Bluetooth LE avec MAUI

applications de contrôle pour Android et cie.

Tam Hanna (Hongrie)

Une application mobile pour est idéale pour contrôler vos propres appareils électroniques. La communication peut être établie via Bluetooth LE pour économiser de l'énergie. Cependant, le développement et la mise à jour de logiciels pour Android et iOS demandent un peu d'effort. C'est là que le cadre MAUI déjà présenté dans un article d'Elektor entre en jeu. Vous pouvez l'utiliser pour programmer des applications indépendamment de la plateforme. Dans cet article, nous allons vous montrer comment utiliser l'interface BLE d'un smartphone.

Le cendrier BopSync, développé par Icy Beats LLC, est contrôlé par une interface Bluetooth LE.

Ceux qui ont une connaissance approfondie du développement .NET trouveront que l'utilisation de MAUI est très pratique pour développer des applications Android et iOS - une introduction générale est disponible dans cet article [1]. Ici, nous voulons « s'approfondir » encore plus en programmant une interface Bluetooth pour Android. Il convient de noter que la bibliothèque utilisée ici est également compatible avec iOS, cependant, les modifications nécessaires dépasseraient le cadre de cet article, c'est pourquoi nous ne les aborderons pas ici.

Configuration de l'environnement de développement

Cet article est basé sur un projet pratique de l'auteur, un développeur indépendant. Un ESP32 sert de « poste distante », qui exécute un programme de contrôle basé sur l'exemple du code [2]. Visual Studio 2022 est utilisé pour développer l'application mobile ; l'exemple de projet actuel est basé sur le modèle d'application .NET MAUI.

Depuis longtemps, Microsoft a décidé de ne pas proposer « directement » une API Bluetooth sous MAUI (ou Xamarin, sur lequel MAUI est basé). C'était une sage décision, car Microsoft a dû avoir une expérience directe du processus de développement des API Bluetooth pour Android, qui était loin d'être « fluide ». Cependant, puisqu'il existe une interface qui permet à Xamarin d'accéder aux éléments natifs du système d'exploitation hôte, il est possible d'utiliser divers paquets NuGet avec des bibliothèques qui visent à compenser cette lacune. Dans les étapes suivantes, nous souhaitons nous appuyer sur la bibliothèque *Plugin.Ble* disponible sur [3]. La première étape consiste à ouvrir la console NuGet, où l'on télécharge la bibliothèque manquante.

Modification du fichier Manifest

En principe, les environnements multiplateformes ne peuvent que partiellement isoler le développeur de la plateforme sous-jacente. Pour la bibliothèque utilisée ici, cela se traduit par le fait que des modifications aux fichiers Manifest respectifs sont nécessaires pour Android et iOS (ce qui n'est pas abordé davantage ici). Ces fichiers indiquent aux systèmes d'exploitation les fonctionnalités que l'application est censée utiliser.

Pour la version 17.6.0 Preview de Visual Studio utilisée ici, un clic sur *AndroidManifest.xml* ouvre un éditeur graphique, qui n'est cependant pas encore tout à fait complet, c'est pourquoi un clic droit sur le fichier et l'ouverture du fichier *Manifest* dans un éditeur de texte est l'approche la plus pratique.

Dans l'étape suivante, l'auteur a remplacé ou ajouté certaines structures aux déclarations existantes (**listage 1**).
À cause du « système de permission » introduit dans Android 6.0, il n'est plus suffisant de modifier le fichier Manifest à ce stade. Au lieu de cela, l'application doit demander à l'utilisateur l'autorisation d'accéder aux fonctions sensibles au moment de l'exécution.
Afin d'éviter le jeu du chat et de la souris entre le système d'exécution MAUI et la gestion des nouvelles autorisations par Google, Microsoft met en œuvre une interface générale pour l'obtention des autorisations.

En principe, il s'agit d'une structure de classe qui contient une liste de constantes de permission. Pour utiliser un nouvel attribut de permission, le développeur n'a qu'à inclure la constante dans une instance *Permissions.BasePlatformPermission*. Le reste de l'interaction avec l'interface utilisateur, qui suit une procédure normalisée, est géré par le *runtime* inclus dans MAUI.
Notre prochaine tâche consiste donc à créer une nouvelle classe `Permissions.BasePlatformPermission`, qui contient les différentes permissions requises (**listage 2**).

Listage 1. Déclarations.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
              <application android:allowBackup="true" android:icon="@mipmap/appicon" android:roundIcon
               ="@mipmap/appicon_round" android:supportsRtl="true"></application>
              <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
              <uses-permission android:name="android.permission.INTERNET" />
              <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
              <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
              <uses-permission android:name="android.permission.BLUETOOTH" />
              <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
              <uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
              <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
              <uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
              <uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
</manifest>
```

Listage 2. Création d'une nouvelle classe Permissions.BasePlatformPermission.

```
public class BluetoothLEPermissions : Permissions.BasePlatformPermission
{
  public override (string androidPermission, bool isRuntime)[] RequiredPermissions
  {
    get
   {
     return new List<(string androidPermission, bool isRuntime)>
   {
     (Manifest.Permission.Bluetooth, true),
     (Manifest.Permission.BluetoothAdmin, true),
     (Manifest.Permission.BluetoothScan, true),
     (Manifest.Permission.BluetoothConnect, true),
     (Manifest.Permission.AccessFineLocation, true),
     (Manifest.Permission.AccessCoarseLocation, true),
        //(Manifest.Permission.AccessBackgroundLocation, true),
    }.ToArray();
    }
 }
}
```

Remarque importante : cette sélection de permissions est valable pour Android 13. Les versions plus anciennes et le Kindle Fire nécessitent un complément de permissions différent.

Recherche d'appareils BLE

Nous pouvons maintenant passer à la recherche d'appareils Bluetooth accessibles. Dans les étapes suivantes, l'auteur suppose que le lecteur est déjà familiarisé avec l'utilisation de Bluetooth LE. Une brève introduction est disponible sur [4].
Pour afficher la liste de tous les périphériques de l'environnement de l'émetteur, l'auteur utilisera une *ListView* en suivant les étapes suivantes : ouvrir le fichier *MainPage.xaml* et ajouter la ListView suivante à un emplacement convenable du code :

```
<ListView x:Name="deviceList">
  <ListView.ItemTemplate>
   <DataTemplate>
    <ViewCell>
    <StackLayout Margin="20,0,0,0"
          Orientation="Horizontal"
          HorizontalOptions="FillAndExpand">
    <Label Text="{Binding}"
          VerticalOptions="Start"
          TextColor="White"/>
    </StackLayout>
   </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Les développeurs habitués à Visual Basic 6 et les environnements similaires doivent changer de perspective lorsqu'ils travaillent avec des boîtes à liste MAUI. Une telle zone de liste MAUI ne consiste pas seulement en une collection d'éléments à afficher - des informations supplémentaires, nommées `ItemTemplate`, sont nécessaires pour l'affichage. Il s'agit d'un balisage XAML simple utilisé par la pile GUI pour chaque ligne de données à afficher dans la ListView. Notre exemple est simple et se contente d'afficher une étiquette à l'écran. La chaîne `Text=""` établit une liaison permettant à l'analyseur XAML d'obtenir directement les données à afficher dans l'élément.
Le fichier de balisage XAML nécessite également un bouton ordinaire qui permettra à l'utilisateur de lancer le processus de balayage.
Dans l'étape suivante, nous pouvons revenir au *Code Behind*, où nous créons quelques classes de support comme suit :

```
public partial class MainPage : ContentPage
{
   int count = 0;
   IBluetoothLE myBLE = CrossBluetoothLE.Current;
   IAdapter myAdapter =
    CrossBluetoothLE.Current.Adapter;
   public ObservableCollection<String> BTLEDevices;
```

Outre les variables requises par la bibliothèque, nous avons également besoin d'une classe `ObservableCollection`. Elle sert de « source de données » au moteur de liaison des données pour la population du ListView. Il convient de noter que le choix d'`ObservableCollection` n'a pas été fait au hasard ; il s'agit d'une classe de collection capable d'envoyer des notifications lorsque des modifications sont apportées à la base de données qu'elle contrôle.
Pour compléter la relation de liaison de données, il faut ensuite modifier le constructeur de la page principale :

```
public MainPage() {
   InitializeComponent();
   BTLEDevices = new ObservableCollection<String>();
   deviceList.ItemsSource = BTLEDevices;
}
```

Ensuite, la technique Code Behind, se charge du clic sur le bouton de recherche. Android intègre un cache de permissions, c'est pourquoi, dans la première étape, nous vérifions si notre application a déjà les permissions nécessaires pour interagir avec l'émetteur Bluetooth. Si c'est le cas, nous appelons la méthode `goScan`, qui se charge de l'exécution du scan.

```
private async void OnScanClicked
   (object sender, EventArgs e) {
   PermissionStatus status = await
     Permissions.CheckStatusAsync
     <BopSyncNetPOC.Platforms.
       Android.BluetoothLEPermissions>();
   if (status == PermissionStatus.Granted){
     goScan();
   }
```

Si la valeur de retour de `CheckStatusAsync` n'est pas positive, nous demandons la permission à l'utilisateur :

```
   else {
   await DisplayAlert("Permission required",
          "Google requires the declaration
           of this permission to perform a BTLE scan.
           Please grant it!", "OK");
   status = await
         Permissions.RequestAsync
           <BopSyncNetPOC.Platforms.
              Android.BluetoothLEPermissions>();
   if (status == PermissionStatus.Granted) {
      goScan();
   }
   else {
      await DisplayAlert("Alert",
           "Permission denied.
            Please reinstall application!", "OK");
    }
  }
}
```

Les textes descriptifs détaillés utilisés ici sont nécessaires pour convaincre l'utilisateur d'autoriser les permissions. L'expérience pratique

a montré que les utilisateurs « confrontés à des difficultés techniques », en particulier, ont tendance à refuser les demandes d'autorisation inattendues dans les fenêtres pop-up, conséquence d'une insécurité alimentée par les médias. En outre, les versions modernes d'Android enregistrent parfois les refus de manière « permanente » - si l'utilisateur clique une fois sur « No », il devra, la plupart du temps, désinstaller et réinstaller l'application avant de pouvoir autoriser à nouveau la demande de permission.
La méthode `goScan` est utilisée ensuite :

```
private void goScan() {
   if (myBLE.State == BluetoothState.On) {
     myAdapter.ScanMode = ScanMode.LowLatency;
     myAdapter.DeviceDiscovered += FoundDevice;
     myAdapter.ScanTimeout = 12000;
     //MUST be called last or ignores settings
     myAdapter.StartScanningForDevicesAsync();
   }
```

Si l'émetteur Bluetooth est déjà actif, nous paramétrons l'objet adaptateur et lançons une recherche en appelant `StartScanningForDevicesAsync`. Il est important de noter que tous les paramètres requis pour l'exécution du balayage doivent être écrits dans la classe avant l'appel de la méthode - les modifications ultérieures sont ignorées.
Si l'émetteur Bluetooth n'est pas actif, une fenêtre s'affiche à la place, invitant l'utilisateur à activer l'émetteur :

```
   else {
   DisplayAlert("Alert",
        "Please switch Bluetooth on!", "OK");
   }
}
```

Le délégué se charge de la localisation des appareils. Nous nous concentrons sur le remplissage de la liste :

```
private void FoundDevice(object sender,
                         DeviceEventArgs e){
   try
   {
     BTLEDevices.Add(e.Device.Name.ToString());
   }
   catch(Exception ex)            {
     var aD = e.Device.NativeDevice;
     //Dont muck around with the GUID in ID
     PropertyInfo aProp =
           aD.GetType().GetProperty("Address");
    BTLEDevices.Add("N/A " + aProp.GetValue(aD, null));
   }
}
```

La majorité des appareils BLE n'ont pas de nom. La routine présentée ici utilise un bloc *try-catch* pour écrire une chaîne par défaut « plus primitive » dans ListView dans ce cas.

Figure 1. La détection Bluetooth LE fonctionne.

À ce stade, le programme est prêt à être testé. La **figure 1** montre à quoi ressemble une exécution du balayage.
Il convient de noter que le placement direct d'éléments natifs dans la partie générale de la solution entraine des résultats bizarres - en particulier, des erreurs signalant l'inexistence de l'espace de nommage `Platforms.Android` se produisent. En effet, certaines opérations de Visual Studio tentent de compiler non seulement la version Android, mais aussi les divers autres projets de platesformes cibles créés dans le squelette du projet MAUI. Pour contourner ce problème, il suffit d'ajouter une protection au niveau du préprocesseur pour le code spécifique à la plate-forme :

```
private async void OnScanClicked
      (object sender, EventArgs e) {
   #if ANDROID
   PermissionStatus status . . .
  #endif
}
```

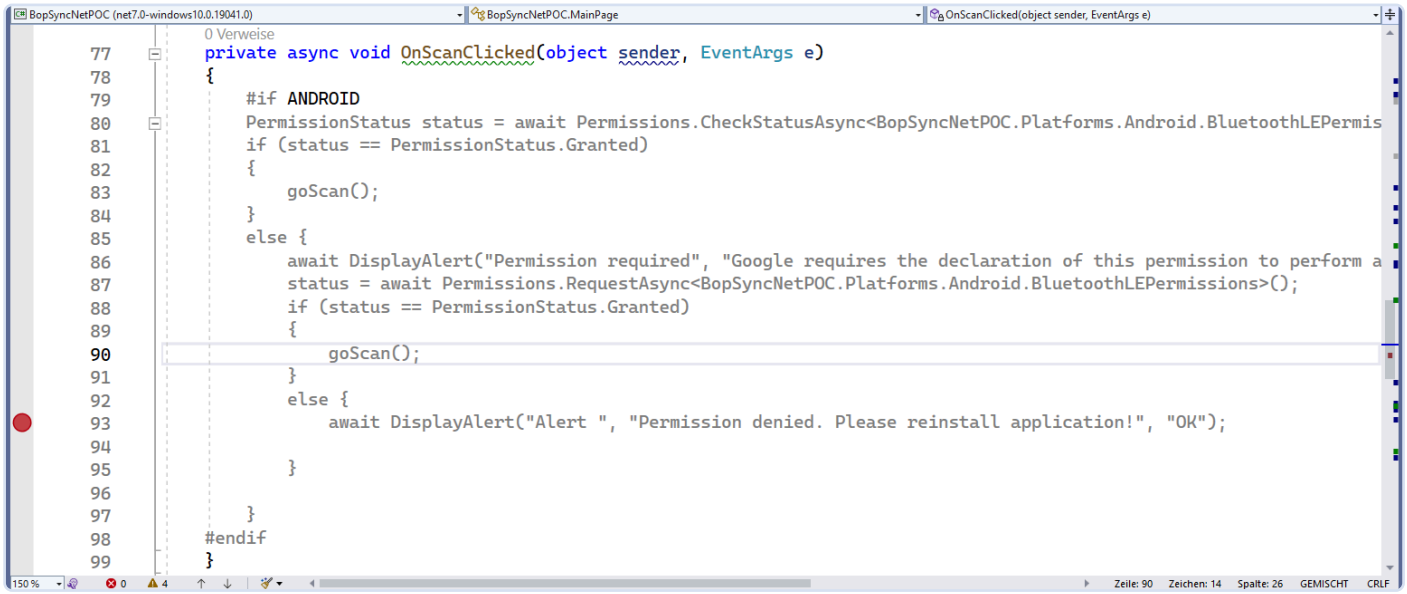


Figure 2. La bataille quotidienne : L'homme contre Visual Studio.

Il est gênant que Visual Studio rencontre des difficultés avec de tels éléments et désactive la complétion syntaxique, comme le montre la **figure 2**.

Identification et établissement de la connexion

Nous sommes maintenant prêts à contrôler le périphérique. L'auteur a ajouté à la page principale une liste supplémentaire dans laquelle il a intégré les instances individuelles de périphériques Bluetooth - elles représentent les stations distantes trouvées par le balayage Bluetooth LE.

Pour l'application de l'auteur, un seul appareil doit être traité à la fois. C'est pourquoi il est judicieux de conserver l'instance dans la classe *Application*. Ouvrez le fichier *App.xaml.cs* et insérez une variable globale :

```
public partial class App : Application
{
   public Plugin.BLE.Abstractions.
   Contracts.IService myBTLEService;
```

La qualification complète est raisonnable parce qu'il y a souvent plus d'une classe portant le nom IService dans l'environnement *.NET*. Si vous déclarez « completely », vous êtes en sécurité.

L'établissement de la connexion commence alors par l'extraction de la classe `IDevice` à partir de la liste des appareils mentionnée ci-dessus :

```
private async void deviceList_ItemSelected
  (object sender, SelectedItemChangedEventArgs e)
{
   IDevice myDevice =
   BTLEDeviceClasses[e.SelectedItemIndex];
```

L'étape suivante est l'obtention d'une instance d'appareil et l'établissement d'une liste de tous les services :

```
try
{
   await myAdapter.ConnectToDeviceAsync(myDevice);
   var services = await myDevice.GetServicesAsync();
   services = services;
   foreach (IService serv in services) {
     String aString = serv.Id.ToString();
     if (aString.CompareTo
        ("000000ff-0000-1000-8000-00805f9b34fb") == 0)
     { //Swap view
       App curApp = (App) Application.Current;
       curApp.myBTLEService = serv;
       await Navigation.
         PushModalAsync(new MainWorkPage()) ;
     }
   }
}
```

Ici, l'auteur a décidé d'identifier la poste distante en vérifiant la disponibilité d'un service spécifique. Si un service doté de l'interface graphique requise est trouvé, l'activité actuellement affichée est modifiée.

Il est important d'utiliser la méthode `Navigation.PushModalAsync`. En effet, l'auteur utilise une page dérivée de [5] dans son application commerciale - l'utilisation de la méthode normale `PushAsync`, recommandée dans la documentation de Microsoft, conduit à d'étranges plantages de la fenêtre.

Le bloc try-catch est nécessaire car il protège contre les problèmes de communication :

```
   catch (DeviceConnectionException ex)
   {
   // ... could not connect to device
   }
}
```

La tâche suivante de l'application consiste à communiquer avec les caractéristiques. Afin de réduire la logique contenue dans les vues, l'auteur s'appuie sur la structure présentée dans l'organigramme de la **figure 3**.

Si vous placez la fonction de communication dans une classe (idéalement statique), vous ne devez pas la créer dans les pages individuelles. L'auteur a de nouveau décidé d'utiliser la classe `App` dans son application ; l'étape suivante est la déclaration :

```
public async void setLightMode(int _what)
{
   var x = await myBTLEService.
      GetCharacteristicAsync(Guid.Parse
         ("0000ff01-0000-1000-8000-00805f9b34fb"));
```

La première étape consiste à utiliser la méthode `GetCharacteristicAsync`, qui détermine une caractéristique spécifique sur la base de son GUID respectif. Comme nous avons effectué une « identification » ci-dessus, l'auteur s'abstient de sauvegarder la valeur de `x` dans cet extrait de code - en pratique, il serait raisonnable de le faire.
Il convient de noter que la bibliothèque serait également capable d'effectuer une recherche « globale ». Le code nécessaire à cette tâche ressemblerait à ce qui suit :

```
public async void setLightMode(int _what)  {
  var ibx = await myBTLEService.GetCharacteristicsAsync();
  foreach (var chara in ibx) {
     var str = chara.Id.ToString();
     str = str;
  }
...
```

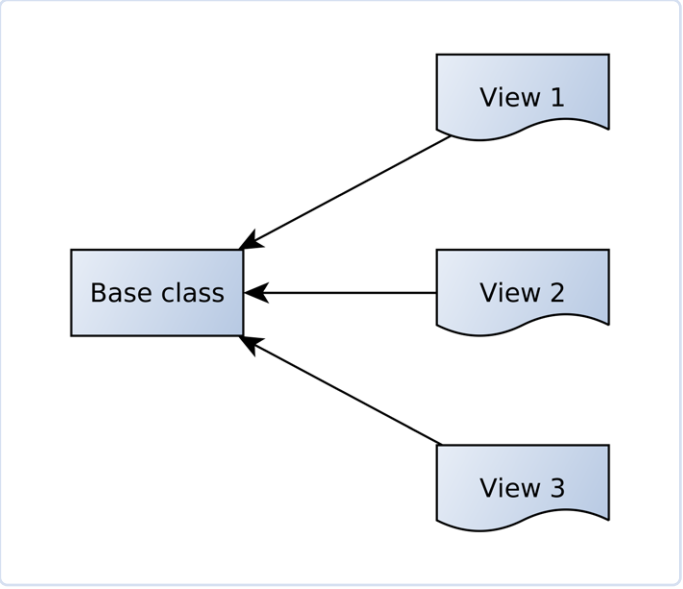


Figure 3. L'isolation de la logique de communication réduit la redondance dans le système.

L'instruction tautologique utilisée dans cet extrait est utile parce qu'elle permet de « récolter » les GUI des caractéristiques individuelles. Si vous ne pouvez pas extraire la caractéristique de l'interface graphique du programme ESP32 sous la forme d'une chaîne, vous pouvez placer un point d'arrêt sur l'instruction tautologique et exécuter le programme associé.
Pour chaque exécution, vous pouvez alors extraire les caractéristiques respectives dans le débogueur - comme le montre la **figure 4** - et (si nécessaire) les comparer aux informations affichées dans le scanner Bluetooth.

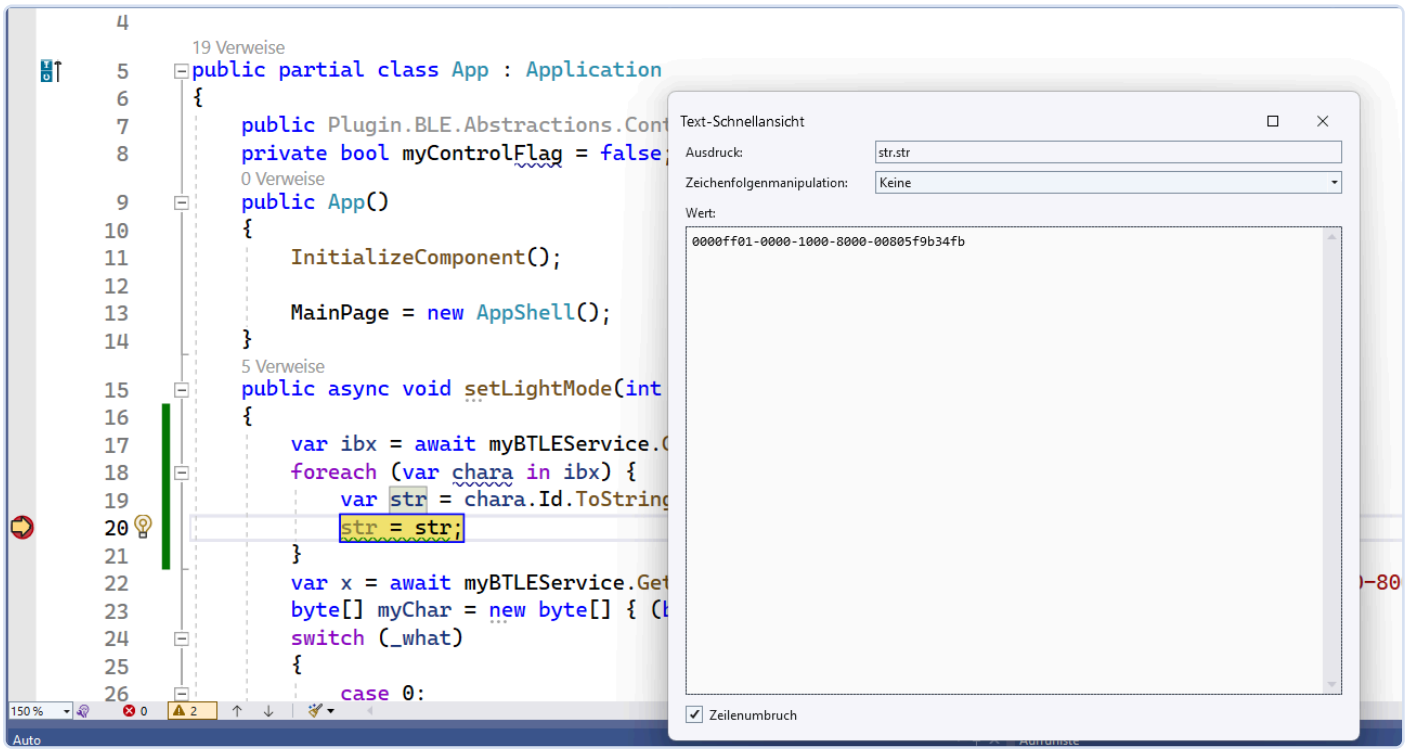


Figure 4. Les instructions tautologiques permettent d'économiser de la frappe !

Les scanners Bluetooth sont utiles

Il serait judicieux de conseiller à toute personne en charge d'une application Bluetooth LE d'installer un scanner Bluetooth LE sur son smartphone Android. L'application permet de se connecter à toutes les stations distantes et de lister les services et caractéristiques qui s'y trouvent. L'auteur utilise l'application nRF Connect dans son entreprise, qui peut être téléchargée gratuitement depuis le Play Store [7].

Pour utiliser l'application scanner, il est recommandé d'avoir un Android ou un ancien iPhone sous iOS 15. À partir d'iOS 16, des modifications ont été apportées à la pile Bluetooth, rendant les appareils moins visibles et limitant la fonctionnalité du scanner!

L'étape suivante consiste à rassembler le playload. Dans l'exemple de l'auteur, le protocole de communication propriétaire est basé sur des flux de données qui transmettent des valeurs ASCII. Le playload est ensuite rassemblé en fonction de la valeur entière fournie avec des constantes de tableau :

```
byte[] myChar = new byte[] { (byte)'0' };
switch (_what)
   {
   case 0:
     myChar= new byte[] { (byte)'0' };
     break;
   case 1:
     myChar = new byte[] { (byte)'1' };
     break;
   case 2:
     myChar = new byte[] { (byte)'2' };
     break;
   case 3:
     myChar = new byte[] { (byte)'3' };
     break;
   case 4:
     myChar = new byte[] { (byte)'4' };
     break;
...
}
```

Enfin, une commande d'établissement de la connexion est nécessaire, dont la structure est la suivante :

```
     var y = await x.WriteAsync(myChar);
   }
```

Envoi de données de l'ESP32 vers le téléphone Android

Enfin, nous voulons réaliser la transmission des données de l'ESP32 via l'interface sans fil. À ce stade, l'auteur aimerait donner un peu plus de détails sur le code ESP-IDF. Les exemples basés sur le système de tables GATT sont, selon l'auteur, très mal documentés ; il y a quelques problèmes confus et des réponses partielles dans le forum.

La question la plus importante à ce sujet est de savoir comment les caractéristiques sont initialisées. Si la constante `ESP_GATT_AUTO_RSP` est transmise, comme spécifié par défaut dans l'exemple, la pile traite les valeurs contenues dans la caractéristique.
L'application reçoit également un événement de lecture comme indiqué ci-dessous ; toutefois, la pile Bluetooth reçoit les valeurs renvoyées d'une mémoire interne et les achemine normalement avant de déclencher l'événement :

```
case ESP_GATTS_READ_EVT:
   ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_READ_EVT");
   break;
```

En théorie, la méthode `esp_ble_gatts_set_attr_value` fournit une fonction qui permet de modifier les valeurs de mémoire stockées dans la pile Bluetooth. Une exécution simple ressemblerait à ce qui suit :

```
void updateBTLECache() {
   esp_err_t ret;
   ret = esp_ble_gatts_set_attr_value
     (heart_rate_handle_table[IDX_CHAR_VAL_A],
        1, (const uint8_t *)"1");
}
```

Le problème avec cette méthode est qu'elle renvoie une valeur nulle même si la chaîne transmise n'est pas valide ou n'a pas encore été initialisée par la pile Bluetooth.
Cela complète déjà la « description du problème » - la configuration ou le remplissage du tableau a lieu relativement tard. Une méthode que la société de l'auteur a trouvée efficace consiste à mettre à jour les valeurs du cache BTLE à chaque fois qu'un nouveau périphérique se connecte à l'ESP32. Le code suivant est idéal à cette fin :

```
case ESP_GATTS_CONNECT_EVT:
   ESP_LOGI(GATTS_TABLE_TAG, "ESP_GATTS_CONNECT_EVT,
            conn_id = %d", param->connect.conn_id);
   updateBTLECache();
   esp_log_buffer_hex(GATTS_TABLE_TAG,
            param->connect.remote_bda, 6);
```

Enfin, voici un extrait de code qui illustre la lecture des valeurs du côté MAUI :

```
public async void setLightMode(int _what)
{
   var x = await myBTLEService.
        GetCharacteristicAsync(Guid.Parse
             ("0000ff01-0000-1000-8000-00805f9b34fb"));
   byte[] z = await x.ReadAsync();
   z = z;
...
```

Il est alors possible de traiter les valeurs fournies sous la forme d'un tableau d'octets.

Accélérer le développement

Nos essais montrent que l'API Bluetooth LE de MAUI peut interagir avec d'autres systèmes cibles facilement. Si vous avez besoin d'une application pour votre système embarqué et que vous la développez avec MAUI, vous pouvez la programmer en C# ou Visual Basic et éviter la complexité des plates-formes natives. En particulier si votre entreprise dispose déjà d'une expertise avec le cadre .NET, cela peut conduire à une accélération significative du processus de développement. Bien entendu, le contrôle Bluetooth LE sur un smartphone ne constitue que la « moitié du travail ». Dans un article publié précédemment [6], l'auteur montre comment programmer un microcontrôleur STM32 avec une interface BLE.

230381-04

Questions ou commentaires ?

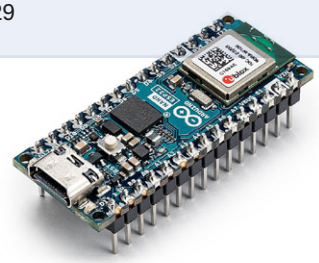
Envoyez un courriel à l'auteur (tamhan@tamoggemon.com) ou contacter Elektor (redaction@elektor.fr).

À propos de l'auteur

Ingénieur dans le domaine de l'électronique, de l'informatique et du logiciel depuis plus de 20 ans, Tam Hanna est designer indépendant, auteur de livres et journaliste (@tam.hanna on Instagram). Dans son temps libre, Tam conçoit et produit des solutions imprimées en 3D et, parmi d'autres activités, il a une passion pour le commerce et la dégustation de cigares haut de gamme.

Produits

> **Arduino Nano ESP32 avec des connecteurs**
> www.elektor.fr/20529



LIENS

[1] V. Krypczyk, « MAUI : Programmation pour PC, tablettes et smartphones «, Elektor 1-2/2024: https://elektormagazine.fr/220442-04
[2] GATT Server Service Table: https://bit.ly/45KupIU
[3] Plugin.Ble library: https://github.com/dotnet-bluetooth-le/dotnet-bluetooth-le
[4] Brief introduction to Bluetooth Low Energy (BLE): https://developer.android.com/develop/connectivity/bluetooth/ble/ble-overview
[5] Navigation.PushModalAsync method: https://github.com/dotnet/maui-samples/tree/main/8.0/Navigation/FlyoutPageSample
[6] T. Hanna, « Bluetooth LE sur le STM32 « Elektor 1-2/2024 : https://elektormagazine.fr/230698-04
[7] Application nRF Connect : https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp