Porting the .NET Micro Framework

A Microsoft Technical White Paper

December 10, 2007

Abstract

If you are planning to develop a small device or a hardware platform for small devices, consider basing it on the Microsoft .NET Micro Framework. The .NET Micro Framework not only inherits the hardware-independence of .NET, it includes two abstraction layers that further isolate applications from hardware details. This makes it straightforward for board makers, device makers, and developers of Windows SideShow-capable devices to port the .NET Micro Framework to new hardware platforms.

This white paper introduces the .NET Micro Framework architecture with a view toward porting it to a new hardware platform. It then discusses the essentials of the porting process, including the structure of the .NET Micro Framework Porting Kit, the required skill sets, estimates for resource commitments, and contacts for obtaining more information on signing a porting agreement or finding a porting specialist.

Legal

Table of Contents

1

The Portable .NET Micro Framework

The .NET Micro Framework is a new execution model for extremely small, resource-constrained devices. It brings the advantages of .NET—the security and reliability of managed code, the ease of development in Visual Studio, and the power of the C# language and the .NET libraries—to an smaller class of device than Microsoft has ever targeted before. To put this in perspective, the .NET Micro Framework is about one fortieth the size of the smallest managed configuration of Windows CE. The footprint of the .NET Micro Framework is a few hundred kilobytes rather than several megabytes.

Another characteristic of .NET, often overlooked because most .NET applications are developed on and run on some flavor of Windows, is hardware-independence. Traditionally, embedded software has been tightly coupled to the hardware on which it runs. In .NET, however, applications are compiled to an intermediate language (IL) rather than to machine code, then are executed by a virtual machine called the .NET Common Language Runtime (CLR).

Rather than starting with an existing .NET platform and trying to slim it down, Microsoft re-imagined .NET as it might look on a device with only a few hundred kilobytes of RAM. The .NET Micro Framework CLR was built from scratch to ECMA specifications and is not derived from any prior implementation.

The .NET Micro Framework not only inherits the concept of the CLR, it incorporates two abstraction layers that help to further isolate applications from the hardware. Separating the hardware-dependent code from the application code allows the .NET Micro Framework to provide great flexibility in hardware selection and increases potential for code reuse from one device to the next.

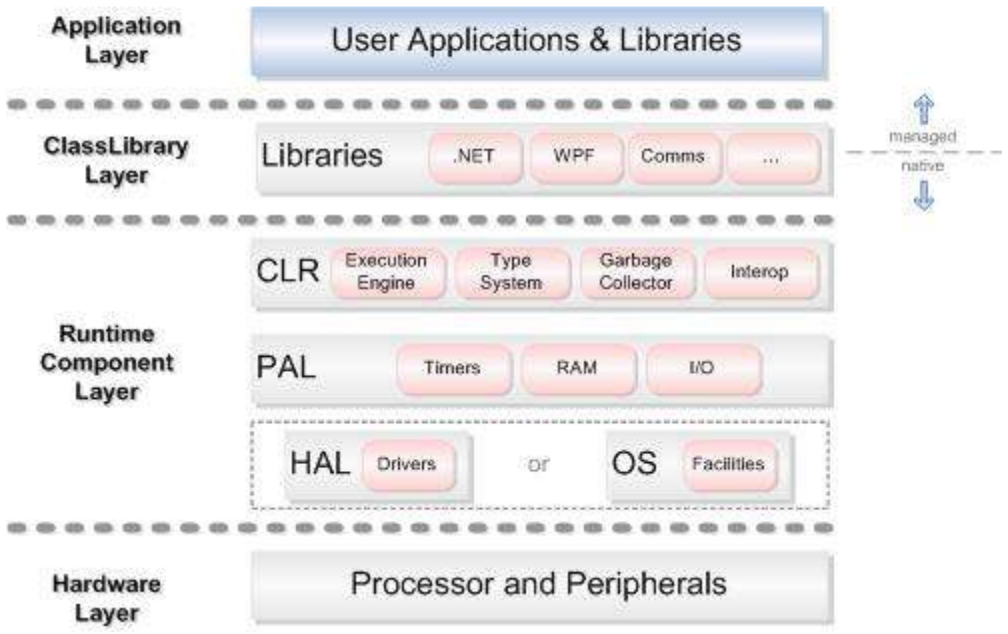
In addition to the CLR, the .NET Micro Framework also provides tight Visual Studio integration (including the ability to deploy managed code to a device and debug it while it's running there) and an extensible emulator that lets developers run and test their embedded applications right on their desktop PCs.

Porting the .NET Micro Framework is the process of adapting it to a new processor, memory configuration, support chips, and/or display and building device firmware capable of running managed applications. A number of successful ports have already been performed by Microsoft and its partners.

If you would like to get the .NET Micro Framework running on an existing hardware platform, or are building a device from scratch and wish to use the .NET Micro Framework as the basis for your product, you may need to perform a port. The purpose of this white paper is to introduce you to the concepts involved in porting and to give you the information you need to decide whether you wish to perform the port in-house or whether it would be better to engage a specialist.

.NET Micro Framework Architecture

To better understand what is involved in porting, you will need some familiarity with the .NET Micro Framework architecture. The platform is divided into the four distinct layers shown in the following diagram.[1]



The *hardware layer* contains the microprocessor and the support circuitry that together constitute your hardware platform. Currently, the .NET Micro Framework runs on several processors based on ARM7, ARM9, and XScale. We plan to support other architectures as time goes on.

In the .NET Micro Framework, two separate abstraction layers isolate the common language runtime (CLR) from the underlying hardware. These are the hardware abstraction layer (HAL) and the platform abstraction layer (PAL). Together, the CLR, the PAL, and the HAL are the heart of the .NET Micro Framework, allowing managed code to run on a small device and supporting both the class library and application layers.

[1] For more information about .NET Micro Framework internals, see the .NET Micro Framework white paper available for download from the .NET Micro Framework page at MSDN.

The HAL is a collection of primitives that provide access to hardware functions. Every peripheral device or interface natively supported by the .NET Micro Framework is represented by a collection of related functions, or API, in the HAL. (In the .NET Micro Framework, sockets-based networking is also considered a hardware function and is a part of the HAL.) A typical HAL consists of 20-30 KB of object code, or on the order of one-tenth of the code in the .NET Micro Framework as a whole. Developing a new HAL implementation for your hardware is usually the primary task involved in porting the .NET Micro Framework to a new platform.

In most devices, the HAL interacts with hardware directly, but the .NET Micro Framework can also run on top of an existing kernel or operating system and implement its functionality by calling through to the OS drivers. Since the CLR handles application multitasking within the managed environment, the underlying operating system does not need to be multithreaded. Some vendors have used this capability for networking or audio features.

On top of the HAL is the PAL. The PAL's job is to provide software services required by the CLR, calling through to the HAL for any needed hardware functionality. PAL services include bootstrapping, timers, memory management, debugging, asynchronous procedure calls, and events. As with the HAL, OS facilities may be used for some or all of these functions if an operating system is available.

Rounding out the .NET Micro Framework architecture are the class library layer and the application layer. The class library layer includes a subset of the .NET class libraries tailored to embedded devices. The assemblies that make up this layer are referred to as *system assemblies.* The application layer consists of the C# managed applications and drivers[2] that run on your device.

Device Firmware, the SDK, and the Porting Kit

The .NET Micro Framework is delivered to application developers in two pieces. Hardware-specific components, along with the .NET Micro Framework CLR and the system assemblies, become part of the device firmware, which may be provided by a hardware vendor with a pre-built module or developed for a specific device application. The .NET Micro Framework software development kit (SDK), which is made freely available by Microsoft, includes the Visual Studio integration components, the extensible emulator, and associated documentation that will be used by developers to build the application layer.

This distinction between the firmware and the SDK is intended to insulate programmers from most of the details of the hardware, allowing them to concentrate on building application-level functionality. The model is similar to ones that have been successful in the enterprise server and desktop application markets, although it is not yet ubiquitous in the embedded space.

Microsoft offers a porting kit with the components you'll need to produce .NET Micro Framework device firmware. The porting kit includes:

- Binaries of the CLR and system assemblies
- Source code for sample HAL and PAL implementations

[2] Managed drivers are discussed later in this paper.

- A build system for compiling and linking your HAL and PAL with the provided CLR and system assemblies
- A simple test application, ProfileDump, a debug bootstrap loader, PortBooter, and their source
- Additional sample applications, tests, tools, and documentation

The porting kit and the SDK are developed in tandem. When new features are finalized in the .NET Micro Framework, a porting kit update is provided to hardware partners so that they can produce new firmware for their device incorporating these features. Shortly afterward, Microsoft makes a new version of the SDK available so that application developers can take advantage of the new features on devices running the updated firmware.

Managed Drivers

With the .NET Micro Framework, HAL drivers are only required for interfaces, such as GPIO or USART, not the devices attached to them. *Managed drivers* can be written for devices connected to any interface supported by the .NET Micro Framework, including GPIO, SPI, I^{2}C, and USART. Managed drivers run in the application space, so developing them requires only the .NET Micro Framework SDK, not the porting kit.

Managed drivers perform well in a wide range of applications, reducing the need for native drivers. Most small devices do not need to communicate with their peripherals at particularly high data rates. With GPS, for example, the default rate established in the NMEA standard is 4800 bps, and although most receivers can be configured to use a higher rate, it is still typically in the kilobits-per-second range. On typical 32-bit embedded processors, performance of managed code is more than adequate for such rates. The non-deterministic nature of the CLR's memory management makes it impossible to guarantee latency, but in practice, the .NET Micro Framework CLR is sufficiently responsive for most devices.

Devices with high throughput or strict latency requirements (for example, a device that needs to stream CD-quality audio to a DAC) may not be a good fit for the .NET Micro Framework. To meet such requirements, you could use an auxiliary processor connected via SPI or I^{2}C for your application's real-time needs and continue to use the .NET Micro Framework for the UI and other non-time-sensitive code. Another approach would be to port the .NET Micro Framework onto a multithreaded real-time OS and run the real-time code in a separate high-priority thread. These techniques preserve the development productivity advantages of the .NET Micro Framework for the bulk of your application, while providing a way to meet the latency requirements for specific functions.

We hope to make the .NET Micro Framework more suitable for performance-critical applications in the future. There will always be some applications, however, for which another Microsoft embedded OS might be a better match. If your application would benefit from a fully native driver stack and/or a true real-time OS, Windows Embedded CE[3] may be a better choice if your target hardware is capable of supporting it.

[3] More information about Windows Embedded CE can be found at the Windows Embedded site.

2

Before You Port

Porting the .NET Micro Framework requires that you have access to programmers with specific skills. You will also need to sign a porting agreement with Microsoft and pay a nominal fee. This section will help you determine if performing your own port makes sense, or whether you should look for outside help from a porting specialist.

Who Should Port

There are three types of organizations that typically perform ports of the .NET Micro Framework.

- **Board developers.** In this business model, the Microsoft hardware partner performs a port, then offers development kits, reference designs, and pre-built modules compatible with the .NET Micro Framework. If you are already in this line of business, supporting the .NET Micro Framework on your hardware has the potential to reach a new, growing market.
- **Device makers.** Established device makers needing hardware more customized than an off-the-shelf module and who have the necessary technical resources may wish to undertake a port themselves. Alternatively, they can have one of Microsoft's hardware partners or a porting specialist assist with the hardware design and the porting effort.[4]
- **Porting specialists.** If you are an experienced embedded developer, you may wish to sign a porting agreement so you can obtain training and learn how to perform ports for others. This could be a profitable new line of business for a consultancy that already has experience with Microsoft embedded platforms and wants to expand its offerings.

[4] See the .NET Micro Framework partners page at MSDN.

Development Ecosystem and Porting Roles

Building devices upon the .NET Micro Framework may involve a number of different developers, companies, or feature teams, not including the .NET Micro Framework team at Microsoft.

- The **porting team** is the group or company that produces the device firmware, incorporating a custom HAL and/or PAL along with the Microsoft-provided runtime components.
- An **infrastructure team** provides infrastructure built on top of the .NET Micro Framework that will be used by application developers. For example, developers working on a SideShow-capable display device may choose to base it on the Windows SideShow Device SDK. The SideShow group inside Microsoft would be an infrastructure team in this scenario. A developer providing a managed driver for a modem or a GPS module can also be considered an infrastructure team.
- The **application team** develops the managed application. Often, this team actually drives development by other teams by specifying the hardware features and form factor needed to support the desired application.

These roles are flexible. Some projects may not need all teams, or one company may perform several functions. There may be more than one infrastructure team—or even more than one porting or application team. A single company building a product on the .NET Micro Framework may also choose to fill all these roles itself.

The lone application developer tinkering with the .NET Micro Framework on a development board, on the other hand, is an application team of one. Since the .NET Micro Framework insulates application developers from hardware details, an application can be created on a generic development board but put into production on an optimized, purpose-built device.

It is important to keep in mind the big picture and the roles you wish to fill when deciding whether to perform a port. It could be that another group or organization could do that part of the job more cost-effectively, or that an existing hardware platform can meet your requirements at reasonable cost.

Porting Scope

As we have seen, porting is the process of building .NET Micro Framework firmware for a previously-unsupported device or board, including a processor and specific support chips or system-on-a-chip (SoC). The scope of the port can have a significant impact on the time and resources required to complete it. We can describe any porting project as *full* or *partial*.

- A **full port** of the .NET Micro Framework involves writing a HAL largely from scratch. A full port may also involve a previously-unsupported processor or processor architecture, requiring the integration of its toolchain with the .NET Micro Framework build process.
- A **partial port** reuses applicable parts of a previous port for which you have source code. For example, if you change the display controller in your device, you will need to provide a new HAL driver for this part and produce new firmware that incorporates it. However, you can use the other drivers you wrote for the previous version of the device.

A partial port may take significantly less time than a full port depending on the level of code reuse.

Staff Requirements

Although porting the .NET Micro Framework does represent a significant resource commitment, the task is substantially simpler than porting Microsoft's larger embedded platforms. Longtime Microsoft embedded developers may recall that creating the initial Windows CE board support packages (BSPs) was quite resource-intensive. The .NET Micro Framework is a much smaller and simpler platform than Windows CE, and the effort required to port it is correspondingly lower. We intend to continue to work toward reducing the porting effort further.

Development of the .NET Micro Framework hardware abstraction layer (HAL) or platform abstraction layer (PAL) should be undertaken by experienced embedded developers with a solid grounding in C and C++. Useful skills for your porting team include experience with JTAG debugging, low-level knowledge of the processor architecture to be used, familiarity with the toolchain (e.g., RealView Developer Suite if your device will be ARM-based), and a good understanding of common embedded interfaces such as I^2C, SPI, and so on. Developers will also need to understand the resource-management issues unique to embedded development, including power consumption issues.

Our experience is that a team's first full .NET Micro Framework port typically requires several programmer-weeks, with an upper bound of 2-3 programmer-months. (Your experience may differ.) With the learning curve scaled, subsequent ports can usually be performed in less time.

Some HAL and PAL APIs have a degree of interdependence and are best assigned to a single developer, but it is feasible to distribute most of the porting workload among members of a small team. At the same time, the task is not so large that it cannot be undertaken by a developer working alone if necessary. A partial port rarely requires more than one or two developers.

Tool Requirements

The .NET Micro Framework porting kit does not include any processor-specific toolchain—the compiler, linker, and other development tools you will use to build your port. You will need the standard, manufacturer-supported tools for the processor or microcontroller you will use in your device, such as RealView Development Suite for ARM. These tools may have significant costs, so be sure to include them in your planning.

Porting Specialists

An experienced porting specialist can significantly reduce the amount of time it will take you to port the .NET Micro Framework. The right porting specialist not only has already surmounted the learning curve, but may also be able to supply HAL drivers for some of the support chips you are using, essentially turning your anticipated full port into a simpler and less time-consuming partial port. For this reason, we highly recommend that you talk to a porting specialist if you are contemplating a full .NET Micro Framework port. We can help you find a porting specialist with experience that suits your needs.

Porting Training and Support

As we mentioned earlier, you will be required to sign a Microsoft porting agreement to gain access to the porting kit, training materials, and support. Payment of a nominal fee is also required. Once you have signed the porting agreement and paid the fee, you will receive the porting kit.

From time to time, Microsoft holds 1-2 day porting training sessions, often at or in conjunction with conferences such as TechEd. These sessions, which are open only to Microsoft .NET Micro Framework porting partners, typically focus on the build process, defining a platform, and writing native drivers for the HAL. If it is not convenient for you to attend one of these sessions, we can provide the training materials to you. Some porting training sessions are videotaped, so we may even be able to provide you with a copy of a recent session.

As a porting partner, you will also have access to Microsoft support engineers, who will answer your technical questions to help you build the platform from your HAL and PAL and our binaries.

3

Porting Essentials

To produce a port, you write drivers for various devices, interfaces, and system functions, compile them, and link them with components provided by Microsoft. As you add drivers, you will be able to run more and more sophisticated test programs and eventually bring up the entire .NET Micro Framework on your device. The architecture of the .NET Micro Framework is modular by design, so that the amount of code you need to write is reasonably small and functionally well-isolated.

The porting process differs slightly depending on whether your .NET Micro Framework port will run directly on the hardware or on top of an existing operating system. When porting to run on hardware, the bulk of your work will be spent developing drivers for a new hardware abstraction layer (HAL). Few if any changes to the platform abstraction layer (PAL) will be needed. When porting onto an OS, the HAL will typically call through to the OS drivers, making the new HAL simple to develop by comparison with an on-the-metal implementation. However, the PAL will likely need some changes.

In both cases, you will also need to provide bootstrap code to initialize the device, although this will usually be simpler when porting onto an OS.

Porting Directly Onto Hardware

When porting directly onto hardware, the process has the following broad steps. You implement the necessary HAL functions in stages, each stage allowing you to bring up a more sophisticated test program or bootstrap loader on your device until you are able to link in the CLR and get the complete .NET Micro Framework running. Using this process gives you manageable chunks of functionality to test at each step, so you can make sure the foundation is working correctly before moving to higher levels.

1. **Configure the build system.** Your first task is to set up a build system so that you can successfully compile and link the generic platform provided with the porting kit. There are a number of environment variables that need to be set; we provide a Windows batch file for this, which you can customize if necessary. We also provide batch files for performing the build.
2. **Create your platform.** A *platform* is simply a definition of the drivers and options that will be used to build the firmware for a specific device configuration. Creating a platform is a matter of copying the generic platform (or another platform that is closer to your target hardware) and updating the configuration files to select the drivers and options you need. As you write new primitives for the HAL and PAL, you will change your platform's linker configuration file to load your libraries instead of the provided stubs.
3. **Get ProfileDump running.** ProfileDump is a minimal test program included with the .NET Micro Framework porting kit. Getting it running on your platform shouldn't require making any changes to the ProfileDump source code; rather, you implement the portions of the HAL it needs and write a bootstrap function that initializes your platform's processor and memory to a known state. The source code to ProfileDump is provided to let you add test functions as necessary to debug your HAL.
4. **Get PortBooter running.** The development loader, PortBooter, accepts the .NET Micro Framework and applications onto a device. As with ProfileDump, the source code to PortBooter is included. To get it running, you will need to implement more of the HAL, including the flash memory and button drivers. After this step, you will be able to deploy the .NET Micro Framework to your hardware and start running managed code on it.
5. **Implement the rest of the HAL.** Your device probably has more than just some buttons and memory. Now's the time to implement drivers for I^2C, SPI, USB, and so on. We recommend focusing on one device at a time and thoroughly testing each before moving on to the next. You can add tests to PortBooter, and you can also run C# applications to exercise each driver at the managed code level.
6. **Switch to TinyBooter.** TinyBooter is the .NET Micro Framework production loader that accepts code downloaded using the MFDeploy tool. TinyBooter supports code signing, though it does not come with source code as does PortBooter. For this reason, your HAL needs to be solid and production-ready before making the switch. Once TinyBooter is working, your port is complete.

Porting Onto an Operating System

When porting the .NET Micro Framework to run on top of an existing OS or kernel, your HAL drivers can often call the OS drivers more or less directly. You may not need to initialize all of the hardware yourself, either, making the bootstrap code significantly simpler. The .NET Micro Framework has only a single thread of native execution and manages all of its own memory, so in general you will not need to use operating system facilities for threading or memory management.

If your port will be running on a multithreaded OS, you will probably need to make some changes to the PAL to address how shared resources are managed. Although the .NET Micro Framework CLR is single-threaded, it is interruptible, and an interrupt service routine (ISR) behaves much like a thread. Since an ISR is the only other "thread" that might be running when the .NET Micro Framework is running directly on the hardware, the standard PAL simply disables interrupts when it needs exclusive access to shared resources. This is probably not appropriate in a .NET Micro Framework implementation running on top of a multithreaded OS. Instead, you will need to rewrite the PAL's locking functions to use the OS's thread synchronization mechanisms. This will need to be done early in the porting process—before you can run even ProfileDump.

Processor Considerations

The .NET Micro Framework is designed to run on 32-bit embedded processors. The 2003 "smart watch" products, which were the first products to employ what would later become the .NET Micro Framework, employed an ARM7 processor, and ARM9 and XScale are also supported today. Processors and MCUs based on the ARM architecture are produced by semiconductor manufacturers from Atmel to Zilog.

The .NET Micro Framework's ARM heritage means that its support for this popular architecture is mature and robust. We also intend to support 32-bit embedded architectures beyond ARM as time goes on. Device makers should have little difficulty finding a suitable processor capable of running the .NET Micro Framework, or adapting it to a new processor if necessary.

Since the .NET Micro Framework CLR handles all memory management for applications, a processor with an MMU is not needed. This may allow you to use a lower-cost part in your device than you might otherwise specify.

Like the full-size .NET, the .NET Micro Framework CLR uses a little-endian byte order. Most embedded processors are either little-endian natively or can be run in a little-endian mode. To date, Microsoft has not seen significant demand for other byte orders, but we are open to building a big-endian version if demand warrants.

Microsoft does not provide source code for the CLR and the .NET class libraries. Instead, you will link binaries of these components with your versions of the HAL and PAL. You must therefore use the same tools we used to build the CLR to build your port of the .NET Micro Framework. Our build process uses the standard vendor-supplied tools for the architectures we support.[5] If you are porting to a processor or architecture that we have not previously supported, or have reason to use tools other than the usual ones, Microsoft will work with you to make sure you have compatible runtime binaries.

Hardware Abstraction Layer APIs

The .NET Micro Framework hardware abstraction layer (HAL) consists of the following major APIs. Not all of these will need to be implemented for every port. Drivers for interfaces that your device does not have (e.g. I^{2}C or SPI) may be "stubbed out."

- **Caching.** Manages caching if supported by the CPU.
- **Clock.** Manages the two clocks, the system clock and the "slow" clock that runs at a fraction of the system clock's speed.
- **Interrupt Controller.** Manages the interrupt state of the processor.

[5] For ARM, the .NET Micro Framework supports RealView Development Suite 3.1.

- **DMA.** Manages the direct memory access controller.
- **GPIO.** Manages general-purpose I/O, single-bit input and output ports.
- **Time.** Manages the system's real-time clock and converts time to various units.
- **EBIU.** External bus interface unit (EBIU) drivers.
- **I^{2}C.** Manages the inter-integrated circuit (I^{2}C) bus.
- **Socket.** Provides socket networking API. A native TCP/IP stack will be released with .NET Micro Framework v2.5; earlier versions require an underlying OS or kernel with networking.
- **USART.** Manages RS-232 serial ports.
- **USB.** Manages the USB interface and endpoints.
- **SPI.** Manages Serial Peripheral Interconnect (SPI) interfaces.
- **System.** Manages the operation of the HAL itself. Includes many constants and low-level functions for memory management and data manipulation.
- **LCD.** Drivers for a display controller. Functions include setting contrast, clearing the display, and blitting.
- **Memory Management Unit.** Manages the processor's MMU if present.
- **Power.** Controls the processor's power usage, including putting it into various sleep or halt modes.
- **Security.** Manages security keys embedded in the hardware on certain processors.
- **Watchdog Timer.** Manages a timer that can be used to automatically reset the device if an application locks it up.

In addition, the HAL also provides APIs for a handful of specialized peripherals. Most ports will need implementations for at most one or two of these.

Platform Abstraction Layer APIs

The platform abstraction layer (PAL) serves to further abstract the functionality exposed by the HAL. It provides a smaller number of APIs than the HAL; these include APIs for asynchronous procedure calls, bootstrapping, event handling, heap management, and debugging. It also offers drivers for flash memory and GPIO buttons (with a debouncing API) and provides zone-based touch screen support.

Additionally, the PAL contains a communications manager API which allows the CLR (and thus .NET applications) to treat various types of communications, such as serial, USB, and sockets, generically. The .NET Micro Framework class libraries also contains a set of related APIs for each of these interfaces. These provide fewer, higher-level functions than the HAL drivers, simplifying the HAL functionality for the needs of the communications manager.

Memory Layout

The ARM toolset uses the concept of *scatter files* to indicate where various pieces of code and data will end up in a device's memory. This process involves defining one or more load regions, each of which might contain one or more execution regions. Load regions are where the code resides when installed on the device, while execution regions are where the code needs to be at runtime. Some code can run from ROM,[6] while some time-critical or low-latency code such as interrupt service routines (ISRs) will need to run from RAM.[7] These considerations are constrained by the CPU and by the device's memory configuration, with application needs for persistent storage complicating the overall picture.

The .NET Micro Framework is downloaded to the device as an image file with all the code and data packed together. Code and data segments must be moved to their final locations in the device's memory each time it starts up. The .NET Micro Framework HAL accomplishes this with a memory loading function called from the main bootstrap function.

On ARM-based platforms, the .NET Micro Framework build process invokes the scatter file processor and passes the scatter information to the linker. From there, it becomes a part of the HAL flash configuration map, which describes the intended use for each sector of the persistent memory. (Sectors are used because even on NOR flash, which is randomly-addressable by byte, a sector is the smallest unit of the memory that can be erased.)

Porting the HAL to an ARM-based platform will require creating or modifying a scatter file to match the memory layout of the .NET Micro Framework to the device's flash configuration. If the processor has an MMU, some RAM must be mapped to address zero so that the ARM vectors can be initialized there.

Non-ARM-based processors use other techniques for arranging memory after a firmware image is copied to a device. If you are porting to a non-ARM platform, it is possible that the memory layout function will need to be very different from the ARM implementation. Some platforms will simply require that the regions be copied by code, rather than using a map file. Microsoft provides appropriate code for supported processors and can provide technical assistance if needed.

Persistent Storage

The .NET Micro Framework provides data persistence through *extended weak references.* Traditional weak references are references to objects that may be discarded if memory becomes tight because they can be recreated when needed at some cost in time—essentially a type of cache. The extended variety of weak reference stores the object reference in persistent memory, such as flash or EEPROM, so that the cached object remains intact across device restarts. This is the preferred solution for persistent data storage on the .NET Micro Framework and is provided in place of a file system.[8]

Because persistent objects can be of any size, the .NET Micro Framework requires that nonvolatile memory be addressable by byte. Therefore, block-oriented flash devices are usually not the best choice for .NET Micro Framework devices; that is, NOR flash memory is preferred to NAND flash.

However, since all access to flash memory is performed through the PAL, it is possible to emulate random accessibility by reading blocks from a block-addressable storage area into a RAM buffer. You could perform this mapping on demand or map the entire flash region into RAM at CLR startup. This

[6] On a .NET Micro Framework device, "ROM" is typically NOR flash memory.
[7] Usually SRAM or SDRAM.
[8] Since a file system is useful for other reasons and is frequently requested by customers, we are investigating providing FAT file system support in a future release of the .NET Micro Framework.

approach may be desirable for some applications despite the performance implications, since NAND flash typically has higher density and better endurance than NOR flash.

If you use this technique, however, persistent storage that has been accessed even once must be considered to be in use for as long as the system is running, because the CLR does not notify the PAL when a given persisted object is no longer in use. Persistent storage must also be kept in sync with RAM whenever it is written to prevent data loss; you must immediately flush any modified blocks to the persistent backing store. Depending on your device's usage patterns and typical object sizes, having to flush an entire mapped block to flash whenever an object in it changes may offset some of the endurance advantage of NAND flash.

Bootstrapping

The bootstrapping sequence will likely need some modification, particularly if you are porting the .NET Micro Framework to run on top of an OS or kernel. The bootstrap process is divided into two main functions. One, called early in the startup process, performs low-level initialization for the system clock, the MMU if one is present, the cache, and other CPU blocks. Additionally, it calls the memory layout function to set up the device's memory layout, as described earlier, using scatter files for ARM processors. The second function, which is called later, initializes all of the drivers in the HAL by calling their individual initialization functions. This is often a convenient place to add other hardware initialization required by your device, assuming the hardware is not needed earlier in the process.

Asynchronous Procedure Calls and Timers

The .NET Micro Framework CLR uses only one execution thread. However, the CLR requires that all driver calls "look" asynchronous—that is, they return immediately, rather than blocking while waiting for hardware I/O to complete. This is accomplished through the use of *asynchronous procedure calls* (APCs). The .NET Micro Framework supports two types of APCs: completions and continuations. Both are provided by the PAL's Asynchronous Procedure Calls API. They work together with the timer driver to service all device drivers. *Completions* are for time-critical tasks; *continuations* are for tasks that can be run when there is time available.

The PAL maintains APCs in two separate lists by type and performs continuations only when there are no completions waiting. The CLR's thread scheduling is driven by a completion, and completions are driven by timers. This makes the PAL timer implementation one of the most critical parts of the .NET Micro Framework. For this reason, we recommend using the standard timer functions whenever possible. If you find it is necessary to develop your own timer functions, as it may be when the .NET Micro Framework is to run on a multithreaded OS, take special care to ensure that they are reliable and perform well.

Power Conservation

The .NET Micro Framework is intended for devices in which power consumption is often a critical factor. If this is a consideration for your device, you should look for places where you could use the processor's various sleep modes or reduce clock speeds to improve runtime. The HAL has an API for entering various power-saving modes; the CLR will call these functions when there are no execution threads, for example, and in most cases they should be implemented appropriately and not stubbed out.

If you are porting the .NET Micro Framework on top of an OS or kernel that does its own power management, this may not be necessary, but even in these cases, there may be steps you can take in your HAL to inform the OS of the CLR's resource needs. You might reduce the .NET Micro Framework process's priority slightly when the CLR is idle, for instance.

Hardware Providers

The .NET Micro Framework allows you to create enumerations for hardware—for example, how many SPI ports are available. *Hardware providers* are assemblies or source files that developers can add to their projects to enable their applications to use descriptive enumerated values specific to your platform, rather than numeric literals, for port or pin assignments.

This process gives application developers a way to discover what interfaces actually exist in the device and refer to them by name. It also enables IntelliSense for these enumerations in Visual Studio. The alternative, typecasting integers or using the defaults provided in the `Microsoft.SPOT.Hardware` namespace, is not as friendly to programmers, so we encourage you to take advantage of this feature.[9]

Emulator Components

You may wish to extend the .NET Micro Framework emulator with custom components that provide simulations of any peripherals your device comes with or supports, particularly if you will be offering pre-built modules. Developers who are building applications for your hardware can then largely run and test their code on their desktop PCs rather than requiring a development board for each developer.

The porting kit is not required to extend the emulator; the .NET Micro Framework SDK has everything you need. Emulator components are written in C# and have the full power of .NET at their disposal. The Microsoft Press book *Embedded Programming with the .NET Micro Framework* (ISBN 9780735623651) includes an excellent chapter on writing emulator components.[10]

[9]You can find a sample hardware provider in the sample project in the Freescale i.MXS DevKit available at freescale.com. The CPU.cs file defines a Microsoft.SPOT.Hardware.FreescaleMXSDemo namespace which the application (found in MXS.cs) uses as one would usually use Microsoft.SPOT.Hardware.
[10] You can read an excerpt from this chapter here.

4

Taking the Next Step

Now you should have the information you need to decide whether to port the .NET Micro Framework. You understand the basic architecture of the .NET Micro Framework and the roles involved in building a device on top of it. You know what's in the porting kit, the basic steps involved in porting, some of the technical decisions and issues involved, and the skills your team will need to successfully complete a port. You have thought about whether you will run the .NET Micro Framework directly on the hardware or on top of an existing operating system or kernel. You've also considered alternatives to porting, including using an existing module or having someone else perform the port, to see if they make more sense for your scenario.

If you have decided to perform a port yourself, your next step is to contact .NET Micro Framework business development at netmfbiz@microsoft.com to get answers to your remaining questions and get the porting agreement. Once you've signed that and paid the porting fee, you can receive training, support, and the porting kit.

If you have decided to work with another company to achieve your goals, see our current list of .NET Micro Framework partners at MSDN. In addition to companies offering porting services, this page lists partners providing compatible development hardware, silicon, and pre-built modules. Contact one or more of these companies for further information and take your next step toward basing your device on the .NET Micro Framework. If you require guidance, contact netmfbiz@microsoft.com with a discussion of the project you have in mind, and we will suggest partners with applicable expertise.

Additional Resources

- .NET Micro Framework at MSDN
- .NET Micro Framework newsgroup