# C# and the .NET Framework: Ready for Real Time?

**Michael H. Lutz,** *Siemens Health Services*
**Phillip A. Laplante,** *Pennsylvania State University*

Microsoft's integrated development environment, Visual Studio.NET, includes a new programming language C# (pronounced "C sharp"), which targets the .NET Framework. Both the .NET Framework and C# are fairly well-documented technologies, but the platform's appropriateness for real-time systems has not received much attention. Microsoft doesn't specifically claim that C# and .NET are intended for real-time systems, but many of the platform's general purpose features—including type unsafe features, thread synchronization, and overflow-sensitive arithmetic—apply to real-time systems. This article will further explore C# and the .NET Framework's suitability for real-time systems.

The C# language aims to provide the power of C++ with the ease of Visual Basic. However, C# is not suitable for hard real-time applications and should be used for firm and soft real-time applications only with care.

## Real-time systems

Practitioners categorize real-time systems as *hard*, *firm*, and *soft*.[1] Hard real-time systems include those in which a single missed deadline might cause catastrophic repercussions. Firm real-time systems tolerate one or more missed deadlines without catastrophic repercussions. In soft real-time systems, missed deadlines only result in performance degradation.

Bart Van Beneden says, "All too often, real-time behavior is associated with raw speed. Popular conclusions are that the faster a system responds or processes data, the more real-time it is."[2] However, these conclusions are incorrect. Real-time systems must foremost address *schedulability* and *determinism*, especially under load.[1] Schedulability indicates a system's ability to satisfy all deadlines. Determinism lets an observer predict the system's next state at any time given its current state and a set of inputs.

## Real-time C#

Real-time Java systems have been studied extensively.[3] As the .NET Framework and C# share traits with the Java Virtual Machine and Java,[4,5] we'll apply to the .NET Framework and C# a synthesis of the approaches taken by these studies.

When examining C# and .NET for real-time systems, you should note the characteristics of the underlying platform—this primarily means Microsoft operating systems. Microsoft and Corel have joined forces in an open source port of the Common Language

Infrastructure and C# to the Unixlike FreeBSD operating system. Additionally, at least two other major efforts are underway to port .NET and C# to Unix and Linux operating systems.

### Operating system considerations

Microsoft released the .NET Compact Framework as a version of the .NET Framework for scaled-down operating systems such as Windows CE 3.0, Embedded Windows XP, and handheld devices. Although this condensed framework's documentation is sparse, it appears that it will include the classes in the .NET Framework applicable to real-time systems.

Microsoft Windows CE 3.0 provides many features for real-time systems, including 256 thread priorities, facilities to counteract priority inversion, and thread latency times comparable to other industry-leading real-time operating systems.[2] Windows CE 3.0 is highly configurable, capable of scaling from small, embedded-system footprints (350 Kbytes) and upwards (for example, for systems requiring user interface support).[6] The minimum kernel configuration provides basic networking support, thread management, dynamic link library support, and virtual memory management. Although a detailed discussion falls outside this article's scope, Windows CE 3.0 clearly provides a powerful real-time operating system for the .NET platform to leverage. Whether the .NET Compact Framework uses all of CE's real-time capabilities has yet to be determined and warrants further study.

### Physical memory access

C# supports *unsafe code*—code that runs outside the Common Language Runtime (CLR) and hence lets pointers refer to specific memory locations.[7] You must *pin* (by using the keyword "fixed") objects that pointers reference, to prevent the garbage collector from altering their location in memory. The garbage collector collects pinned objects but does not move them. This capability would tend to increase schedulability, and it also allows for direct memory device access to write to specific memory locations—a necessary capability in embedded real-time systems.

### Garbage collection and memory management

The typical C algorithm traverses a data structure (typically a tree or hash table) to find a suitable location for allocation. It allocates the memory, then updates the data structure with the new object's presence.

.NET manages dynamic, heap-based memory with a stack. A system pointer, `NextObjPtr`, always points to the location for the next dynamic memory allocation. So, allocating memory in .NET is a simple, streamlined process. Microsoft's measurements support this claim.[8]

.NET offers a generational approach to garbage collection intended to minimize thread blockage during mark and sweep.[8] Microsoft's garbage collector and other vendors' .NET implementations will improve this algorithm over time and might even specialize it for various needs (such as real-time systems). The algorithm is built on the assumption that processing a portion of the heap is less expensive than processing the entire heap. Even with .NET's generations, however, it's impossible to know the collection's exact schedule, nor is it possible to guarantee each collection's cost. This nondeterministic behavior might pose the most significant barrier to hard real-time systems written in C#.

### Thread schedulability and determinism

C# and the .NET platform do not support many of the thread management constructs that real-time systems, particularly hard ones, often require.[9,10] Even Anders Hejlsberg (Microsoft's C# chief architect) states, "I would say that 'hard real-time' kinds of programs wouldn't be a good fit (at least right now)" for the .NET platform.[11] For instance, the Framework does not support thread creation at a particular instant in time with the guarantee that it completes by a particular point in time. C# supports many thread synchronization mechanisms but none with this precision.

Windows CE 3.0 has significantly improved thread management constructs.[6] If properly leveraged by C# and the .NET Compact Framework, it could provide a reasonably powerful thread management infrastructure. Current enumerations for thread priority in the .NET Framework, however, are largely unsatisfactory for real-time systems. Only five levels exist: `AboveNormal`, `BelowNormal`, `Highest`, `Lowest`, and `Normal`. Contrast this to Windows CE 3.0, designed specifically for real-

**Even with .NET's generations, it's impossible to know the collection's exact schedule, nor is it possible to guarantee each collection's cost.**

**Type safety is integral to the .NET platform; it catches program errors at compile time and proactively catches them at runtime.**

time systems with 256 thread priorities. Microsoft's `ThreadPriority` enumeration documentation also states that "the scheduling algorithm used to determine the order of thread execution varies with each operating system."[7] This inconsistency might cause real-time systems to behave differently on different operating systems.

Clearly, Microsoft must extend .NET's thread priority enumeration to fully leverage the thread priority levels in Windows CE 3.x. Most Windows implementations (including Windows 2000 and XP) support 32 thread priorities, which is considered inadequate for many real-time systems.[6]

C# supports an array of thread synchronization constructs:[10]

- *Lock*. A lock is semantically identical to a critical section (a code segment guaranteeing entry to itself by only one thread at a time).[7]
- *Monitor.* Lock is shorthand notation for the monitor class type.
- *Mutex*. A mutex is semantically equivalent to a lock, with the additional capability of working across processes spaces. The downside to mutexes is their performance cost.
- *Interlock*. You use interlock, a set of overloaded static methods, to increment and decrement numerics in a thread-safe manner.

Related to thread management is execution-time predictability—the schedulability we have been discussing. Real-time systems designers should include *runtime code analysis*, which proactively determines the expected runtime of critical code segments.[12] If the platform realizes that the code will not execute in a predefined timeframe, it throws an exception. This lets the software react appropriately (for example, by performing an *imprecise computation*, returning less accurate results, which is often better than no results) and is superior to reactively realizing a missed deadline.[11] The .NET Framework falls short of most real-time developers' needs in this regard.

However, if the .NET Compact Framework eventually allows for 256 priority levels and if the operating system addresses .NET thread priority inversion as it does for unmanaged threads, developers will be able to address some real-time deadline constraints. Additionally, .NET threads are easy to code, offer synchronization mechanisms, and are type safe (instead of raw function pointers, .NET uses delegates). C# might prove attractive to architects as a candidate for soft and firm real-time systems.

### Priority inversion

*Priority inversion* implies that through synchronization constructs (for example mutexes), lower-priority threads might block higher-priority threads—a situation preferably avoided. Priority inversion might lead to *starvation*, where higher-priority threads receive significantly less CPU time than their lower-priority counterparts or even deadlock. Both situations are classic real-time systems problems.[1] To deal with priority inversion, many operating systems, including most Windows implementations (for example, CE), temporarily boost the priority of lower-priority threads in these cases, freeing synchronization objects for higher-priority thread execution as quickly as possible. This solution is known as the *priority ceiling protocol.*[1] Because the .NET platform, specifically the CLR, relies on the underlying operating system for its thread management mechanisms, the CLR should benefit from Windows' priority inheritance mechanisms.

In tests conducted for this article, however, it did not appear that thread inheritance works properly. The test configuration consisted of Windows 2000 SP1, .NET v1.0.3705 (release one), running on an 800-MHz Dell Inspiron 8000 with 523 Mbytes of physical memory. These negative results occurred consistently for a variety of testing approaches.

The tests dealt exclusively with C#'s lock statement. This statement is equivalent to Win32 *critical sections.*[10] A critical section (and lock) is an intraprocess synchronization mechanism that serializes threads through designated sections.

The tests generated multiple threads, allowing a normal-priority thread to enter a lock before all other threads. Meanwhile, a set of lower-priority threads blocked against this normal-priority thread's lock. The processes' primary thread simply waited until all worker threads completed their tasks.

These experiments showed that the lower-priority threads' blocked duration (wall-

```
using System.Runtime.InteropServices;
namespace CSharp_ConsoleApp
{
   public class TimingStuff
   {    [DllImport("kernel32")]
        private static extern int QueryPerformanceCounter(ref Int64 X);
        // Constructor
        public TimingStuff()
        {
            if( QueryPerformanceCounter( ref m_t1 ) == 0)
            {
                 throw new Exception( "QueryPerformanceCounter failed" );
}   }    }    }
```

**Figure 1. Calling the** `QueryPerformance-Counter` **Win32 application programming interface.**

clock time) varied *directly* with their priority, as opposed to an inverse relationship—the desired result if the priority inheritance protocol was invoked properly. Specifically, when we increased the priority of the blocked threads in subsequent tests, Windows should have temporarily increased the running thread's priority to speed it through the critical section. Instead, the thread owning the lock (normal priority) owned the lock longer as higher priority was given to the blocking threads. This is the problem of priority inversion almost by definition.

According to Microsoft documentation on Windows 2000, Windows CE 2.1, CE 3.0, XP, and so forth, Windows handles priority inversion through priority inheritance. Jeffrey Richter, a columnist for *MSDN* magazine, says the CLR leverages the underlying operating system's thread management system, so priority inheritance should theoretically behave similarly.[5] According to these experiments, however, at least in the case of .NET's lock mechanism, priority inheritance was not easily noticeable.

### Timers

Timers in C# resemble the existing Win32 timer's functionality. When constructed, timers are told how long to wait in milliseconds before their first invocation and are also supplied an interval (in milliseconds), specifying the period between subsequent invocations.[5] These timers' accuracy are machine dependent and not guaranteed, reducing their usefulness for real-time systems.

### Hardware characteristics discovery

Kelvin Nilsen notes that "development of real-time Java programs is especially difficult. The developer of a Java application has no idea how powerful the CPU [is] and how much memory will be available in the Java Virtual Machine environment in which the application is to run."[13]

Many of these shortcomings do not apply to C# and the .NET platform. For instance, C# (through .NET's `System.Runtime.InteropServices` library) supports underlying system application programming interface invocation through `PInvoke`. For example, Figure 1 shows how to call the `QueryPerformanceCounter` Win32 API, accessing the system's high-resolution performance counter. The Win32 API provides the CPU characteristics, memory usage, and so on, at both the machine and process levels. Furthermore, via the `System.Diagnostics` library, C# developers can access the system's performance counters, consisting of hundreds of system-level measurements. Microsoft has added numerous .NET-specific performance objects: CLR Data, CLR Memory, CLR Networking, ASP .NET counters, and a set of JIT (just in time) compiler counters—all accessible programmatically and through the Performance Manager user interface (Perfmon.exe).

### Type safety

Type safety is integral to the .NET platform; it catches program errors at compile time and proactively catches them at runtime—a valuable asset for real-time systems. In .NET, Richter notes that type safety assures that "allocated objects are always accessed in compatible ways. Hence, if a method input parameter is declared as accepting a 4-byte value, the CLR will detect and trap attempts to access the parameter as

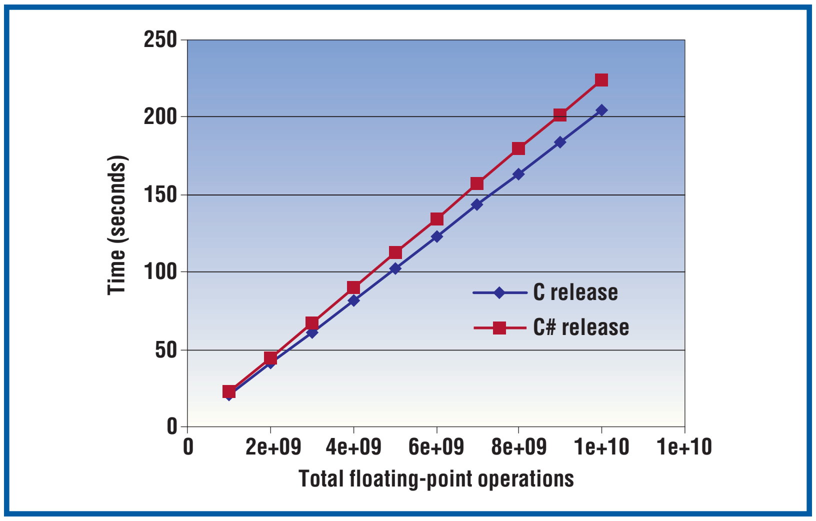


**Figure 2. Cumulative wall-clock runtime.**

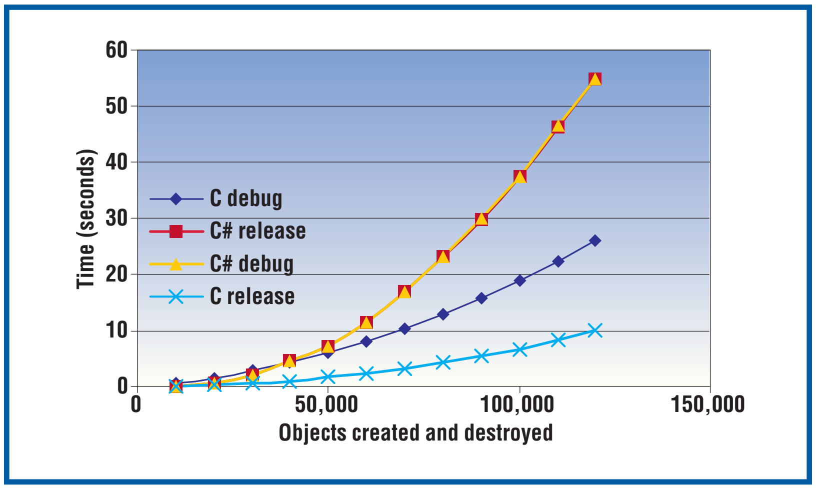


**Figure 3. Cumulative wall clock runtime versus objects created and destroyed (object time increases during the test).**

an 8-byte value. Similarly, if an object occupies 10 bytes in memory, the application can't coerce this into a form that will allow more than 10 bytes to be read."[5]

Richter continues, noting type safety also indicates that "execution flow will only transfer to well-known locations (namely, method entry points). There is no way to construct an arbitrary reference to a memory location and cause code at that location to begin execution. Together, these eliminate many common programming errors and classic system attacks such as exploiting buffer overruns."

### Exception handling

The .NET Framework contains structured *exception handling*. Exception handling is based on the "try, throw, catch, finally" approach that many mainstream object-oriented languages use.[10] Exception handling in .NET spans languages and is extensible, meaning a developer can extend the error messages based on a specific application.

Thrown exceptions create exception objects caught in catch blocks.[5] So, in C# error handling, an exception is literally an object. Furthermore, this approach spans all languages targeting the .NET platform, including VB.NET, C#, and Managed C++. In the current Win32/COM (Component Object Model) programming model, a myriad of error-handling mechanisms exist, including `HRESULTs` (COM), the implementation of special interfaces to get a more meaningful error message (COM),[7] and calls to `GetLastError` (Win32). .NET supports one well-understood, structured exception-handling technique common to all languages.

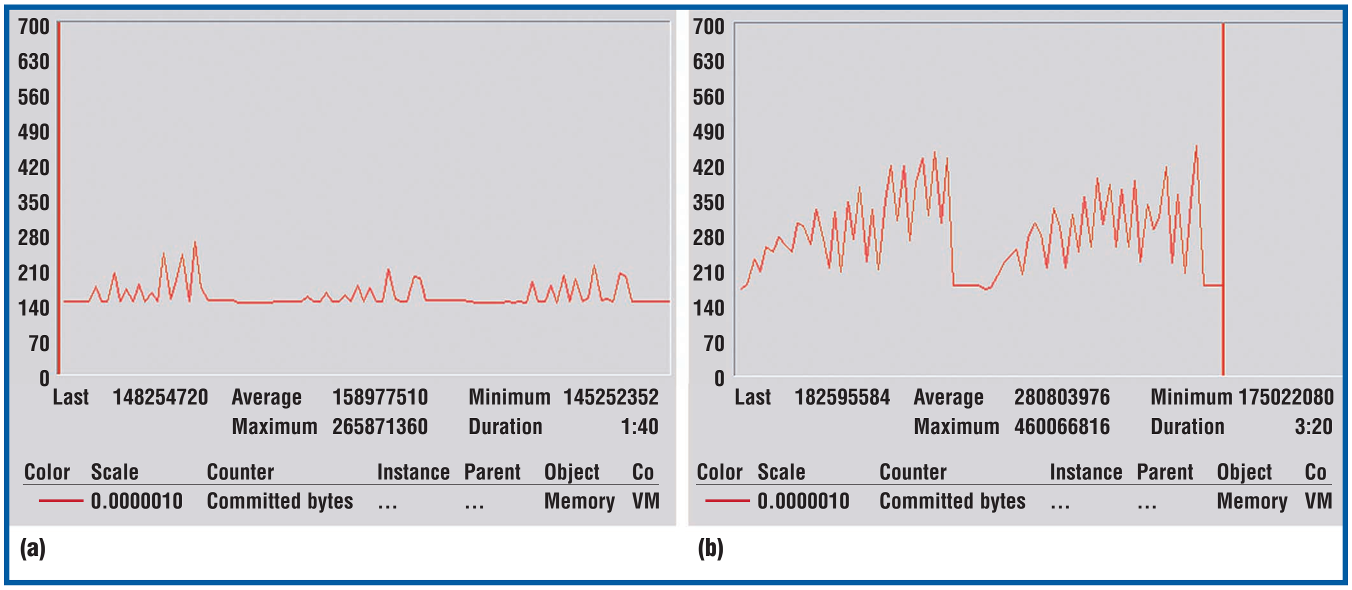
## Performance tests

Although performance is not the essence of real-time systems, it's an important aspect.[14] The faster a process executes, the easier it is to meet associated deadlines, although speed alone does not guarantee a 100-percent success rate.

We conducted two experiments comparing C#'s performance against C. All tests were built with version 55603-652-0000007-18846 of C++. NET, and we built and executed C# .NET on the .NET platform, version 1.0.3705. These versions correspond to .NET's first generally available, nonbeta version. All builds were optimized release builds. The tests ran on an 800-MHz Inspiron 8000, running Windows 2000 Professional, SP 1, with 523 Mbytes of physical memory.

The first test consisted of 10 billion floating-point operations. Figure 2 shows the cumulative wall clock runtime.

The second test dealt with memory management. We generated and released a linked list containing 5,000 nodes in both C and C# 24 times (48 total). Every other list saw an increase in node size. So for the first two sets of 5,000 nodes, each node contained a simple numeric value and a string of length zero. For the second set of two lists, each node contained a simple numeric again and a string of 2,500 bytes, and so on, increasing the string size by 2,500 bytes every other list iteration.



(a)

```
Last     148254720   Average    158977510   Minimum  145252352
                     Maximum    265871360   Duration      1:40

Color  Scale       Counter           Instance  Parent  Object  Co
       0.0000010   Committed bytes   ...       ...     Memory  VM
```

(b)

```
Last     182595584   Average    280803976   Minimum 175022080
                     Maximum    460066816   Duration      3:20

Color  Scale       Counter           Instance  Parent  Object  Co
       0.0000010   Committed bytes   ...       ...     Memory  VM
```

**Figure 4. Memory footprint for a linked-list memory consumption test written in (a) C and (b) C#.**

To make the tests as comparable as possible, we originally wrote the code in C# and then copied it and rewrote it in C. The resulting algorithms differed in two primary ways. First, the C# algorithm used an object oriented approach in which the nodes consisted of a class—for example, the linked list was a class. We wrote the C solution strictly in the structural paradigm with structs, pointers, and so on.

Second, we explicitly freed the 5,000-node linked list after each of the 24 iterations in C. In C#, the garbage collector was not called directly, so the CLR managed memory itself. We simply set the reference to the full linked list (5,000 nodes) to null in C#, then reused it for the new set of 5,000 nodes. No deletes are necessary in C#; the delete keyword doesn't even exist in C#.[7] Figure 3 shows our results.

Figures 2 and 3 show C# floating point performance to be virtually identical to C, but .NET memory management requires further tuning to reach C efficiency levels.

## Memory footprint

During the memory management performance tests, we took a sampling of the machine's *committed bytes*. This statistic offers insight into the volume of memory consumed during execution. Microsoft's Performance Manager defines *committed memory* as follows:

> *Committed Bytes is the amount of committed virtual memory, in bytes. (Committed memory is physical memory for which space has been reserved on the disk paging file in case it needs to be written back to disk). This counter displays the last observed value only; it is not an average.*[7]

The Performance Manager's resulting metrics offer insight into C's memory footprint versus C#'s. Figure 4a shows the machine's committed bytes throughout three sample test runs of the memory management algorithm described previously in C. The baseline committed bytes are about 150 Mbytes. Interestingly, memory consumption appears to streamline over time. The second and third runs appear to consume less memory than the first even though the executing code and input were identical. The high watermark for all three tests is approximately 280 Mbytes.

C# behaves quite differently (we only executed two runs, see Figure 4b). It does not appear that the .NET memory management infrastructure can streamline memory use over time, although we need more testing to verify this observation. The baseline memory usage appears to be equal to the C tests (150 Mbytes), although obviously the high watermark is greater. The C# high watermark is approximately 430 Mbytes, a 50-percent increase over C.

## Suggestions for Further Study

C# and the .NET Framework are not currently appropriate for hard real-time systems. However, this article does uncover some characteristics of C# and the .NET platform warranting further study for soft and firm real-time systems. Recommended topics include

- Performance impact of object-oriented style versus procedural (C-style) programming.[1,2]
- .NET performance comparisons on various Windows operating systems.
- .NET interrupt and thread-switching latency.
- .NET's garbage-collection behavior under load.
- Adding real-time thread-scheduling application programming interfaces to .NET.
- Worst-case cost analysis for C# and .NET.
- The .NET Compact Framework for real-time systems.
- Enhancing the CLR (Common Language Runtime) for real-time systems, potentially following the Real-Time Specification for Java.

### References

1. HA. Kuketayev, "The Data Abstraction Penalty (DAP) Benchmark for Small Objects in Java," *J. Object-Oriented Programming*, www.adtmag.com/joop/article.asp?id=4597&mon=8&yr=2001.
2. A.D. Robison, "The Abstraction Penalty for Small Objects in C++," abstract of talk at POOMA, 1996, www.acl.lanl.gov/Pooma96/abstracts/robison.html.

These tests illustrate C's ability to use physical memory more efficiently. Particularly for embedded systems where efficient resource consumption is paramount, C# appears to have significant ground to cover before reaching C efficiency levels.

C# and the .NET platform are not currently appropriate for hard real-time systems for several reasons. It has unbounded execution of its garbage-collected environment and lacks threading constructs to adequately support schedulability and determinism, among other things. Our experiments also showed that the thread inheritance didn't work.

Provided that these shortcomings are not critical, C# and .NET might be appropriate for some soft and firm real-time systems. (See the "Suggestions for Further Study" sidebar.) For example, a single-threaded application that can allocate memory entirely during program initialization mitigates the issues of priority inversion and garbage collection execution. Indeed, C#'s ability to interact with operating system APIs, shield developers from complex memory management logic, and floating-point performance approaching C also helps. However, any real-time system would require disciplined programming to cope with the problems that this study has uncovered.

## References

1. P.A. Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd ed., IEEE CS Press, Los Alamitos, Calif., 1998.
2. B. Van Beneden, "Examining Windows CE 3.0 Real-Time Capabilities," *Dr. Dobb's J.*, vol. 331, Dec. 2001, pp. 66–72.
3. B. Venners, "The Lean, Mean, Virtual Machine," *JavaWorld*, June 1996, www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html.
4. E. Meijer and J. Miller, "Technical Overview of the Common Language Runtime (or Why the JVM Is Not My Favorite Execution Environment)," Microsoft, 2001; http://docs.msdnaa.net/ark/Webfiles/WhitePapers/CLR.pdf.
5. J. Richter, "Microsoft .NET Framework Delivers the Platform an Integrated, Service-Oriented Web," *MSDN Magazine*, Sept. 2000; http://msdn.microsoft.com/msdnmag/issues/0900/Framework/default.aspx.
6. P. Yao, "Windows CE 3.0: Enhanced Real-Time Features Provide Sophisticated Thread Handling," *MSDN Magazine*, Nov. 2000, http://msdn.microsoft.com/msdnmag/issues/1100/RealCE/default.aspx.
7. Online documentation for .NET and SDK, www.msdn.microsoft.com/library/default.asp.
8. J. Richter, "Garbage Collection in the .NET Framework" (parts one and two), *MSDN Magazine*, Nov./Dec. 2000; www.msdn.microsoft.com/msdnmag/issues/1100/gci/default.aspx.
9. G. Bollella et al., *Real-Time Specification for Java*, Addison-Wesley, Boston, 2000.
10. A. Troelsen, *C# and the .NET Platform*, Apress, Berkeley, Calif., 2001.
11. "Conversations on .NET," Microsoft; www.msdn.microsoft.com/library/en-us/dndotnet/html/dotnetconvers.asp?frame=true.
12. K. Nilsen, "Issues in the Design and Implementation of Real-Time Java," *Java Developer's J.*, vol., 1, no. 1, 1996, pp. 44–57.
13. K. Nilsen, "Adding Real-Time Capabilities to Java," *Comm. ACM*, vol. 41, no. 6, June 1998, pp. 49–56.
14. E. Bertolissi and C. Preece, "Java in Real-Time Applications," *Proc. 10th IEEE Real Time Conf.* 1997, IEEE CS Press, Los Alamitos, Calif., 1997.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

## About the Authors

**Michael H. Lutz** is a software engineer for Siemens Health Services. His research interests include real-time systems, software design and architecture, and mitigating software complexity. He received a BS in computer science and engineering from Bucknell University and an MS in engineering from Pennsylvania State University. He is a member of the IEEE Computer Society. Contact him at 51 Valley Stream Parkway, Malvern, PA 19355; michael.h.lutz@siemens.com.

**Phillip A. Laplante** is an associate professor of software engineering at Penn State University's Great Valley School of Graduate Professional Studies. His research interests include real-time and embedded systems, image processing, and software requirements engineering. He cofounded the journal *Real-Time Imaging*, edits the CRC Press Series on Image Processing, and is on the editorial board of four journals. He received his BS, MEng, and PhD in computer science, electrical engineering, and computer science, respectively, from the Stevens Institute of Technology and an MBA from the University of Colorado. He is a member of the ACM and SPIE, a senior member of the IEEE, and a registered professional engineer in Pennsylvania. Contact him at Pennsylvania State Univ., 30 E. Swedesford Rd., Malvern, PA 19355-1443; plaplante@psu.edu.