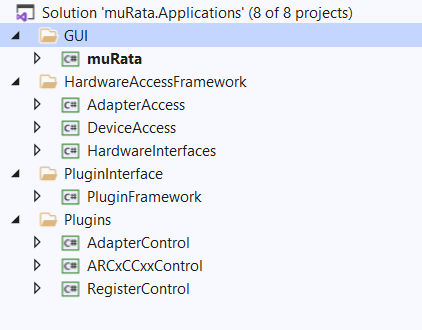
**MuRata Apps:**

**Solution** : muRata.Applications.sln

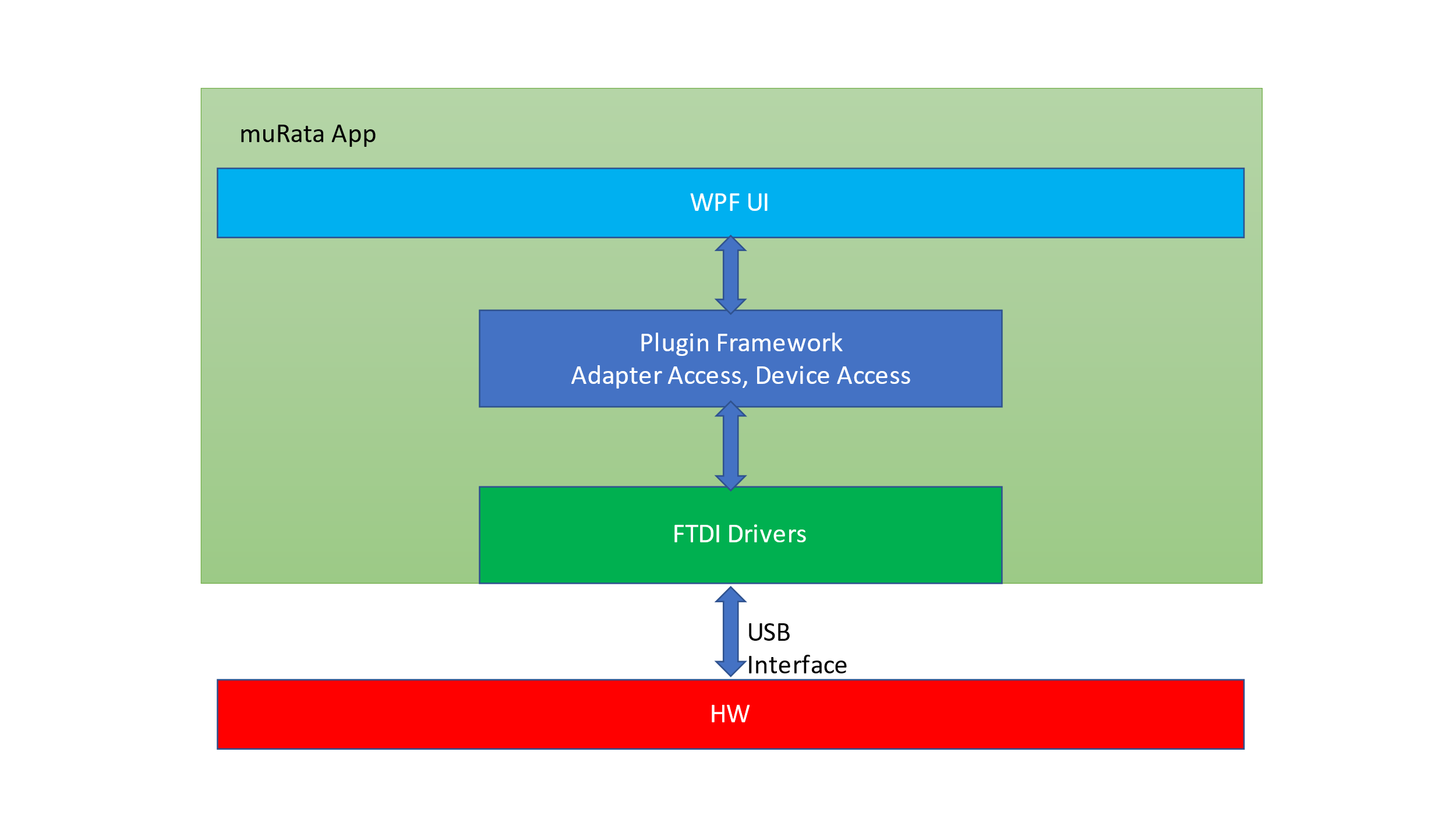
(Path : psemi\HEAD\Solutions\muRata.Applications\ muRata.Applications.sln)

**muRata.Applications.sln -** **WPF MVVMLight Design pattern by GalaSoft.**

1. **GUI** – Represents the user Interface layer of MuRata app.
2. **HardwareAccessFramework** – Specifically for adapter access, device access and hardware interfaces
3. **PluginInterface** – Consist of the PluginFramework
4. **Plugins** – For AdapterControl, ARCxCCxxControl, RegisterControl



**Architecture:**

**Supported Protocols :**

1. I2C
2. PMBus

**Adapters**:

Aardvark

FTDI

TDI

PMBob

Aardvark

PMBus

I2C

PMBus

I2C

I2C

VirtualAdapter – Used for demo purpose (Supports I2C protocol)

**FTDI Wrappers**:

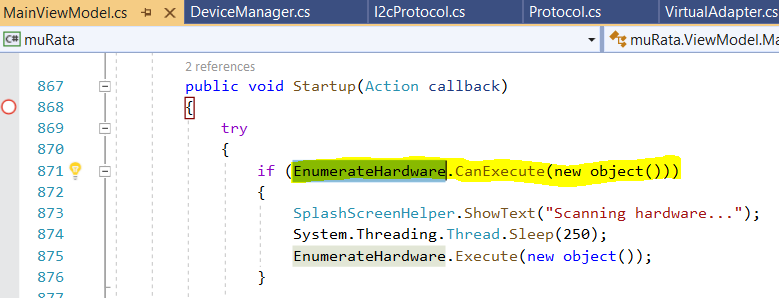
FTD2XX\_NET.cs – I2C

FTD2XX\_NET\_PMBus.cs – PMBus

All the necessary xml files are located at "… psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Devices\\Application.adz" in the format of Compressed Amega Disk File.

**Code Flow**

**1) Enumerate through hardware in order to get the adapter info and devices info.**



**EnumerateHardware is responsible for loading the Active Adapter, Active Device info’s**

appConfig = "… psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Devices\\Application.adz" -

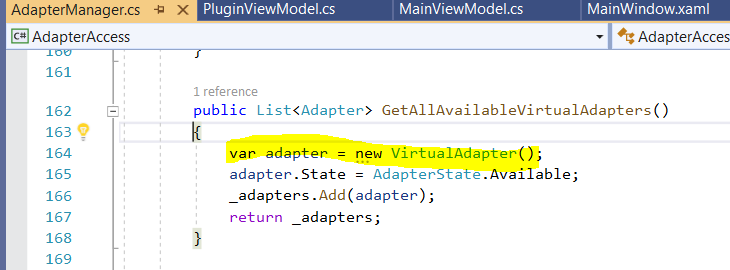
"AppConfig.xml";

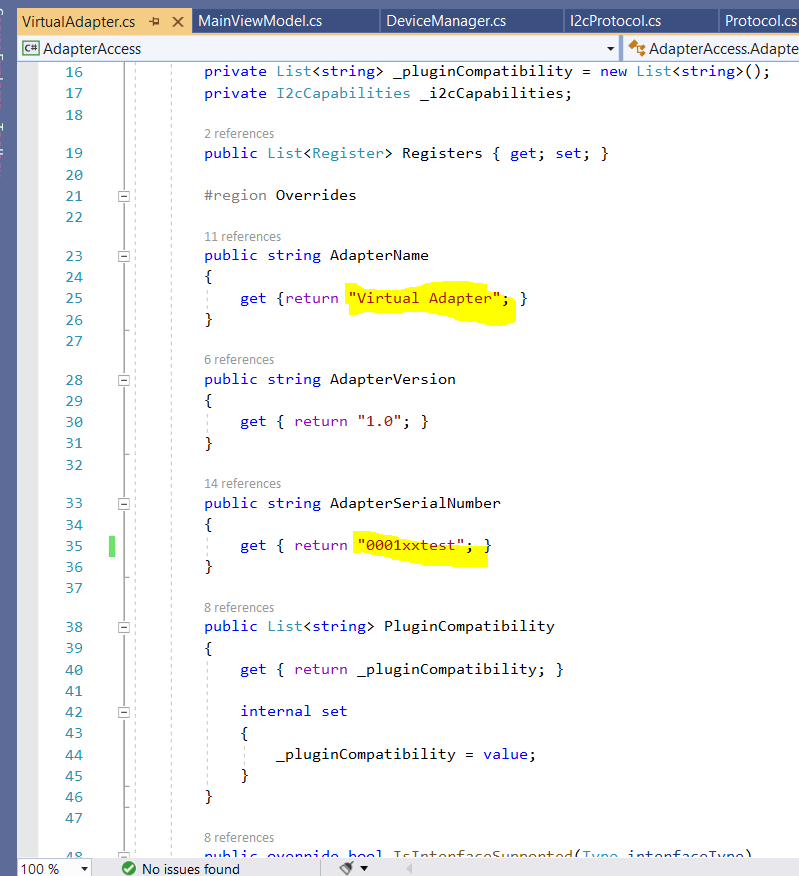
\_appManager.DeviceInfo = appConfig.Descendants("Device");

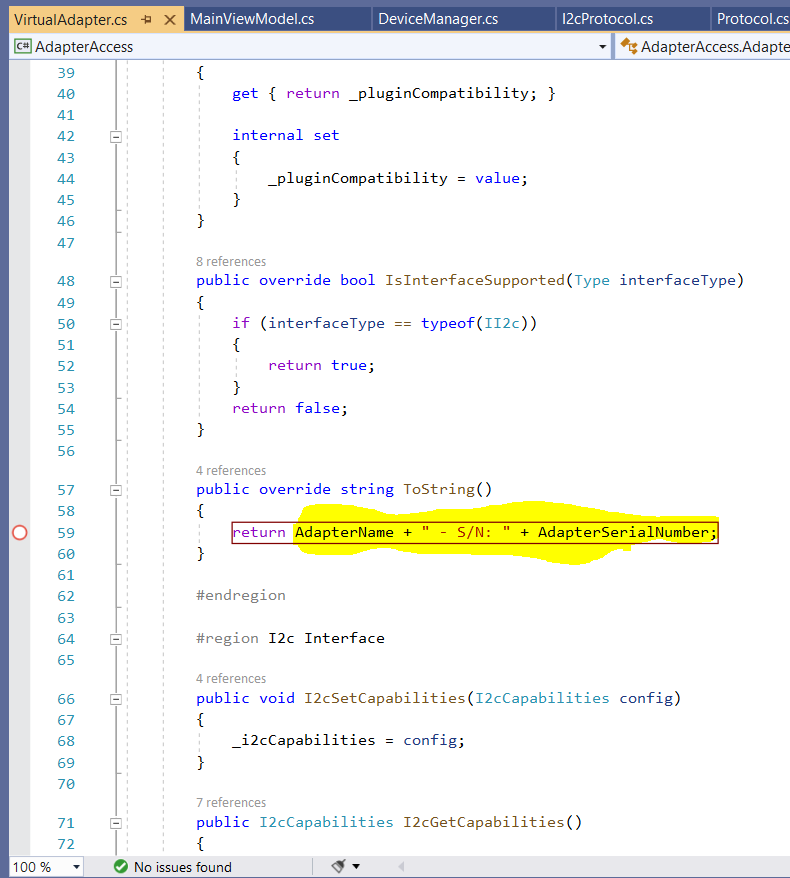
\_appManager.IdentityRegisters = appConfig.Descendants("Setting");

\_deviceManager = new DeviceManager(\_adapterManager, \_isInternalMode, \_appManager.DeviceInfo, \_appManager.IdentityRegisters);

**Adapters will be obtained from :**







if (\_isDemoMode)

{

\_availableAdapters = \_adapterManager.GetAllAvailableVirtualAdapters();

}

else

\_availableAdapters = \_adapterManager.GetAllAvailableAdapters(true, IncludeI2c, IncludePmBus);

}

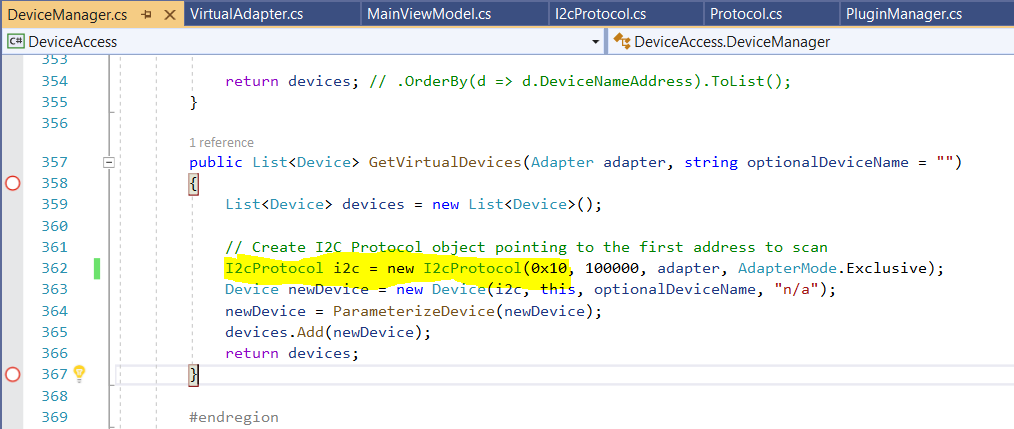
**Devices will be obtained from :**

if (\_isDemoMode)

{

\_availableDevices = \_deviceManager.GetVirtualDevices(a, DeviceInfoName);

}



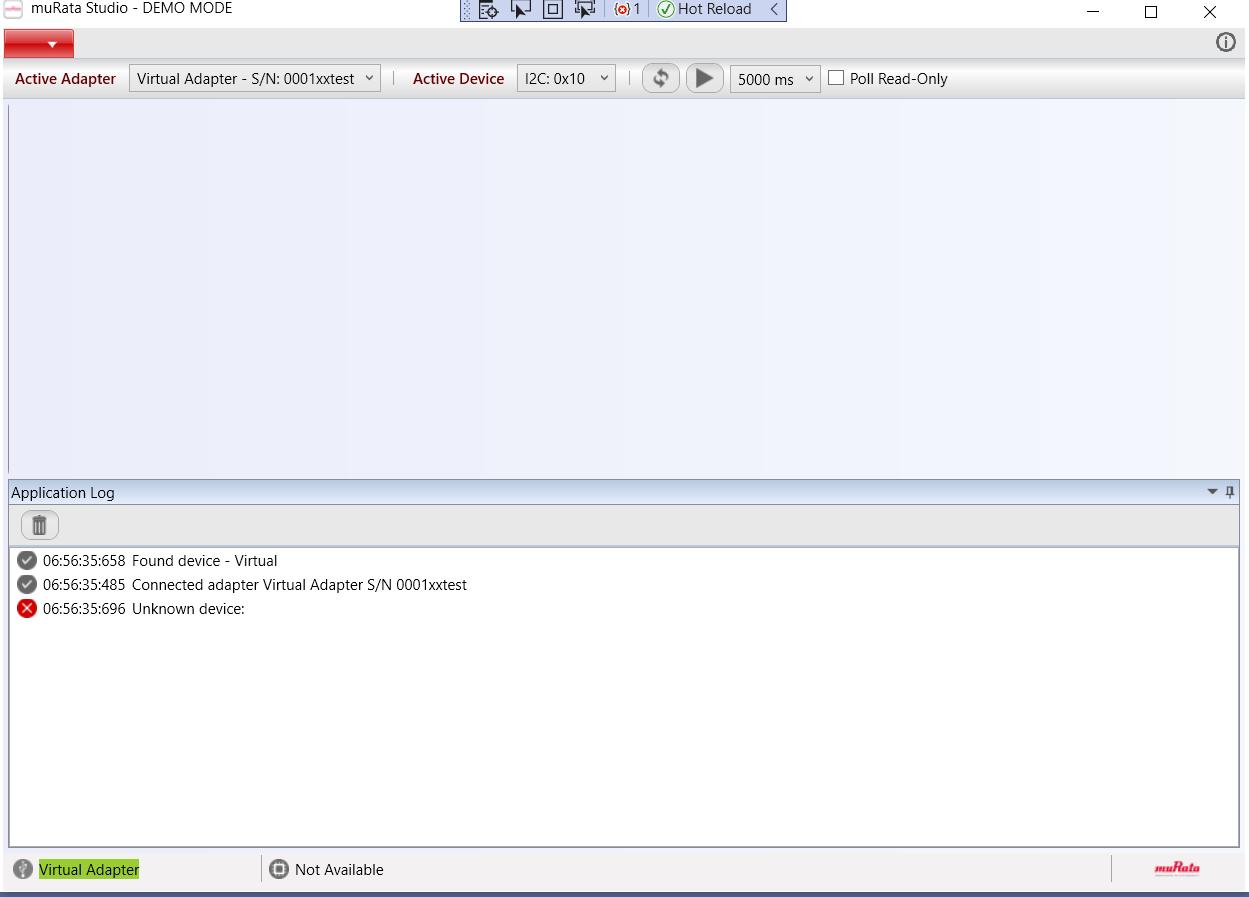
Adapter info and available device info is hard corded from the code section.

ObservableDictionary<Adapter, List<Device>> AdaptersAttachedDevices

ActiveAdapter = (IAdapter)AdaptersAttachedDevices.Keys. ElementAt (0);

AdaptersAttachedDevices.ElementAt(0). Value.ForEach(d => Devices.Add(d));

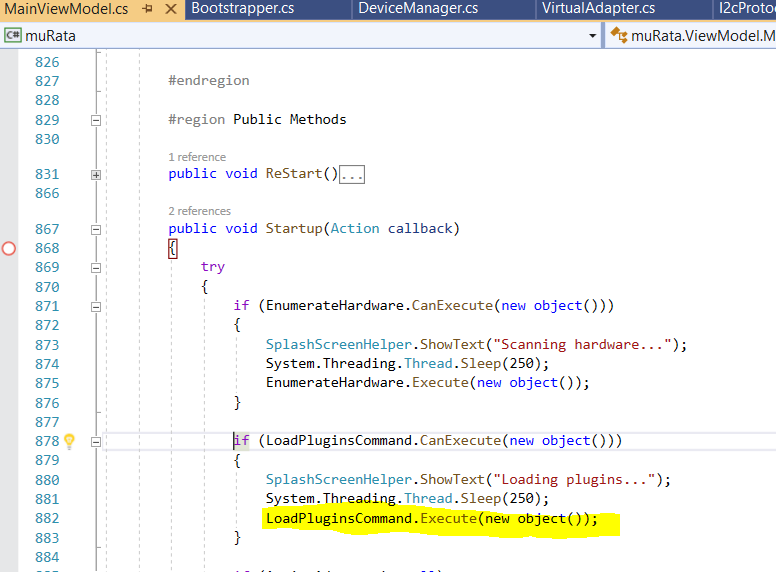
ActiveDevice = Devices [0];

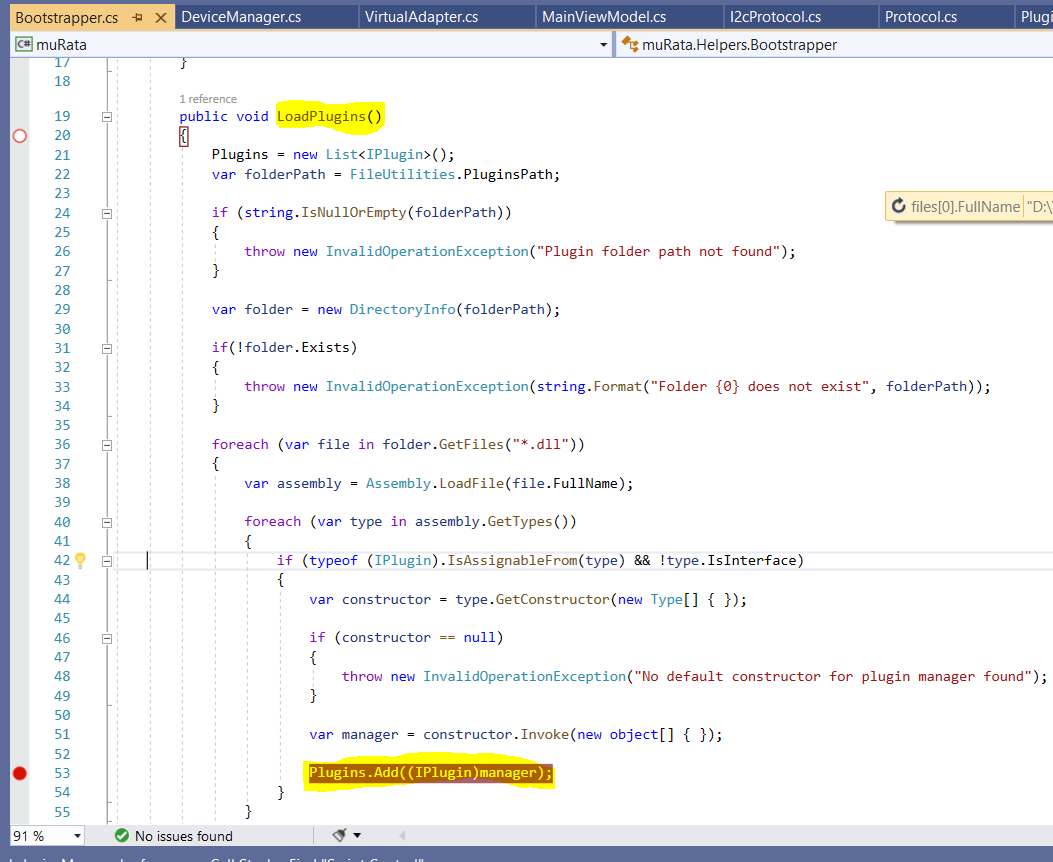


**2)Load the PlugIns**

Plugins are loaded during application StartUp as shown below.

"……\\psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Devices\\Application.adz" - "Adapters.xml"





FileUtilities.PluginsPath : "... [\\psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Plugins](../../../..//psemi/HEAD/Apps/muRata/bin/Debug/Plugins)"

All the below three plugins are loaded :

1. AdapterControl.dll
2. ARCxCCxxControl.dll
3. RegisterControl.dll

Adzfile.FullName = "……\ \psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Devices\\Application.adz"

using (ZipFile zip = ZipFile.Read(adzfile.FullName))

{

ZipEntry entry = zip[adaptersFile];

entry.Extract(ms);

ms.Position = 0;

appConfig = XDocument.Load(ms);

IEnumerable<XElement> adapters = appConfig.Descendants("Adapter");

….

}

XElement thisAdapter = adapters.FirstOrDefault(a => a.Attribute("Name").Value == iAdapter.AdapterName);

………………….………………….………………….………………….………………….………………….………………….

**3)Load the corresponding plugin based on PluginCompatibility control .**

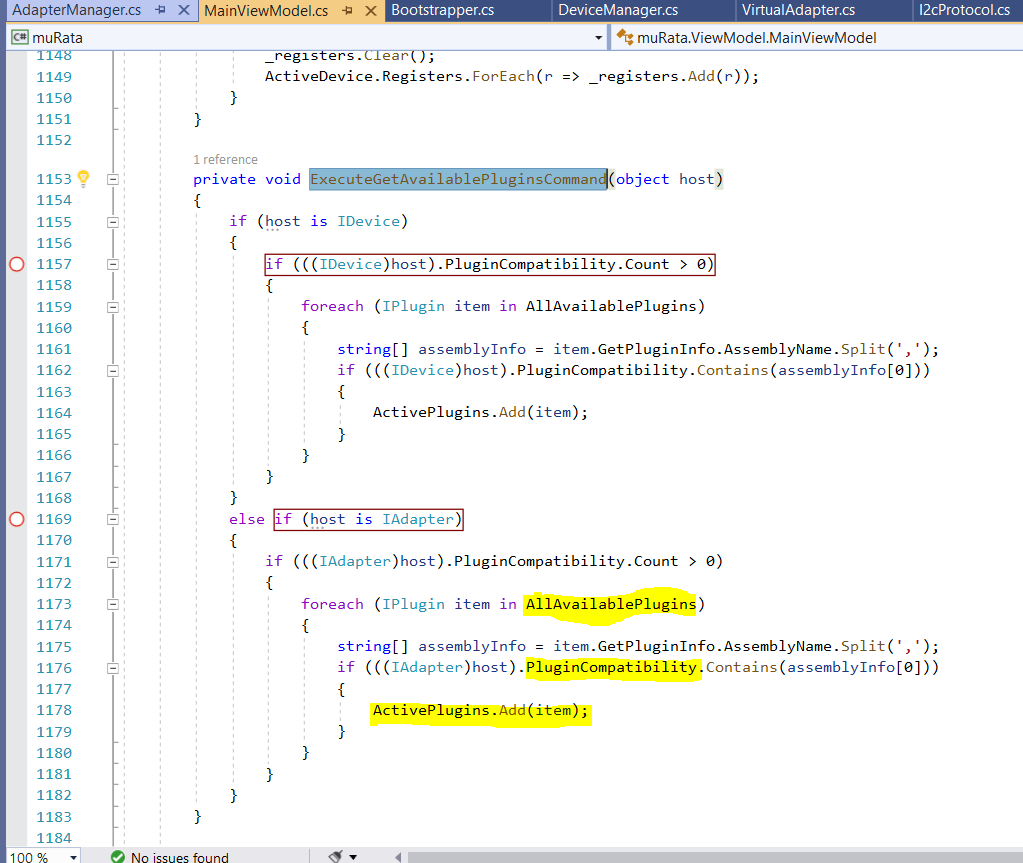
IEnumerable<XElement> plugins = thisAdapter.Descendants("Plugin");

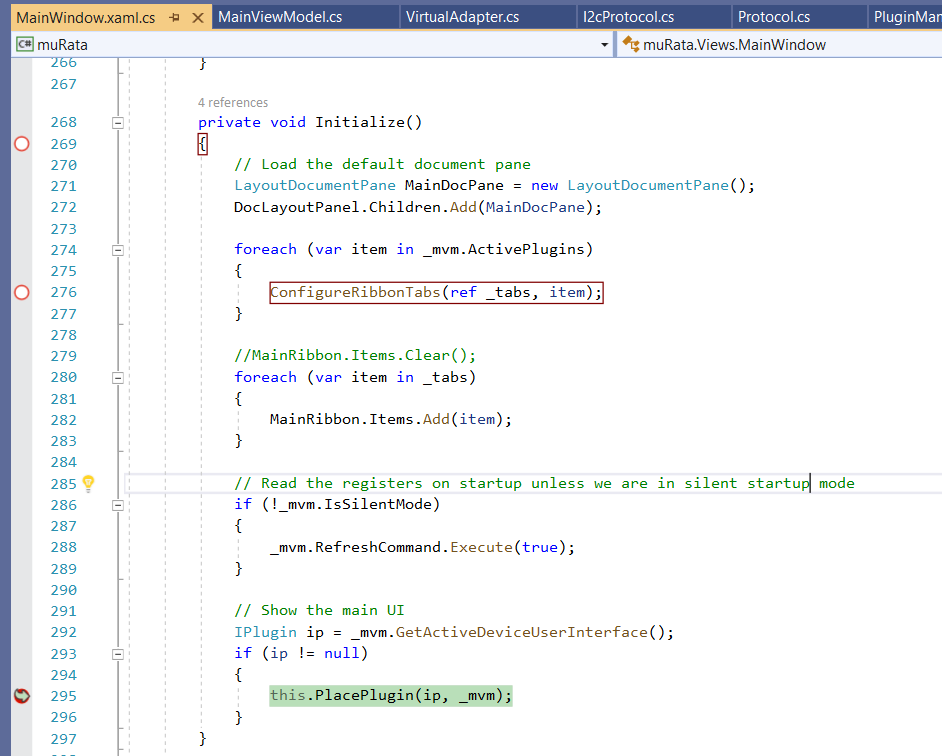
foreach (string plugin in plugins)

{

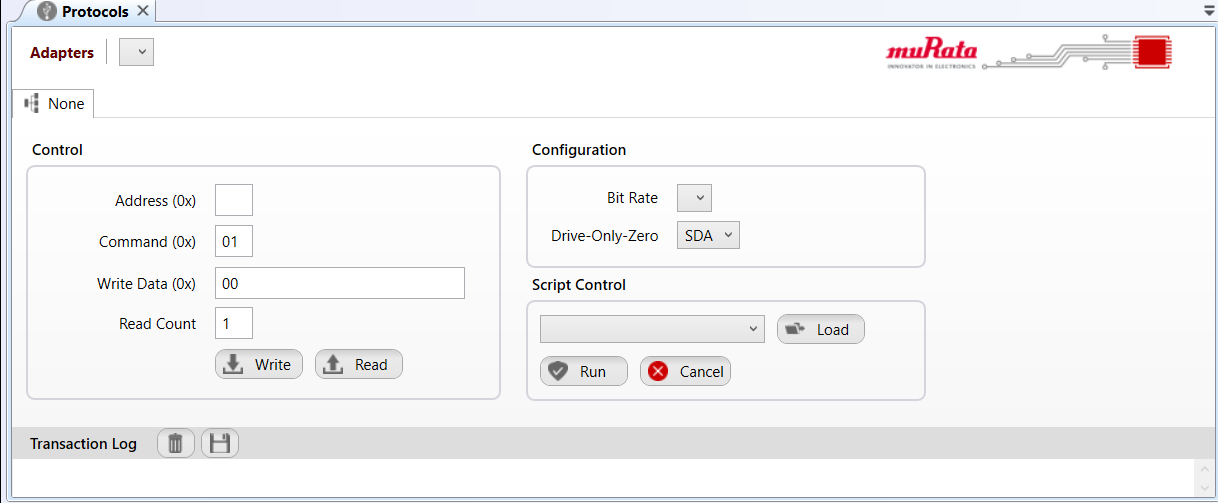
iAdapter.PluginCompatibility.Add(plugin);

}





All the values in the Control, Configuration and Script Control section will be loaded from the PluginViewModel of the corresponding control properties binded to xaml file.



**Control**:

Address(0X) value corresponds to I2cAddress in PluginViewModel

Command (0X) value corresponds to RegisterAddress in PluginViewModel

WriteData(0X) value corresponds to Data in PluginViewModel

ReadCount value corresponds to ReadCount in PluginViewModel

**Configuration:**

Bit rate value corresponds to BitRateDescriptions in PluginViewModel

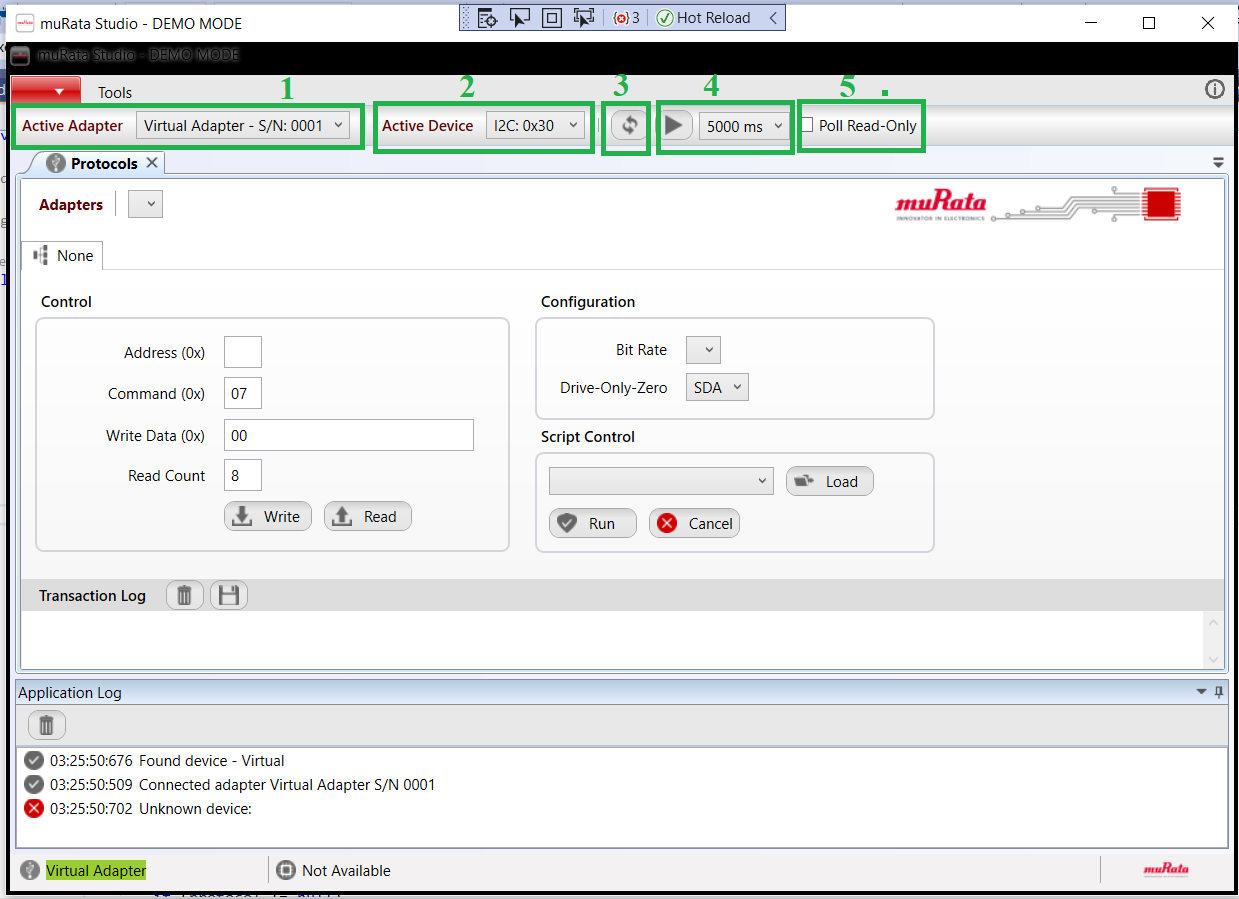
Drive-Only-Zero value corresponds to hard coded combo box items (OFF, SCL, SDA ,Both SCL and SDA)

**Script Control:**

Combobox loads the Scripts in PluginViewModel

LoadScriptCommand load the scripts from the Path : “…..\\psemi\\HEAD\\Apps\\muRata\\bin\\Debug\\Plugins\\Scripts"

**Virtual Adapter UI :**



1) Active Adapter: Display the active adapter name. Can be changed from the drop-down menu

2) Active Device: Display the active device name. Can be changed from the drop-down menu

3) Refresh: Button click will refresh the current connection by reading all the registers again.

4) Polling Interval in milliseconds

Play button: TogglePollingCommand

Drop down list: Lists four interval timings (500, 1000,2000,5000). Can be changed from the drop-down menu

5) Poll Read only: Indicates that the poll status is read only (Not sure about this)

Flow Chart for device detection

Initialize

Scan Hardware

Enumerate USB Devices

Found USB Device//  
Adapter

Load Plugins

Select Adapter Plugins

Select Device Plugins

Connect to the Adapter

Load data into the GUI controls

Load GUI

Yes

No

**Drive-Only-Zero**

Enabling Drive-Only-Zero ensures that the SDA line is driven by the I2C master only when it is supposed to be driven LOW, and tristateit when it is supposed to be driven HIGH. This feature is available only in FT232H chip. Trying to enable this feature using function I2C\_Init will have no effect on chips other than FT232H

**What is SDA and SCL pins?**

This is just two wires, called **SCL** and **SDA**. **SCL** is the clock line. It is used to synchronize all data transfers over the I2C bus. **SDA** is the data line. The **SCL** & **SDA** lines are connected to all devices on the I2C bus.

**Address**Memory address of the device

**Command**Command to be sent to the device

**Write Data**Data to be written to the device

**Read count**Number of bytes to read?

**Bitrate**  
Bitrate for read//write operations

**Poll**  
Check if the device is ready..

**Poll read Only**Does not send any data to device while polling

**Refresh**Refresh the connection

**Script Control**Read or write data in a batch using a xml script file.

**Transaction Log**  
Log of the read//write operations

**Application Log**Log about adapters and devices. Like if an adapter and a device is found. Connection status etc.

**Script Control Method**

The following method persforms multiple read/write operations using a xml script.

private void RunScript(string parameter)

{

if (parameter == null)

{

return;

}

\_cancelSource = new CancellationTokenSource();

Script script = \_scriptMananger.Scripts.Find(s => s.Name == parameter);

if (script == null)

return;

foreach (var item in script.Actions)

{

switch (item.Operation)

{

case OperationType.Delay :

var delayAction = item as DelayAction;

Thread.Sleep(delayAction.WaitMs);

break;

case OperationType.Read :

case OperationType.Write :

var transaction = item as Transaction;

// Pre-set the target address to the active adapter

byte addr = \_activeProtocol.TargetAddress;

// Reassign the target address if it is specified in the script

if (script.SlaveAddress.HasValue)

{

addr = (byte)script.SlaveAddress;

}

for (int i = 0; i < transaction.RepeatCount; i++)

{

if (item.Operation == OperationType.Write)

{

WriteBlock(addr, transaction.Command, transaction.Data);

}

else

{

ReadBlock(addr, transaction.Command, transaction.Length);

}

Thread.Sleep(transaction.InnerDelayMs);

if (\_cancelSource.IsCancellationRequested)

{

return;

}

}

break;

}

}

}

**Write block method**Writes a block to the device

private void WriteBlock(byte slaveAddress, byte registerAddress, byte[] data)

{

string dt = DateTime.Now.ToString("hh:mm:ss:fff");

try

{

\_activeProtocol.TargetAddress = slaveAddress;

\_activeProtocol.WriteBlock(slaveAddress, registerAddress, data.Length, ref data);

Dispatcher.BeginInvoke(

(Action)(() =>

{

LogEntries.Insert(0, new LogEntry()

{

DateTime = dt,

ProtocolName = \_activeProtocolType.ToString(),

Direction = "W",

Address = slaveAddress.ToString("X2"),

Register = registerAddress.ToString("X2"),

Length = data.Length.ToString().PadLeft(2, '0'),

ForeColor = \_writeColor,

Data = BitConverter.ToString(data)

});

}));

}

catch (Exception ex)

{

Dispatcher.BeginInvoke(

(Action)(() =>

{

LogEntries.Insert(0, new LogEntry()

{

DateTime = dt,

ProtocolName = \_activeProtocolType.ToString(),

Direction = "W",

Address = slaveAddress.ToString("X2"),

Register = registerAddress.ToString("X2"),

Length = "00",

ForeColor = \_errorColor,

Data = ex.Message

});

}));

}

return;

}

**Read block method**Reads a block from the device

private void ReadBlock(byte slaveAddress, byte registerAddress, int length)

{

//string dt = DateTime.Now.ToString("yyyy-dd-MM hh:mm:ss:fff");

string dt = DateTime.Now.ToString("hh:mm:ss:fff");

try

{

byte[] data = new byte[length];

\_activeProtocol.TargetAddress = slaveAddress;

\_activeProtocol.ReadBlock(registerAddress, length, ref data);

Dispatcher.BeginInvoke(

(Action)(() =>

{

LogEntries.Insert(0, new LogEntry()

{

DateTime = dt,

ProtocolName = \_activeProtocolType.ToString(),

Direction = "R",

Address = slaveAddress.ToString("X2"),

Register = registerAddress.ToString("X2"),

Length = length.ToString().PadLeft(2,'0'),

ForeColor = \_readColor,

Data = BitConverter.ToString(data)

});

}));

}

catch (Exception ex)

{

Dispatcher.BeginInvoke(

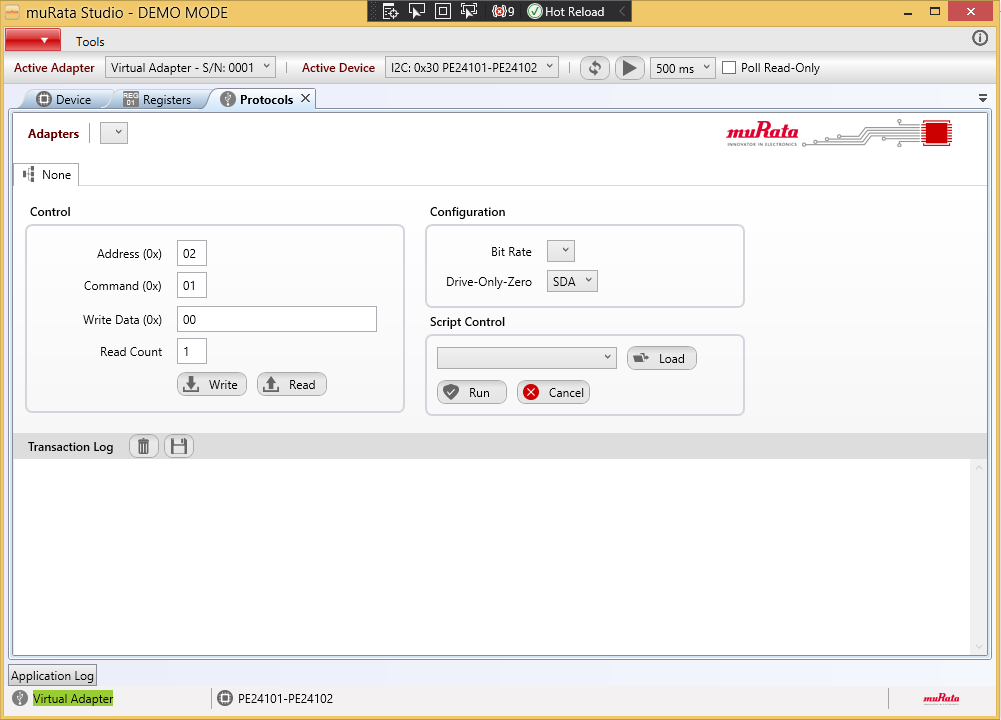
(Action)(() =>

{

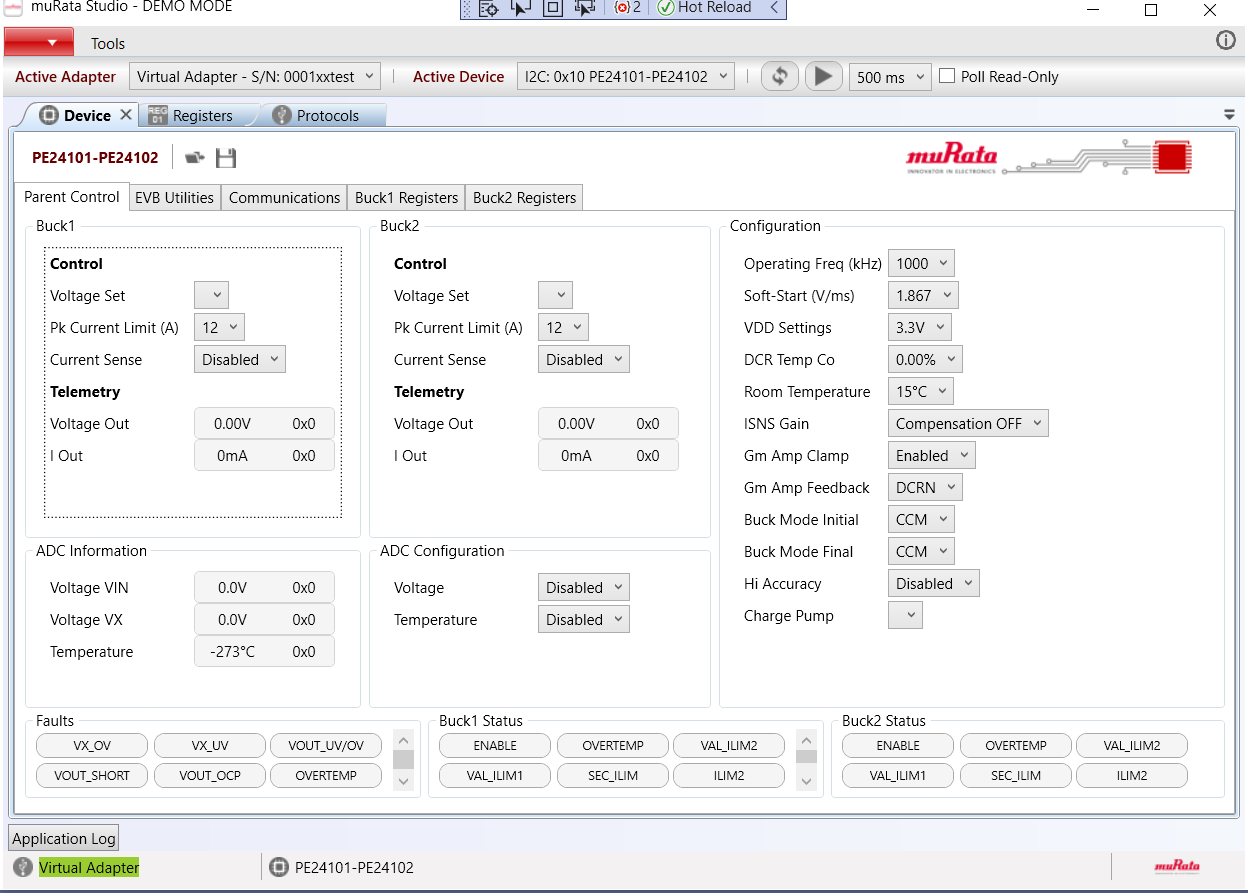
LogEntries.Insert(0, new LogEntry()

We are now able to see all the tabs that is Protocols, Devices and Registers

**Protocols Tab**



**Devices Tab**

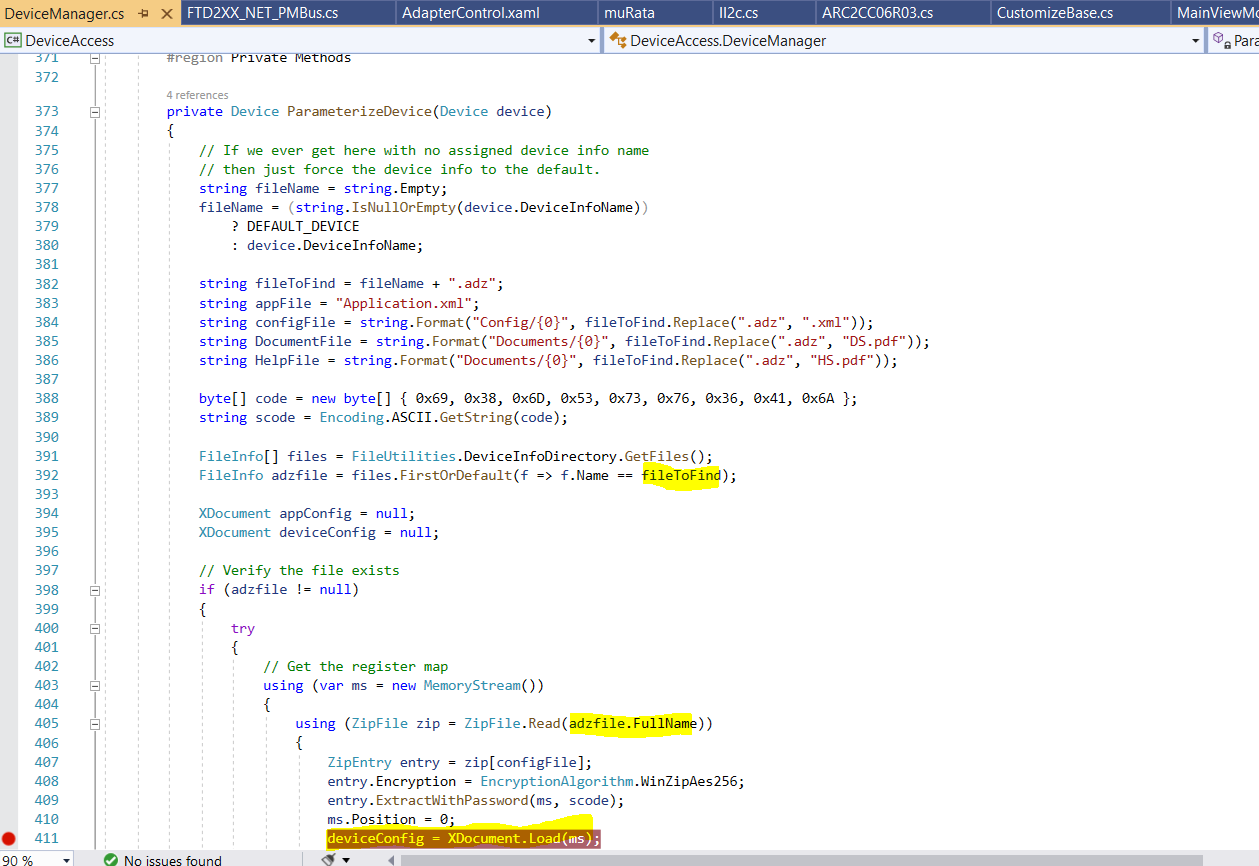


The UI displays the information regarding the device read from the configuration file.

psemi\HEAD\Apps\muRata\bin\Debug\Devices\ARC2CC06-R03.adz”.

**Loading config file for ARC2CC06-R03**

The files are loaded via ParameterizeDevice() method in DeviceManager.cs file



Each device have UiElements in Device.cs property which will be filled from the corresponding device configuration file

public XElement UiElements

{

get { return \_ui; }

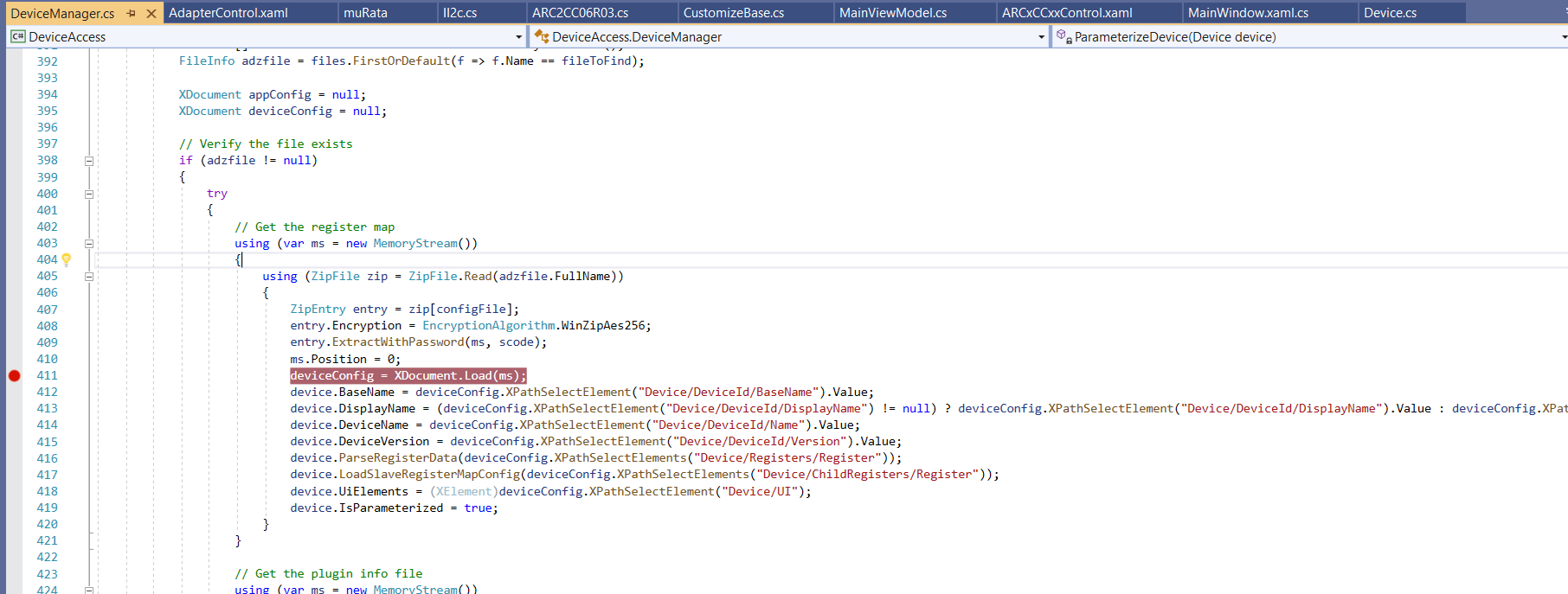
internal set

{

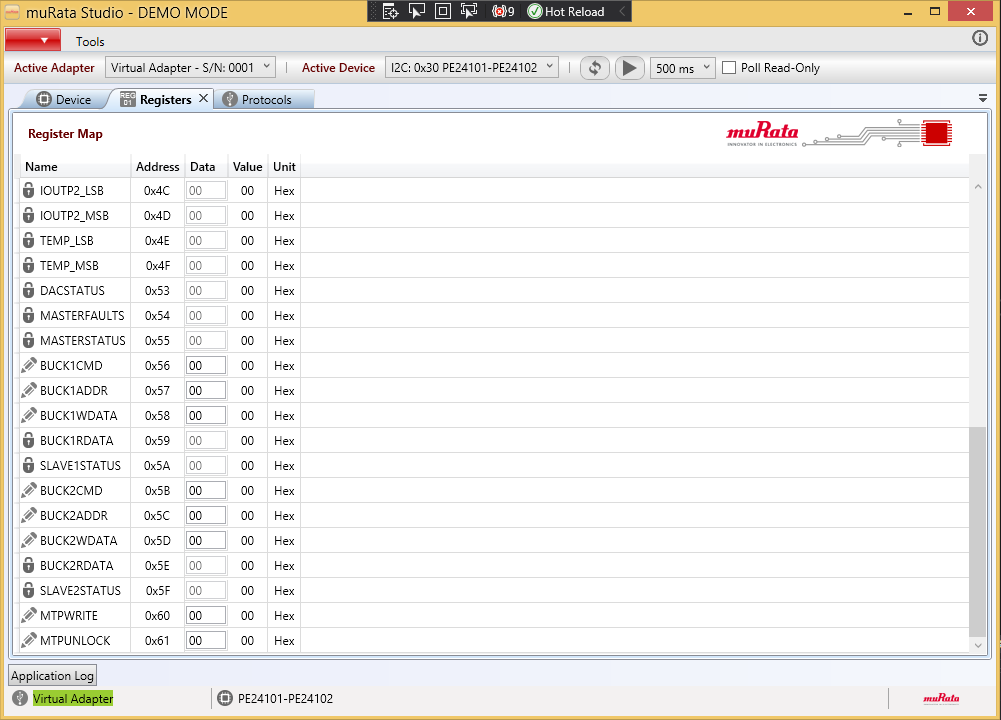
\_ui = value;

}

}



**Registers Tab**



REGISTERS

The device configuration file consists of Registers section which will list out the registers for that device.

<Registers>

<Register>

//...............Register info

</Register>

<Register>

//...............Register info

</Register>

<Register>

//...............Register info

</Register>

</Registers>

**Eg of Register :**

<Register Name="VSET1" Address="0X00" DataType="H" Size="1" Format="X2" Unit="Hex" ReadOnly="False" Private="False" Description="This register holds the setting for the target voltage of Buck 1.">

<Bit Mask="0x0080" Name="VSET1\_7" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0040" Name="VSET1\_6" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0020" Name="VSET1\_5" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0010" Name="VSET1\_4" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0008" Name="VSET1\_3" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0004" Name="VSET1\_2" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0002" Name="VSET1\_1" Description="Configures target voltage for Buck1" />

<Bit Mask="0x0001" Name="VSET1\_0" Description="Configures target voltage for Buck1" />

</Register>

Each register consist of 8 bits .

**Register xml details:**

1) Name – Register name

2) Address – Target address/address of the device memory space.

3) DataType – Available Datatypes are L11, S, F, H, U

4) Size – Provide the information about bit.

5) Format – String types

6) Unit - For information about unit

7) ReadOnly – Provide information about whether the register is editable or not.

8) Private – True means it is not available to the user and is only available fr developer. False means, register available for user

9) Description – Details about register

**Bit details :**

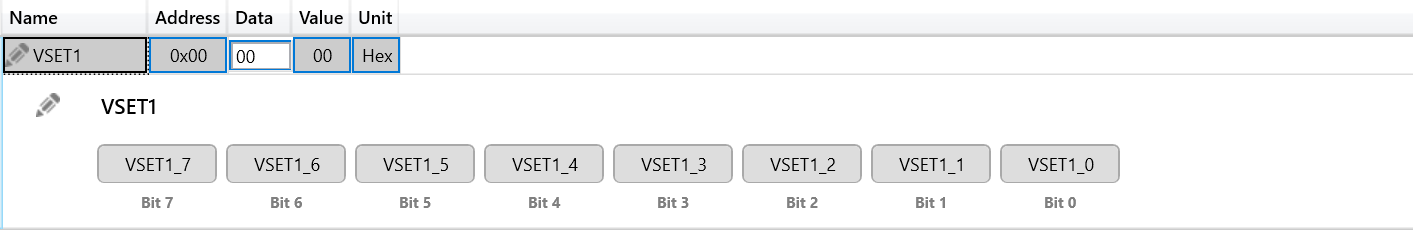
1) Mask – Bit will be masked to allow the operation on it. This bit mask value is mentioned in the UI section against a control if user want to do the operation only on one bit.

If it doesn’t depend on bit, in UI panel section it will be defined as 0XFF, so that all bit are available for operation.

2) Name – Bit name

3) Description – Information about the bit .

Based on the device configuration file information a Register will be added to Register control Plugin and will be displayed to user as shown below.

Based on the bit type, user has the provision to change the bit values so that value changes will be passed on to the device.

**Command Line Arguments in Murata App :**

**1) /demo( Demo mode)** - Allows the user to load the Virtual Adapter inorder to get an idea about how the device UI will look like. As it is the visual display of dummy device GUI, this will not let the user to perform any communication with the device .

**2) /dev (Development mode)** - Allows the user/operator to view the Registers defined as Private in device configuration file.

**3)/silent** - Allows the user to prevent reading all the registers at the startup of app . Else it will read all the registers irrespective of their access type(readonly & access types - sendbyte/receivebyte/rblock). Additionally if this is set, it assumes that the device is connected instead of communicating via protocol. Also, if it is set, it will use the default slave address for san else add range to scan .

**4)/i2cauto** - Obsolete - It automatically fetches AutoAddress of the devices by looping through all interfaces on the available adapter.

**5)/r -** If it is I2c based, then when device presence is confirmed, based on this argument, it will do the locking and unlocking of device by writing to the bytes of access register.

**Running Command Line Arguments in installed application.**

if we want to directly run the command line arguments  via command line, then we can use the below arg.

1) Open the command line from the Muratstudio installed folder location.(**C:\Program Files (x86)\muRata Corporation\muRata Studio**)

2) Execute the below line via command line, so that it will load the virtual adapter with the device mentioned.

**start muRataStudio.exe /demo /d="ARC2CC06-R03"**