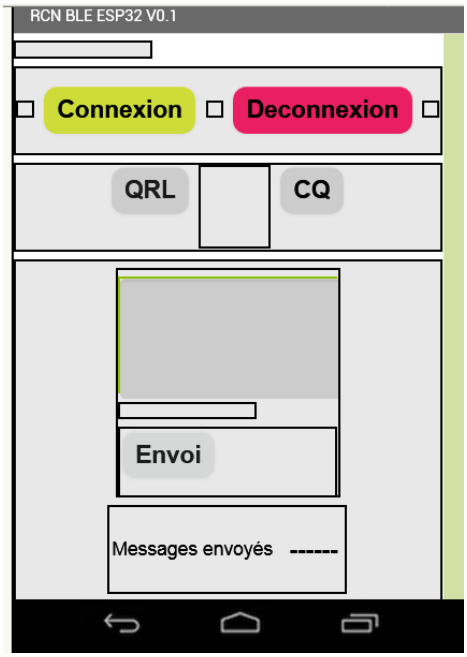
Super Arduino et CW

La réalisation d'un générateur CW va servir à décrire, une réalisation type, avec une carte Super Arduino de base avec processeur ESP32, et la mise en œuvre d'outils de développement du logiciel écrit en langage Arduino, et un « Buzzer » pour émettre les « dih » (.) et les « dah » (-).

Ce générateur est en communication par Bluetooth avec un smartphone servant à rédiger le texte envoyé à la carte Super Arduino, et l'émission avec un buzzer des « dih » et « dah » du « Morse ».

Le logiciel du Smartphone sous Android est réalisé avec AppInventor2 du MIT, environnement de développement totalement graphique, permettant la réalisation de logiciel Android très simplement, sans avoir à connaitre Android.



Le bouton **Connexion** permet au Smartphone de se connecter en Bluetooth LE à la carte SuperArduino émettant le Morse avec son Buzzer.

Lorsque le bouton **CQ** est appuyé, il passe en vert et envoie la séquence CQ CQ DE F5KAR F5KAR+K.
Un nouvel appui arrête l'émission en Morse de la séquence et refait passer le bouton en gris.

De même le bouton **QRL** pour la séquence QRL ? QRL ?

Le bouton **Envoi** envoie le texte qui aura préalablement été saisi par l'opérateur dans la zone de texte au-dessus de ce bouton

Enfin le nombre de messages envoyés en Morse est fourni par la carte SuperArduino en Bluetooth et affiché sur le Smartphone

Contenu

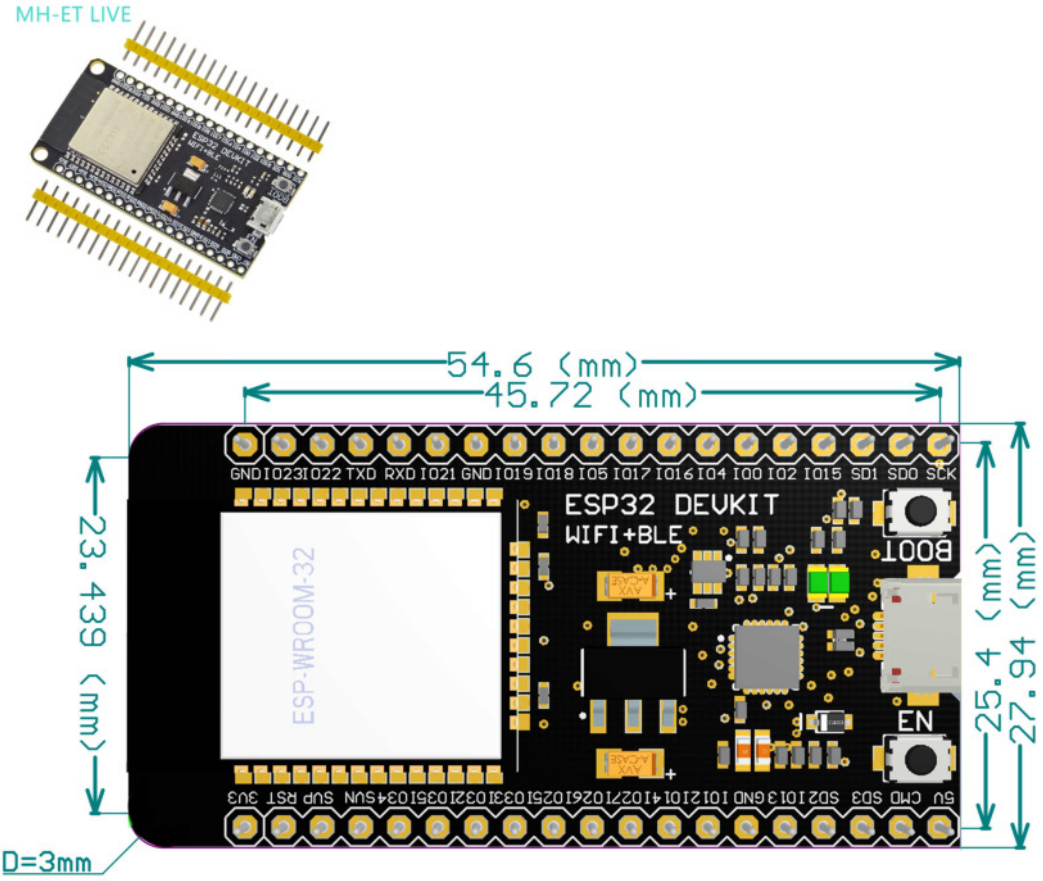
Matériel....2
Logiciel SuperArduino....3
D'une lettre à son code Morse....3
Communication Bluetooth....7
Initialisation du traitement de la carte SuperArduino....8
Traitement de la carte SuperArduino....9
Logiciel Android sur Smartphone....11
AppInventor2....11
Conception interface utilisateur....11
Traitement réalisé....13
ESP32-CW-VSF Version avec réglages....17
Dépôt logiciels....18
ESP32-CW....18
ESP32-CW-VSF....18

Matériel

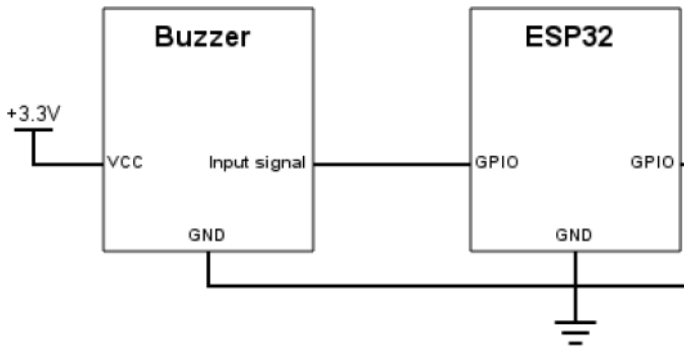
Ce processeur 32 bits double cœur (*à comparer au processeur 8 bits ATMEGA 328 d'une carte Arduino Nano 3.0*) inclut Wifi 802.11b/g/n et Bluetooth LE (Low Energy).

La carte ESP32 DEVKIT de MH-ET-LIVE possède 520 Ko de mémoire vive et 4 Mo de mémoire flash contenant le logiciel téléchargé, une interface USB, les antennes Wifi et Bluetooth, et 2 boutons poussoirs (Reset et Chargement logiciel), pour un coût de 6 €

Le développement du logiciel de cette carte s'effectue avec l'IDE Arduino comme pour les cartes Arduino standards (Nano, Uno,…)



Un « buzzer » passif est connecté à une broche GPIO du processeur ESP32.

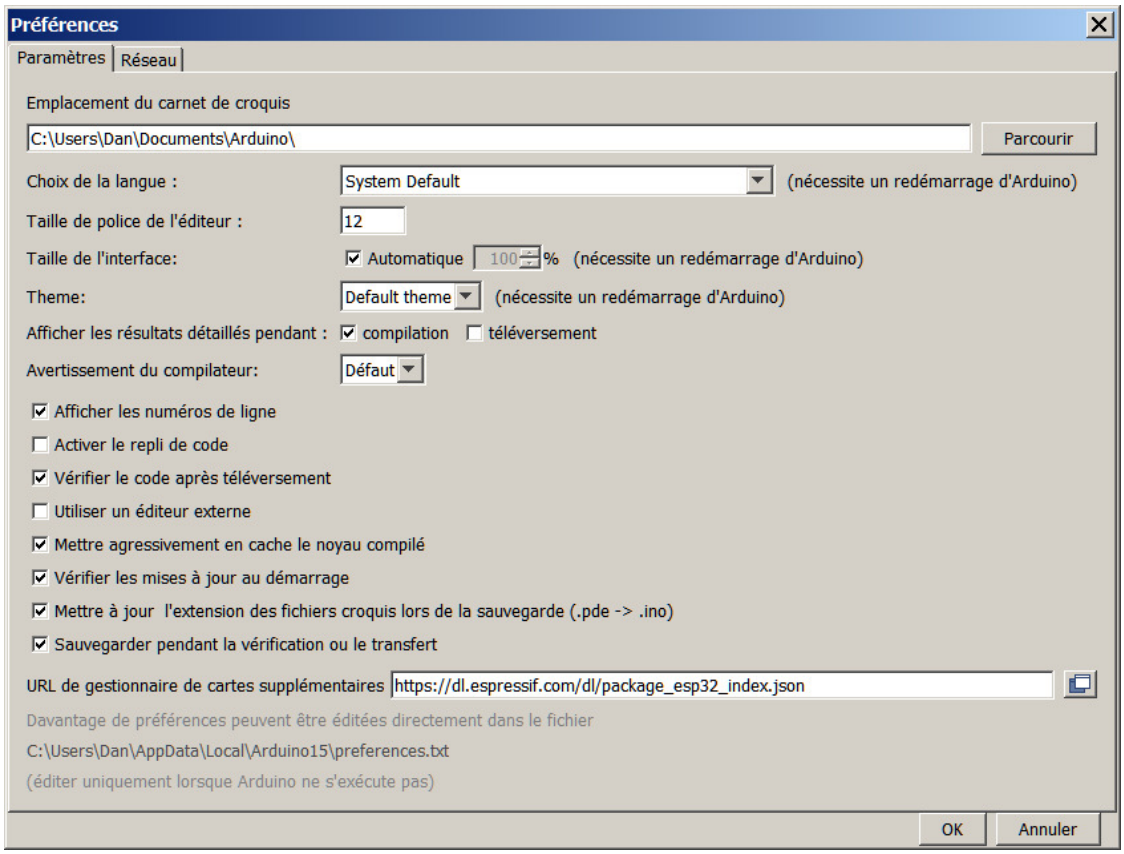


Logiciel SuperArduino

Il faut d'abord télécharger l'environnement habituel de développement intégré Arduino (Integrated Developement Environment) sur Internet https://www.arduino.cc/en/Main/Software.

Après lancement du logiciel Arduino, l'installation des éléments supplémentaires pour l'ESP32 se fait très simplement en remplissant le champ « URL de gestionnaire de cartes supplémentaires avec `https://dl.espressif.com/dl/package_esp32_index.json`

Plus de détail https://github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/boards_manager.md



D'une lettre à son code Morse

Examinons d'abord la 1ére partie du logiciel qui va faire correspondre à chaque lettre son code Morse.

Le tableau MorseTable est utilisé pour transformer les caractères reçus du Smartphone en code Morse.

Les caractères échangés entre le Smartphone et la carte ESP32 sont codés en ASCII, chaque lettre étant représentée par un nombre entre 0 et 127.

Le nombre est exprimé en base hexadécimale (16), c'est-à-dire que le 1er chiffre est à multiplier par 16 et non par 10 comme pour les nombres utilisés habituellement.

Ainsi le code ASCII du caractère A est 0x41 (le 0x indique que nous utilisons une base hexadécimale) soit 4*16+1 = 65 se trouve dans le tableau ci-dessous à la ligne 5éme ligne (marquée 40) et à la 2éme colonne (marqué 1) (en informatique on compte souvent à partir de 0 et non pas de 1)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Les caractères avec un fond rose sont appelés caractères non imprimables ; En général ils servaient dans les années 1960-1970 à contrôler la transmission entre l'émetteur et le récepteur. Nous ne les utiliserons pas dans le logiciel

Le code ASCII de chaque caractère est utilisé comme index dans le tableau.
Ainsi le code ASCII du caractère A

```
const char* MorseTable[] = {
  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, //  de 0x0 à 0x07
  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, //  de 0x08 à 0x0F
  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, //  de 0x10 à 0x17
  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, //  de 0x18 à 0x1F
  // space, !, ", #, $, %, &, '
  "s", "-.-.--", ".-..-.", NULL, NULL, NULL, NULL, ".----.",
  // ( ) * + , - . /
  "-.--.", "-.--.-", NULL, ".-.-.", "--..--", "-....-", ".-.-.-", "-..-.",
  // 0 1 2 3 4 5 6 7
  "-----", ".----", "..---", "...--", "....-", ".....", "-....", "--...",
  // 8 9 : ; < = > ?
  "---..", "----.", "---...", "-.-.-.", NULL, "-...-", NULL, "..--..",
  // @ A B C D E F G
  ".--.-.", ".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
  // H I J K L M N O
  "....", "..", ".---", "-.-", ".-..", "--", "-.", "---",
  // P Q R S T U V W
  ".--.", "--.-", ".-.", "...", "-", "..-", "...-", ".--",
  // X Y Z [ \ ] ^ _
  "-..-", "-.--", "--..", NULL, NULL, NULL, NULL, "..--.-",
  // ' a b c d e f g
  NULL, ".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
  // h i j k l m n o
  "....", "..", ".---", "-.-", ".-..", "--", "-.", "---",
  // p q r s t u v w
  ".--.", "--.-", ".-.", "...", "-", "..-", "...-", ".--",
  // x y z { | } ~ DEL
```

```
  "-..-", "-.--", "--..", NULL, NULL, NULL, NULL, NULL, // de 0x78 à 0x7F
};
```

Les caractères envoyés en Bluetooth depuis le Smartphone arrivent dans l'ESP32 dans l'objet « rxValue », la « chaine de caractères » étant contenue dans rxValue.c_str()

```
    strcpy(buf ,  rxValue.c_str());   // une copie des caractères reçus est faite dans buf
    buflen = strlen(buf);             // calcul de la longueur de la chaine de caractères

    //

    for ( index = 0 ; index < buflen ; index++ )    // Parcours caractère par caractère, de la chaine
                                                       de caractère reçue
    {
     ch = buf[index];                  // la variable ch contient le caractère à émettre en code Morse

     if (ch == 0x85) ch = 'a';                      //  On remplace le caractère à par le caractère a
     if ((ch == 0x82 || ch == 0x8A )) ch = 'e';    //  On remplace les caractères é et è par par le
                                                       caractère e
     if (ch >= 0x7F) ch = '?';                   // Les caractères de code ASCII > ou = à 127 sont
                                                       remplacés par  ?
     EmettreTiretPoint(TableMorse[ch]);        // Utilisation du tableau MorseTable, le code
                                                       ASCII du caractère étant utilisé comme index
     delay(Point_Duree*3);                     // Règle 3 du Morse
    }
    delay(Point_Duree * 7);                  // fin du dernier mot reçu Règle 4
```

Règles du Morse

1. Un Tiret est égal à trois Points.
2. L'espacement entre deux éléments d'une même lettre est égal à un Point.
3. L'espacement entre deux lettres est égal à trois Points.
4. L'espacement entre deux mots est égal à sept Points.

Les procédures Point et Tiret vont générer respectivement le dih et le dah sur le Buzzer en le faisant vibrer au rythme de 500 fois par seconde (Buzzer ON pendant 1 ms et Buzzer OFF pendant 1ms ensuite, répétés la durée du Point ou du Tiret)

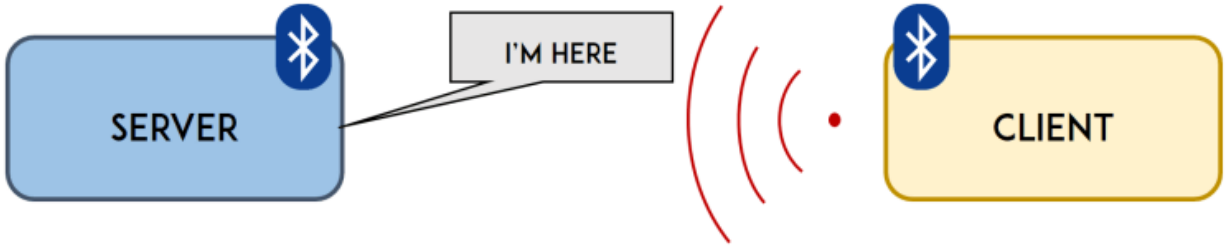
```
int Point_Duree = 30 ;    // Le point dure environ  60 ms (30 fois 1 ms +30 fois 1 ms)
int Tiret_Duree = Point_Duree * 3;  //  Règle 1

void Point()   // génére le son point sur le buzzer
{
 digitalWrite(LED, ALLUME);   // La LED est allumé le temps de l'émission du Point

 for (k = 0; k < Point_Duree; k++) // Durée de l'emission d'un point
 {
  digitalWrite(BUZZER, HIGH); // Emission d'un son à 500 Hz avec 1 ms haut pour la sortie
Buzzer
  delay(1);//delay 1 ms
  digitalWrite(BUZZER, LOW); //   et 1 ms bas pour la sortie Buzzer
```

```
   delay(1);// delay 1 ms
  }
  digitalWrite(LED, ETEINT);   // La LED est eteinte à la fin de l'emission

}

void Tiret()
{
  digitalWrite(LED, ALLUME);

  for (k = 0; k < Tiret_Duree; k++) // Durée de l'emission d'un tiret
  {
    digitalWrite(BUZZER, HIGH); // Emission d'un son à 500 Hz avec 1 ms haut pour la sortie
Buzzer
    delay(1);//delay 1 ms
    digitalWrite(BUZZER, LOW); // et 1 ms bas pour la sortie Buzzer
    delay(1);// delay 1 ms
  }
  digitalWrite(LED, ETEINT);

}
```

La procèdure **EmettreTiretPoint** émet les dih et les dah correspondant au caractère à émettre en utilisant l'élément du tableau MorseTable pointé (*morseCode)

```
void EmettreTiretPoint(const char * morseCode)
{
  int i = 0;
  while (morseCode[i] != 0)
  {
    if (morseCode[i] == '.') {
      Point();delay(Point_Duree);      // Règle 2
    } else if (morseCode[i] == '-') {
      Tiret();delay(Point_Duree);     // Règle 2
    }
    else if (morseCode[i] == 's') {   // espace entre 2 mots (caractère blanc), rien est émis par le
Buzzer
      digitalWrite(LED, ETEINT);     // La LED est éteinte
      delay(Point_Duree * 7);    // Règle 4
    }
    i++;
  }
}
```

Communication Bluetooth



Dans le cas des logiciels décrits dans cet article, la carte SuperArduino joue le rôle de serveur Bluetooth et le smartphone le rôle de client Bluetooth.

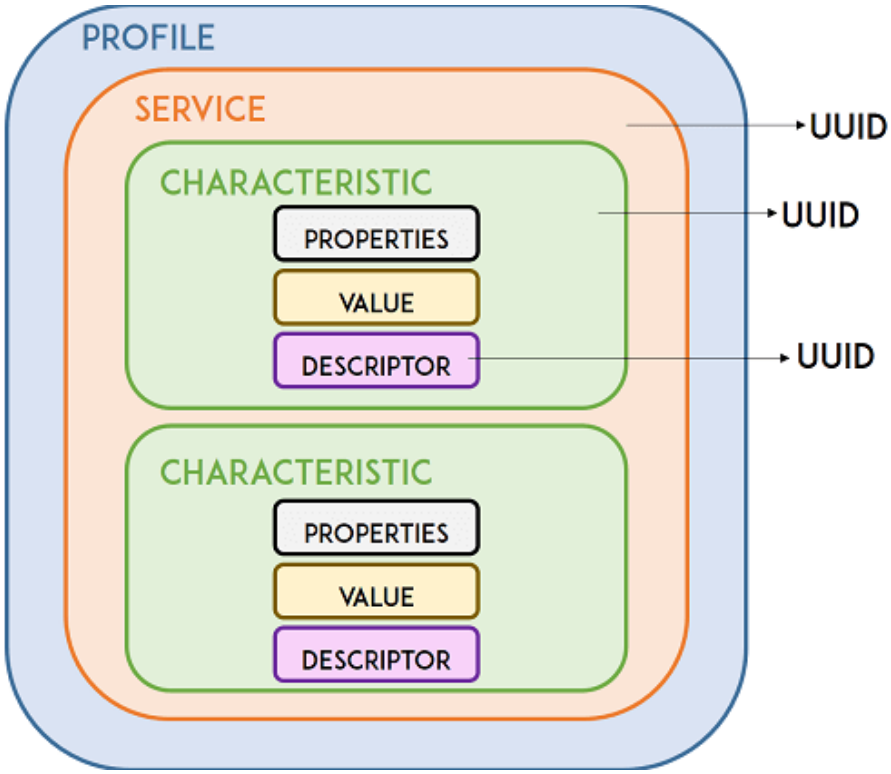
Le serveur annonce sa présence sur les « ondes » et peut ainsi être « trouvé » par les clients

Un client recherche les serveurs à portée et lorsqu'il trouve le serveur recherché, il établit une connexion et se met à l'écoute du serveur.

La communication entre 2 équipements Bluetooth utilise une structure de données hiérarchisée, appelée Profile composé d'un ou plusieurs services.

Chaque service contient généralement plusieurs « caractéristiques ».

Un service est simplement un ensemble de valeurs provenant par exemple de capteurs tel que Pression sanguine, Pouls, Poids, niveau de la batterie, etc. ….



Chaque service, chaque caractéristique et chaque descripteur ont un UUID (Universally Unique IDentifier, en Français identifiant universel unique), permettant à tous les équipements Bluetooth d'informer les autres équipements, des services et données associées échangées.

(Les UUID sont définis sur le site https://www.uuidgenerator.net)

La carte SuperArduino est utilisée comme serveur Bluetooth et est programmée avec un service UART (*similaire à cette bonne vieille liaison série COM1*) avec une caractéristique réception de caractères et une autre caractéristique transmission de caractères

```
#define SERVICE_UUID  "6E400001-B5A3-F393-E0A9-E50E24DCCA9E" // UART service
#define CHARACTERISTIC_UUID_RX "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"
```

2 classes de rappels (Callbacks en anglais) doivent être écrites pour le serveur, pour définir son comportement lors d'une connexion et lors d'une déconnexion d'un client.

Ici le comportement est très simple et consiste à positionner la variable « deviceConnected » à vrai ou faux

```
class ServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };

    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};
```

Pour chaque caractéristique, une classe de rappel doit aussi être écrite.
Ici la chaine de caractère reçue du Smartphone est stockée dans la variable rxValue
Si le 1er caractère reçu est 0, on décide que le SuperArduino doit cesser d'émettre du Morse (veille = VRAI)

```
class Callbacks: public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      rxValue = pCharacteristic->getValue();
        // Si le 1er caractère reçu est 0, le programme passe en veille
        if (rxValue[0] == char('0')) veille = VRAI ;
        else veille = FAUX;

      }
    }
};
```

Initialisation du traitement de la carte SuperArduino

Dans la partie setup() du programme Arduino, il est nécessaire de créer le périphérique Bluetooth que nous nommons « BLE ESP32 », ensuite de créer le serveur associé à ce périphérique en y associant sa classe de rappel.

```
// Creation du périphérique BLE
  BLEDevice::init("BLE ESP32"); // création et initialisation avec BLE ESP32 comme nom de périphérique

  // Crée le Serveur BLE
  BLEServer *pServer = BLEDevice::createServer();
```

```
pServer->setCallbacks(new ServerCallbacks());
```

Puis le service UART est créé en utilisant SERVICE_UUID, son UUID.
Sont également créés les caractéristiques de transmission de caractères (TX) et de réception (RX) avec sa classe de rappel à la réception de caractères

```
// Creation du Service BLE
BLEService *pService = pServer->createService(SERVICE_UUID);

// Creation de la caracteristique BLE en envoi (TX)
pCharacteristic = pService->createCharacteristic(
                    CHARACTERISTIC_UUID_TX,
                    BLECharacteristic::PROPERTY_NOTIFY
                  );


// Creation de la caracteristique BLE en récception (RX)
BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                       CHARACTERISTIC_UUID_RX,
                                       BLECharacteristic::PROPERTY_WRITE
                                     );

pCharacteristic->setCallbacks(new Callbacks());
```

Puis il faut démarrer le service

```
// Démarrer le service
pService->start();

// Commencer l'annonce du service
pServer->getAdvertising()->start();

Serial.println("V0.1 Attente de la connexion d'un client bluetooth");
```

La LED s'éteint lorsque le setup est terminé et que la carte SuperArduino est prête à communiquer avec un client Bluetooth

```
digitalWrite(LED, ETEINT);
```

<u>Traitement de la carte SuperArduino</u>

Lorsque la connexion avec le client a été réalisé et que le SuperArduino n'est pas en veille (veille = FAUX)

```
void loop() {
  if (deviceConnected) {
    if (veille == FAUX ) {
      // iteration...
      txValue += 1 ; // txValue représente le nombre de message Morse émis par le Buzzer

      // Conversion de l'entier txValue en chaine de caractère ASCII
      char txString[8];
```

```
itoa(txValue, txString, 6); //

pCharacteristic->setValue(txString);

pCharacteristic->notify(); // Envoie la valeur à l'application Smartphone.
```

Puis on continue par le traitement de la chaine de caractère rxValue décrite précédemment (EmettreTiretPoint(TableMorse[ch])

Logiciel Android sur Smartphone

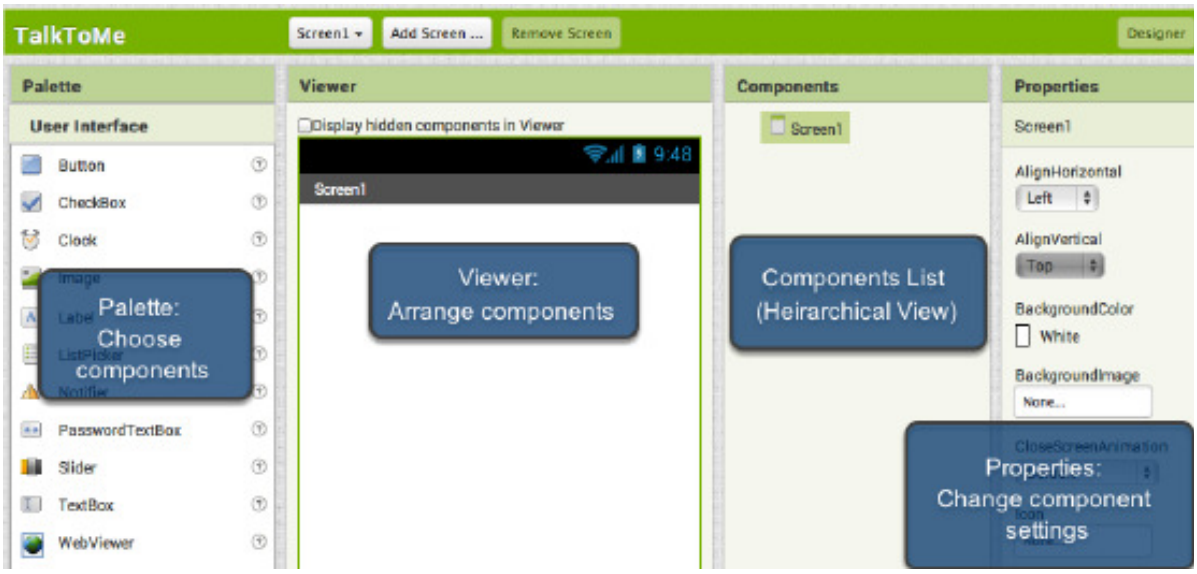
AppInventor2

Ce logiciel client Bluetooth est développé avec AppInventor2 disponible sur le site http://ai2.appinventor.mit.edu

Après avoir créé un compte utilisateur, un projet nouveau à définir s'affiche avec un écran Screen1 créé.

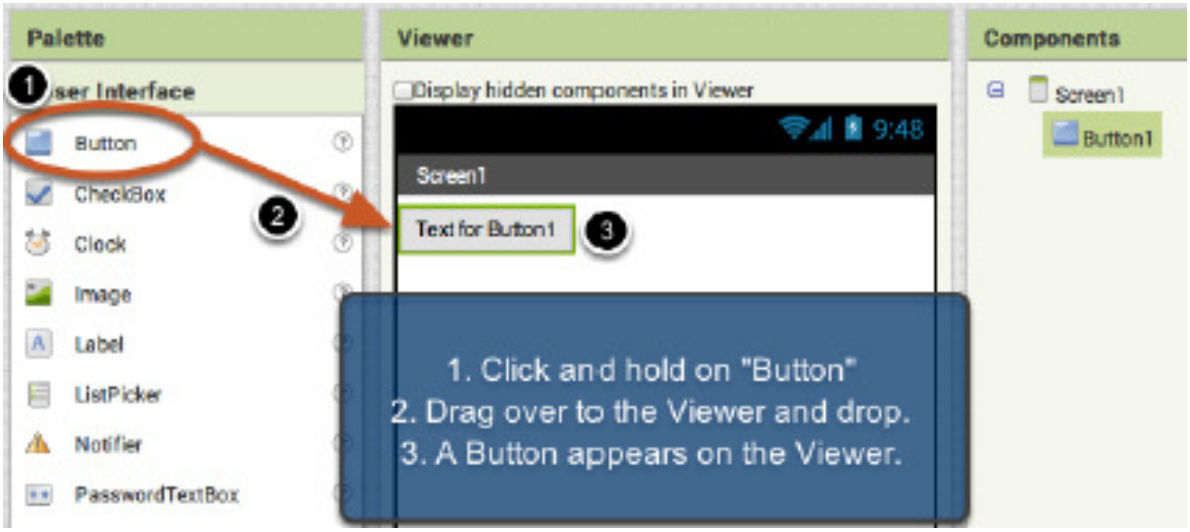
Conception interface utilisateur

A gauche la palette des composants graphiques disponibles, au milieu la vue écran de l'application et à droite la liste des composants et leurs propriétés.

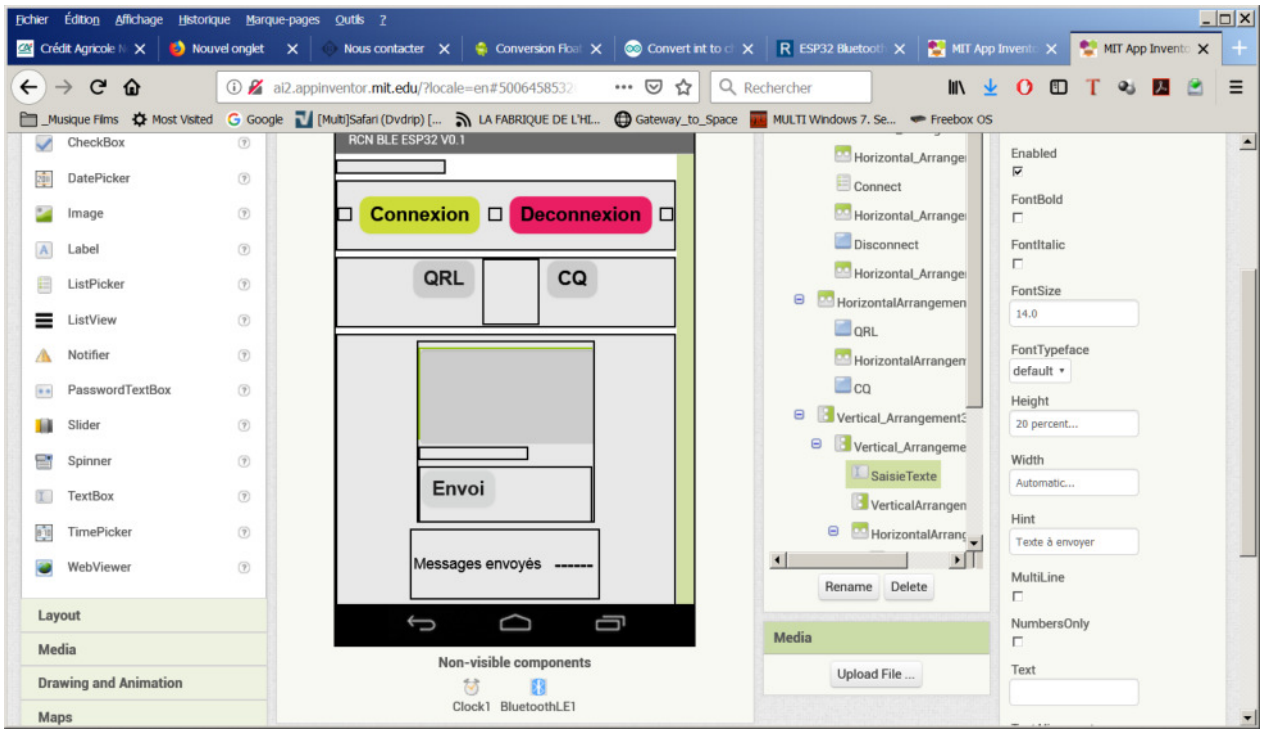


La création du logiciel s'effectue par « glisser -lâcher » du composant depuis la palette dans l'écran puis définition des propriétés du composant.

Dans l'exemple ci-dessous, un bouton est ajouté à l'écran



L’application sur le Smartphone présente l’écran suivant :



Il est composé de boutons (Connexion, Deconnexion, QRL, CQ, Envoi) d’une TextBox (*nommée SaisieTexte*) pour saisir le texte à envoyer par Bluetooth à la carte SuperArduino et 2 libellés pour afficher le nombre de messages envoyés en Morse.

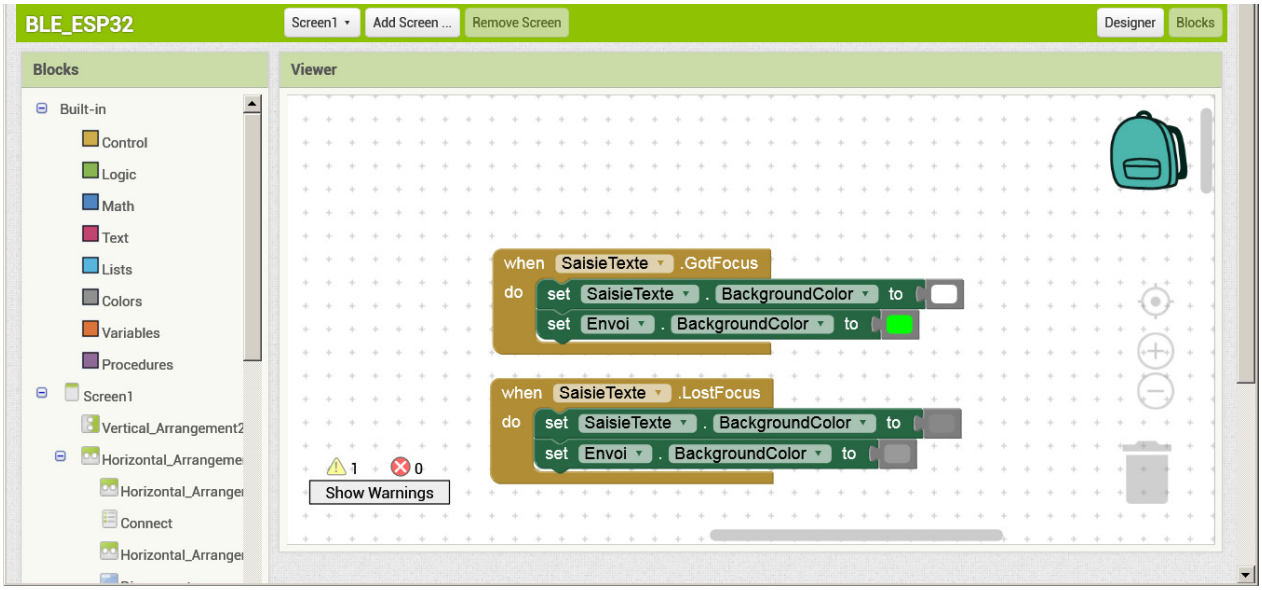
Les composants sont placés dans l’écran à l’aide de la souris.
Sont inclus 2 composants n’ayant pas d’interface graphique visible, une horloge et la communication Bluetooth LE (Low Energy).

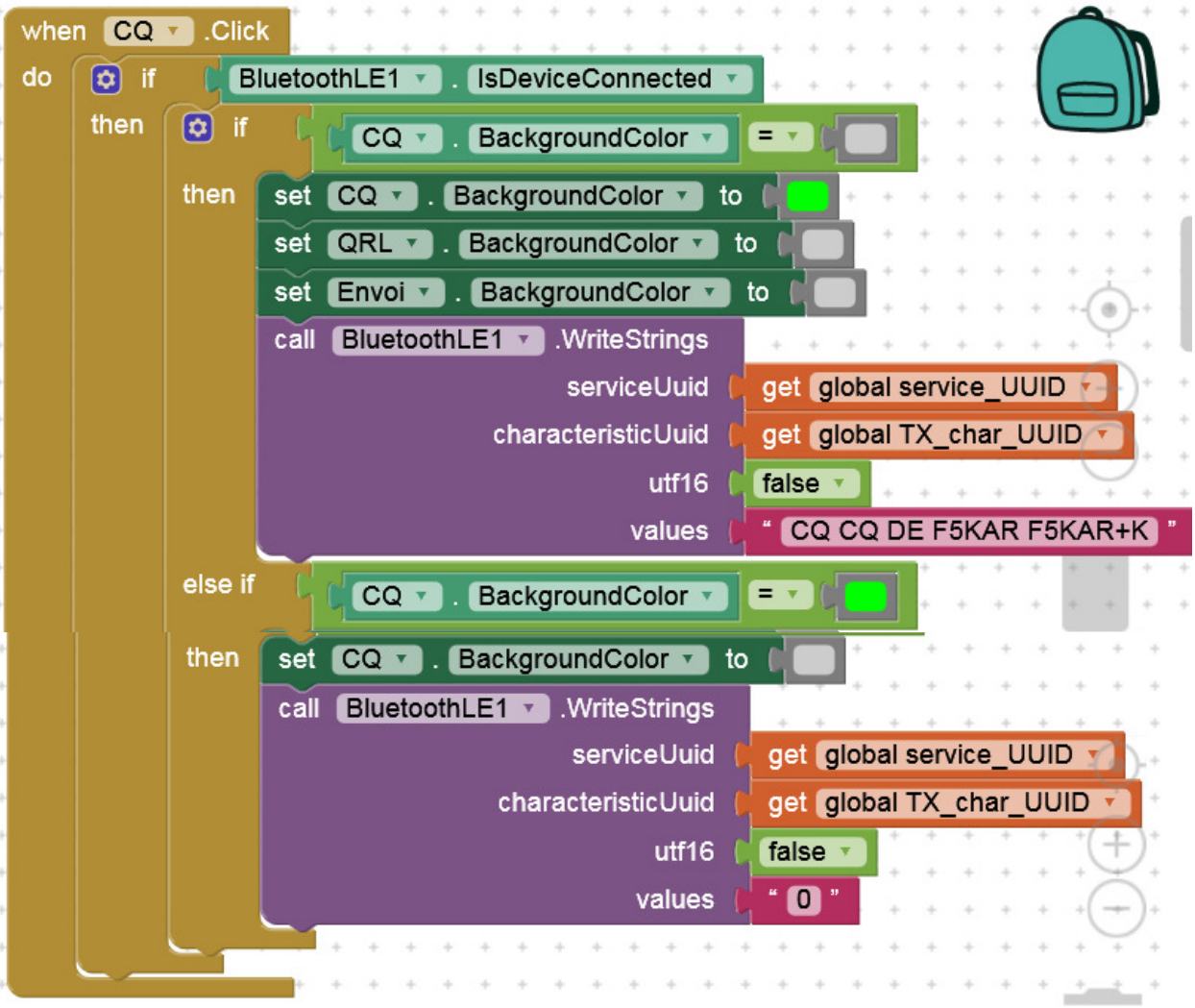
Traitement réalisé

L'utilisateur peut appliquer à chaque composant des blocs de traitement.

Dans l'écran ci-dessous le traitement appliqué à la TextBox appelé SaisieTexte concerne le Focus (curseur à l'intérieur de la TextBox)

- Lorsque le curseur est dans la TextBox , le fond devient blanc et le bouton Envoi passe en vert
- Lorsque le curseur n'est plus dans la TextBox , le fond devient gris et le bouton Envoi passe en gris

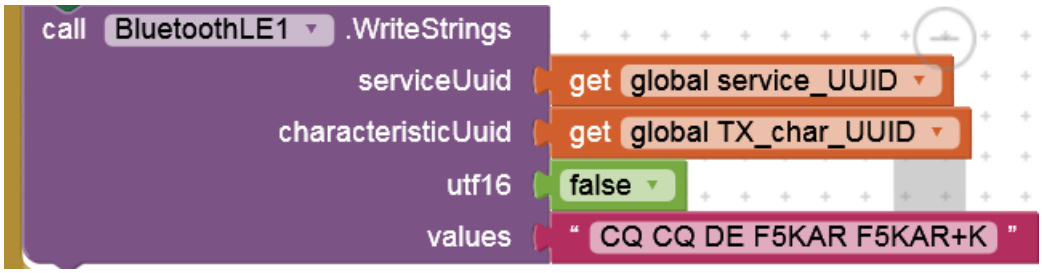




Un traitement plus élaboré est réalisé sur le bouton CQ. Quand l'utilisateur appuie sur ce bouton, tout le traitement situé dans le bloc do est réalisé.

Détaillons ce traitement ;

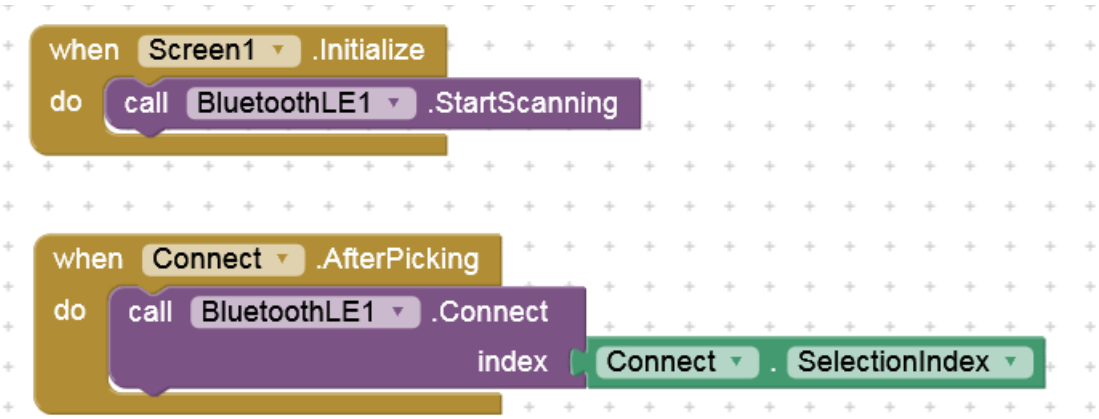
Si le composant BluetoothLE1 est connecté et si la couleur de fond du bouton CQ est grise, alors on passe la couleur de fond de ce bouton en vert et la couleur des 2 autres boutons QRL er Envoi en gris, pour informer l'utilisateur que le bouton CQ est actif.



Puis le composant BluetoothLE1 écrit (WriteStrings) la chaine de caractère « CQ CQ DE F5KAR F5KAR+K » attachée au paramètre values en utilisant les UUID du service et de la caractéristique correspondant à ceux définis dans le logiciel de la carte SuperArduino

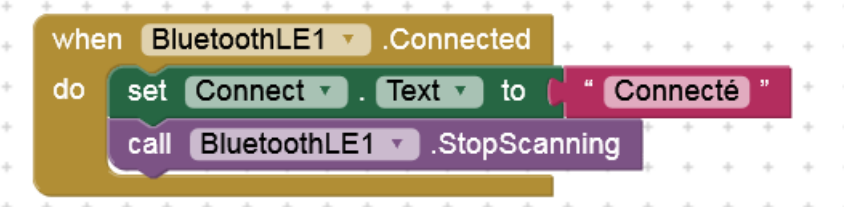
Le TX_char_UUID du Smartphone est 6E400002-B5A3-F393-E0A9-E50E24DCCA9E ce qui correspond au UUID_RX du SuperArduino ; Le smartphone envoie la chaine de caractère et la carte superArduino reçoit cette chaine de caractère.

Traitement Bluetooth

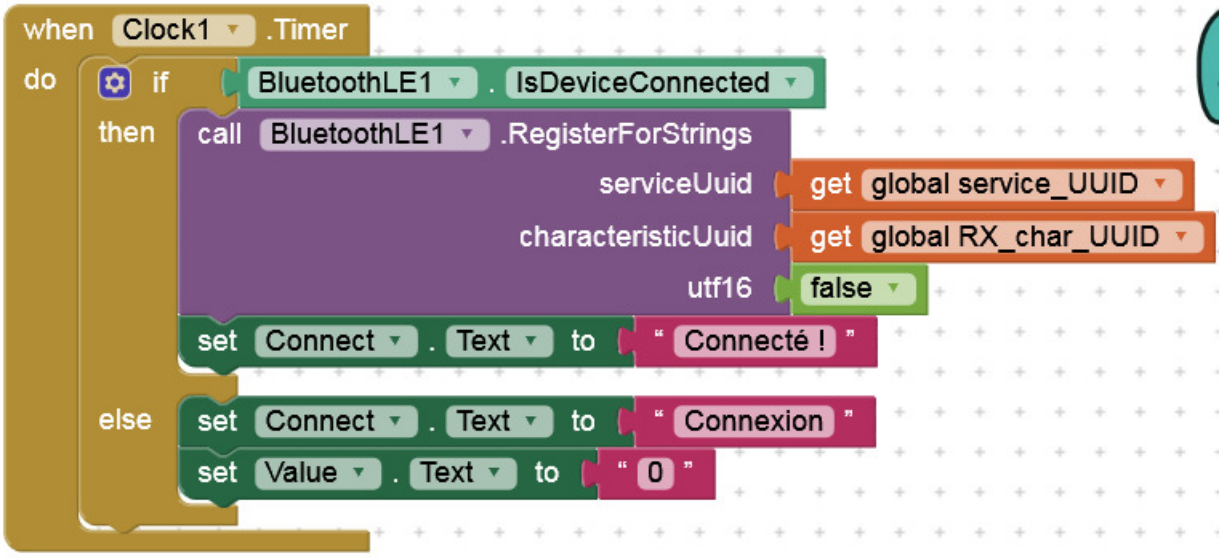


A l'initialisation de l'écran de l'application Android sur le smartphone, le Bluetooth démarre une recherche (scanning) des serveurs Bluetooth se trouvant à portée.

Après appui sur le bouton Connect, la connexion est effectuée avec le serveur choisi par l'utilisateur.



Le bouton Connect voit son libellé changé de Connexion à Connecté



Après connexion, si le périphérique Bluetooth est effectivement connecté, le composant Bluetooth est configuré pour recevoir une chaine de caractères (ce sera le nombre de messages émis en Morse par la carte SuperArduino) avec le service et la caractéristique définis.
Le bouton Connect voit son libellé passer à « Connecté »

Dans le cas où le périphérique Bluetooth n'est pas connecté, le libellé du bouton **Connect** passe à « Connexion » pour indiquer à l'utilisateur que le périphérique Bluetooth n'est plus connecté et doit demander une nouvelle connexion.

L'ensemble des blocs est visible en chargeant le fichier BLE_ESP32_CW_V1.aia dans AppInventor2 (http://ai2.appinventor.mit.edu)

ESP32-CW-VSF Version avec réglages

Pour ceux qui souhaitent approfondir ce sujet, une version avec des réglages a été développée.

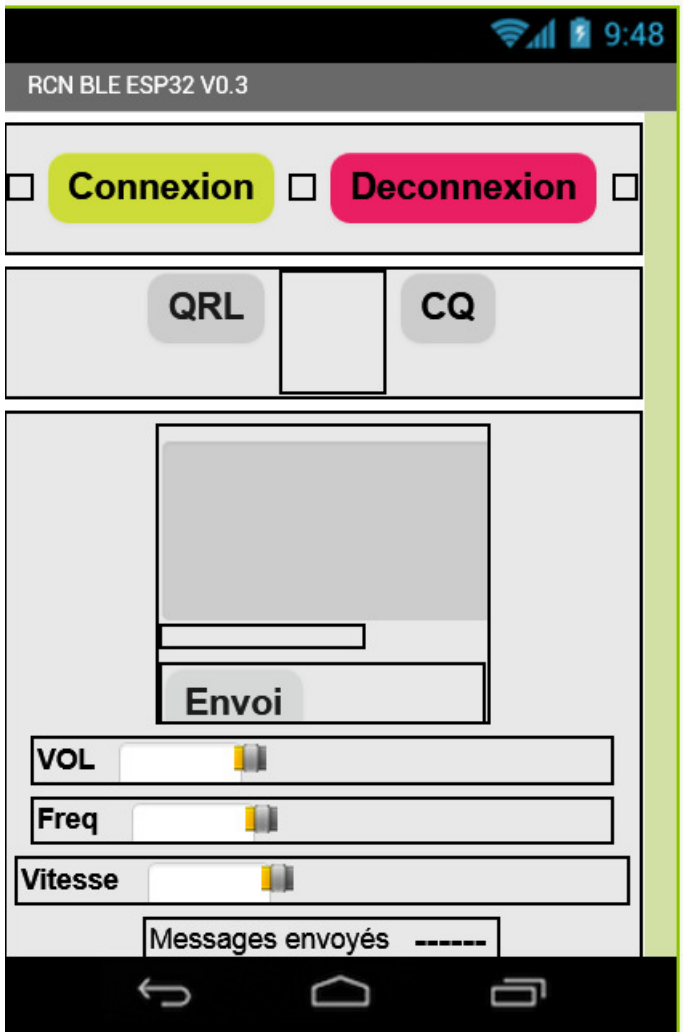
La génération du son par le buzzer ne s'effectue plus avec des delay() mais utilise la fonction matérielle PWM disponible dans l'ESP32.

En réglant la fréquence et le rapport cyclique des impulsions PWM, la fréquence du son et le volume sonore sont modifiés.

En réglant la vitesse, la durée du Point est modifiée (la durée du Tiret étant toujours égale à 3 fois la durée du Point .

Le service Bluetooth LE UART a été enrichi de 3 caractéristiques supplémentaires pour envoyer du Smartphone à la carte EST32 les valeurs de ces 3 réglages.

Les 3 curseurs de réglages sont visibles dans l'interface utilisateur ci-dessous.



Dépôt logiciels

Ces logiciels ont été placés sur le serveur github pour que vous puissiez les essayer, les modifier et les enrichir.

ESP32-CW

Les logiciels sont stockés dans le répertoire github https://github.com/dnguyen76/ESP32-CW

BLE_ESP32_CW_V1.aia	Logiciel source pour produire l'application apk avec AppInventor2
BLE_ESP32_CW_V1.apk	l'application apk à installer sur le smartphone
esp32_ble_X_CW.ino	Logiciel source pour produire l'application à charger sur la carte ESP32 avec IDE Arduino

ESP32-CW-VSF

Les logiciels sont stockés dans le répertoire github https://github.com/dnguyen76/ESP32-CW-VSF

BLE_ESP32_CW_VSF.aia	Logiciel source pour produire l'application apk avec AppInventor2
BLE_ESP32_CW_VSF.apk	l'application apk à installer sur le smartphone
esp32_ble_X_PWM_VSF.ino	Logiciel source pour produire l'application à charger sur la carte ESP32 avec IDE Arduino