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

Table of Contents

Scope:

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 setup:

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.

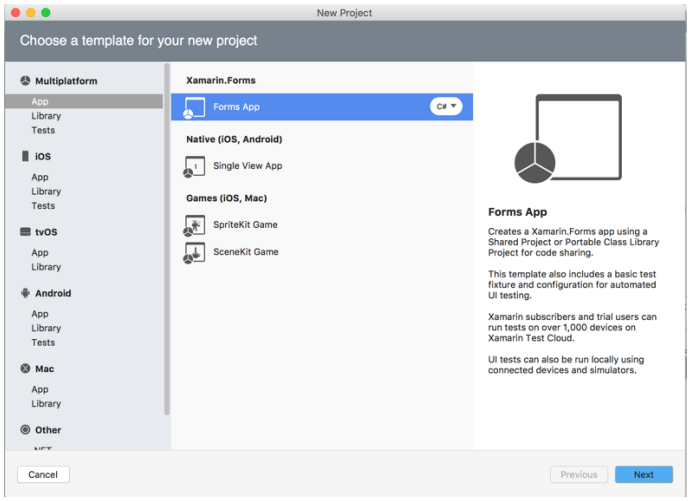
This document was developed with Xamarin Studio running on a Mac. On Windows, it is best to run Xamarin inside Visual Studio.

Clone the reference repo in Github.

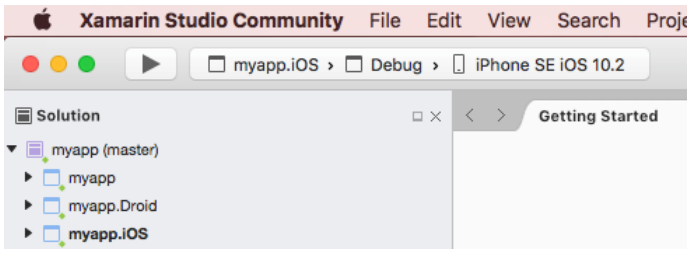
- https://github.com/ralemy/xamarin-tsl-handheld

Setting Up your own project:

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

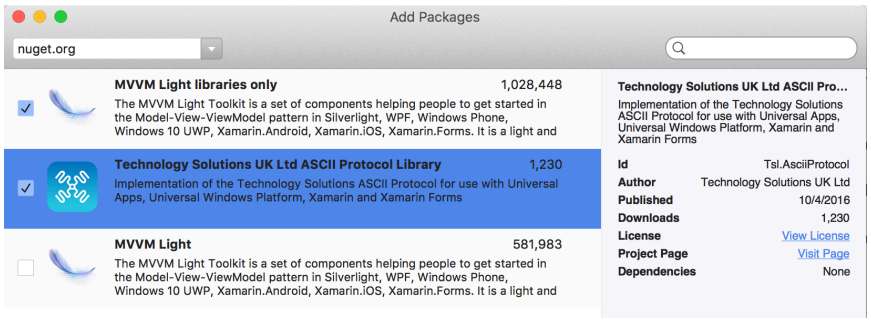


Give your app a name, I use “myapp” here, but it can be anything based on what you want your app to do.

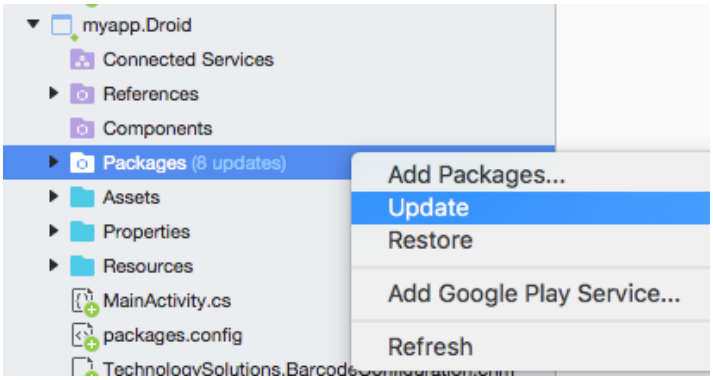


For all projects (myapp, myapp.Droid, myapp.iOS, etc) , right click and add the following packages:

- Tsl.AsciiProtocol, and MvvmLightLibs



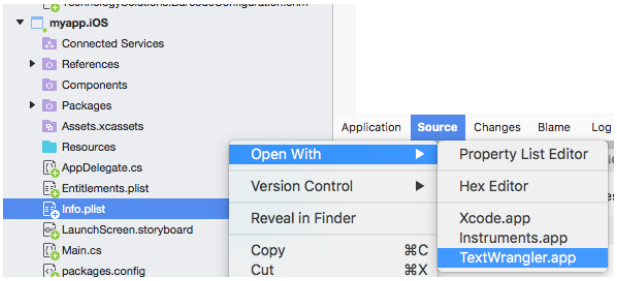
Open each project and update the packages if necessary:



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:



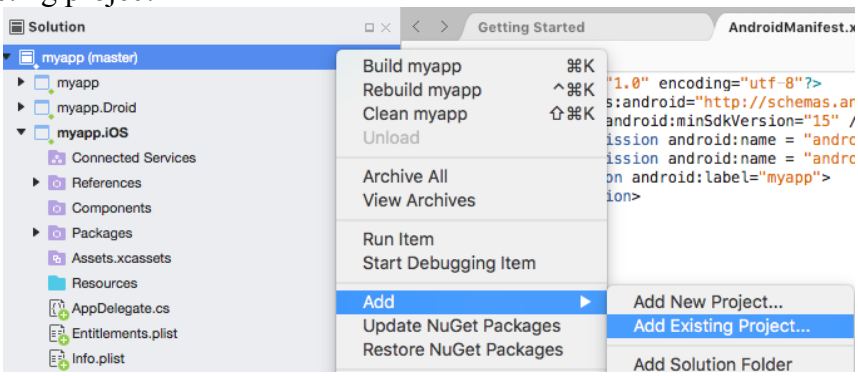
Add the following code to the dict element at the end of the file:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
<string>com.uk.tsl.rifd</string>
</array>
```

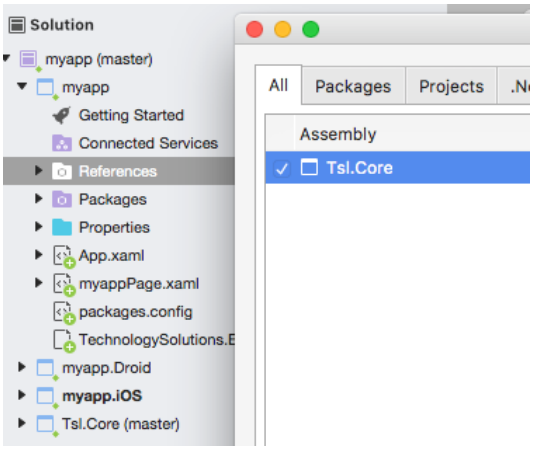
So your file will look like this:

```
45      <key>UISupportedExternalAccessoryProtocols</key>
46      <array>
47          <string>com.uk.tsl.rifd</string>
48      </array>
49  </dict>
50  </plist>
```

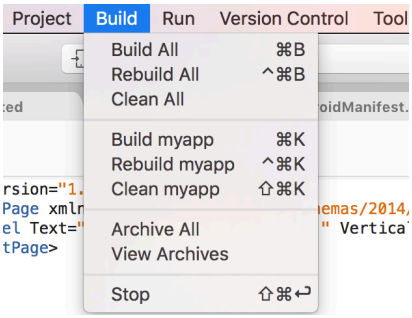
The next step is to import the Tsl.Core project into your solution. Right-click on the solution, select “add existing project”



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. Do the same for Droid and iOS projects.



Finally, from the build menu select Clean All and then Rebuild All.



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 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

Behind the scenes, the Tsl.Core project uses the SimpleIoc (inversion of control) class from the MvvmLight framework. However we will encapsulate it with the ViewModelLocator in case we want to change it in the future.
ViewModelLocator itself registers quite a few dependencies (see below). Therefore, the first things that the app should do is to call the InjectDependencies() on that class to make the base dependencies available.
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();
    }
```

As you add your own services, viewmodels, and other dependencies, you will register them in the RegisterDependencies() method. By using the Register<T>() function of the ViewModelLocator. More on this later.

Adding new pages to the app:

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.

Three files are usually involved in a mobile page

- Xaml file, which is an XML file that defines the layout of the page
- Xaml.cs file, which is a C# file that contains the code behind the xaml file.
- ViewModel file, which is a C# file that controls the UI logic of the view and the data for the view.
    - This file is optional but if it is present then it has to be bound to the xaml file. Binding can be achieved from the xaml or the xaml.cs file.

For this document for each page we always make a ViewModel file and always inject it as a dependency and bind it as context of the xaml file in the xaml.cs file.

To Navigate between pages, 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.

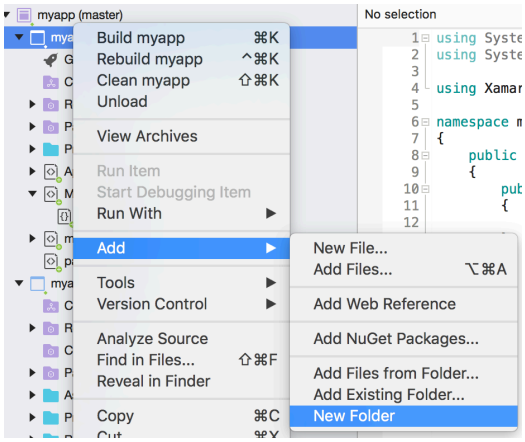
The main project can then get this service and use it to set the main page of the application, and register any extra pages that the project has. One of the first steps is to register the pages that Tsl.Core provides, which can be done by calling the RegisterPages() method as shown below:

```
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;
}
```

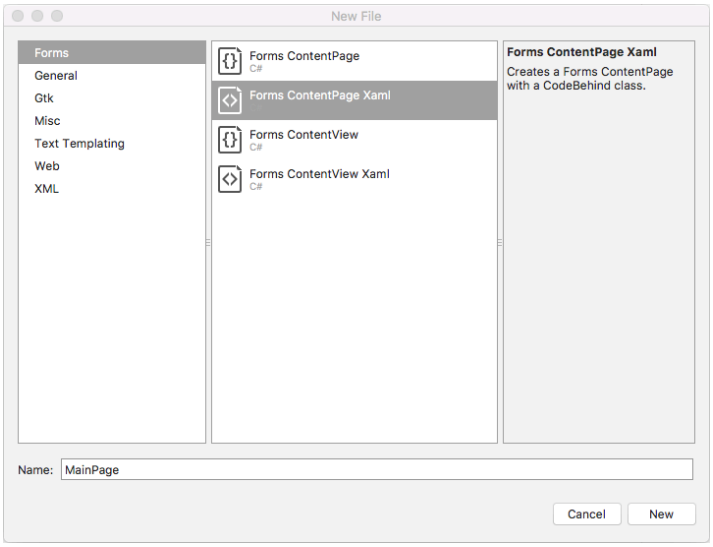
Sample 1: Adding a Main Page

In this section we will change the main page of the application and gradually add elements to it to demonstrate the functionality provided by the Tsl.Core framework.

The first step is to add a new page and call it Main Page. Right click the main project and add a new folder named Views:



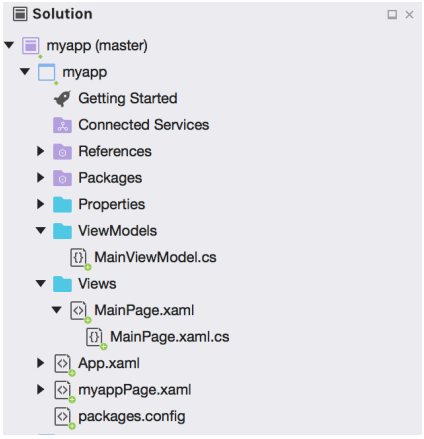
Right click on Views folder, select Add -> new file, and specify the file to be a ContentPage Xaml under the name MainPage:



This will add two files to the main project under Views folder: MainPage.xaml and MainPage.xaml.cs.

The Main Page requires a ViewModel file, so add a new folder named ViewModels and add an Empty Class named MainVeiwModel to that folder.

The final result should look like this:



Edit the source code and make the MainViewModel class inherit from MvvmLight's ViewModelBase:

```
public class MainViewModel : ViewModelBase
  {
     public MainViewModel()
     {
     }
  }
```

Then, Inject an MainViewModel as a depency in App.xaml.cs

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

In the MainPage.xaml.cs file, get the dependency and set it as the binding for the page:

```
      public MainPage()
      {
         InitializeComponent();
         BindingContext = ViewModelLocator.GetDependency<MainViewModel>();
      }
```

The page is pretty empty. Add a Layout and a Label to it for now.

```
<?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">
   <ContentPage.Content>
      <StackLayout Padding="30">
         <Label Text="Here is the First Page of My app" />
      </StackLayout>
   </ContentPage.Content>
</ContentPage>
```

Finally, we will refactor App.xaml.cs to use this page as the main page:

```
    public App()
    {
        InitializeComponent();
        RegisterDependencies();
        MainPage = RegisterPages(new NavigationPage(new MainPage()));
    }
```

And we can go ahead and delete the myappPage.xaml and myappPage.xaml.cs that was created automatically.

Sample 2: Connecting to the handheld reader

Tsl.Core defines a Connect Page that allows for selecting, connecting to and disconnecting from TSL Handheld readers. To get to this page, you need to navigate to it from somewhere in your application. For simplicity, we will add a button to our main page that navigates to this Connect page.

First, replace the Label on the page with a Button and assign the Command for that button to bind to GoToConnectPage:

```
<StackLayout Padding="30">
    <Button Text="Select Reader" Command="{Binding GoToConnectPage}" />
</StackLayout>
```

Then, we need to do a few things in the MainViewModel file:

- Get the INavigationService in the constructor and save it privately
- Add a NavigateToConnectPage method that navigates to the Connect Page
- Create a GoToConnectPage RelayCommand and set it to call the above method

```
public class MainViewModel : ViewModelBase
{
    private readonly INavigationService _navigator;
    public ICommand GoToConnectPage { get; private set; }

    public MainViewModel(INavigationService navigator)
    {
        _navigator = navigator;
        GoToConnectPage = new RelayCommand(NavigateToConnectPage);
    }

    void NavigateToConnectPage()
    {
        _navigator.NavigateTo(ViewModelLocator.ConnectPageKey);
    }
}
```

Now run the app and click on the button. It will navigate to the Connect Page which allows listing, adding, connecting and disconnecting from readers.

Sample 3: Querying the Handheld Reader

The Tsl.Core framework encapsulates the reader functionality using ReaderService singleton which is injected during the registration process. To demonstrate its use, we add a label to our main page to display the name of the connected reader, if any. If no reader is connected, the label would indicate that.

To start, add a Label to the MainPage.xaml and set its text binding to ReaderName property.

```
<StackLayout Padding="30">
   <Label Text="{Binding ReaderName}"/>
   <Button Text="Select Reader" Command="{Binding GoToConnectPage}" />
</StackLayout>
```

To get notified when a new reader is connected, inject the ReaderService to the constructor of MainViewModel and register it to receive ReaderConnection Messages. This is done by calling the RegisterForReaderConnections() method of the ReaderService with two arguments, first the ViewModel and then a delegate to be run when the reader is connected or disconnected.

This delegate function will examine the content of the message it receives. If the content is null, then the reader is disconnected. Else, the content contains a TslReaderInfo object which contains the versions and serial numbers of the reader.

The model then uses this information to correctly update the ReaderName property and have it bind to the Label in the MainPage. The final code looks like this:

```
private readonly ReaderService _readerService;
private string _readerName = "Not Connected";
public string ReaderName
{
   get { return _readerName; }
   private set
   {
      Set(() => ReaderName, ref _readerName, value);
   }
}

public MainViewModel(INavigationService navigator, ReaderService readerService)
{
   _navigator = navigator;
   _readerService = readerService;
   GoToConnectPage = new RelayCommand(NavigateToConnectPage);
   _readerService.RegisterForConnectionEvents(this, msg => {
      ReaderName = msg.Content == null ? null : _readerService.ConnectedReader.DisplayName;
   });
}
```

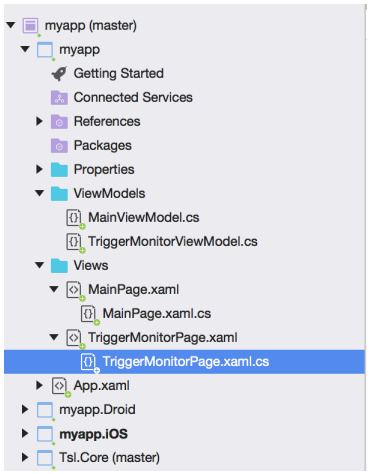
In a production app, the _readerService.UnregisterFromConnectionEvents(this); will be used to remove the event handler when it is no longer needed.

Sample 4: Monitoring the Handheld Trigger Button

The ReaderService exposes functionality to allow for configuration of the trigger switch actions. The trigger switch has two main actions, Single and Double click. By default, Single Click is bound to RFID Inventory and Double Click is bound to Barcode read. As an example, we add another button to the MainPage that navigates to a new page in our app, in which we monitor the Switch if a reader is connected and report the status as it changes.

First, right click on the Views directory and add a Form ContentPage Xaml file with the name TriggerMonitorPage. This will add the two files (xaml and xaml.cs) for us.

Then, right click on the ViewModels directory and add an empty class called TriggerMonitorViewModel. Here is the final file structure:



Now, edit the App.xaml.cs file to register the page to the navigation system and inject its ViewModel:

```
Page RegisterPages(NavigationPage navigationPage)
{
   var navigation = ViewModelLocator.GetDependency<INavigationManager>();

   navigation.SetMain(navigationPage);
   ViewModelLocator.RegisterPages(navigation);

   navigation.Register(TriggerMonitorPage.PageKey, typeof(TriggerMonitorPage));

   return navigationPage;
}

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

Add a static PageKey string to the page and bind the context in TriggerMonitorPage.xaml.cs

```
public partial class TriggerMonitorPage : ContentPage
{
    public static readonly string PageKey = "TriggerMonitorPage";
    public TriggerMonitorPage()
    {
        InitializeComponent();
        BindingContext = ViewModelLocator.GetDependency<TriggerMonitorViewModel>();
    }
}
```

Now add a button to MainPage.xaml and set the Command to GoToTriggerMonitor:

```
<ContentPage.Content>
    <StackLayout Padding="30">
        <Label Text="{Binding ReaderName}" />
        <Button Text="Select Reader" Command="{Binding GoToConnectPage}" />
        <Button Text="Monitor Trigger Switch" Command="{Binding GoToTriggerMonitorPage}" />
    </StackLayout>
</ContentPage.Content>
```

And in the MainPage define the Command and the method to navigate and assign them to each other:

```
public ICommand GoToTriggerMonitorPage { get; private set; }

public MainViewModel(INavigationService navigator, ReaderService readerService)
{
    _navigator = navigator;
[….]
    GoToTriggerMonitorPage = new RelayCommand(NavigateToTriggerMonitorPage);
[…]
}

void NavigateToTriggerMonitorPage()
{
    _navigator.NavigateTo(TriggerMonitorPage.PageKey);
}
```

Now, add a Label to the TriggerMonitorPage.xaml to show the state of the trigger and bind it to property Trigger Action:

```
<?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.TriggerMonitorPage">
    <ContentPage.Content>
        <Label Text="{Binding TriggerAction}" />
    </ContentPage.Content>
</ContentPage>
```

ReaderService exposes a SwitchConfig object, which has the following parameters:

Parameter	Type	Description
AsyncReporting	Boolean	When true, AsyncReportHandler will be fired each time the trigger is pressed.
HapticFeedback	Boolean	When true, there would be a haptic feedback (vibration) when the trigger is pressed.
Double Press	Enum	Action to perform on double click of trigger. Can be one of: Off, Barcode, Inventory, Read, Write, AntennaDefault or User.
Single Press	Enum	Action to perform on single click of trigger. Same options as Double Press.
SwitchStateChanged	EventHandler	Triggered when switch changes state.

Here is what needs to be done in TriggerMonitorViewModel

- It has to derive from ViewModelBase
- It has to get ReaderService in constructor and store it privately
- It has to register to ReaderConnectionEvents and when a reader is connected configure the trigger button behavior
- It has to define a TriggerAction property that would bind to the Label and show the state of trigger.

Here is how the code looks like. Note the use of AsyncReporting and SwitchStateChanged:

```
public class TriggerMonitorViewModel : ViewModelBase
{
   ReaderService _reader;

   string _triggerAction = "Not Connected";
   public string TriggerAction
   {
      get { return _triggerAction; }
      set
      {
         Set(() => TriggerAction, ref _triggerAction, value);
      }
   }

   public TriggerMonitorViewModel(ReaderService reader)
   {
      _reader = reader;
      _reader.RegisterForConnectionEvents(this, msg =>
      {
         if (msg == null) TriggerAction = "Not Connected";
         else RegisterSwitch(_reader);
      });
   }

   void RegisterSwitch(ReaderService reader)
   {
      TriggerAction = "Not Started";
      reader.SwitchConfig.AsyncReporting = true;
```

```
        reader.SwitchConfig.SwitchStateChanged = (sender, e) =>
        {
            switch (e.State)
            {
                case SwitchState.Double:
                    TriggerAction = "Double Click";
                    break;
                case SwitchState.Single:
                    TriggerAction = "Single Click";
                    break;
                case SwitchState.Off:
                    TriggerAction = "Off";
                    break;
            };
        };
        reader.Configure(reader.SwitchConfig);
    }
}
```

Sample 5: Adding a Slider to Change Antenna Output Power

Once again, we add a new button to the MainPage to take us to a new InventoryPage. The initialization steps are similar to the previous sample, so they are just listed here:

- Add a InventoryPage.xaml to the Views folder (which will add the xaml.cs file too)
- Add a InventoryPageViewModel to the ViewModels folder
- Register the InventoryPageViewModel class with the Dependency injector
- Register the InventoryPage with the Navigation manager
- Add a button and a command to MainPage to navigate to the Inventory Page

We now add to the InventoryPage.xaml a Label to show the power level, and a slider to change it, and a button to send the changes to the reader. Since we want the slider to jump every 0.5 db, we will give it a name:

```
<StackLayout Padding="30">
    <Label Text = "{Binding OutputPower, StringFormat='{0:N1} dBm'}"/>
    <Slider Maximum="30" Minimum="10" Value="{Binding OutputPower}" x:Name="PowerSlider"/>
    <Button Text = "Update Reader" Command="{Binding UpdateConfig}" />
</StackLayout>
```

In the code behind (InventoryPage.xaml.cs), we will jump the slider value at 0.5 dBm intervals when its value changes.

```
public InventoryPage()
{
    var powerStep = 0.5;
    InitializeComponent();

    PowerSlider.ValueChanged += (sender, e) => {
        var newStep = Math.Round(e.NewValue / powerStep);
```

```
        PowerSlider.Value = newStep * powerStep;
    };
    BindingContext = ViewModelLocator.GetDependency<InventoryModelView>();
}
```

In the ViewModel, we create the OutputPower variable and the UpdateConfig command. We make the button disabled unless updating the config makes sense, i.e.

- The first time user enters this page
- After a change in the value of the slider if a reader is connected
- Each time a new reader is connected.

```
private readonly ReaderService _readerService;
public ICommand UpdateConfig { get; private set; }

private bool _shouldUpdate = true;
public bool ShouldUpdate
{
    get { return _shouldUpdate;}
    set
    {
        _shouldUpdate = value;
        (UpdateConfig as RelayCommand).RaiseCanExecuteChanged();
    }
}

private double _outputPower = 29.0;
public double OutputPower
{
    get { return _outputPower; }
    set
    {
        if (Set(() => OutputPower, ref _outputPower, value))
            ShouldUpdate = true;
    }
}

public InventoryModelView(ReaderService readerService)
{
    _readerService = readerService;

    UpdateConfig = new RelayCommand(ConfigReader,
                                () => _readerService.ConnectedReader != null && ShouldUpdate);

    _readerService.RegisterForConnectionEvents(this, msg =>
    {
        ShouldUpdate = true;
    });
}

async void ConfigReader()
{
    _readerService.InventoryConfig.Power = Convert.ToInt32(OutputPower);
    _readerService.InventoryConfig.IncludeRssi = true;
    await _readerService.Configure(_readerService.InventoryConfig);
```

```
    ShouldUpdate = false;
}
```

Note that while we allow for 0.5 dBm step in power, TSL driver only accepts integer values, so we have used Convert.ToInt32() at the end. Nevertheless, the example is aimed at showing the value of the code-behind xaml.cs file in adding functionality to the page.

Sample 6: Reading and Displaying a RAIN RFID Tag EPC

In this sample we will add two Labels to our Inventory Page to show the Latest EPC that was read and the number of scans attempted. While we are at it, we refactor our slider to read the maximum and minimum values from the framework.

```
<Label Text="{Binding OutputPower, StringFormat='{0:N1} dBm'}" />
<Slider Maximum="{Binding MaxOutputPower}"
        Minimum="{Binding MinOutputPower}"
        Value="{Binding OutputPower}"
        x:Name="PowerSlider" />
<Button Text="Update Reader" Command="{Binding UpdateConfig}" />
<Label Text="{Binding ScanCount, StringFormat='Scan Count: {0:N1}'}"/>
<Label Text="{Binding LastEpc, StringFormat='Last EPC: {0}'}"/>
```

We then will add the new bindings to our InventoryViewModel. For Max and MinOutputPower, the corresponding read-only properties of the InventoryConfig are used. For receiving scan data we register an event handler with ReaderService, and when we get data we see if there are any tags (and if so assign the last one to our LastEpc property). We also check if the Complete flag is true which means a scan round is done.

```
public int MaxOutputPower
{
    get { return _readerService.InventoryConfig.MaxOutputPower; }
}
public int MinOutputPower
{
    get { return _readerService.InventoryConfig.MinOutputPower; }
}

private string _lastEpc = "";
public string LastEpc
{
    get { return _lastEpc; }
    set { Set(() => LastEpc, ref _lastEpc, value); }
}

private int _scanCount = 0;
public int ScanCount
{
    get { return _scanCount; }
    set { Set(() => ScanCount, ref _scanCount, value); }
}
```

```
public InventoryModelView(ReaderService readerService)
{
    _readerService = readerService;
[…….]
    _readerService.RegisterTagListener((sender, e) => {
        var i = e.Tags.Count();
        if (i > 0)
            LastEpc = e.Tags.ElementAt(i - 1).Epc;
        if (e.Complete)
            ScanCount += 1;
    });
}
```

The _readerService also exposes an UnregisterTagListener to remove a listener from tag events.

Sample 7: Reading and Displaying a Barcode

Reading Barcodes is essentially the same as reading tags. When the user double clicks the trigger button, the barcode reader is activated and gives the user 9 seconds to read a barcode. It will abort if a barcode is not read in 9 seconds. This timeout is configurable from 1 to 9 seconds.

ReaderService exposes a BarcodeConfig to set these parameters. The Configure() method is overloaded to accept a BarcodeConfig and send it to the reader. ReaderService also has two methods, RegisterBarcodeListener() and UnregisterBarcodeListener(), which allow a class to listen for barcode events.

In order to demonstrate these, we will add another Label to our xaml file and bind it to a LastBarcode property:

```
<Label Text="{Binding LastBarcode, StringFormat='Last Barcode: {0}'}"/>
```

Then we add the LastBarcode Property to our InventoryViewModel and register the listener to fire when a new barcode is scanned.

```
private string _lastBarcode = "";
public string LastBarcode
{
    get { return _lastBarcode; }
    set { Set(() => LastBarcode, ref _lastBarcode, value); }
}

public InventoryModelView(ReaderService readerService)
{
    _readerService = readerService;
[…….]
    _readerService.RegisterBarcodeListener((sender, e) =>
    {
        if (!String.IsNullOrWhiteSpace(e.Barcode))
            LastBarcode = e.Barcode;
```

```
    });
}
```

There is also an UnregisterBarcodeListener to remove the listener from the queue if the object is disposed.

Sample 8: Barcode scanning and EPC inventory without Trigger

Sometimes it is desirable to put the reader into barcode scanning or inventory mode without waiting for user to press or double click the trigger. ReaderService exposes two methods for that, one is GetBarcode() that forces a barcode scan, and the other is GetInventory() that forces an Inventory round.

To demonstrate them, let's add two buttons to our InventoryPage.xaml:

```
<Button Text="Run One Inventory" Command = "{Binding RunInventory}"/>
<Label Text="{Binding LastBarcode, StringFormat='Last Barcode: {0}'}"/>
<Button Text="Scan a Barcode" Command="{Binding ScanBarcode}"/>
```

Then we define the new commands in the InventoryViewModel file:

```
public ICommand ScanBarcode { get; private set; }
public ICommand RunInventory { get; private set; }
```

And set them in the constructor. We use lambda functions here to define less functions and show the point. The first function is executed when the button is clicked, the button will be disabled if hasReader() function returns false.

```
Func<bool> hasReader = () => _readerService.ConnectedReader != null ;

ScanBarcode = new RelayCommand(() => _readerService.GetBarcode(_readerService.BarcodeConfig),
                        hasReader);

RunInventory = new RelayCommand(() =>_readerService.GetInventory(_readerService.InventoryConfig),
                         hasReader);
```

ViewModelLocator Reference:

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 pages with the Navigation Service

ConnectPageKey	Static string to allow navigation to Connect Page
Register<T>()	Static method to allow for registering a dependency
GetDependency<T>()	Static method to get a dependency previously registered
ReaderService	Used to encapsulate communication with Handheld
TslReaderInfo	Information about the handheld, such as it's version and serial numbers

ReaderService Reference:

ReaderService is the main object for encapsulating the reader. It is the object that dispatches Messages and exposes an api for reader communication. It is injected by the ViewModelLocator so classes can ask for it by injecting in their constructor or calling GetDependency method of ViewModelLocator.

ReaderService exposes the following Properties:

Type	Name	Description
InventoryConfig	InventoryConfig	Contains the settings for the inventory operation (e.g power)
SwitchConfig	SwitchConfig	Contains the settings for the Trigger switch configuration
BarcodeConfig	BarcodeConfig	Contains the settings for barcode scanning functionality
INamedReader	ConnectedReader	The current connected reader or null if no reader is connected.
TstReaderInfo	ReaderInfo	Information on current connect reader serial numbers and versions, or null if no reader is connected.

It also exposes the following methods:

Return	Name	Description
void	RegisterForConnectionEvents (recipient, handler)	Registers a recipient for connection events and calls the handler when events happen
void	UnregisterFromConnectionEvents (recipient)	Unregisters the recipient from connection events.
void	RegisterTagListener(handler)	Registers handler to be called when a RAIN tag is read.
void	UnregisterTagListener(handler)	Removes the handler form chain for RAIN tag discovery.

void	RegisterBarcodeListener (handler)	Registers handler to be called when a Barcode is read.
void	UnregisterBarcodeListener (handler)	Removes the handler from chain for Barcodes found.
void	GetBarcode(BarcodeConfig)	Runs one barcode scan using the parameters specified by the config. Will call the handlers registered with RegisterBarcodeListener() if a barcode is found.
void	GetInventory(InventoryConfig)	Runs one inventory scan using the parameters specified by the config. Will call the handlers registered with RegisterTagListener() with tags found and scan completed messages.

Feature Road Map:

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	5	Sets the switch monitor, the barcode and the 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 serialization / deserialization of json objects