C# iOS and Android application for TSL 1128 handheld: Step by Step

# Abstract:

This document explains the framework for developing mobile applications for TSL 1128 handheld RFID/Barcode readers. It uses the C# language via the Xamarin framework to develop code that can run both on iOS and Android platforms. It introduces a plugin-based framework that can be extended to include specialized functionality for different use cases in various domains.

# Installation and Import of the Reference Repo:

Follow the Installation instructions based on your development environment

* <https://developer.xamarin.com/guides/cross-platform/getting_started/requirements/>
* Follow instructions on the same website to setup your simulator and physical devices for mobile development.

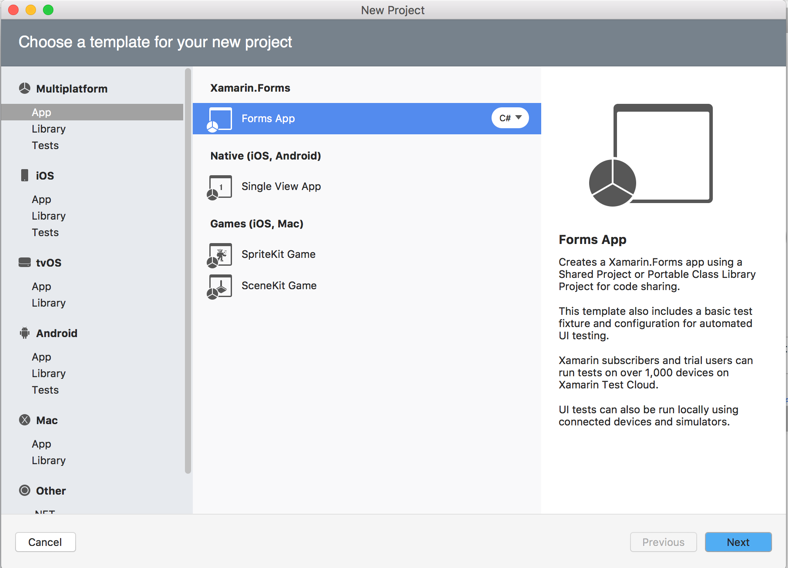
This document was developed with Xamarin Studio running on a Mac. On Windows, it is best to run Xamarin inside Visual Studio.

Clone the reference repo in Github.

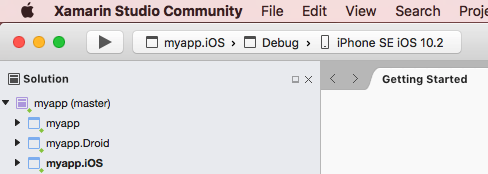
* <https://github.com/ralemy/xamarin-tsl-handheld>

# Setting Up your own project:

In your own repo (a separate directory), Create a new solution based on Xamarin Forms App:

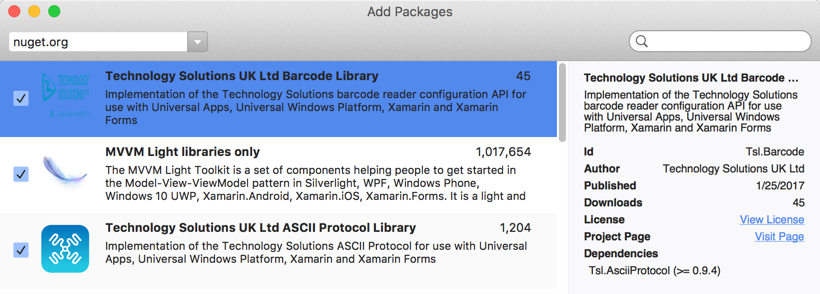


Give your app a name, I use “myapp” here, but it can be anything based on what you want your app to do.

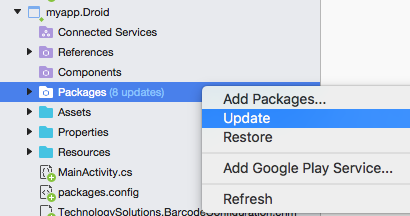


For all projects (myapp, myapp.Droid, myapp.iOS, etc) , right click and add the following packages:

* Tsl.Barcode, Tsl.AsciiProtocol, and MvvmLightLibs



Open each project and update the packages if necessary:

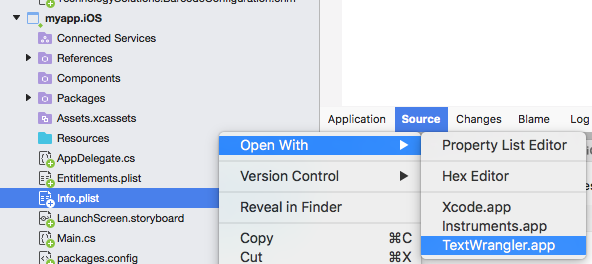


Add the following code to myapp.Droid /Properties/AndroidManifest.xml

<uses-permission android:name = "android.permission.BLUETOOTH" />

<uses-permission android:name = "android.permission.BLUETOOTH\_ADMIN" />

Edit myapp.iOS/info.plist with an external editor:



Add the following code to the dict element at the end of the file:

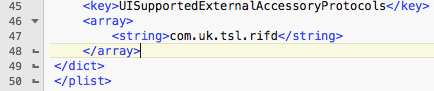
<key>UISupportedExternalAccessoryProtocols</key>

<array>

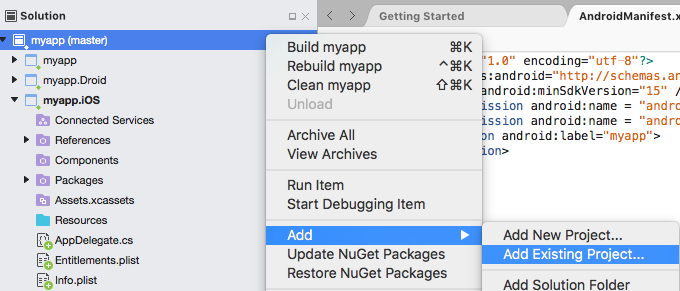
<string>com.uk.tsl.rifd</string>

</array>

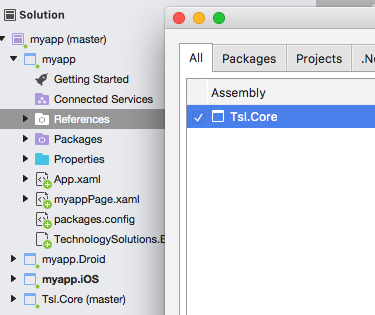
So your file will look like this:



# The next step is to import the Tsl.Core project into your solution. Right-click on the solution, select “add existing project”



Now navigate to the reference repo and select Tsl.Core/Tsl.Core.csproj. This will add Tsl.Core as the fourth component of your solution. Then go ahead and double click on references directory in your main project, and add Tsl.Core as a reference.



# How it is arranged:

This project uses the following technologies:

* Xamarin Forms, to create apps with C# that run both on iOS and Android.
* MVVMLight, for Dependency Injection and MVVM design pattern.
* TSL.AsciiProtocol and Tsl.Barcode for communication with the TSL reader.

When creating a new app, the steps explained in the “Setting Up Your Own Project” section will create a Xamarin Forms application which has both android and iOS sub projects. We will import the Tsl.Core project which provides us with general functionality for connecting to the handhelds and using the Mvvmlight framework. We then edit the main project to integrate that with the rest of our application.

# How it Works:

The main entry to the application is in Tsl/App.xaml.cs (shows as Tsl/App.xaml/App.xaml.cs in Xamarin Studio)

There are two things this file has to do, First, it has to make sure all needed dependencies are injected. Second, it has to register all required app pages.

## Injecting Dependencies

For dependency injection, this project uses the SimpleIoc (inversion of control) class from the MvvmLight framework. Each plugin is expected to provide a method that runs as a singleton and registers dependencies. For Tsl.Core project, this is the static method InjectDependencies() which is found in Tsl.Core.ViewMode.ViewModelLocator class. The method uses a static property to make sure that the registration happens only once.

The Constructor for the App Class in the main project has to call the dependency registration method on each plugin. Therefore, the first thing to do would be to edit App.xaml.cs and add a method to register the Tsl.Core dependencies:

public App()

{

InitializeComponent();

RegisterDependencies();

MainPage = new myappPage();

}

void RegisterDependencies()

{

ViewModelLocator.InjectDependencies();

}

If you have more plugins, you would call their registration method after the InjectDependencies() call in the RegisterDependencies method.

ViewModelLocator class in Tsl.Core package has a static GetDependency method that returns a specific dependency if it is registered. This can be used for all dependencies registered using the same technique, even when they were registered with other plugins.

Tsl.Core.ViewModel.ViewModelLocator Registers the following dependencies:

|  |  |
| --- | --- |
| **Dependency** | **Responsibility** |
| INavigationService | Used to Navigate to different pages on the app per MVVMLight protocol |
| INavigationManager | Used to register the app pages with the framework |
| IUIRunner | Used to force an action to be executed on the UI thread of the app |

## Registering Pages:

Mobile apps can have multiple pages. Plugins can also add their own pages, for example the Tsl.Core will add a page to select the handheld and connect to it or disconnect from it. The MvvmLight framework requires a class that implements INavigationService to handle the navigation between pages. Tsl.Core contains such implementation and injects it as a dependency.

The main project should specify the main page and ask each plugin to register any pages they have. This is needless to say best done in constructor, so the above constructor is refactored to do just that (myappPage is the default main page created automatically by Xamarin Studio):

public App()

{

InitializeComponent();

RegisterDependencies();

MainPage = RegisterPages(new NavigationPage(new myappPage()));

}

Page RegisterPages(NavigationPage navigationPage)

{

var navigation = ViewModelLocator

.GetDependency<INavigationManager>();

navigation.SetMain(navigationPage);

ViewModelLocator.RegisterPages(navigation);

return navigationPage;

}

## Components of a Xamarin Page

This document only discusses the xaml pages in Xamarin. Xaml pages consist of the following components:

* Xaml file, which is an XML file that defines the layout of the page
* Xaml.cs file, which is a C# file that connects the layout to a ViewModel.
* ViewModel file, which is a C# file that controls the UI logic of the view and the data for the view
* Model file, which is a C# data structure that defines the reference data for the page.

The page may also use services, providers, other models, etc. as needed by the functionality it implements.

### Xamarin xaml file:

This XML file defines the layout of the page. There are multiple types of layout, the simplest being the ContentPage. There is quite a few types of pages, explaining all of them is out of the scope of this document, but more information can be found in the link below:

<https://developer.xamarin.com/samples/xamarin-forms/Navigation/>

When myapp was first created, Xamarin automatically placed a myappPage.xaml file in the main project. This is a ContentPage and looks like following:

<?xml version="1.0" encoding="utf-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

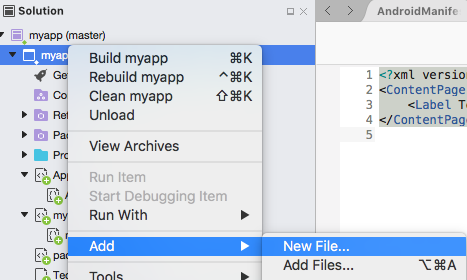
xmlns:local="clr-namespace:myapp" x:Class="myapp.myappPage">

<Label Text="Welcome to Xamarin Forms!"

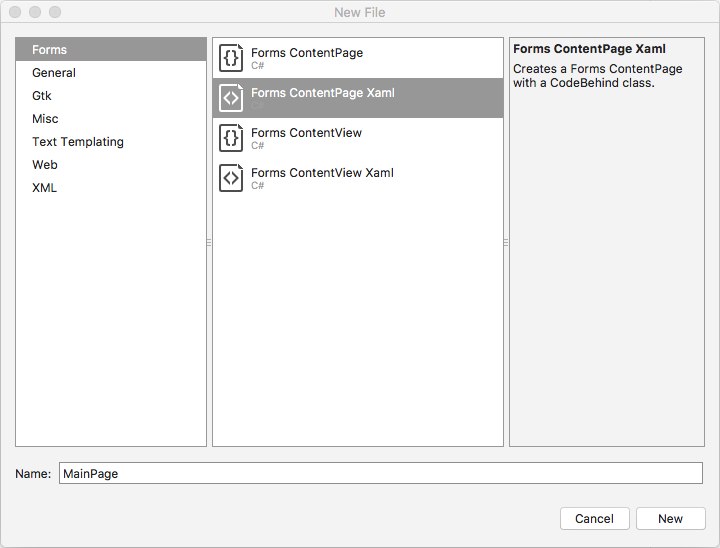
VerticalOptions="Center" HorizontalOptions="Center" />

</ContentPage>

We will add a new file to the main project:



And specify that to be a ContentPage Xaml under the name MainPage:



This will add two files to the main project: MainPage.xaml and MainPage.xaml.cs. Now, we will refactor App.xaml.cs to use this page as the main page:

public App()

{

InitializeComponent();

RegisterDependencies();

MainPage = RegisterPages(new NavigationPage(new MainPage()));

}

the myappPage.xaml and myappPage.xaml.cs files are no longer needed, so let’s go ahead and delete them.

Editing The MainPage.xaml, it is just another empty ContentPage element:

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="myapp.MainPage">

<ContentPage.Content>

</ContentPage.Content>

</ContentPage>

Notice the x:Class=”myapp.MainPage” attribute. This is how the XML file knows which class is associated with it. Let’s now add a button to navigate to the Connect Page for selection of the handheld.

<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="myapp.MainPage">

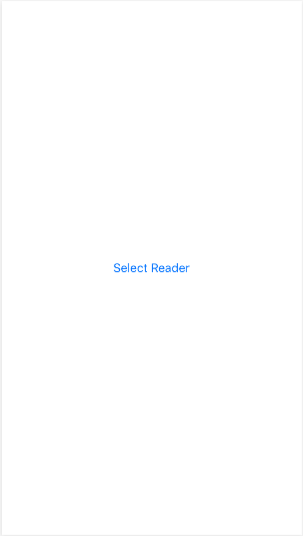
<Button VerticalOptions="Center"

HorizontalOptions="Center"

Text="Select Reader"/>

</ContentPage>

It will look like this:



### Defining and injecting page ViewModel

The next step is to define behavior for the button. For that to happen the page needs a ViewModel. We will create a folder for all ViewModels, and a C# class in it called MainViewModel.cs which extends the ViewModelBase. The NavigationService registered by the Tsl.Core project is injected in the constructor.

To support a callback when the button is clicked, it has an ICommand property that it initializes in the constructor to point to a method that will then navigate to the ConnectReader page registered earlier by the Tsl.Core project.

public class MainViewModel :ViewModelBase  
    {  
        public ICommand ConnectCommand { get; private set;}  
        INavigationService navigator;  
  
        public MainViewModel(INavigationService navigator)  
        {  
            this.navigator = navigator;  
            ConnectCommand = new RelayCommand(ExecuteConnect);  
        }  
        public void ExecuteConnect()  
        {  
            navigator

.NavigateTo(ViewModelLocator.ConnectPageKey);

        }  
    }

This class is injected as a dependency in the App.xaml.cs where dependencies are registered:

void RegisterDependencies()  
        {  
            ViewModelLocator.InjectDependencies();  
            ViewModelLocator.Register<MainViewModel>();  
        }

And it is bound to the view in the constructor of the MainPage (MainPage.xaml.cs)

public MainPage()  
        {  
            InitializeComponent();  
            BindingContext = ViewModelLocator

.GetDependency<MainViewModel>();

        }

To use the Command we just added, in MainPage.xaml we will bind the button to execute it when pressed:

<?xml version="1.0" encoding="UTF-8"?>  
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"   
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"   
        x:Class="myapp.MainPage">  
    <Button VerticalOptions="Center"   
        Command="{Binding ConnectCommand}"  
        HorizontalOptions="Center"   
        Text="Select Reader"/>  
</ContentPage>

## Feature Value Table:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Feature | Value | Comment |
| **1** | **Encapsulate MVVMLight Frame work** | **Epic** | **So that we are not bugged down to one framework.** |
| 1.1 | Encapsulate Dependency Injection | 2 | Use MvvmLight Inversion of control behind the scenes |
| 1.2 | Implement Navigation Service | 5 | Add registration, setup, and navigation endpoints |
| 1.3 | Implement a plugin protocol for ViewModels, Services, and Pages | 5 | Standard methods and DI protocol for encapsulating functionality and UI implementation detail |
| 1.4 | Implement sample code for navigation pages | 2 | End to end example for a view, viewmodel, and a model providing a functionality on a Xamarin app |
| **2** | **Encapsulate TSL Reader Framework** | **Epic** | **Abstract the details of most common use cases of Handheld devices** |
| 2.1 | Implement Handheld Connection Page | 5 | Connects, disconnects, and lists available TSL 1128 Handheld readers |
| 2.2 | Implement Handheld Configuration Page | 5 | Sets the power, rssi report during app operation |
| 2.3 | Implement Handheld callbacks | 3 | Sets the barcode and transponder finder |
| **3** | **Encapsulate Backend Communication** | **Epic** | **Abstract sending the data from the app to backend** |
| 3.1 | Implement Restful Get, Post, Put | 2 | Have a sample UI for config, but allow the user to replace it with their own. |
| 3.2 | Implement Restful authentication with basic http | 2 | Secure the password inside the app to be able to set it in config page |
| 3.3 | Implement Restful authentication with token | 2 | Allow for expansion of authentication to include OAuth and other protocols |
| 3.4 | Implement RabbitMQ client | 5 | Make sure it runs both on iOS and Android. If not, change to ZeroMQ |
| 3.5 | Implement Json data representation | 3 | Provide sample code for marshall/ unmarshall of json objects |