage: pyinstaller [-h] [-v] [-D] [-F] [--specpath DIR] [-n NAME]
                   [--add-data <SRC;DEST or SRC:DEST>]
                   [--add-binary <SRC;DEST or SRC:DEST>] [-p DIR]
                   [--hidden-import MODULENAME] [--collect-submodules MODULENAME]
                   [--collect-data MODULENAME] [--collect-binaries MODULENAME]
                   [--collect-all MODULENAME] [--copy-metadata PACKAGENAME]
                   [--recursive-copy-metadata PACKAGENAME]
                   [--additional-hooks-dir HOOKSPATH]
                   [--runtime-hook RUNTIME_HOOKS] [--exclude-module EXCLUDES]
                   [--key KEY] [--splash IMAGE_FILE]
                   [-d {all,imports,bootloader,noarchive}]
                   [--python-option PYTHON_OPTION] [-s] [--noupx]
                   [--upx-exclude FILE] [-c] [-w]
                   [-i <FILE.ico or FILE.exe,ID or FILE.icns or "NONE">]
                   [--disable-windowed-traceback] [--version-file FILE]
                   [-m <FILE or XML>] [--no-embed-manifest] [-r RESOURCE]
                   [--uac-admin] [--uac-uiaccess] [--win-private-assemblies]
                   [--win-no-prefer-redirects]
                   [--osx-bundle-identifier BUNDLE_IDENTIFIER]
                   [--target-architecture ARCH] [--codesign-identity IDENTITY]
                   [--osx-entitlements-file FILENAME] [--runtime-tmpdir PATH]
                   [--bootloader-ignore-signals] [--distpath DIR]
                   [--workpath WORKPATH] [-y] [--upx-dir UPX_DIR] [-a] [--clean]
                   [--log-level LEVEL]
                   scriptname [scriptname ...]
pyinstaller: error: the following arguments are required: scriptname

(pyinstaller-demo) C:\Users\ines>
```

Activer Windows

Si nous avons pas eu ce message, a ce moment il faut revoir tout les etapes precedentes.

Pour executer la derniere commande, il faut changer le repertoir en allant au chemain du script de l'interface :

```
(pyinstaller-demo) C:\Users\ines>cd C:\Users\ines\PycharmProjects\interface_pyserial_executable_pycharm
```

Figure 3.8 Changement du repertoir vers le script de l'interface graphique

Ensuite nous exécutons la commande suivante pour avoir notre exécutable. Nous avons choisi une icône qui convient avec notre sujet :

```
(pyinstaller-demo) C:\Users\ines\PycharmProjects\interface_pyserial_executable_pycharm>pyinstaller "C:\Users\ines\PycharmProjects\interface_pyserial_executable_pycharm\exec
utable_interface_pycharm.py" --onefile --icon="C:\Users\ines\PycharmProjects\interface_pyserial_executable_pycharm\40_104848.ico"
```

Figure 3.9 La commande qui permet de crée un exécutable de l'interface avec une Icon spécifique

Notre exécutable sera alors placé dans le dossier **Dist** à l'emplacement de notre script de l'interface.

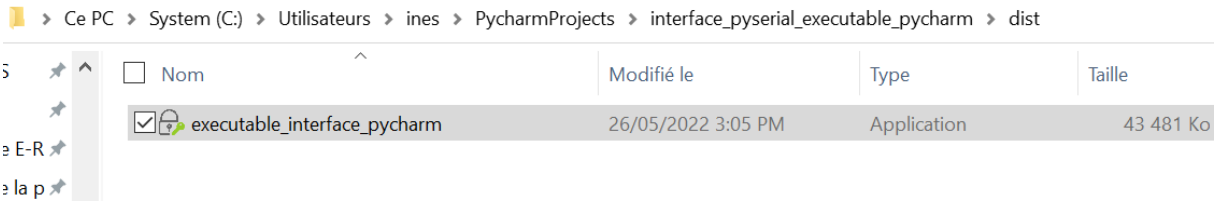

Figure 3.10 Positionnement de l'exécutable dans les fichiers

Résultat finale de l'exécutable :

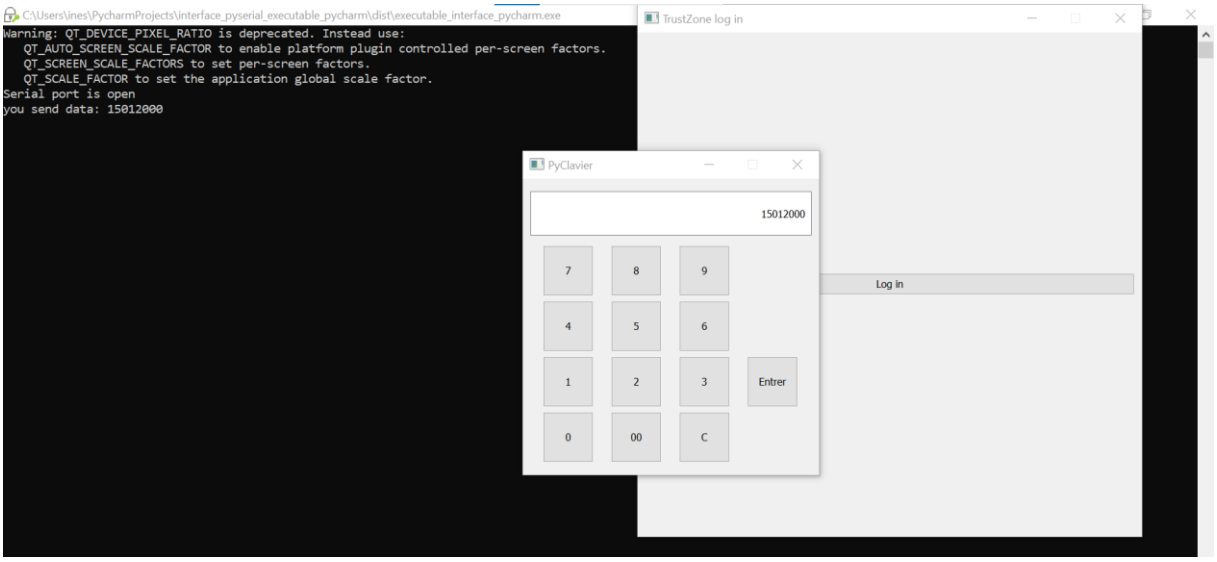

Figure 3.11 Résultat de l'exécutable

Conclusion

Pour conclure, durant ce stage j'ai eu l'occasion de réaliser plusieurs missions pour la mise en œuvre d'une application basée sur la technologie TrustZone. En résumé nous avons réalisé une interface graphique et son exécutable (application) qui donne la main à l'utiliser pour écrire son code secret, puis le transmettre à la carte STM32 en utilisant le port série, qui va à son toure prendre le code secret et le comparer avec le code secret de sa partie sécurisée, et décide si la porte s'ouvre ou pas.