Software Updates in the .Net Micro Framework

A developers guide

Abstract:

The .Net Micro Framework version 4.2 includes a software update framework that will support remote firmware updates, assembly updates and other OEM defined updates (such as key provisioning). Rather than implementing a full end-to-end software update solution that will surely not fit everyone's needs, the framework will allow the OEM to determine the best solution for their device. To support our customers we have added a couple sample remote firmware update solutions in the Porting Kit (which will be discussed later in this document).

1 Software Update Framework for .Net MF (MFUpdate)

The .Net Micro Framework firmware update procedures in the past have all relied on either TinyBooter or the OEM's own firmware update utility (like SAM-BA). Both of these utilities require a local connection to the device and are not extensible. In .Net Micro Framework v4.2 we have added the ability to perform remote firmware updates with a flexible update framework (dubbed "MFUpdate"). The framework was designed to provide the core facilities required for an update without designating the update transport or protocol. MFUpdate provides facilities for update packet management (with verification), persistent storage, image verification and the plumbing for image backups.

The Version 4.2 Porting Kit comes with a couple of sample remote firmware update solutions: an HTTP(s) device server option and an updated .Net MF debugger which communicates with the Porting Kit tool MFDeploy. In addition, MFDeploy will continue to support local (or wired) connections as well, including a new option for installing firmware to a file. This will enable SD card updates for supported devices. We will likely ship with at least one SD card update sample.

2 MFUpdate Features

The following is a list of the features of the MFUpdate framework:

- Update packet management
  - Identifies missing packets
  - Handles adding packets to the update
  - Manages packet verification
- Persistent storage
  - Supports persistent storage of packets so that a delay or restart of the update can be handled without restarting the download
- Validity checking
  - Support for validating packets
  - Support for validating full update
- Backup support (TBD)
  - Enable backup of current firmware (if supported)

All of the previous features are pluggable, so the developer can choose between our implementations or create their own.

3 Supported Update Types

The following are the update types that will be supported out of the box for the .NET Micro Framework. However, the MFUdpate framework was designed to be extensible so that developers can create their own update types.

- Firmware Updates

Firmware updates are currently supported in the v4.2 Beta with multiple sample applications. The current samples use a compressed image file to update the firmware. The build system was changed to produce this file automatically when building a solution. For the TinyCLR project this produces several .nmf files: one for each load region and one (TinyCLR.nmf) that concatenates all the others for a single file deployment.

The Beta samples currently do not show of update validation feature; because there was not enough time to add it. The RC and final release will include at least one sample of an update signature check for secure update verification.

- Assembly Updates
  This feature is not fully developed yet, but it will support both single assembly updates and full deployment replacement updates. For the single assembly deployment scenario we will modify the assembly load and link code to load only one assembly with the same major minor version. The build and revision numbers will be used to determine which assembly to load (larger numbers will be used).

  For deployment replacement updates a DAT file (concatenated PE files on 4 byte boundaries) will be used to update the entire deployment sector. This update type is not intended for ER_DAT region updates. Since the ER_DAT is part of the firmware, a firmware update is required.

- Key Updates
  This feature is not implemented in the Beta but will ship with the RC and final release. This update type allows users to update device keys in the field. The updated key should be wrapped so as to avoid compromising its identity. This protocol is left up to the porting kit user and update key service provider. We will provide a sample to illustrate how this can be done.

4 Managed Update Framework

The following sections will discuss the general outline of the MFUpdate framework starting with the managed code classes and moving to the native code structure.

4.1 MFUpdate abstract class

MFupdate is the abstract base class for the various update types. We will support firmware, assembly and key updates in the box and provide extensibility for adding additional update types. Currently for the v4.2 Beta only Firmware updates are fully implemented for the supported iMXS_net_open platform. The Emulator will also handle firmware updates up until the install. This enables easy testability for the packet management and validation functionality. Subclasses of this method will use the MFUpdate constructor to identify which update type they represent. This allows new update types to be added without having to change any of the existing update framework.

Methods:

- AddPacket
  This method adds an MFUpdatePkt object (with packet index and validation data) to the update mechanism.

- GetUpdateProperty
  This method retrieves an update or update type specific property given the property name. This feature was designed in a way that the marshaling of the object from native to managed code should behave much like Marshal.PtrToStructure.

- SetUpdateProperty
  This method sets an update or update type property for the given property name. This feature was designed to handle marshaling from managed to native code much like the Marshal.StructureToPtr method.

- GetMissingPacketEnumerator
  This method returns an IEnumerator object which can be used to enumerate over the set of missing packet numbers. This is used to aid in update recovery or interrupted update scenarios.

- DeleteUpdate
  This method deletes the current update and all its packets in persistent storage.

- ValidateUpdate
  This method is used to validate the full update prior to install. If the result of this command is a success, then the update should be ready for install.

- InstallUpdate
  This method starts the install process after validating the full update. For firmware updates this will include a device reset.

4.2 MFFirmwareUpdate Class

The Microsoft.SPOT.MFUpdate.MFFirmwareUpdate class is intended to assist your application in create a firmware update service. The MFFirmwareUpdate class inherits from the MFUpdate base class and also provides the current firmware properties and enables you to quickly create a firmware update.

MFFirmwareUpdate Constructor Parameters:

- provider - The string name of the target update provider that supports this install.
- updateID - The unique update identifier for this update.
- version - The version of the update (major, minor, build, revision).

- firmwareType - The MFUpdateSubType for the update. Used to indicate which type of firmware update is being installed (system assemblies, CLR, configuration, or user defined).
- updateSize - The total size of the update.
- pktSize - The update packet size.

MFFirmwareUpdate Methods:

- CurrentFirmwareVersion - This static method allows you to get information about the currently installed firmware (described below).

FirmwareVersion class:

The FirmwareVersion class is a singleton that provides read-only data about the current firmware's version. The follow is a list of the member variables and their descriptions:

Members:

- Version - The major, minor, build and revision versions if the firmware
- OemName - A string representing the OEM name for the firmware (defined by the Porting Kit developer).
- OemId - The identification number of the OEM as defined in the Porting Kit OEM_MODEL_SKU structure.
- Model - The model number of the device as defined in the Porting Kit OEM_MODEL_SKU structure.
- SKU - The SKU number of the device as defined in the PortingKit OEM_MODEL_SKU structure.
- IsBigEndian - Boolean value indicating if the device is big endian or little endian.

The firmware version data can be used by the MFUpdate application in determining if an update is required.

4.3 MFAssemblyUpdate Class

The MFAssemblyUpdate class is intended to simplify updating or adding new assemblies. As discussed earlier there are basically two options for updating assemblies: replacement of the entire deployment region, or providing an update assembly. The distinction is made using the MFUpdateSubType parameter in the constructor of the MFAssemblyClass. The first option completely erases the deployment sector prior to installation, so you must have all the required deployment assemblies in the update for it to work properly. In order to assist the application for this process, the MFAssemblyUpdate provides a list of the currently installed assemblies. This way the application can forward this information to the update server to assure a complete deployment assembly list.

The second type of update can either be a new or updated assembly. Adding new assemblies can be useful on a system that uses reflection to load know interfaces or types. You can imagine a Micro Framework UI application that uses this functionality to download new UI applets. In fact we actually

had this behavior in the early days of the SPOT watches where you could configure the application lineup on your watch. In the case of an updated assembly there are some restrictions that need to be observed:

Restrictions for Assembly Updates:

- Major or Minor version changes are considered a new assembly and will result in two versions of the assembly to be loaded (linking will happen based on these versions)
- Build or Revision changes will result in the only newest (based on these values) assembly being loaded.
- When updating assemblies with internal/interop methods, the updated assemblies must have the exact same signature as the original or the runtime linking will fail. This means that new types cannot be added and method signatures cannot be changed.
- The updated assembly should only add new functionality so that dependent assemblies are not affected.

MFAssemblyUpdate Constructor Parameters:

- provider - The string name of the target update provider that supports this install.
- updateID - The unique update identifier for this update.
- version - The version of the update (major, minor, build, revision).
- assemblyType - The MFUpdateSubType for the update. Used to indicate a replacement of the deployment region or simply an assembly addition or update.
- updateSize - The total size of the update.
- pktSize - The update packet size.

MFAssemblyUpdate Methods:

- GetInstalledAssemblies - Gets the list of assemblies (name and version) that are currently on the device. This can be used to determine if an update is needed or which assemblies are needed in the update.

4.4 MFKeyUpdate Class

The MFKeyUpdate is intended to be used to update keys on the device. This could be used for updating authentication, validation, or provisioning keys. This feature is not fully implemented yet and may not be ready for RC.

4.5 Extensibility

In order to extend the managed update framework you simply need to subclass the MFUpdate class (or one if its subclasses) and provide any extra information required for the update via the Get/SetUpdateProperty methods. The get/set update property methods will serialize your property objects into a binary structure compatible with a similar C/C++ structure (provided your property object includes only simple types).

4.6 Adding Your Own Update Software

While we provide a couple of samples of remote firmware updates, it is intended that our customers will implement their own update application. The reason for this is simple: we cannot possibly come up with a solution that will meet all of our customers' needs. We also do not want to mandate which protocols a device needs to support in order for updates to work. Therefore we added the support facilities needed and samples to assist our customers in creating an update application. This section will show you how to use those facilities to create your own software. It is also strongly recommended that you look at our sample HttpServer application as it currently uses the MFUpdate framework to provide an update solution.

The main things you need to know about the managed MFUpdate framework are the following:

- The "provider" string parameter to the MFUpdate constructor needs to match the corresponding IUpdatePackage::ProviderName in the native level.
- The "updateID" parameter should be unique for a given update
- The "version", "updateType", "udateSubType" parameters should also be supported by the native implementation
- The AuthenticationCommand method allows you to perform handshake calls to the native code implementation. This can be used if you need to generate a nonce or sign a token or other authentication handshaking required for your applications authentication. The final resulting authentication token should be passed to the Open method which will call the native codes Authenticate method which will allow the update to work. Note that updates will not work until the open command has been called with valid authentication data (determined by the native code).
- Get/Set Property methods and the Open method take objects as parameters. These objects are serialized into a binary structure that is compatible with a similar C/C++ structure. Therefore these objects should not contain any variable size objects like lists or non-fixed size arrays. Also note that we do not attempt to fix alignment issues, so if you know the resulting binary code requires a buffer you will need to manually add it in your C# structure. The sample below shows you how the serialization works.

```
C#
----------------------
class SerNestClass
{
    int b;
    char[] name = new char[16];
}

class SerClass
{
     int i;
     SerNestClass cls;
}
```

```
C++
-------------------------
struct SerNestStruct
{
    INT32 b;
    char name[16];
}

struct SerStruct
{
    INT32 i;
    SerNestStruct cls;
}

Binary
----------------------
00 00 00 00     ← SerClass.i
00 00 00 00     ← SerClass.cls.b
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ← SerClass.cls.name
```

- MFUdpate handles interrupted updates so you can call the GetMissingPacketsEnumerator to process only the packets that are not already on the device. Note, that if the interruption to the update was because of a power failure and only a partial packet was updated the update will fail. We only check for packet validity while adding the packet.

5 Native Update Framework

The native code portion of the update framework was designed to be configurable and extensible. It is made up of a set of interfaces that allow the porting kit user to choose between our set of implementations or implement their own.

5.1 Native Code Interfaces and Structures

5.1.1 IUpdateProvider

The IUpdateProvider interface structure provides a configuration and extension point for adding update providers.

Methods:

- InitializeUpdate - Initializes the given update by adding corresponding update components interface pointers (IUpdateProvider, IUpdateStorageProvider, IUpdateBackupProvider, etc).
- GetProperty - Gets the property value for the given property name.
- SetProperty - Sets the property value for the given property name.
- InstallUpdate - Verifies and Installs the update.
- Extension - Void pointer to allow for extensions to the interface.

5.1.2 IUpdateValidationProvider

- ValidatePacket - Validates the packet using the given validation data specific to the update type and provider.
- ValidateUpdate - Validates the full update using the given validation data specific to the update type and provider.
- AuthCommand - Enables the application level to perform handshake commands between the remote update provider and the devices update firmware.
- Authenticate - Authenticates the update with given application specific authentication data.
- Extension - Void pointer to allow for extensions to the interface.

5.1.3 IUpdateStorageProvider

Methods:

- Initialize - Initializes the storage system (likely only called by MicroBooter).
- Create - Creates an update storage location with the given update information.
- Open - Opens an update storage with the given update information.
- Close - Closes the update storage for the given handle.
- Delete - Deletes the given update storage object.
- GetFiles - Gets the files for the given update criteria. Call with storageIDs parameter equal to NULL to get the number of matching files.
- IsErased - Determines if the given section of memory is erased.
- Write - Writes the given data to the indicated memory offset.
- Read - Reads the given amount of data from the given offset.
- GetHeader - Gets the update header information from the update storage.
- GetEraseSize - Gets the erase size for the storage (mainly used for block storage devices to indicate what the erase boundaries are).
- Extension - Void pointer to allow extensions to the interface.

5.1.4 IUpdateBackupProvider

We do not provide any samples for the backup facility at this point. We may or may not have anything for the official release. The backup provider infrastructure assumes there is only one backup supported per update type.

Methods:

- CreateBackup - Creates a backup for the given update type. This is intended to be called during the installation of the update so that a recovery can be attempted if the install fails. There should only be one backup per update type.
- RestoreBackup - Restores a backup for the given update type
- DeleteBackup - Deletes the backup for the given update type.
- Extension - Void pointer to allow extensions to the interface.

5.1.5 IUpdatePackage

Methods:

- ProviderName – Name of the update package (used to route updates).
- Update – The IUpdateProvider implementation for the package.
- Validation – The IUpdateValidationProvider implementation for the package.
- Storage – The IUpdateStorageProvider implementation for the package.
- Backup – The IUpdateBackupProvider implementation for the package.
- Extension - Void pointer to allow extensions to the interface.

5.1.6 MFUpdateHeader

Fields:

- UpdateID - The unique identifier for the update.
- Version - The major, minor, build and revision version numbers for the update.
- UpdateType - The update type (firmware, assembly, etc.).
- UpdateSubType - The update sub type (indicates what type of firmware, assembly, etc.).
- UpdateSize - The size of the entire update.
- PacketSize - The size of all the packets (except perhaps the last packet).

5.1.7 MFUpdate structure

Fields:

- Header - The MFUpdateHeader that describes the update.
- StorageHandle - The update storage handle corresponding to the update.
- Flags - Internal flags used to track the state of the update.
- IUpdatePackage - A pointer to the IUpdatePackage object for the update.
- Extension - A void pointer for extending the update structure. This can be used to append update provider specific data to an update during the call to IUpdateProvider::InitializeUpdate.

5.2 Configurability

The native level of the update framework is configurable in a couple of ways. First, any of the interface structure implementations can be replaced by the porting kit user. Second, any the interface methods can be set to null: representing an unsupported operation.

5.3 Extensibility

The native level of the update framework is also extensible in a couple ways. New update types and update providers can be added by porting kit users to handle new or existing update types. In addition, the 'Extension' parameter of the interface structures and the update object itself allows the user to append extensible information.

5.4 MicroBooter Sample Installer

The v4.2 ships with at least one full end-to-end MFUpdate sample (iMXS_net_open). This platform has a new project type called MicroBooter. This is a stripped down version of the TinyBooter that performs to major tasks: bootstrapping and installing a compressed image that was downloaded by the CLR using the MFUpdate framework. There is one other optional component to MicroBooter which is to provide a failsafe if the image was corrupted during an install (without backup support). This feature is basically an SREC server that listens to the platforms debugger port for SREC packets. MFDeploy has been modified to enable uploading of SREC. If you remove the failsafe portion, you can obtain a very small code size (much smaller than TinyBooter) which leaves you more space to handle remote updates even on very small devices. For the final release, we intend on having one of our smaller platforms as a sample for remote updates using this technique.

6 MFUpdate Samples

As mentioned earlier, the Micro Framework contains two end-to-end remote firmware update samples: a modified debugger/MFDeploy sample; and an HTTP(s) device server sample. The following sections will describe the samples in detail.

6.1 MFDeploy Updates

MFDeploy and the corresponding device side debugger component have been updated to support the new update framework. In addition, for TCP/IP connections a new feature has been added to upgrade the connection to SSL for deployment. To support this new feature you will see a slightly different look for MFDeploy (see Figure 1 below). The new UI enables you to select a certificate that will be used for the SLL connection. Note, SSL is only used when required (firmware updates, memory access methods, etc.). In the application configuration file you have the ability to determine if the SSL connection should require mutual authentication or just authenticate MFDeploy (see below). One other thing to note, while MFDeploy is the client with respects to the TCP/IP connection, for SSL it becomes the server (since MFDeploy should always be authenticated).

```
<setting name="SslRequireClientCert" serializeAs="String">
  <value>True</value>
</setting>
```

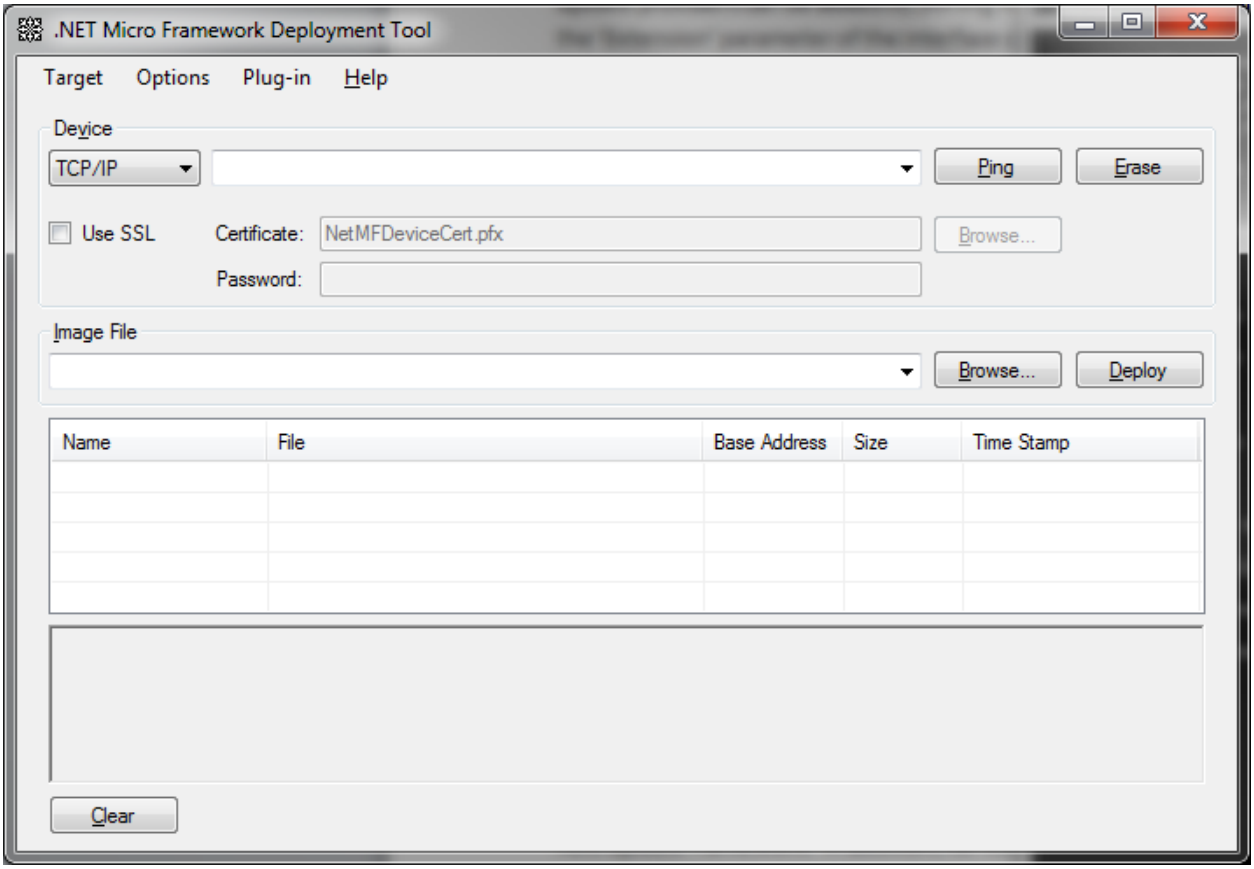

Figure 1: MFDeploy SSL UI Changes

While MFDeploy still supports connections to TinyBooter, it also now supports uploading directly to the CLR and deploying directly to MicroBooter (which we will discuss later). For devices that support MFUpdates, MFDeploy can download the image directly to the CLR. In order to reduce the footprint needed and the amount of data sent, we use a compressed image that is generated during the porting kit build of a solution. We will discuss how to configure your solution later in this document. This new file type (.nmf) can either be a single compressed image region or can be a concatenation of multiple compressed regions (with 4 byte alignment). Our provided compression algorithm usually compresses the image from 40-50%, but you can use your own algorithm as long as you update the corresponding boot loader code (MicroBooter) on the device side.

These .nmf files can be used in conjunction with MFDeploy or the HTTP(s) server sample to remotely update the devices firmware (if the device has support). The .Net MF debugger was updated to support new commands and responses that correspond to the native portion of the MFUpdate framework. The changes allow MFDeploy to perform a remote firmware update (when the device's debugger port is TCP/IP).

New Debugger commands:

- UpgradeToSsl - Upgrades the transport from TCP/IP to SSL
- MFUpdate_Start - Starts the MFUpdate process for the given update information.
- MFUpdate_AddPacket - Adds a packet with given index, size and validation data (CRC).
- AuthCommand - Enables authentication handshake commands (currently it is only used to make sure SSL is supported on the device)
- Authenticate - Authenticates the device (in the case of SSL this is just verifies the SSL connection has been made).
- GetMissingPkts - Gets the list of missing packets for the given update.
- MFUpdate_Install - Initiates the install process (for firmware updates, the response will be received after update validation but prior to the actual install since the install requires a reboot).

In order to test this sample out of the box, you will need to have an iMXS platform with network support. However, later in the document we will discuss how to change a solution to provide support for MFUpdates.

Performing the update:

From the MFDeploy side of things you simply need to open one of the MFUpdate firmware update files (.nmf) that are generated by the porting kit build for your platform's TinyCLR project. The changes required to the TinyCLR.proj file to enable this option will be discussed in the following sections. The resulting files are: TinyCLR.nmf (in the root build output bin directory), ER_FLASH.nmf, ER_DAT.nmf, and ER_CONFIG.nmf (in the output's bin\TinyCLR.bin\ directory). The .nmf file is a compressed file (using LZ77 compression) prepended with a CompressedImage_Header structure. The TinyCLR.nmf file is a concatenation of all the other firmware compressed files (with 4-byte alignment). MicroBooter is able to handle multiple compressed images in the same update. MFUpdate and MFDeploy have provisions to handle interrupted updates. Therefore if the connection is broken after a partial download, then MFDeploy can pick up where it left off on the next update attempt. Also, MFDeploy uses SSL to provide authentication of the user and the device. In addition, MFDeploy uses the key form the SSL certificate to digitally sign the firmware update. This signature is then validated by MFUpdate on the device side prior to install.

6.1.1 MFDeploy Configuration

If you plan on supporting remote update it is strongly advised that you use SSL or other security measures to at least assure authentication of the provider. While encrypting the firmware may not be required, authenticating where the firmware is coming from and authenticating the firmware itself is important. This is why we choose SSL for our MFDeploy sample. For our sample we provided a self-signed certificate that is loaded by MFDeploy and the device to initiate the SSL connection. It is required that you create your own self-signed certificate or obtain a valid certificate if you intend on using MFDeploy in production.

In order to use your own certificate, you can simply browse to the location of the certificate (using the browse button). You can also update the MFDeploy.exe.config file to set the default certificate location (see below).

```
    <setting name="SslCert" serializeAs="String">
      <value>NetMFDeviceCert.pfx</value>
    </setting>
```

Then you simply enter the certificate password (we do not save the password), select the firmware update file (.nmf) and transport and press the 'Deploy' button. MFDeploy will connect with the device at the given TCP/IP address and request for it to upgrade to SSL. Then MFDeploy will authenticate as the server and create the SSL connection. From this point on the deployment packets will be sent over the SSL connection. And when the final packet is sent a signature of the update (using the certificate's private key) will be sent for validation. If validation succeeds the device will reboot into MicroBooter and the image will be installed.

6.1.2 Device Configuration

6.1.2.1 Device Build Changes

Building an MFUpdate deployment file (.NMF):

In order to produce the compressed image files (.nmf) used by MFDeploy, you will need to modify your solution's TinyCLR.proj file to add the following groups.

```
<PropertyGroup>
   <ExtraTargets>$(ExtraTargets);CompressImage</ExtraTargets>
  <CompressImageFlashSym>EntryPoint</CompressImageFlashSym>
   <CompressImageDatSym>TinyClr_Dat_Start</CompressImageDatSym>
   <CompressImageCfgSym>g_ConfigurationSector</CompressImageCfgSym>
</PropertyGroup>

 <ItemGroup>
   <CompressImageSymdef Include="$(BIN_DIR)\TinyCLR.symdefs" />
   <CompressImageFlash  Include="$(BIN_DIR)\TinyCLR.bin\ER_FLASH" />
   <CompressImageDat    Include="$(BIN_DIR)\TinyCLR.bin\ER_DAT" />
   <CompressImageCfg    Include="$(BIN_DIR)\TinyCLR.bin\ER_CONFIG" />
 </ItemGroup>
```

If you rebuild you should see four new .nmf files in solution's output directory. Note, if you solution does not build an ER_DAT file then you can remove the CompressImageDat item in the ItemGroup above.

- TinyCLR.nmf - A concatenation of the other compressed .nmf files
- ER_FLASH.nmf - A compressed TinyCLR image file.
- ER_DAT.nmf - (optional) - A compressed image file of the system assemblies.
- ER_CONFIG.nmf - A compressed image file for the configuration sector.

TinyCLR.nmf will be in the same location as TinyCLR.axf and you will find the others in the TinyCLR.bin directory.

6.1.2.2 Device Changes

This section will show you how to configure your solution to support MFUpdates. Please note that the Beta and possibly the RC versions of Solution Wizard may not work properly with the MFUpdate feature. We will fix this issue prior to shipping.

SSL Configuration:

If you are intending to use the MFDeploy's SSL connection, you will need to configure your platform to support upgrading to SSL. It is also important to note that the on the device side, we have only added support for OpenSSL. If you are using the RTIP libraries, you will need to add support for upgrading to SSL. However if you are using OpenSSL you simply need to provide the SSL certificates and target host name. This is done by implementing the IDebuggerPortSslConfig interface for your solution.

IDebuggerPortSslConfig methods:

- GetCertificateAuthority - Gets the certificate authority certificate and certificate length used to validate the server certificate (MFDeploy's certificate).
- GetTargetHostName - Gets the host name string associated with the servers certificate (the certificates 'CN' value).
- GetDevcieCertificate - Gets the device certificate and certificate length to be used in the SSL connection.

The certificates can be in BASE64 or binary format and can be either be PEM, DER, or, PKCS12.

Your implementation of the IDebuggerPortSslConfig can either be added to your TinyCLR.cpp file or you can create a new configuration project in your solution's "DeviceCode\DebuggerPort\" directory. Please use the iMXS_net_open solution as a template. When Solution Wizard is updated it will automatically generate the configuration file in your solution's directory.

MFUpdate Configuration:

In order to support MFUpdate you will need to provide the list of IUpdatePackages supported by your solution. Again you can either add this to your TinyCLR.cpp file or add a new project file in the solution's "DeviceCode\MFUpdate\" directory. Please use the iMXS_net_open solution as a sample. An

IUpdatePackage interface object describes the set of update facilities used by the MFUpdate feature. When an update is created it is passed a provider name which must correspond to one of the IUpdatePackages for the solution. From that point on it uses the facilities designated by the update package. The following is the iMXS_net_open solution's implementation of the the IUpdatePackage list. Please note that it has multiple packages. The "NetMF" package supports the MFDeploy and the "HTTPSUpdate" supports the managed HttpServer sample application. Since the HTTP server can uses SSL it performs its validation at the application level and native validation is simply a CRC check to make sure the data was not corrupted. In order to support MFDeploy, you only need to include the first package.

```
static const IUpdatePackage s_UpdatePackages[] =
{
  {
    "NetMF",
    &g_MicroBooterUpdateProvider,
    &g_SslUpdateValidationProvider,
    &g_BlockStorageUpdateProvider,
    NULL,
  },
  {
    "HTTPSUpdate",
    &g_MicroBooterUpdateProvider,
    &g_CrcUpdateValidationProvider,
    &g_BlockStorageUpdateProvider,
    NULL,
  }
};

const IUpdatePackage* g_UpdatePackages     = s_UpdatePackages;
const INT32         g_UpdatePackageCount = ARRAYSIZE(s_UpdatePackages);
```

Other changes to your Solution:

Until Solution Wizard work is complete, you will need to manually add the MFUpdate libraries to your solution.

- Update your block storage map to include enough BLOCKTYPE_UPDATE blocks to handle the firmware update's size (see below).

```
const BlockRange g_I28F640J3_16_BlockRange[] =
{
  { BlockRange::BLOCKTYPE_BOOTSTRAP    , 0,  0 },   // TinyBooter
  { BlockRange::BLOCKTYPE_CODE         , 1, 18 },   // TinyCLR runtime
```

```
    { BlockRange::BLOCKTYPE_DEPLOYMENT , 19, 30 },
    { BlockRange::BLOCKTYPE_DEPLOYMENT , 31, 40 },
    { BlockRange::BLOCKTYPE_UPDATE     , 41, 58 },
    { BlockRange::BLOCKTYPE_SIMPLE_A   , 59, 59 },
    { BlockRange::BLOCKTYPE_SIMPLE_B   , 60, 60 },
    { BlockRange::BLOCKTYPE_STORAGE_A  , 61, 61 },
    { BlockRange::BLOCKTYPE_STORAGE_B  , 62, 62 },
    { BlockRange::BLOCKTYPE_CONFIG     , 63, 63 }    // g_ConfigurationSector
};
```

- Modifications to TinyCLR.proj
    - If you intend on creating a managed application that uses MFUpdate include the following, otherwise they are optional
        - Add the “MFUpdate.featureproj”
        - Add the CLR interop library

```
    <ItemGroup>
      <RequiredProjects Include="$(SPOCLIENT)\CLR\Libraries\SPOT_Update\dotNetMF.proj" />
      <PlatformIndependentLibs Include="SPOT_Update.$(LIB_EXT)" />
    </ItemGroup>
```

    - Add the MFUpdate PAL library

```
<ItemGroup>
  <RequiredProjects Include="$(SPOCLIENT)\DeviceCode\PAL\MFUpdate\dotnetMF.proj" />
  <DriverLibs Include="MFUpdate_PAL.$(LIB_EXT)" />
</ItemGroup>
```

    - Add the MFUpdate Provider libraries required for MFDeploy

```
<ItemGroup>
  <RequiredProjects Include="$(SPOCLIENT)\DeviceCode\Drivers\MFUpdate\dotnetMF.proj" />
  <DriverLibs Include="MicroBooterUpdate.$(LIB_EXT)" />

  <RequiredProjects Include="$(SPOCLIENT)\DeviceCode\Drivers\MFUpdate\Storage\dotnetMF.proj" />
  <DriverLibs Include="BlockStorageUpdate.$(LIB_EXT)" />

  <RequiredProjects Include="$(SPOCLIENT)\DeviceCode\Drivers\MFUpdate\Validation\SSL\dotnetMF.proj" />
  <DriverLibs Include="UpdateValidationSSL.$(LIB_EXT)" />
</ItemGroup>
```

    - Add your implementation of the IDebuggerPortSslConfig configuration

```
<ItemGroup>
  <DriverLibs Include="DebuggerPort_[YOUR_SOLUTION_NAME].$(LIB_EXT)" />
```

```
   <RequiredProjects
Include="$(SPOCLIENT)\Solutions\[YOUR_SOLUTION_NAME]\DeviceCode\DebuggerPort\dotnetmf.proj" />
 </ItemGroup>
```

- Add your implementation of the IUpdatePackage configuration

```
<ItemGroup>
   <DriverLibs Include="MFUpdate_iMXS_net_open.$(LIB_EXT)" />
   <RequiredProjects
Include="$(SPOCLIENT)\Solutions\iMXS_net_open\DeviceCode\MFUpdate\dotnetmf.proj" />
</ItemGroup>
```

6.2 HTTP(s) Firmware Update

The HttpServer sample that has shipped with the .NET Micro Framework for several versions has been updated to support uploading a firmware update image (.nmf file). The web page that is served by the Http(s) server now contains a separate file download field that when posted by the browser will start an MFUpdate procedure using the managed code portion of the framework. The sample works for both HTTP and HTTPS. The HttpServer sample is not intended to be used in production because it really only performs device validation since it is the SSL server. It is only intended to show how to use the managed MFUpdate classes.

In order to test this sample you will either need to update your solution to support MFUpdates or use the emulator. The emulator will not perform the firmware install for obvious reasons, but it will allow you to debug and see how the MFUpdate framework works. For the emulator, you can use any .nmf file you like or just create a fake .nmf file with whatever data you so choose. The advantage of using the emulator is that you can easily debug both ends of the update process (MFDeploy and the emulator) in Visual Studio.

If you want to try this sample on your solution you first need to update your device with the changes listed in 6.1.2 Device Configuration except for the "*SSL Configuration*" section. You will need to include the "HTTPSUpdate" IUpdatePackage ("NetMF" is not required) and you will need the CRC update validation library instead of the SSL update validation library. This sample should work with OpenSSL or RTIP libraries.

7 Provided Update Facilities

7.1 Update Providers

The following are subsections describe the sample MFUpdate providers supplied in the .NET MF Porting Kit. These providers implement the IUpdateProvider interface.

7.1.1 MicroBooter Provider

The MicroBooter update provider enables Firmware and Assembly updates.

Methods:

- InitializeUpdate - Only supports firmware and assembly updates.
- GetProperty - Not implemented; returns false.
- SetProperty - Not implemented; returns false.
- InstallUpdate - For firmware updates, this method saves an object in the device configuration sector that indicates to MicroBooter an update is ready. Then the device is rebooted and MicroBooter (described later) takes over. For assembly updates, the MicroBooter provider installs the assemblies in the deployment sector (erasing first if the UpdateSubType indicates a replacement) and then resets the CLR.

7.2 Update Validation Providers

The following subsections describe the sample MFUpdate validation providers supplied in the Porting Kit. These providers implement the IUpdateValidationProvider interface

7.2.1 SSL

The SSL validation provider enables SSL authentication and encryption for debugger updates by upgrading the communication port to SSL. In addition it validates the image by using the SSL certificates public key to verify the image signature.

Methods:

- AuthCommand - Used by the debugger to get the authentication type (SSL).
- Authenticate - Initiates the SSL session and saves the servers SSL public certificate for update validation.
- ValidatePacket - Validates the packet using by checking its CRC.
- ValidateUpdate - Validates the update by verifying the update signature using the server's public certificate.

7.2.2 CRC

The CRC validation provider is used by the HttpServer managed code sample to provide a rudimentary data integrity check for the update. In this case the application is handling the authentication (via HTTPS), so the validation is a simple CRC check.

Methods:

- AuthCommand - Supports the GET_AUTH_TYPE command and returns TYPE__CRC.
- Authenticate - Not implemented; returns true.
- ValidatePacket - Validates the packet using by checking its CRC.
- ValidateUpdate - Validates the update by checking its given CRC value.

7.3 Update Storage Providers

The following subsections describe the update storage providers supported in the Porting Kit.

7.3.1 Block Storage

The block storage provider uses the block storage system to store MFUpdate packets. It is specifically wired to store the packets to blocks regions with the UPDATE block usage type.

Methods:

- Initialize - Initializes the Block Storage system (only called by MicroBooter).
- Create - Creates a new MFUpdate storage with given update parameters.
- Open - Attempts to open an existing MFUpdate in the update storage region.
- Close - Closes the given update storage (for this implementation this is a no-op).
- Delete - Deletes the given existing update storage.
- GetFiles - Gets update storages for the given update type.
- IsErased - Determines if the data at the given offset and size of the update storage is erased.
- Write - Writes the given data to the given offset of the storage device.
- Read - Reads the data at a given offset of the update storage.
- GetHeader - Gets the update header for the given update storage.
- GetEraseSize - Gets the erase block size for the given update storage.

7.3.2 File System

The file system storage provider sample is not currently in the Beta but will be available for the RC. It will allow updates to be stored on the devices file system. In addition, we will also add support for writing MFUpdate files (.nmf) to an SD card (via MFDeploy). On the device side we will provide a sample managed application that recognizes the update , validates it and installs it on the device.

Methods:

- Initialize - Initializes the file system (only called by MicroBooter).
- Create - Creates a new MFUpdate storage file with given update parameters.
- Open - Attempts to open an existing MFUpdate file in the file system.
- Close - Closes the mfupdate file.
- Delete - Deletes the given update file.
- GetFiles - Gets a list of mfupdate files in the file system matching the update type.
- IsErased - Determines if the data at the given offset and size of the update storage is erased.
- Write - Writes the given data to the given offset of the update file.
- Read - Reads the data at a given offset from the update file.
- GetHeader - Gets the update header for the given update file.
- GetEraseSize - Gets the erase block size for the given update file.

7.4 Boot Loaders

The following subsections will describe the boot loaders supported in the .NET Micro Framework. At this point the only MFUpdate supported boot loader is MicroBooter.

7.4.1 MicroBooter

MicroBooter is a new boot loader application that was designed to work exclusively with the MFUpdate feature. It has two major responsibilities: bootstrapping the device; and installing updates. MicroBooter also has an optional feature that enables MFUpdate to download SREC files directly to MicroBooter in case there was a faulty install of the CLR. Because MicroBooter does not need many of the facilities that are required for TinyBooter it has a much smaller footprint. The main difference between TinyBooter and MicroBooter is how it installs new firmware. The TinyBooter requires the download of the new firmware to occur when running TinyBooter. It then installs the firmware directly into place and then checks for validation. Therefore if the validation fails you are left with no firmware on the device. The MicroBooter relies on the MFUpdate feature to store the update and perform validation. Then MIcroBooter simply installs the update which includes image validation and decompressing the image into its final location. Naturally the MFUpdate requires more storage memory than TinyBooter, but it also enables remote updates because the communication and storage takes place in the CLR. If TinyBooter had to add remote capabilities it would at least double the size of the image because the network stack would need to be duplicated in both applications: the CLR and TinyBooter.

7.4.2 TinyBooter

Tinybooter has been the standard boot loader and update mechanism for the .NET Micro Framework for most of its lifetime. It is a standalone application which can update a firmware image by receiving SREC data from MFDeploy. The image validation is a signature check that occurs after the image is installed. TinyBooter will still be supported and is still the main boot loader for a majority of the platforms. However, if your devices needs remote firmware update capabilities or you would like image validation to occur before installation, you will likely want to use the MFUpdate with MicroBooter.