# Microsoft’s.NET MicroFramework

Product Positioning and Technology Whitepaper

Donald Thompson, Software Architect

Colin Miller, Group Manager

Updated: March 21, 2007

Introduction

For years, Microsoft has had two offerings in the embedded space: XP Embedded and Windows CE. These operating systems have formed the platform for many successful devices, such as PDAs, Smart Phones, automatic teller machines, and network switching equipment. Microsoft introduced the .NET Framework, which brings the power and reliability of managed code to scale from large servers to embedded devices. However, many devices are still too small/resource constrained to support these platforms. To serve the market for even smaller devices, Microsoft has developed a new platform Microsoft .NET Micro Framework. (.NET MF). The .NET Micro Framework addresses this untapped area by providing a small, efficient implementation of the .NET runtime for smaller devices. This has great advantages to developers who can maintain the same managed code, use the same world class tools, and continue to participate in the world’s largest developer community.

Microsoft’s.NET MicroFramework 1

XPe, WinCE, and .NET MicroFramework 4

Technical Overview 5

The .NET MicroFramework CLR 6

Platform Integration and Porting 7

.NET MicroFramework HAL 7

Libraries 8

User Interface Shell 8

Visual Studio Integration 9

System Requirements 10

Runtime 10

Development environment 10

Appendix A: .NET MicroFramework CLR in Depth 12

Basic Features 12

Extended Features 12

Omitted Features 12

Type System 13

Intrinsic Types 13

Framework Types 13

Reflection 13

Serialization 13

Execution Engine 14

Responsibilities 15

Threading 15

Synchronization 15

Timers 15

Exception Handling 16

Application Domains 16

Memory Management 16

Persistent Storage 16

Resource Management 17

Interop 17

Security 17

Crypto 18

Patching 18

Appendix B: .Net MicroFramework HAL in Detail 19

Continuations 19

ISR/DSR 19

Callbacks and Events 19

Drivers 19

Sound 19

Managed Drivers 19

Communications 20

Remote Procedure Call (RPC) 20

**The .NET Platform for Smart, Connected Devices**

From sensor networks to smart watches, there are new applications that require low hardware cost and low power consumption, but would benefit from the flexibility, reliability, code reuse, and great tools of .NET. The .NET MicroFramework is a bootable runtime that extends the Microsoft embedded story to devices that are too resource-constrained to run Windows CE. The .NET MicroFramework is designed to work on devices where hardware capabilities are too limited to be practical with the full .NET or .NET Compact Framework running on Windows XP Embedded or Windows CE.

The Microsoft .NET MicroFramework has the potential to do for small embedded devices what Visual Basic first did for the PC – open the environment to a much larger pool of developers by simplifying what has been a very complex development environment. At the same time, this solution is integrated with the same Visual Studio tools that developers use for the other Microsoft embedded platforms. This means that in vertically-integrated solutions, the same resources that work on the servers and the intermediate devices, can also work on the smallest devices in a cost-effective way. The .NET MicroFramework is the platform that brings these advantages to devices too small for the other Microsoft embedded solutions.

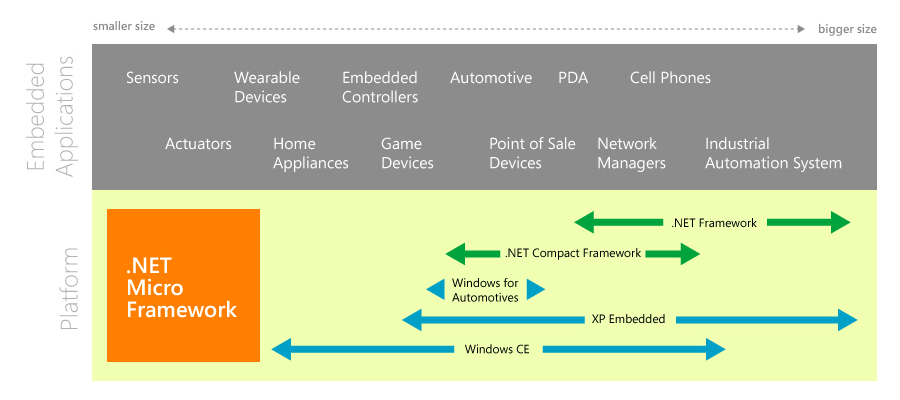
New generations of 32-bit processors are being introduced that are very power efficient and price competitive with 8- and 16-bit microcontrollers. This trend, combined with declining memory prices, makes it possible for a reliable, managed code environment to be the right choice for many projects. For any given application then, the choice is whether it is more cost effective to increase the variable costs (the costs to manufacture each device) slightly for the advantages of lower development cost, better system reliability, and the extensibility that a general platform provides. While there will continue to be a large segment of the embedded market for which the recurring costs of materials is paramount, the segment for which ease of development, code reuse, flexibility, and other advantages of higher level systems is growing markedly.

More power-efficient radios and communications protocols, along with the continued reduction in the cost of integrating these communications capabilities, are enabling a new class of connected small devices that are integrated with other devices in an active environment. Many opportunities are opening in deploying new networks of small devices in areas like industrial and home automation, healthcare, and retail.

These solutions are increasingly composed of many small devices interacting with each other and larger systems. With the .NET MicroFramework, you can now build a solution that uses .NET to support this broad range of devices. Communications and processing code can be moved from one part of the system to the other seamlessly. This type of application also benefits from the efficiency and robustness of using the tools and managed code environment enabled by .NET.

**Figure 1**

**Microsoft Embedded Products**



## XPe, WinCE, and .NET MicroFramework

Microsoft’s continuum of processing capabilities for embedded applications now runs from Windows XPe (potentially with the .NET Framework) to Windows CE (potentially with the .NET Compact Framework) to the bootable runtime of the .NET Micro Framework. Where XPe and CE have very broad and deep support for operating system extensions, the .NET MicroFramework is focused on being as small and as power efficient as possible while providing the full benefits of a managed code environment. This means that there are a number of differences, largely in the extensibility of the platform. Windows CE supports a broad range of configurations through its Platform Builder which allows configurations from about 600K through 64M. Configurations that support managed code require at least 12 M of RAM.

The .NET MicroFramework supports a subset of the .NET Framework and runs with or without an underlying OS so the footprint can be smaller. Current configurations of the MicroFramework require about 300K of RAM. In addition, no Memory Management Unit is required so the MicroFramework can run on less expensive processors.

The strengths of the platform are:

* Lower hardware cost than other high end platforms
  + Much smaller memory footprint
  + Less expensive processors (no MMU)
* Lower development costs than other embedded platforms
  + Managed code
  + Broader developer pool familiar with C# and .NET development
  + Better toolset through Visual Studio
  + Full support for on-device debugging in Visual Studio
  + Code reuse with other parts of the solution
  + Initial development on the PC using extensible emulation tools
* Lower power consumption than high-end platforms, improving battery–life on small devices

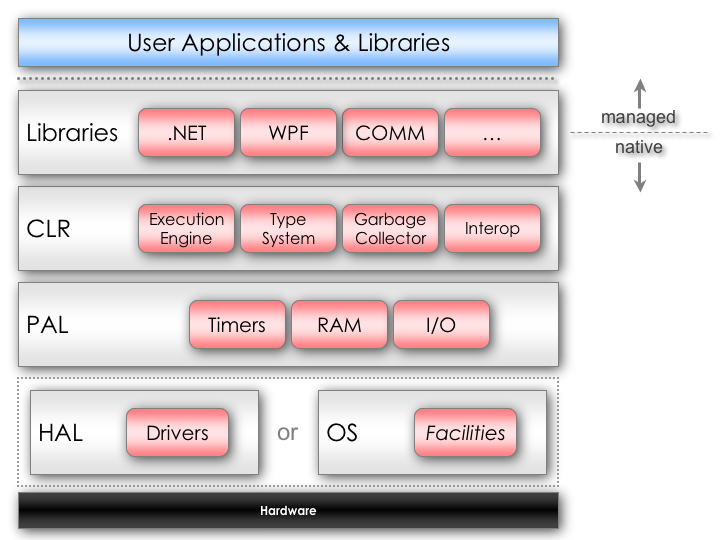
.NET MicroFramework is a first step for Microsoft into this domain. It is an area with continued investment and ongoing research into finding ways to scale our offerings down to smaller form factors to enable the use of higher level languages and world class tools that span the full range of software development.

# Technical Overview

As stated above, the .NET MicroFramework is a bootable .NET runtime. The architectural overview in Figure 2 below shows the components that make up the runtime. In the following sections, we talk first about the .NET MicroFramework Common Language Runtime (CLR) since that is the core of the system. We also discuss the library support which defines the breadth of applications that can be created on this platform. Finally, we examine the platform integration which defines the magnitude of the porting effort to new platforms and how the runtime can stand alone without an underlying OS or can be ported onto an existing OS.

**Figure 2**

**The Basic Architecture for .NET MicroFramework**



***A ‘bootable’ runtime***

The MicroFramework provides a subset of full OS features, so it does not require an OS and is referred to as a ‘bootable’ runtime. Making the runtime ‘bootable’ means that booting support, interrupt handling, threading and process management, heap management, and other environmental support functions that are typically provided by the operating system have been added to the .NET MicroFramework so that it can run directly on hardware. Runtimes such as the .NET Framework run on top of a general and fully featured OS and utilize the support of that OS but provide a more protected application environment. The OS services that are needed by the application are provided up through the runtime. In creating a bootable runtime, the .NET MF provides a subset of the full OS features directly rather than relying on the underlying OS. This means that in place of the OS, the .NET MF , sitting on a small HAL (20-30K). The services provided by the .NET MF are a subset of a full general OS with this subset being selected as services required to run applications on small devices.

## The .NET MicroFramework CLR

The core of this platform is the .NET MicroFramework Common Language Runtime (CLR); a small, optimized “managed code” runtime based on industry-standard ECMA specifications. Managed code is different from native code since it runs within the context of the CLR which provides an environment where the execution is safe from common execution errors. When native code executes, the operating system really doesn’t know anything about the code and therefore can only execute it blindly. Managed code includes a great deal of information about the code so that the CLR can prevent errors.[[1]](#footnote-1)

By supporting managed code, the .NET MicroFramework CLR allows for rapid development and safe execution of application code using modern programming languages and tools, namely **C#** and **Visual Studio .NET**, allowing development teams to more effectively focus their resources on delivering customer value.

Where appropriate, links to Microsoft Developer Network (MSDN) documentation have been provided *in situ* for background and common concepts shared across .NET implementations.

For general information on .NET and managed code, please visit:

* [.NET Overview](http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx)
* [ECMA Standards](http://msdn.microsoft.com/net/ecma/)
* [Managed Code technical talk](http://msdn.microsoft.com/theshow/episode.aspx?xml=theshow/en/Episode035/manifest.xml)

The .NET MicroFramework CLR serves the same purpose as its larger cousins ([.NET Framework](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthecommonlanguageruntime.asp) and [.NET Compact Framework](http://msdn.microsoft.com/smartclient/understanding/netcf/default.aspx)): to load and execute managed code. The benefits of managed code are many, but the main advantage is that the runtime environment can provide safety: security, protected resources, validation, recovery, isolation, etc. Through the [Base Class Library (BCL)](http://msdn.microsoft.com/netframework/programming/bcl/default.aspx) and language features, developers become more productive by not having to worry with infrastructure issues as is typical in native code development.

The .NET MicroFramework CLR brings these desirable characteristics to resource-constrained embedded devices. The key implementation distinctions can be found in .NET MicroFramework CLR’s design goals:

* Minimize software footprint
  + Run without a traditional OS by integrating functionality directly into the runtime engine
  + Selective inclusion of runtime and library functionality appropriate to application/device
* Executable from ROM and FLASH memories
* Optimized for energy efficiency (i.e., for battery-powered devices)

## Platform Integration and Porting

There are two potential approaches to implementing the MicroFramework on a specific hardware platform. The platform can be ported on an existing OS and function as a normal runtime. Alternatively, the platform includes a Platform Adaptation Layer (PAL) and a Hardware Adaptation Layer (HAL) that support the runtime directly on the hardware. Deciding which direction is appropriate depends entirely on the specifics of the application. The PAL and HAL provide two layers at which the integration of the platform can be made. The PAL support higher level integration including asynchronous communication calls, high level timers, list and data structures, and more. The PAL is implemented on the HAL which supports all the low level interfaces and works directly with the hardware. The HAL provides generic access to all peripherals and drivers. Drivers are common across all platforms and operate through opaque queued I/O. This design shields the driver from ISR and queue-locking details.

The porting process requires the changes at primarily the HAL level to either adapt the drivers to new hardware or to implement the equivalent support through the underlying OS. The Porting Kit includes the HAL and PAL source for a default reference platform. Currently, the platform is built using the ARM toolset (versions 1.2 and 3.0 are supported) with support for ARM7 and ARM 9 as well as Win32 through simulated drives for the emulation story.

### .NET MicroFramework HAL

When no underlying OS is present, the .NET MicroFramework HAL provides the necessary infrastructure for booting and interfacing with the hardware and peripherals. The functionality of a traditional OS has been factored between .NET MicroFramework HAL and CLR forming what could be considered an embedded OS. The .NET MicroFramework HAL provides the critical interface to hardware (IRQs, timers, I/O), but does not have a full kernel to manage processes and threads or any heaps of its own; it defers these functions to the .NET MicroFramework CLR.

The simplified .NET MicroFramework HAL kernel supports two execution modes: single-threaded application and interrupt service routine (ISR) – each has its own set of unique rules that must be followed.

There is no scheduler *per se*, as there is only a single application thread. .NET MicroFramework CLR provides all user applications (the shell, Network protocol, alerts, games, etc.) execute within the runtime, which has its own facilities for providing traditional multithreading. Having a single application thread also removes locking overhead. However, this means that the application running on .NET MicroFramework HAL (i.e., .NET MicroFramework CLR) must explicitly yield execution periodically (idle time) in order for continuations to be processed (cooperative multitasking).

Another key design decision relates to the expectation of low CPU utilization to reduce power consumption. This has resulted in simpler code, lower power consumption, deferred service routines (DSR) versus "do it now" versus "sleep in place" approaches, reducing clock speed while doing short spins, and other tricks.

## Libraries

**.NET Base Class Libraries**

The functionality of the .NET base class libraries has been covered in detail on the technical discussion of the .NET MicroFramework CLR. In summary, the MicroFramework supports some of all of:

|  |
| --- |
| System.Reflection |
| System.Threading |
| System.Collections |
| System.Resources |
| System.Runtime |
| System.Diagnostics |
| System.IO |
| System.Globalization |

### User Interface Shell

Extensive support has been provided for development of user interfaces for these small devices. This includes input, layout (content-sizing, text flow, controls), fonts, images (bitmaps, GIFs), pens, brushes, colors, drawing primitives (rectangles, circles, etc.), window management, popups / alerts, etc.

The application model loosely follows that of the Windows Presentation Foundation, though without the declarative markup features (XAML). This means that developers familiar with applications development on Vista will be comfortable writing UI for applications running of these small devices.

The **layout system** refers to the protocol and associated API that enables sizes and positions to be allotted for user interface elements to render in. As with everything else in the MicroFramework, the platform is optimized for embedded devices. The primary difference is the lack of division between “Core” and “Framework”, for two reasons:

1. Size and performance: by adding additional layers, the cost to run on the small form factors becomes excessive
2. Due to the smaller form factors and the tying to a device release, the interoperability and lifetime/upgrade scenarios for software built on this stack are greatly reduced.

The layout system adds support for user-specified sizing, positioning, margins, etc., of specific layout elements.

The primary goals of this system are:

* Size to Content --- allows controls to size themselves according to the content they present, dynamically at runtime (as opposed to statically at authoring-time)
* Universal Properties --- allow developers to control (programmatically) the behavior and appearance of layout elements in a consistent, intuitive, and succinct manner
* Globalization Support --- allows products developed for different language / cultures to easily be adapted without programming overhead
* Multiple Device Support --- allows products with different display capabilities to share a common mechanism, reducing development time and complexity

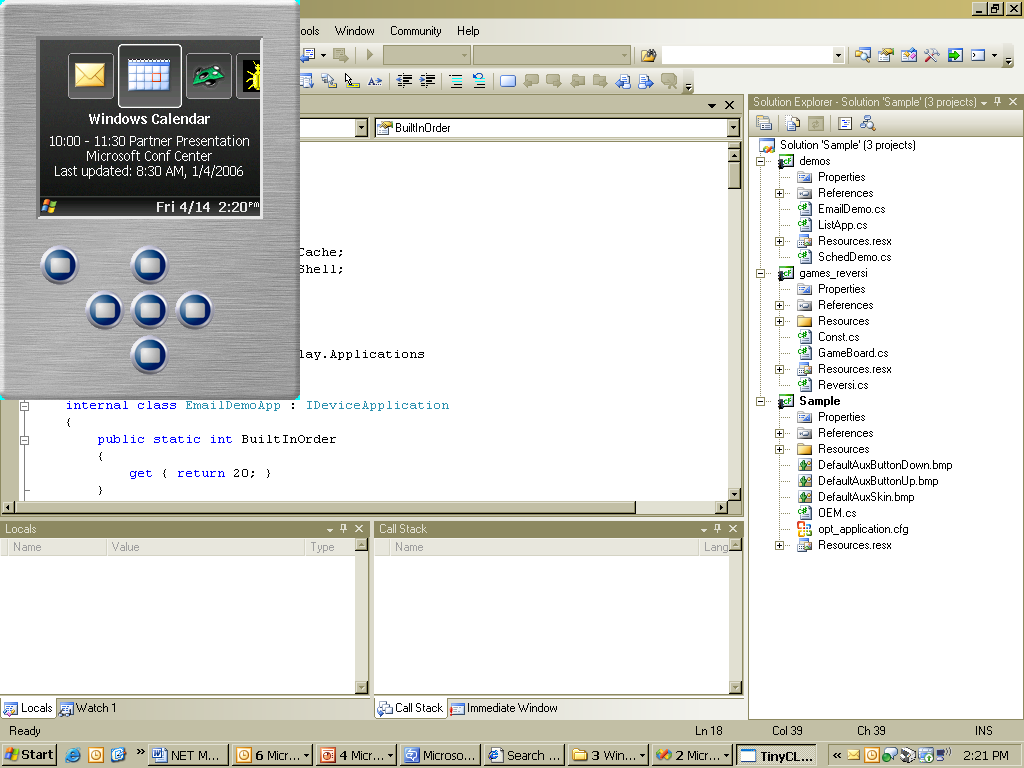
## Visual Studio Integration

Development for the .NET MicroFramework platform is done using [Visual Studio.](http://lab.msdn.microsoft.com/vs2005/default.aspx) . This integration supports full emulation of your device on the PC, building and deploying the assemblies to the device, and debugging the code running on the device through USB or serial connections – all from within Visual Studio.

Initial development can be done on the PC through an **emulation** environment provided which allows application developers to write much of their code without worrying about the device. This emulation environment is designed to be extensible so that any device can be emulated on the PC. Development at this level offers extensive utilities, such as logging, heap analyzers, profilers, to assist in understanding resource utilization and performance of managed code.

**Figure 3**

**Extensible Emulation in Visual Studio 2005**

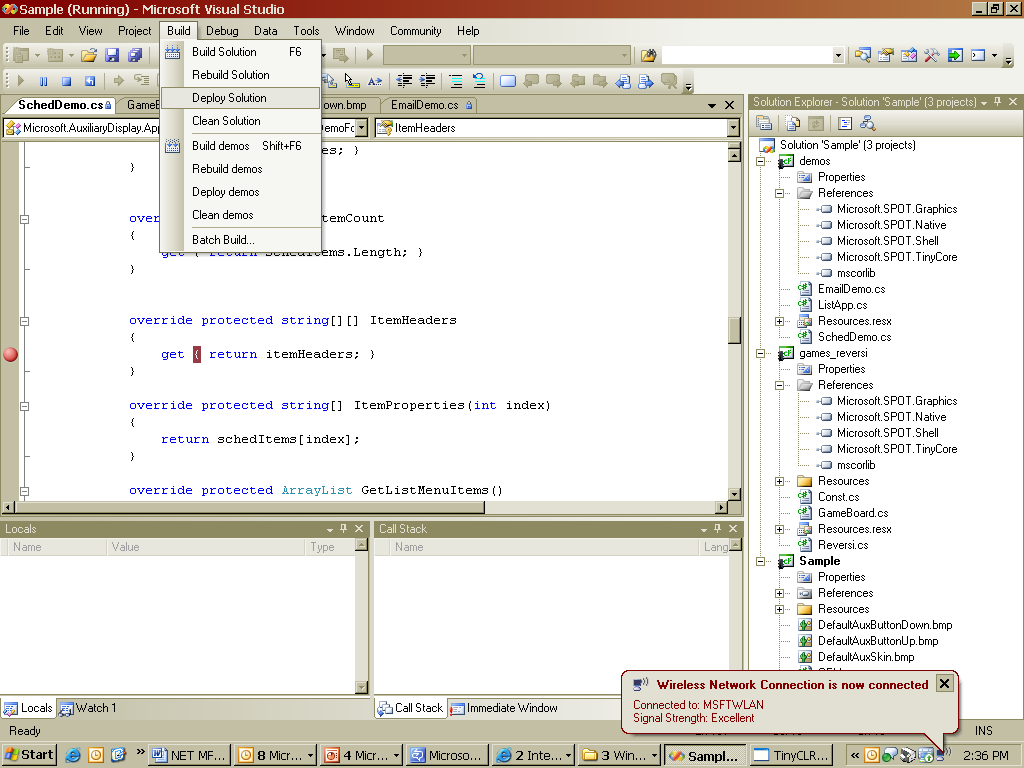


While PC emulation has lots of advantages, it is not a perfect substitute for running the code on the device. So, we have extended the project system of Visual Studio 2005 to specifically target development on these devices, including flashing and interactive code-level debugging. Using a serial or USB connection, the **RPC** subsystem provides a live communication channel to the device, allowing downloading the application to the device within Visual Studio, runtime variable inspection, breakpoints, and all of the expected features found in visual debugging. For example, if you have a device that communicates with the PC over Bluetooth, you could be debugging both sides of the communications in that same session of Visual Studio stepping through code running on the PC and then following breakpoints right into code running on the device.

The [assemblies](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassemblies.asp?frame=true) (binaries) generated by Visual Studio are not usable directly. The **Metadata Processor** tool is used to convert the standard .NET assembly into the format required by the .NET MicroFramework CLR. This post processing further reduces the size of the assemblies. This step is fully automated in the Visual Studio integration and occurs as seamless part of deploying the assemblies to the device.

**Figure4**

**Build, Deploy, and Debug on the device in Visual Studio**



## System Requirements

### Runtime

Currently the platform runs on several ARM 7 and ARM 9 processors. The suggested minimal memory configurations at present are 300K RAM and 1 Meg of Flash. The devices also need to have serial or USB support to support downloading and debugging the applications during development.

### Development environment

The development environment requires Visual Studio 2005 to support the full integration with the devices. The SDK installs add-ins on top of Visual Studio so the install will fail unless Visual Studio is already installed. These add-ins support on-device debugging. In addition, project templates are installed for MicroFramework projects. A serial or USB connection between the PC and the device is also required.

# Appendix A: .NET MicroFramework CLR in Depth

.NET MicroFramework CLR implements all of the major features found in the full CLR (execution engine, type system, memory management, etc.). It omits a few features that were deemed inappropriate for the this class of device, and adds a few features that are specific to the this class of device. Rather than providing an exhaustive enumeration of all differences, this description is limited to the most salient.

## Basic Features

* Numeric types, Class types, value types, arrays (mono-dimensional only), delegates, events, references, weak references
* Reflection (extremely important, it's the basis for the extensibility of the system)
* Serialization (not true .NET serialization, but a more compact version)
* Garbage collection (GC) and defragmentation of memory (vital for safe, recoverable application environments)
* Non-incremental Mark and Sweep GC (Generational is overkill for small RAM when energy-efficiency is primary)
* Self-describing data (as opposed to separate metadata, leads to faster GC)
* Deep exception handling (very important for fault tolerance)
* Time-sliced thread management (20ms quantum, interruptible, prioritized)
* Multiple AppDomains

## Extended Features

* Execution constraints (limits call durations; prevents lock-ups)
* Strings are represented internally as UTF-8 and exposed as Unicode
* Value types are emulated by runtime (not first class), allowing for GC optimization and saving RAM
* Global, shared string table for well-known values (types, methods, fields) reduces RAM/ROM, and cross-references between assemblies
* Patching, which includes assembly differencing, method and resource substitution, and new types
* No virtual tables for method resolution saves RAM
* "Not so weak" references are prioritized and persisted (RAM, EEPROM, FLASH)
* Specialized garbage collector provides "automatic" data management (stale content, etc.) and persisted storage management
* "Weak" delegates mean that runtime manages dangling references
* Custom serialization engine relies heavily on new attributes to guide encoding

## Omitted Features

* Machine-dependent types (unsafe instructions) prevent direct memory access by managed code
* SMP involves simpler runtime and is inappropriate for our goals (single processors are the norm)
* MSCORLIB functionality reduced to what is appropriate for these devices
* Multi-dimensional arrays (sparse, jagged)
* Exception handling in native code; processor faults (traps) and external watchdog only
* MMU support

## Type System

.NET MicroFramework CLR implements a Common Language Specification-compliant [common type system](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthecommontypesystem.asp?frame=true) used to represent and manage classes and other data structures used by the system and applications. In addition to supporting built-in (intrinsic), framework, and user defined types, the type system also stores [metadata](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmetadataselfdescribingcomponents.asp?frame=true) used by the system to perform all other operations, such as program loading and execution, memory management, patching, etc. The type system is tightly bound to the memory management and execution engine subsystems.

### Intrinsic Types

The ECMA standard requires support for basic language and runtime features, including:

* Numeric types --- integers (signed and unsigned, 8/16/32/64 bits) and floating point / doubles
* String types --- UTF8 encoding (exposed as Unicode)
* Class types --- including lock semantics
* Binary blobs --- image and sound data, for example
* Collections --- Array Lists, Enumerators,…
* Arrays --- mono-dimensional only at this time
* Delegates --- function pointers
* Events --- multiple dispatch
* References --- including weak references

.NET MicroFramework CLR also offers a so-called *extended weak reference*. A reference is merely a pointer to some other object. A weak reference is a pointer to an object that might be reclaimed during Garbage Collection (CG), under sufficient memory pressure. This is useful if the referenced object is large, but can be easily recreated as needed. An extended weak reference allows for the referenced object to be stored / retrieved automatically by the runtime using persisted memory, such as across a device reboot. This mechanism greatly simplifies the programming patterns found when references are used. See Persistent Storage below for more.

### Framework Types

.NET MicroFramework CLR offers a subset of the BCL and [.NET Framework](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthenetframeworkclasslibrary.asp). This includes many of the objects defined in the [System](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystem.asp) namespace. As previously mentioned, this consists of ~70 classes representing ~420 methods, as opposed to the ~1450 classes and ~22500 methods of the full implementation. In practice, for many applications in this class of device, this has proven to be sufficient. It is relatively straightforward to add support for special features as needed.

No support is yet provided for databases (System.Data.\*), web (System.Web.\*), XML (System.XML.\*), etc.

## Reflection

[Reflection](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconReflectionOverview.asp) is the ability to obtain metadata for every object in the system, including classes, methods, fields, etc. This functionality is made available to application programs and is used for serialization (see below).

## Serialization

[Serialization](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrserializingobjects.asp?frame=true) is the ability to convert an active object (in memory) into a compact representation for storage or transmission such that the object can be re-created at a later time or on a different device. The serialization engine in .NET MicroFramework CLR was specifically designed to create very compact representations, as this mechanism is used to communicate to the device as a “wire protocol”. Desktop and server systems aren’t as concerned with this as they typically have much greater bandwidth than small devices. Using attributes (a language feature), it is possible to provide *hints* to the serialization engine which can produce optimally compact representations. These include:

* Flags, bit field describing the serialization of this object:
  + SerializationFlags.PointerNeverNull, the pointer to the serialized object is never null.
  + SerializationFlags.ElementsNeverNull, the elements in the serialized object are never null.
  + SerializationFlags.FixedType, the serialized object can only be an instance of the specified class, and not an instance of a derived class.
* ArraySize, when the element is an array, this is the size of the serialized array. A value of -1 indicates that the array extends to the end of the stream.
* TypeOfElements, the Type of the elements in the array.
* BitPacked, the number of bits in which this object value is bit-packed.
* RangeBias, the bias adjustment for the serialized value.
* Scale, the scale adjustment for the serialized value. For time, it's in ticks.

To serialize an object:

byte[] Microsoft.SPOT.Reflection.Serialize( object o, Type t )

To deserialize it:

object Microsoft.SPOT.Reflection.Deserialize( byte[] v, Type t )

## Execution Engine

As with all managed code runtimes, the .NET MicroFramework CLR provides a virtual machine abstraction that is targeted by the compiler tools from the source programming language. In order to execute programs, a special engine is required to load, prepare, and perform the instructions specified in the program (the [Intermediate Language](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconMicrosoftIntermediateLanguageMSIL.asp) [IL]). One of the key benefits of this design is that it allows for code to be developed independently of the actual hardware on which it runs.

Since programs are specified in terms of a virtual machine, the execution engine must translate encoded operations into *native* operations (i.e., instructions that can be performed by actual hardware). There are two basic approaches (and a myriad of variations) to accomplishing this: *interpret* the code as it is executing or [*compile*](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconexecution.asp) it to native code, as it is needed ([Just In Time](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconjitcompilation.asp) [JIT]). .NET MicroFramework CLR interprets program code. There are pros and cons to these approaches (beyond the scope of this document). For .NET MicroFramework CLR, the benefits were seen as:

* Smaller code footprint
  + Managed code is represented in less space than native code
  + Avoids two copies of the program (IL + compiled)
  + Less engine code required to implement
* Safer
  + Fine-grained execution (instruction level) never passes control from engine (i.e., increased determinism)
  + Less complex engine code (i.e., fewer moving parts)

The primary disadvantage of the interpreter approach is its decreased execution speed, as (theoretically) more code is executed. To help overcome this, .NET MicroFramework CLR has been designed to allow a combination of IL and native code (i.e., mixed-mode execution) through an interop mechanism (see below). When execution speed is critical, native code can be written and invoked from managed code.

### Responsibilities

The execution engine is responsible for coordinating the actions of the various parts of the system:

* it instantiates the heap as part of the boot sequence
* it keeps track of the heap cluster that returned the memory for the last allocation, to speed up the next allocation and also to minimize fragmentation to the heap
* it owns the list of all the weak references, used during GC to properly update them
* it owns the list of all the timers and it updates and schedules the next timer to trigger, implementing a timer cache to avoid re-evaluating the timer settings all the times
* it owns the list of end points
* it owns the list of all the threads, organized in a ready, waiting, and zombie list
* it owns the list of all the objects needing finalization, and it controls the thread used to execute the finalizers
* it controls the global lock used to implement the semantic behind the GloballySynchronized attribute
* it owns the list of all the other dual objects
* it owns the set of breakpoints in a debugger-enabled build

It also provides helper methods for commonly used operations:

* create new object
* new thread and thread rescheduling
* variable initialization from metadata
* object cloning needed to emulate the ByVal semantic
* access to member fields through reflection
* lock/unlock objects
* sleep a thread
* suspend a thread waiting for events
* determine if an object is an instance of a type

### Threading

.NET MicroFramework CLR provides [multi-threading](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthreading.asp?frame=true) support, even when the underlying platform does not. While not a true multi-threaded kernel, the execution engine simulates one by offering time-sliced context switching using 20ms quantum. Threads are prioritized and interruptible (due to instruction level interpretation).

### Synchronization

Accesses to shared resources between multiple threads of execution are [synchronized](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmanagedthreadingsupport.asp?frame=true) to avoid corrupting state.

### Timers

.NET MicroFramework CLR offers support for various timers, through System.Threading.Timer and Microsoft.SPOT.ExtendedTimer. The various behaviors include:

* Recurring vs. one-shot
* Absolute time vs. relative time
* Int32 vs. TimeSpan
* Event-based (like second, minute, hour, time zone change, etc.)

.NET MicroFramework CLR timers are implemented in a slightly different way than the standard runtime in that they do not attempt to invoke callbacks based on a frequency. Instead, .NET MicroFramework CLR timers either clean themselves up or reschedule themselves for another firing. This avoids the situation where timer overhead is greater than the work being performed in the callback and makes for a more stable, reliable system. It is also possible for the application to compensate for the latency incurred by the system in executing timers, allowing them to be more accurate.

### Exception Handling

One of the keys to creating a safe execution environment is the ability to effectively handle exceptional runtime situations, which might occur as a result of unforeseen operating conditions. Applications are allowed and encouraged to use the exception handling mechanisms found in traditional .NET programs. The execution engine manages the dispatching and clean up associated with exceptions. In addition, it has special mechanisms for protecting the integrity of the system as a whole from unruly programs, by gracefully cleaning up state that could be causing a program to misbehave. Ultimately, a program will be “blacklisted” if it continues to fail according to a pre-determined heuristic.

### Application Domains

[Application domains](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconapplicationdomains.asp) (AppDomains) provides isolation between applications.

## Memory Management

One of the advantages of managed code is that memory management is tied to the type system, making it possible to track memory at the object level. This allows the system to reclaim “dead” objects and removes this burden from the programmer. In addition, memory can be “compacted” to create contiguous free space for newly allocated objects, thereby avoiding out of memory conditions due to heap fragmentation. This is known as Garbage Collection (GC).

The approach to GC used in .NET MicroFramework CLR differs from the [automatic memory management](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconautomaticmemorymanagement.asp) provided by the CLR in several respects. Instead of Generational, the .NET MicroFramework CLR uses a non-incremental Mark and Sweep algorithm. This approach was chosen due to the limited availability of RAM in the targeted small devices. More sophisticated approaches were explored, but the tradeoffs didn’t warrant them. Several optimizations have been made to make garbage collection fast and efficient, such as embedding metadata directly into the heap blocks in order to avoid table look-ups.

Support for persistent storage (see below) has been added to memory management system in .NET MicroFramework CLR, making it trivial to store and recover data from non-volatile memory, across device resets and reboots.

### Persistent Storage

As previously mentioned, for small devices it is sometimes necessary to persist application information between active instances (applications may start and stop based on user action or during a reset/reboot). For example, the Smart Watch applications (i.e., channels) are not all active simultaneously; the user can quickly switch between them. The channels each require configuration information as well as received content to be easily recoverable in order for them to function.

The .NET MicroFramework CLR’s GC mechanism has been enhanced to include support for this situation by writing / reading objects from non-volatile storage (EEPROM and FLASH). This acts as simply a backing store for memory (similar to paging in traditional systems). The objects are prioritized and time stamped, so that non-volatile storage can itself be garbage collected. Extremely important items (i.e., highest priority) are, in practice, never removed.

Many safeguards have been implemented to ensure robustness in this mechanism. Due to the physical characteristics of FLASH chips, it is possible to change individual bits from the “1” state to the “0” state, but the inverse is not so. For this reason, the system uses two identical storage banks. Each bank is treated like a tape, where most of the updates happen at the “tail”. When the end of a bank is reached, the other bank is erased, all the data is copied from the active bank to the spare bank, and then the role of the two banks is reversed. Writing data in the bank as if it were a tape also helps increase the lifetime of the part, since they are generally finite.

Support for traditional file systems is not yet provided, but could easily be added as a library component.

## Resource Management

A [resource](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingusingresources.asp?frame=true) is any non-executable data that is deployed with an application. Resources make localization straightforward, and serve as the basis for fonts, bitmaps, and sounds. Where possible, a consistent treatment is given to .NET MicroFramework CLR resources as the desktop CLR. The primary differences are:

* no support for extensible resources
* resources are identified by Int16, as opposed to strings (reducing footprint)
* a different ResourceManager class is provided, in the SPOT.Native namespace, to handle managed resources from within code

Visual Studio “auto-generates” type-safe code for accessing resources, thereby avoiding easy to make mistakes when accessing resources by number.

Resources can be shared across applications, but each application is responsible for consistent naming conventions and orderings.

## Interop

It is often necessary or desirable to invoke native code functions from managed code, generally for reasons of performance. This process is called [interop](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconinteroperatingwithunmanagedcode.asp).

The .NET MicroFramework CLR supports this mechanism by allowing individual methods to be marked as “native.” The developer provides native code implementation and has access to the .NET MicroFramework CLR object model. There are number of restrictions imposed on this style of implementation due to the lack of true multi-threading support in .NET MicroFramework HAL, stack size. This is an area of ongoing development.

## Security

There are several features of the platform that are designed to assist the user in creating secure devices. One is App Domains which are lightweight processes. This is ensures that one application can not interfere with the execution of another either by failing or by directly accessing code or resources from another app.

Another feature is that all assemblies that are loaded onto the platform have to be signed. Currently, they all have to be signed with a single key. This insures that only applications for the one publisher are loaded into the system.

## Crypto

The platform contains several libraries to help secure the system. RSA libraries are included for key generation. A symmetric encryption algorithm is supported through XTEA – an extension of the Tiny Encryption Algorithm which corrects a weakness in the original TEA Algorithm from [David Wheeler](http://en.wikipedia.org/wiki/David_Wheeler) and [Roger Needham](http://en.wikipedia.org/wiki/Roger_Needham) of the [Cambridge Computer Laboratory](http://en.wikipedia.org/wiki/University_of_Cambridge_Computer_Laboratory). The hashing algorithm used is Davies-Meyer modified to support the block size of XTEA. In the original watch devices that were built on this platform, each device has a symmetric key and an RAS key pair to support secure delivery of all content only to devices that are authorized to receive it.

## Patching

The .NET MicroFramework CLR supports several types of patching. This is important for remotely updating the software on a deployed product where flashing a new image is not feasible. While a patch to the runtime (including HAL and drivers) is possible, it is less common than patching managed code. There are a few restrictions to patching managed code. The main rule is that a change cannot remove things. So you cannot delete a method or remove a field. Some additions are allowed, but no new fields are permitted. Disallowed changes include:

* Delete a type
* Change the super class of a type
* Change the set of interfaces implemented by a type
* Change the kind of a type: a struct cannot be changed to a class, or an enum to a value type
* Add, remove, or modify any fields of a type
* Add or delete methods of a type
* Add or delete resources
* Change any of the custom attributes

# Appendix B: .Net MicroFramework HAL in Detail

## Continuations

Continuations are the primary mechanisms for deferring work from an ISR, which must be executed very quickly, to the application thread, which typically isn't under strict time constraints. Continuations are implemented as a FIFO queue that can be scheduled within the application thread. The typical target is to execute less than 10 mSec of work with specifiable minimum contiguous duration.

ISR/DSR  
ISRs are maskable and have a worst case latency of ~100 uSec. The longest ISR in the system is ~1mSec.

Deferred service routines (DSR) allow processing to be performed at a specific time or when a particular event occurs. They can be processed on either the APP thread or from within an ISR. A time-deferred call has a granularity of 40uSec. An idle CPU is treated as a consumable resource.

## Callbacks and Events

.NET MicroFramework HAL supports callbacks, which will fire from continuation processing on the application thread. An "event conditional sleep" mechanism allows the application thread to place the system in a dormant but alertable state. This allows the system to save power during idle periods.

The .NET MicroFramework HAL can communicate with the .NET MicroFramework CLR through event masks. Before entering into the interpreter loop, the .NET MicroFramework CLR checks for signals in any of the event masks. Should it find activity, it will dispatch into the appropriate handler code for further context-specific processing.

## Drivers

The existing drivers include: buffered RS232 I/O, SPI (13.8MHz) as a shared resource for peripherals, DirectBand Radio (QPSK), monochrome LCD (120x96), battery monitoring (temperature + voltage = state of charge), Bluetooth, 802.15.4, Flash memory (parallel), EEPROM memory (serial), OEM-specific (heart rate monitor, air pressure sensor, touch screen), calibrated accurate time, boolean outputs (backlight, vibrator), boolean inputs (buttons), and PWM outputs (Piezo, slow-start backlight, vibrator).

Drivers have been written for a number of specific parts (various EEPROMs, LCDs, battery monitors, etc.) using different buses and interconnects (SPI, HPPI, GPIO, etc.). .NET MicroFramework HAL also supports DMA, on .

### Sound

Support for Piezo-based sound effects is provided. An experimental polyphonic sound generator has also been developed.

## Managed Drivers

.NET MicroFramework CLR makes it possible to write drivers using managed code. Managed implementations exist for GPIO, PWM/VTU, I2C, SPI, USB 1.1, and USART.

## Communications

Small devices are much more interesting when they can communicate with each other and as components in larger deployments. As part of the initial drops of the SDK, we have sample drivers for several personal area networking protocols. These include BlueTooth for communications with PC and cellphones and 802.15.4 for communication with sensor Networks.

## Remote Procedure Call (RPC)

The MicroFramework platform offers an RPC-like mechanism for communication between a host system (i.e., PC) and a small device. This is a low-level mechanism implemented using an endpoint address/port approach.

Each endpoint has an address, m\_addr. A list is maintained of pending messages, m\_messages, both from and to the endpoint. The field m\_cmd of a Message object defines the role and direction. The managed layer calls GetMessage to fetch the next message from the queue, SendMessage / SendMessageRaw / ReplyRaw to create an outgoing message. The applications normally use the endpoints through the Microsoft.SPOT.RpcServer and Microsoft.SPOT.RpcClient helper classes.

The RPC layer ensures reliable delivery of messages across the link (serial or otherwise).

1. This is a high level description of managed code. There are a number of other differences such as the use of a common intermediate language (MSIL) as well as the language differences found in C# and other managed code languages. [↑](#footnote-ref-1)