Spreading your .NET

How managed code could speed embedded system development. By **Matthew Caffrey** and **Richard Parris**.

Short product lifecycles and intense price pressure in markets for mobile devices demand that embedded developers should continually search for faster tools and techniques. However, developing an embedded system typically requires engineers to understand the intricacies of the software platform and processor hardware.

One solution may be to draw on the experience of desktop developers, who benefit from automation of basic functionality and resource management tasks. This allows them to focus on the application, create compelling features and reuse IP across multiple projects.

Embedded developers usually write native code, coupled tightly to the microprocessor and application programming interfaces (APIs) specific to the chosen software platform. This is usually seen as the best way to meet lofty performance demands with scarce system resources.

Desktop developers may take a different approach – building the application using managed code. The APIs are defined as part of a runtime environment that exists as a layer between the application and the operating system. The runtime environment is usually portable across a number of processors and software platforms, thereby allowing the code to be ported just as easily. Hence code can be reused to reduce errors, amortise development costs and generally speed application development.

The runtime environment also provides certain built in functionality, such as memory management and web service support, in addition to the included APIs. When developing native code, these features usually have to be added explicitly, a labour intensive task which does not enhance directly the market appeal of the end application.

Managed code development

Microsoft .NET is a desktop oriented environment designed to support development of managed code applications. The .NET framework runs on the target device and includes a highly featured runtime environment called the Common Language Runtime, as well as class libraries that support functions such as GUI development, database access and web services (see Figure 1).

Source code, which can be developed in any one of some 20 compatible languages, is compiled to an intermediate language (IL) during development. This IL code is typically compiled at runtime by a Just In Time compiler, which creates native code for the appropriate processor. The .NET framework is compatible with Windows operating systems, including Windows XP, Windows 2000/98ME and Windows Server 2003.

A runtime environment developed specifically to meet the resource limitations and performance demands of embedded applications can help embedded developers access many of the advantages of working with managed code. For example, the Microsoft .NET Compact

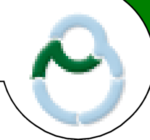
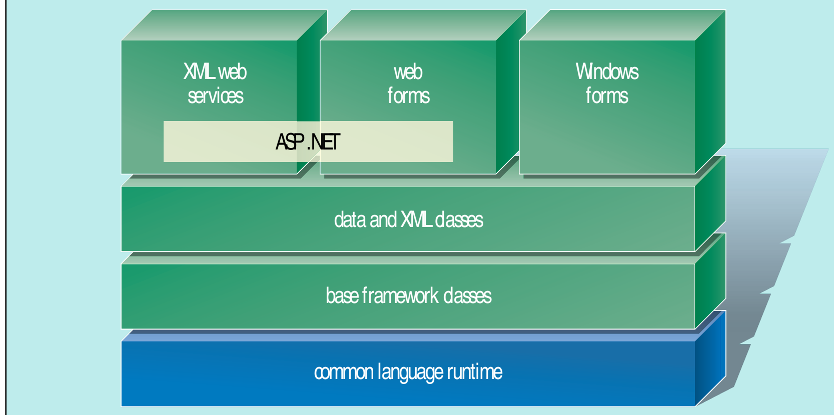


Figure 1: The .NET Framework for desktop development



framework is a subset of the .NET Framework, supporting application development in C# or Visual Basic. It includes the base class libraries, plus additional libraries specific to mobility and device development. The .NET Common Language Runtime is unchanged, as it was always intended to run efficiently on small devices with limited memory, resources and power budgets. Overall, the complete .NET Compact Framework binary size is about

"Code can be reused to reduce errors, amortise ... costs and generally speed application development."

Matthew Caffrey, **DSP Design**, Richard Parris, **Abacus**

1.5Mbyte – compared to 30Mbyte for the .NET Framework – and is therefore suitable for many small devices.

The .NET Compact framework requires Windows CE.NET, Microsoft Pocket PC/2002/2002 Phone Edition, or Smartphone 2003. Since .NET Compact requires the Graphical Windowing and Event Subsystem for its operation, it cannot be used with headless platforms and is therefore not supported for certain Windows CE.NET configurations, including Residential Gateway and Tiny Kernel.

The greatly reduced binary size of .NET Compact Framework is partly achieved by supporting a limited range of capabilities, compatible with the general demands of embedded devices targeting Windows functionality. For example, .NET Compact Framework supports only 28 of the 35 desktop controls provided in .NET Framework. These controls are optimised for embedded size and performance requirements, and not all .NET properties, methods or events are supported.

Although the .NET Compact framework functions much the same as .NET Framework, there are differences that help reduce the overall binary size. For example, support for XML is valuable, although XML documents can only be read or written in the forward direction and more care is required when planning navigation and size of memory for XML documents.

On the other hand, there is enhanced support for certain features common to smart devices. For example, APIs are included that simplify sending and receiving information over an infrared port. There is also help for developers creating TCP client/server applications.

The .NET Framework consists of a set of class library assemblies, which provide access to system functionality. One of these class libraries is Windows Forms, which helps desktop developers configure rich user interfaces with intuitive features such as high resolution, multiple display styles, powerful navigation features, support for notifications and more.

These capabilities are critical, since the user interface is often a crucial differentiator for handheld products. In addition, designers of handheld products typically work hard to make the best use of a small area. The controls supported by .NET Compact aim to satisfy these demands, but are implemented differently for greater efficiency in terms of size and performance. Most common controls are supported, including buttons, text boxes, scroll bars and forms.

Memory management

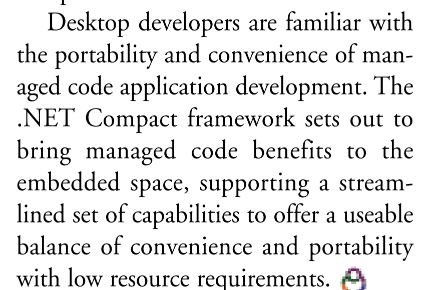
When creating native code, memory management can be technically challenging as well as time consuming. The resulting code is also a haven for bugs and security holes. In addition to the memory management capabilities inherent in the operation of a managed code application, .NET and .NET Compact also have the Garbage Collector feature.

Garbage Collector automates memory management by removing objects from memory automatically when they are no longer required. As a runtime component, Garbage Collector sweeps the memory periodically and destroys components that are no longer required. However, this action is non deterministic; there are no guarantees about when an object will be destroyed. This may be a problem in time critical applications.

Conclusion

As a rule, writing an application in native code results in faster execution as well as optimised resource requirements including a low memory footprint. On the other hand, native code ties the application closely to the software environment and processor characteristics.

Desktop developers are familiar with the portability and convenience of managed code application development. The .NET Compact framework sets out to bring managed code benefits to the embedded space, supporting a streamlined set of capabilities to offer a useable balance of convenience and portability with low resource requirements.



Author profiles:

Matthew Caffrey is director of DSP Design. Richard Parris is Microsoft Embedded business development manager for Abacus Embedded Systems Group.