Building a Custom Version of the .NET Micro Framework for a Device Emulator

A developers guide

Abstract

The .Net Micro Framework version includes an extensible emulator that can be customized on the standard peripheral interfaces provided by the managed object model (OM) and the Hardware Abstraction Layer (HAL). This document explains how to rebuild the .NET Micro Framework runtime to extend both the OM and the HAL.

1 Rebuilding the Runtime and Loading the HAL in the Emulator

In this document we will explain how to create a custom Windows-hosted version of the .NET Micro Framework CLR, and use it in a MF Custom Emulator. Some examples of suitable customer types that are interested by this article are

a) The Porting Kit user that wants to change an existing functionality at the runtime level or driver level or introduce a new standard function in the .NET MF object model and driver model

b) The Porting Kit user who wants to add his own native functions to the MF CLR, and needs an emulator that has the same native functionality that he is developing for some actual device.

c) The Porting Kit and SDK user that wants to fix a bug at the runtime level and create a new emulator

The Porting Kit (PK) documentation describes in some detail the software architecture of the MF runtime and how it resides between the managed system libraries and the drivers' layer. For a device, the runtime builds into a set of static libraries, selected and listed in a project file for a specific solution and linked into a device image. The selection process can be helped by the SolutionWizard application in the PK. For the emulator, the runtime is instead built into a Win32 DLL and loaded by a host process that, among other things, takes care of parsing the command line and loading the emulator configuration. The configuration can be changed to emulate typical HW characteristics such as different displacement of runtime memory, or LCD sizes, etc…

The Win32 DLL is at all effect the same .NET Micro Framework runtime that runs on any device, but it is compiled for the Windows instruction set with the options of the *%SPOCLIENT%\tools\target\ Microsoft.SPOT.System.x86.Targets* file of the PK.

In order to emulate faithfully enough the device capabilities the runtime for Win32 relies on the same HAL API declarations that a device would, and specifically the ones declared in *%SPOCLIENT%\DeviceCode\Include*. Obviously though, the implementation of the HAL API is engineered to suit a Windows platform and also to accommodate extensibility, as the first customer scenario above requires.

Just as any device image builds as a solution in the PK with an associated set of settings and a project file, e.g. *%SPOCLIENT%\Solutions\SAM7X_EK\TinyCLR\TinyCLR.proj* for the SAM7X512 processor, so does the emulator. The solution for the emulator is *%SPOCLIENT%\Solution\Windows2*, where the '2' is required by pure historical reasons.

The *Windows2* solution comes with a settings file, an associated instruction set, i.e. 'x86' and an architecture settings file, just like any other device. The architecture settings file is minimal and lies in the *%SPOCLIENT%\DeviceCode\Targets\OS\Windows* directory, a sibling of the directories that would normally contain the MCU drivers' code and configuration. In the case of the emulator, the MCU is ideally the Win32 platform. Rather than actual drivers, the emulator only requires including a few system DLLs and header files: a good example is provided by the usage of Winsock for networking emulation rather than an actual MAC driver library. The *Windows2* solution compiles into a Win32 DLL with the name Microsoft.SPOT.CLR.dll.

The emulator is an executable and as such it needs a hosting process for the .NET MF Win32 runtime DLL. The hosting process will take care of parsing the command line parsing and the emulator configuration, something the device does not normally do, at least for a native port.

Summarizing so far we have the following extensibility points:

1) The solution
2) The emulator configuration
3) The emulator implementation of the standard drivers API
4) The emulator host process and UI

When should one take advantage of these extensibility points? There is no definite rule, but it is rather easy to choose a strategy based on requirements and desired flexibility.

Duplicating a solution is always a good idea when making changes to the files of the solution. Solutions are just containers of libraries and, in the case of a device image, memory settings. For example, when adding an interop component, it is necessary to add the libraries for it to the *TinyCLR.proj* project file in the solution directory. Having a different solution prevents the user from injecting undesired behavior to the standard solution and binaries, in this case *Microsoft.SPOT.CLR.dll*.

The emulator configuration is for everybody to use and it is mostly useful when trying different HW settings such as memory constraints, size of RAM, LCD size, number of serial ports, and so forth. The emulator configuration file is called as the emulator executable plus the '*.config*' extension and should be located in the same directory of the emulator process.

Modifying the emulator implementation of standard HAL APIs is for the really advanced users that want to add a new standard driver API to the .NET Micro Framework along with the managed application object model, e.g. a CAN bus object model and related driver interface. This extension will require modifying or augmenting the standard HAL interfaces in *%SPOCLIENT%\DeviceCode\Include* as well tweaking the emulator implementation and interface declaration for those APIs. And their configuration support. This is a rather substantial amount of work with the potential of breaking the whole processing chain and should be undertaken only if one plans to redistribute the resulting components for other users to extend them or customize them. If a user simply wants to add an interop component, he can just get away with adding the interop component library in the *TinyCLR.proj* file and does not need to modify the standard HAL interface. In fact, most of the time an interop components relies on existing drivers APIs and it is exposed to the managed application through a custom object model, rather than a more general infrastructure with attached extensible emulation. Examples of custom interop components are available in the PK in the *%SPOCLIENT%\Product\Sample* directory.

The emulator process and UI should be customized by anybody implementing a device emulator and doing so is fully supported by VS2010 and the .NET Micro Framework project templates. Also we provide a few samples along with the SDK as well, noticeably the temperature emulator sample and the sample emulator code that constitutes our standard emulator.

Let's now dive into the code base and explore our options from an operational point of view.

Implementing a custom device emulator means implementing a hosting process for the Win32 DLL generated by the original *Window2* solution, i.e. *Microsoft.SPOT.CLR.dll*. This task can be accomplished entirely in Visual Studio by creating a new device emulator project. The DLL *Microsoft.SPOT.CLR.dll* is in fact redistributed with the SDK as well. Attending to this task only entails addressing the extensibility points 4) and partially 2), when the emulator configuration file is edited by the user, which is rather obvious.

Instead, implementing a custom version of the .NET Micro Framework means to create a new version of the *Microsoft.SPOT.CLR.dll*, either by recompiling the Windows2 or creating a clone of it, and host it in a device emulator process. This activity deals with extensibility point 1). It is a more complete activity than implementing a device emulator only.

The good news is that, as some of the tasks an emulator carries out are common to most device emulators. We do provide facilities to implement the hosting process correctly and with the same level of functionality and flexibility. Also we try to implement most of the functionality in managed code, specifically C#, for maximum productivity. Let's analyze the support we provide.

The first interesting code to look at is the one in the directory *%SPOCLIENT%\Solutions\Windows2\TinyCLR*: in this directory one will find the actual implementation of all drivers APIs needed by the emulator and the emulator command line parsing. Also there is a file, *NetAssemblyInfo.cpp*, for attaching versioning information to the runtime DLL. The command line parsing can effectively be used for a command line emulator, but it is not documented yet. The most interesting commands are the /load: command which lets the emulator load assemblies and then execute them. Incidentally it is interesting to observe that when Visual Studio launches an emulator session, it does nothing but spawning a Win32 process with a command line that lists all the assemblies that the managed application needs to load. You could do so yourself on a DOS shell and run an MF application on your desktop standalone, with no debugger. Indeed, this is what we do when we want to debug the CLR runtime on Windows. Back to the code in the TinyCLR directory: upon inspecting any driver's code, it is immediately apparent that the API implementations are nothing but stubs and do all look something like this:

```
BOOL USART_RemoveCharFromTxBuffer( int ComPortNum, char& c ) {

return EmulatorNative::GetISerialDriver()->RemoveCharFromTxBuffer( ComPortNum, (wchar_t %)c);

}
```

Let’s keep this in mind and observe that in the *TinyCLR.proj* file of the Windows2 solution we see the property `<ManagedCode>true</ManagedCode>`. This tells us that this code is where we leave the native code C++ code of the NET MF runtime to jump into the managed code of the extensible emulator.

We also notice that we do that by relying on the `EmulatorNative` type and accessing an interface for the specific driver at hand, in the example above the `ISerialDriver` interface. The `EmulatorNative` type implements the `IEmlator` interface, which, along the other drivers’ interfaces is defined and coded at *%SPOCLIENT%\CLR\Tools\EmulatorInterface*. This is in fact the extensibility point 3) above, where one would have to add a new driver interface, e.g. `ICANBusDriver` for the yet not-supported standard CAN bus object model. The most important thing to understand here is that the `EmulatorNative` type is responsible for accessing the emulated driver interfaces and that it is in facts the gate-keeper to the emulated HAL. Swapping the `EmulatorNative` implementation effectively swaps the HAL. The `EmulatorNative` type is implemented in the TinyCLR directory of the Windows2 solution as well, and therefore the files in this directory would have to be modified in order to modify the behavior of the HAL or, like in the example of the CAN bus, a new file should be added to implement a new API and that addition should be reflected in the `EmulatorInterface` type.

Looking that EmulatorNative.cpp we see that this type is initialized by a IHAl interface (below),

```
void EmulatorNative::Initialize( IHal^ managedHal )
```

This allows the `EmulatorNative` to dispatch the calls to the so the appropriate implementation of HAL APIs. The question therefore is where and how a new IHAL gets supplied. The IHAL component is at all effects the collection of the drivers’ API of extensibility point 3). It is at this point that the final type comes into play, and the most important one: the `Microsoft.SPOT.Emulator.Emulator` type that implements the device emulator reusable functionalities. The IHAL component is part of the Emulator object and it is initialized through some levels of indirection when the Emulator instance is created in the code at the %SPOCLIENT%\Framework\Tools\Emulator directory. The code at this directory is practically the skeleton of the hosting process and takes care of parsing the emulator configuration and dispatching the runtime calls through the `EmulatorNative` object to the user supplied implementation of the drivers’ API interfaces. So many levels of indirection are necessary to go from the native code of the runtime, to the native API specification of the HAL and finally to the user defined implementation in managed code, taking in account the user supplied configuration as well. We can observe that the default constructor for the Emulator uses gets the `IEmulator` interface from the `LoadDefaultHAL()` method, which looks for the Microsoft.SPOT.CLR.dll standard Hal implementation

```
private static IEmulator LoadDefaultHAL()
{
IEmulator emu = null;
string emulatorTypeName = "Microsoft.SPOT.Emulator.EmulatorNative";
// Construct an assembly reference name using our own version number
// and public key, since we use the same for both Emulator.dll and CLR.dll
AssemblyName halWindowsName = new AssemblyName();
Assembly exec = Assembly.GetExecutingAssembly();
```

```
halWindowsName.Name = "Microsoft.SPOT.CLR";
halWindowsName.CultureInfo = System.Globalization.CultureInfo.InvariantCulture;
halWindowsName.Version = exec.GetName().Version;
halWindowsName.SetPublicKey(exec.GetName().GetPublicKey());
        try
{
Assembly halWindowsNativeAssembly =
Assembly.LoadFile(
Path.Combine(
Path.GetDirectoryName(exec.Location),
halWindowsName.Name + ".dll")
);
emu = halWindowsNativeAssembly.CreateInstance(emulatorTypeName) as IEmulator;
…
…
```

As you can notice this method literally looks for the Microsoft.SPOT.CLR.dll by name.

Therefore, in order to choose ones HAL we will need to use a different constructor and create our own instance of our `EmulatorNative` type. Remember though that we do not necessarily need to come up with our own `EmulatorNative` type. If we only want to customize the device emulator UI, add an emulated device on a standard driver interface or change the emulator configuration file you can very well reuse the standard HAL and only create a device emulator project through Visual Studio.

Let's now instead go through all the process of creating a new version of the .NET Micro Framework, e.g. a new version of *Microsoft.SPOT.CLR.dll*. The major steps involved are few, and you can start applying them from the point it makes sens for your specific scenario: e.g. if you do not need to modify the code in the Windows2 solution you do not need to duplicate it. Let's begin.

First, you will create a clone of the solution that builds the .NET Micro Framework CLR that ships with the MF PK and SDK products change a handful of solution-specific names and other properties to make this new CLR unique. Then, you will add interop projects and/or prebuilt interop libraries to his new solution project, creating in effect a CLR with a superset of the features provided in Microsoft's own Win32 MF CLR. Finally, having built this solution and created a native DLL, you will create a new MF Custom Emulator project or re-use an old one, specifying in the constructor for your emulator class which Win32 CLR to use. When you build this emulator project, the resulting EXE will need to be registered with the MF SDK for use in running Micro Framework projects.

2 Creating a new Win32-targeted CLR solution

(We will call the solution "MyCLR" for the purposes of this document):

1) Go to %SPOCLIENT%\Solutions, and run: xcopy /ei Windows2* MyCLR.
2) Go to %SPOCLIENT%\Solutions\MyCLR. Rename Windows2.settings to MyCLR.settings. Do the same for Windows2.settings_no_floating_point.
3) Open MyCLR.settings. Change the value of TARGETPLATFORM property, which in Microsoft's default solution is "Windows2", to some name of significance to your team, for instance "MyCLR". Change also the PLATFORM property (??). Change the value of the IncludePaths item which says "Solutions\Windows2\" to "Solutions\MyCLR\". Close MyCLR.settings.

4) Open dotnetmf.proj. Change the value of the Directory property to “Solutions\Windows2” to “Solutions\MyCLR”. Change also the value of the MfSettings file. Close dotnetmf.proj.
5) In the TinyCLR directory in your solution, you will find AssemblyInfo.cs. Open it, and edit it to add assembly attributes that will be attached at build time to your new CLR DLL, attributes like version number, the name of your company, a descriptive string of some significance to your product team or your customers.
6) Finally, open TinyCLR\TinyCLR.proj, and at a minimum, change the name of the DLL that this project will shortly produce

   1) Change the AssemblyName property to something meaningful to your team. This will be the name of the DLL that contains your custom CLR.
   2) Change the Directory property to match the actual location of this project file, in this case “Solutions\MyCLR\TinyCLR”.
   3) Change the MFSettingsFile property to refer to this containing solution: $(SPOCLIENT)\Solutions\MyCLR\MyCLR.settings.

7) You will need to sign your emulator to later use it in a custom emulator project, and to do so you need to add the following lines to the NetAssemblyInfo.cpp file:
[assembly:AssemblyKeyFileAttribute("TestPublicKey.snk")];
[assembly:AssemblyDelaySignAttribute(true)];
then create a TetPublicKey.snk key pair with sn.exe, a tool that comes with the Visual Studio SDK
8) Install the PK crypto pack

Having done so, you may now build your new CLR. We have not added any new interop libraries to it yet, of course, but that is done in exactly the same way you would do it for a device-based solution, and is therefore outside the scope of this note. Let’s assume you’ve done so already, and move along.

2.1 Build your CLR

msbuild /t:Build /p:MfSettingsFile=%spoclient%\solutions\MyCLR\MyCLR.settings /p:FLAVOR=Debug

You now have a new Win32 implementation of the MF CLR in your %SPOCLIENT%\BuildOutput tree.

2.2 Creating a new Custom Emulator that uses MyCLR.DLL

1) Start Visual Studio, if it’s not already running. Select New Project from the File menu, and choose the Emulator template from the collection of Micro Framework project types to create.
2) Add to your Emulator project a reference to your new CLR DLL and the system assembly Microsoft.SPOT.Emulator.Interface.
3) In the main source file of your emulator project, find the constructor for your subclass of Microsoft.SPOT.Emulator.Emulator. Change it to look like below

```
class Program : Emulator
{
public Program() : base( new EmulatorNative() ) { }
…
}
```

4) Build the emulator project.
5) Re-sign the emulator project with *sn.exe* (this tools comes with the Visual Studio SDK, use *–R* option).
6) Now you need to install the emulator as one of the emulators from the SDK running in Visual Studio. To do so add a registry key at `HKLM\Software[\Wow6432Node]\Microsoft\.NETMicroFramework\[VERSION]\Emulators\[YOUR_EMULATOR_NAME]` that looks like the one from the existing SDK. Add the Name and Path parameters to your emulator executable.
7) Go back to your manage application project, select the properties for the project and select the new emulator from the drop down list

3 Additional resources

Check documentation at http://msdn.microsoft.com/en-us/library/hh399820.aspx for building the managed part of the emulator and/or use the existing configuration support.