

Seminar
Embedded Systems
(WS 06/07)

Programming Embedded Systems

Leopold-Franzens-University of Innsbruck
Institute of Computer Science

Rödlach Philippe,
Hausberger Thomas
11.01.2007

Inhalt

1	Requirements on programming languages for an embedded system	2
2	Programming languages	3
2.1	Ada	3
2.1.1	Brief description	3
2.1.2	Programming Embedded System with Ada	4
2.1.3	Example: The Ada concurrency model	6
2.2	C und C++	8
2.2.1	Brief description	8
2.3	Java, Real Time Java, J2ME	9
2.3.1	Brief description to Java	9
2.3.2	Java2ME	9
2.3.3	Real-time Java	10
2.4	B#	11
2.4.1	Brief description	11
2.5	Forth	12
2.5.1	Brief description	12
2.6	Embedded Eiffel	13
2.6.1	Brief description	13

1 Requirements on programming languages for an embedded system

Developing an embedded system is hard work. Reliability is essential; indeed, embedded software may control a safety- or security-critical system where an error can have catastrophic consequences. New requirements will almost surely pop up, so the software must be maintainable. Real-time constraints and memory limitations make time and space predictability and run-time performance important. Many embedded systems comprise activities that are performed concurrently, either with actual parallelism or through multiplexing on a single processor. And most deal at some point with hardware-specific details like interrupt handling and data layout.

Language musts

Real-time embedded applications require the following:

- General features for promoting reliability, maintainability, reusability, and other broad software engineering goals.
- Specific features for real-time and embedded applications.
- A lack of features that interfere with goals such as predictability of program execution and performance.

Examples of general-purpose features are compile-time type checking, support for encapsulation and information hiding, namespace management (avoiding clashes between the names of locally defined entities and either predefined or external entities), parameterizable templates and object-oriented programming features. C's type checking is rather weak, and it offers no support for any of the other areas listed. In contrast, both Ada 95 and Java were designed from the outset to promote sound software engineering practices. Ada has somewhat more extensive capabilities than Java — for example, Ada supplies strongly typed scalars, operator symbol overloading, and generic templates.

Features that are needed for embedded systems development are the ability to program at the machine level (dealing with addresses and "raw storage"), write interrupt service routines, and execute similar functionality. This is where C does provide the needed features. Ada offers comparable functionality, specifically through its Systems Programming Annex. The Java language is weak in this area, since the ability to directly access hardware resources would compromise both the language's safety and portability. Interestingly, more features don't automatically mean better functionality. If a language feature is more general than is required, there's the danger it may incur a run-time cost even if the feature is not used. Other features may interfere with the requirement for safety or predictability:

- The ease with which type checking can be defeated in C is an example of unwanted generality that compromises reliability. The solution is for the programmer to adhere to certain stylistic guidelines (possibly detected/enforced by an external tool).
- Ada has a general run-time model, and the full support may incur an unwanted cost. Anticipating such an issue, the language also includes a directive that identifies features that aren't used by the program. A program that uses a feature that is excluded is rejected by the compiler. The compiler's implementation can thus provide a specialized version of the run-time support knowing that certain features will not be used.
- The main issue with Java derives from its foundation in object-oriented programming. The problem isn't so much with efficiency but rather with the apparent unpredictability or latency incurred by garbage collection. The approach taken in the Real-time specification for Java (RTSJ) is to allow the program to define memory areas that aren't subject to garbage collection and to define certain kinds of threads that aren't allowed to access the garbage-collected heap.

2 Programming languages

2.1 Ada

2.1.1 Brief description

Ada is a structured¹, statically typed² imperative³ computer programming language designed by a team led by Jean Ichbiah of CII Honeywell Bull under contract to the United States Department of Defense during 1977–1983. It addresses many of the same tasks as C or C++, but with one of the best type-safety systems available in a statically typed programming language. Ada was named after Ada Lovelace, who is often credited with being the first computer programmer.

Ada was originally targeted at embedded and real-time systems. The Ada 95 revision designed by S. Tucker Taft between 1992 and 1995, improved support for system-, numerical- and financial-programming.

Notable features of Ada include strong typing⁴, modularity mechanisms (packages), run-time checking, parallel processing (tasks), exception handling, and generics. Ada 95 added support for object-oriented programming, including dynamic dispatch⁵.

Ada supports run-time checks in order to protect against access to unallocated memory, buffer overflow errors, off by one errors, array access errors, and other avoidable bugs. These checks can be disabled in the interest of efficiency, but can often be compiled efficiently. It also includes facilities to help program verification. For these reasons Ada is widely used in critical systems, where any anomaly will lead to very serious consequences ie accidental death or injury. Examples of systems where Ada is used include avionics, weapons (including thermonuclear weapons) and spacecraft.

Ada also supports a large number of compile-time checks to help avoid bugs that would not be detectable until run-time in some other languages or would require explicit checks to be added to the source code.

Ada's dynamic memory management is safe and high-level, like Java and unlike C. The specification does not require any particular implementation. Though the semantics of the language allow automatic garbage collection of inaccessible objects, most implementations do not support it. Ada does support a limited form

¹ Structured programming can be seen as a subset or subdiscipline of procedural programming, one of the major programming paradigms. It is most famous for removing or reducing reliance on the GOTO statement (also known as "go to").

² In static typing all expressions have their types determined prior to the program being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

³ Imperative programming is a programming paradigm that describes computation as statements that change a program state.

⁴ Strong typing doesn't allow a value of one type to be treated as another, for example treating a string as a number.

⁵ Dynamic dispatch is the process of mapping a message to a specific sequence of code (method) at runtime. Also commonly known as polymorphism.

of region-based storage management. Invalid accesses can always be detected at run time (unless of course the check is turned off) and sometimes at compile time.

The syntax of Ada is simple, consistent and readable. It minimizes choices of ways to perform basic operations, and prefers English keywords (e.g. "OR") to symbols (e.g. "||"). Thus it avoids terse constructs such as "||", "++", and "+=" (of the C language), and enforces that each conditional statement be closed. The rationale is that code for a complex system must be readable by reviewers and maintainers. Reviewers may include domain experts who are not highly software literate. Code for complex systems is typically maintained for many years, by programmers other than the original author. It can be argued that the language design benefits apply to most software projects, and most phases of software development, however when applied to complex, safety critical projects, benefits in correctness, reliability, and maintainability take precedence over (arguable) costs in initial development.

2.1.2 Programming Embedded System with Ada

To meet those demands, a programming language needs high-level features that support sound software engineering and provide the necessary generality, but without sacrificing run-time performance. Such a language must also provide low-level mechanisms that are more typically found in assembly language programming.

Ada was designed to satisfy these sometimes-conflicting requirements, and recent enhancements in the new Ada 2005 standard have improved it. Ada makes the embedded-system developer's job more manageable, and it can be a better development choice than C, C++ or Java.

Ada was designed from the start to promote reliability and maintainability, with features that emphasize readability over writability and that detect errors early. Many checks are performed by the compiler--for example, to ensure that the uses of a data object are consistent with its type. Errors that are not detectable at compile time (such as an out-of-bounds) are caught at run-time through compiler-generated checks. Those checks can often be eliminated automatically through compiler optimizations or, if the programmer has verified that they will not fail, manually through specific directives.

Ada's emphasis on readability and reliability is in contrast with the C family of languages, including C++. Java detects buffer overrun errors, but its weakly typed primitive type facility allows data misuse errors that would be caught in Ada. And unlike both Java and the C-based languages, Ada allows programmers to specify a constrained range for a scalar variable (for example, integers in the range 0 through 100), which aids both readability and reliability.

The evolution of programming languages has been accompanied by two major development approaches: procedural programming, in which a system's architecture is dictated by the kinds of processing that must be performed, and object-oriented programming, in which a system's architecture is dictated by the kinds of entities that must be processed and their relationships. Some embedded systems can be modeled through a procedural-programming approach; others may best be captured through object orientation in order to facilitate enhancements and maintenance.

Ada, like C++, can be used for both procedural and object-oriented programming. C, by contrast, lacks object orientation, and purely procedural programming in Java is rather clumsy.

Concurrent programming is intrinsically more difficult than sequential programming. Testing is complicated, since there are many more possible control paths and since new sorts of errors can arise, such as race conditions, deadlock and accessing a data object while it is being modified.

Ada has a high-level concurrency model that is designed to help avoid such errors. Concurrent activities (tasks) can communicate with each other either directly (rendezvous) or through protected operations on protected objects (see later in this topic). A protected operation is performed with mutually exclusive access to a protected object, and the semantics help prevent race conditions. A predefined object-locking policy can be implemented extremely efficiently because of the requirement that tasks not block while executing protected operations.

Ada 2005 has extended the Ada tasking model. It defines some new task-dispatching policies and provides a framework in which multiple policies can coexist. It also allows applications to monitor and control execution time on a task-by-task basis, with run-time detection of budget overruns.

Many languages (such as C and C++) do not support concurrency directly and instead require the programmer to obtain the desired facilities through libraries. This interferes with portability. Others, most notably Java, have a low-level concurrency mechanism that is error-prone.

Embedded systems often have to perform low-level processing: dealing with storage addresses, laying out data structures with specific fields occurring at specific offsets, querying or specifying the size of data objects, handling interrupts, using specialized hardware instructions, treating data as "untyped" storage elements. All of those capabilities are found in Ada. Moreover, and in contrast to C and C++, the Ada rules make it clear to the reader of the program that such system-specific and perhaps potentially unsafe features are being used. Low-level programming in Java requires native code and a corresponding loss of protection.

Embedded systems typically have stringent requirements for a small memory footprint and high efficiency. "Simple" languages, such as C, that might meet those requirements are problematic with respect to reliability, and they lack some necessary features (such as concurrency support and exception handling). Higher-level languages like C++, Java and full Ada come with run-time libraries that may be large or complex. Ada supplies a mechanism, the "Restrictions pragma"⁶, that lets the programmer select exactly those features that are needed. No run-time libraries are included for features outside the chosen subset.

A useful subset for embedded high-integrity real-time systems is the Ravenscar profile⁷, a reliable concurrency model standardized in Ada 2005 as a set of Restrictions pragmas. Ravenscar is general enough to express the kinds of concurrency patterns that are needed but is simple enough to be supported by a small, reliable and efficient run-time library. The Ravenscar profile meets the requirements for determinism, schedulability analysis and memory boundedness, suiting it for both hard-real-time⁸ and high-integrity embedded systems.

Also pointers are important in low-level programming. But if the pointer facility is not strongly typed, an object of one type may be viewed (through a pointer) as having a different type. If an object becomes inaccessible (for example, by being popped off the run-time stack) but a pointer to the object still exists, the result is a notorious bug known as a dangling reference. And with dynamic allocation, storage reclamation is a concern.

Ada addresses those issues via a strongly typed pointer mechanism that includes scope accessibility checks to prevent dangling references. The Restrictions pragma may be used to indicate a constrained use of pointers (for example, dynamic allocations only occurring during system initialization). Since aggregate data can go on the stack, Ada does not require the overhead of a run-time garbage collector.

Ada is unique in explicitly addressing the mixed-language requirement of modern systems. The Ada standard provides interfacing mechanisms that allow an Ada program to import subprograms or global data from, or export them to, other language environments. Ada also lets the programmer specify that a data structure is to be laid out based on the conventions of a compiler for another language.

Thus, it's possible to combine Ada efficiently, reliably and portably with other languages in the same program. For example, if a scientific library in Fortran is needed in an Ada application, there is no need to convert the library to Ada; it can simply be called from the Ada code. In the other direction, if an existing Ada application is to be extended with new functionality that is to be implemented in another language (such as C++), the simplest approach is to use the Ada interfacing mechanisms so that the new code can

⁶ Pragma = compiler directive

⁷ The Ravenscar Profile is an analyzable subset of Ada tasking that's suitable for hard real-time and high-integrity applications.

⁸ A system is said to be real-time if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. The classical conception is that in a hard or immediate real-time system, the completion of an operation after its deadline is considered useless - ultimately, this may lead to a critical failure of the complete system. A soft real-time system on the other hand will tolerate such lateness, and may respond with decreased service quality (e.g., dropping frames while displaying a video).

call the existing Ada subprograms. There is no reason to undertake the risky and costly strategy of converting Ada to the other language.

2.1.3 Example: The Ada concurrency model

Ada is organized into one core and several annexes. The *core* is required of all Ada implementations and contains the definitions of all language constructs. The annexes define additional facilities but never new syntax. The additional facilities they provide are typically in the form of *packages* (modules) that define abstract data types and *pragmas* (compiler directives) that specify required behavior. Both packages and pragmas are defined in the core.

The core includes the definition of concurrency in the form of *tasks* representing threads of control, and *protected objects* that provide mutual exclusion and condition synchronization in a high-level, yet efficient, manner. Consider the typical producer/consumer programming example shown in Listing 1. Tasks represent the producer and consumer threads, with a protected object representing the shared buffer. In this example we use a reusable generic unit defining a bounded buffer type. *Generic units* are essentially templates for program units, with various aspects factored out of the code. (Indeed, in C++ generic units are called templates and serve much the same purpose.)

Listing 1 Typical producer/consumer programming example

```

1. package Characters is new Concurrent_Bounded_Buffers (Character);
2.
3. Holding : Characters.Bounded_Buffer (Size => 10);
4.
5. task Producer;
6.
7. task Consumer;
8.
9. task body Producer is
10. begin
11.   for Next_Value in Character range 'A' .. 'Z' loop
12.     Holding.Put (Next_Value);
13.   end loop;
14. end Producer;
15.
16. task body Consumer is
17.   Next_Value : Character;
18. begin
19.   loop
20.     Holding.Get (Next_Value);
21.     Ada.Text_IO.Put (Next_Value);
22.     exit when Next_Value = 'Z';
23.   end loop;
24.   Ada.Text_IO.Flush;
25. end Consumer;
```

Our generic unit is shown in Listing 2. The package template is named `Concurrent_Bounded_Buffers` and has one generic parameter named `Element` (see Line 2, Listing 2) representing the type of values held within a buffer object. Given this template, the main program `Demo_Buffers` can instantiate the generic into a usable concrete unit (Line 1, Listing 1), in this case a package named `Characters` because we instantiate the generic to obtain buffers of `Character` values. The package exports a protected type, such that many protected objects of that type can be easily declared. In particular, the protected type `Bounded_Buffer` is used to declare the object `Holding` in the main program (Line 3, Listing 1). When a `Bounded_Buffer` object is declared we must also specify the capacity of the buffer. Thus the buffer `Holding` can contain at most 10 characters at once.

Listing 2 Generic bounded buffer unit used in Listing 1

```

1. generic
2.   type Element is private;
3.   package Concurrent_Bounded_Buffers is
4.
5.     type Content is array (Positive range <>) of Element;
6.
7.     protected type Bounded_Buffer (Size : Positive) is
8.       entry Put (Item : in Element);
9.       entry Get (Item : out Element);
10.    private
11.      Values   : Content (1..Size);
12.      Next_In  : Positive := 1;
13.      Next_Out : Positive := 1;
14.      Count    : Natural  := 0;
15.    end Bounded_Buffer;
16.
17. end Concurrent_Bounded_Buffers;

```

The producer body is declared on Line 9 and the consumer body on Line 16 of Listing 1. These task bodies define the behavior of the two threads of control. The producer simply inserts 26 characters into the buffer by calling the Put operation on object Holding (Line 12, Listing 1). Because at most 10 characters can be in the buffer at once, the producer might temporarily block at some point, depending on how the tasks are scheduled. The consumer task loops indefinitely, reading characters from the buffer via procedure Get (Line 20, Listing 1) and printing them until it reads the character Z. If the Holding buffer is empty before the last character is consumed, the consumer will temporarily block. Eventually the tasks terminate and the program as a whole terminates.

An important point about protected objects is that they're not threads of control. Rather, they're passive, data-oriented objects in the same way that semaphores are data instead of threads. As a result, no expensive task switching is involved when invoking one of their operations. In contrast, tasks in the now-outdated Ada 83 model could only communicate directly, via the rendezvous mechanism, thereby causing a number of task switches on each invocation. In particular, the shared buffer had to be another task in Ada 83 because no synchronization mechanism existed other than the rendezvous mechanism defined in the language. In Ada 95, protected objects provide operations that are executed by callers without this task switching overhead.

The body of the generic package in Listing 3 contains the body of the protected type Bounded_Buffer. As mentioned, each protected operation is executed by a caller with mutually exclusive access so no other task can be executing a modifying call on the same object at the same time. Thus, no race conditions are possible on the shared data inside the protected object. Note the logical expressions on Lines 5 and 12 of Listing 3, referring to encapsulated objects Count and Size. These "barriers" express condition synchronization by enabling or disabling the execution of the associated routine until the proper conditions are met. In this example a call to Put is allowed to execute only when the buffer is not full. Execution of Get is similarly enabled only when the buffer is not empty.

Listing 3 Implementation of generic bounded buffer unit

```

1. package body Concurrent_Bounded_Buffers is
2.
3.   protected body Bounded_Buffer is
4.
5.     entry Put (Item : in Element) when Count < Size is
6.     begin
7.       Values (Next_In) := Item;
8.       Next_In := (Next_In mod Size) + 1;
9.       Count := Count + 1;
10.    end Put;
11.
12.   entry Get (Item : out Element) when Count > 0 is
13.   begin
14.     Item := Values (Next_Out);
15.     Next_Out := (Next_Out mod Size) + 1;
16.     Count := Count - 1;
17.   end Get;
18.
19.   end Bounded_Buffer;
20.
21. end Concurrent_Bounded_Buffers;

```

2.2 C und C++

2.2.1 Brief description

More than any other, C has become the language of embedded programmers. This has not always been the case, and it will not continue to be so forever. However, at this time, C is the closest thing there is to a standard in the embedded world.

Successful software development is so frequently about selecting the best language for a given project, it is surprising to find that one language has proven itself appropriate for both 8-bit and 64-bit processors; in systems with bytes, kilobytes, and megabytes of memory; and for development teams that consist of from one to a dozen or more people. Yet this is precisely the range of projects in which C has thrived.

Of course, C is not without advantages. It is small and fairly simple to learn, compilers are available for almost every processor in use today, and there is a very large body of experienced C programmers. In addition, C has the benefit of processor-independence, which allows programmers to concentrate on algorithms and applications, rather than on the details of a particular processor architecture. However, many of these advantages apply equally to other high-level languages. So why has C succeeded where so many other languages have largely failed?

Perhaps the greatest strength of C is that it is a very "low-level" high-level language. The "low-level" nature of C was a clear intention of the language's creators.

Of course, C is not the only language used by embedded programmers. In the early days, embedded software was written exclusively in the assembly language of the target processor. This gave programmers complete control of the processor and other hardware, but at a price. Assembly languages have many disadvantages, not the least of which are higher software development costs and a lack of code portability. In addition, finding skilled assembly programmers has become much more difficult in recent years. Assembly is now used primarily as an adjunct to the high-level language, usually only for those small pieces of code that must be extremely efficient or ultra-compact, or cannot be written in any other way.

C++ is an object-oriented superset of C that is increasingly popular among embedded programmers. All of the core language features are the same as C, but C++ adds new functionality for better data abstraction and a more object-oriented style of programming. These new features are very helpful to software developers, but some of them do reduce the efficiency of the executable program. So C++ tends to be most popular with large development teams, where the benefits to developers outweigh the loss of program efficiency.

Nonetheless, C has a number of drawbacks that may be significant:

Language insecurities. The loopholes in C's type model are well known; for example, allowing pointers to be treated as integral data and vice versa. Although it's possible to avoid such loopholes by applying a style-enforcement tool, more secure languages will prevent the hard-to-find bugs that loopholes create and will thereby decrease development time and promote higher reliability.

Lack of support for large-scale systems. C has weak facilities for "programming in the large," that is, developing applications comprising hundreds of thousands, indeed perhaps millions of lines of code. A relatively small portion of most embedded systems needs the low-level facilities provided by C. The majority of the logic is higher-level processing that's better supported by languages that offer greater type safety and better capabilities for module structuring, namespace management, encapsulation, and object orientation.

Absence of needed functionality. C has no language support for concurrency (multithreading), a serious omission for real-time systems. The programmer needs to use an external application programming interface or real-time operating system (RTOS), thus compromising portability. Moreover, the programmer needs to be sensitive to whether otherwise-safe libraries can be used in a multithreaded application. Languages with an integrated concurrency model can better deal with these problems.

Summary

Advantages using C for embedded systems:

- Easy access to hardware
- Good control over resources (i.e. memory, CPU)

- Small and simple to learn
- Compilers available for almost every processor

Disadvantages:

- Error prone, especially for beginners
- Nearly no runtime checks
- Little reusability and maintainability
- Language insecurity
- Lack of support for large-scale systems
- Absence of needed functionality

2.3 Java, Real Time Java, J2ME

2.3.1 Brief description to Java

Java is an object-oriented programming language developed by Sun Microsystems in the early 1990s. Java applications are, in the official implementation, compiled to bytecode, which is compiled to native machine code at runtime. The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities.

2.3.2 Java2ME

As time and technology moved on, Sun recognized the need to collect the device-oriented platforms under one umbrella. At JavaOne in 1999, Sun introduced the Java 2 Micro Edition. J2ME is not a specific virtual machine, API, or specification. Instead, J2ME provides a modular, scalable architecture to support a flexible deployment of Java technology to devices with diverse features and functions. A J2ME "configuration" targets devices with a specific range of capabilities. A "profile" selects a configuration and a set of APIs to target a specific domain of applications. By selecting the best configuration and profile, a vendor can produce a wide range of flexible applications. Since lightweight appliances do not need to support the entire Java 2 platform, their resource requirements (and therefore cost) will be reduced. At the same time, by allowing modular extensions, J2ME allows vendors to differentiate themselves by producing innovative applications and incorporating value-added features.

2.3.2.1 J2ME Configurations

A J2ME configuration defines an API and a virtual machine optimized to service devices that fall into a particular range of capabilities and resources. Two configurations have been defined, the Connected, Limited Device Configuration and the Connected Device Configuration. At this time, there is no configuration for disconnected devices.

The Connected Device Configuration expects devices with substantial resources. In particular, it requires at least 512K ROM and 256K RAM, and a device that can support a complete JVM implementation. This category of device includes most modern PDAs.

The Connected, Limited Device Configuration provides a platform for more resource constrained, but still network-connected devices. (The network connection may be intermittent.) This category of device includes cell phones.

2.3.2.2 J2ME Profiles

Profiles define additional class libraries and APIs needed to enable domain-specific applications on a particular configuration. Profiles provide the vertical specialization built upon the horizontal configurations. For instance, the Mobile Information Device Profile requires at least the Connected, Limited Device Configuration. It enables development of applications to provide wireless access to information.

The most basic profile is the Foundation Profile. The Foundation Profile requires the Connected Device Configuration. It explicitly supports devices with no user interface whatsoever, but which do have networking support.

2.3.3 Real-time Java

Real-time application development requires an API set and semantics that allow developers to correctly control the temporal behavior of application, i.e., how it will behave in real-world time. A real-time edition of Java must therefore provide some semantic JVM enhancements and a new API set appropriate for real-time applications. It is not surprising that the main obstacle in achieving real-time characteristics for Java is its garbage collector. A real-time garbage collector became a revolutionary and central component of Sun's recently-released Java real-time edition RTS 1.0; although its first implementation does not include one (it is expected in the next release). Java RTS addresses other issues, making strong deterministic guarantees for thread scheduling, synchronization overhead, lock queuing order, class initialization, and maximum interrupt response latency. Java RTS is intended only for suitable underlying operating systems, which means that only a real-time operating system is appropriate for implementing the JVM.

It's also worth mentioning that existing J2SE application will successfully run under Java RTS because the RTSJ specification only restricts the semantics of the JLS and JVM specification to a subset; it does *not* allow syntactic extensions which could break existing applications.

To make the real-time garbage collector predictable, you should understand how your program demands memory from the heap, because the garbage collector and your program both use it. The program makes garbage, the garbage collector turns garbage into free memory, and they have to interact in the heap. Since they can't do that at the same time, there must be some "synchronization" between the two. Thus, you have to tell the garbage collector something about how fast your program will create garbage so it knows how fast it must collect it. Getting that number can be somewhat tricky, but you must consider memory, no matter what you do.

Changes to meet real-time requirements

- *Direct memory access.* Java RTS allows direct access to physical memory, making it similar to J2ME. No surprise there: one of the main target platforms of real-time Java is embedded systems. This means that now you can create device drivers written in pure Java. Although memory access is not directly a real-time issue, physical memory access is desirable for many applications. Java RTS defines a new class that allows programmers byte-level access to physical memory, as well as a class that allows the construction of objects in physical memory. One might think that physical memory access is the point where Java gives up its main principles--reliability and safety--and takes a step back towards C, but this isn't the case. Java maintains strong security protections by controlling memory bounds and data contents.
- *Asynchronous communications.* Java RTS provides two forms of asynchronous communication: asynchronous event handling, and asynchronous transfer of control. Asynchronous event handling means the developer can now schedule the response to events coming from outside the JVM. Asynchronous transfer of control provides a carefully controlled way for one thread to interrupt another thread in a safe manner.
- *High-resolution timing.* There are several ways of specifying high-resolution time including absolute and relative time. Nanosecond accuracy is available for time scheduling and measurements.
- *Memory management.* There are two new types of memory areas that help prevent unpredictable delays commonly caused by traditional garbage collectors in real-time applications: immortal memory and scoped memory. Immortal memory holds objects without destroying them, except when the program ends. This means that objects created in immortal memory must be carefully allocated and managed, as with C programs. Scoped memory is used only while a process works within a particular section, or scope, of the program (such as in a method). Objects are automatically destroyed when the process leaves the scope. Neither immortal nor scoped memories are garbage collected, so using them avoids problems of GC interference. The Java RTS also provides limited support for providing memory allocation budgets for threads using memory areas.

Maximum memory area consumption and maximum allocation rates for individual real-time threads may be specified when the thread is created.

- *Real-time threads.* As mentioned previously, Java RTS supports two new thread models: real-time threads (`javax.realtime.RealtimeThread`) and no-heap real-time threads (`javax.realtime.NoHeapRealtimeThread`). Both thread types cannot be interrupted by garbage collection. These thread classes have 28 levels of priority and, unlike standard Java, their priority is strictly enforced. Real-time threads are synchronized and are not subject to so-called "priority inversion" situations where a lower priority thread has a block on a resource needed by a higher priority thread and thus prevents the higher priority thread from running. Thorough testing has proven that Java RTS completely avoids any priority inversions, which is crucial for mission-critical applications.

2.4 B#

2.4.1 Brief description

B# (pronounced "be sharp") is a tiny, object-oriented, and multi-threaded programming language that is specially dedicated for small footprint embedded systems.

Because B# has its roots in the C family of languages, it will be immediately familiar to C, C++, Java, and C# programmers. In addition to supporting modern object-oriented features such as namespaces, abstract and concrete classes, interfaces, and delegates, the B# language caters to the embedded systems programmer with efficient boxing/unboxing conversions, multi-threading statements, field properties, device addressing registers, interrupt handlers, and deterministic memory defragmenter.

Each of these features is directly supported by the constructs of B# and its underlying virtual machine in order to create, use, and reuse more portable and decoupled software components across different embedded systems applications. This series of articles highlights and illustrates these key B# concepts with short programming excerpts, but leaves the details of other more common features to a more complete overview.

Each B# program consists of one or more source files. Each file is composed of types which, in turn, are composed of members. Classes, interfaces, and delegates are examples of different types and fields, methods, properties, interrupt handlers, and device registers are examples of different members.

B# has a unified type system. All B# types inherit from a single root object type. Thus, all types share a set of common operations and values of any type can be manipulated in a consistent manner. B# is also strongly-typed and supports both value and reference types.

B# has five value types: boolean (`bool`), character (`char`), signed integer (`int`), unsigned integer (`uint`), and floating-point (`float`). For the sake of simplicity and internal stack evaluation efficiency, the five value types are all represented and manipulated as (maximum) 32-bit values.

Whereas value types contain variable data, reference types contain addresses of objects. B# provides a set of predefined reference types familiar to Java or C# developers including object, array, and string. Two additional types, delegate and `ioreg` for device addressing, are also included. The object type is similar to the untyped pointer (`void*`) in the C programming language.

The type system of B# has the ability to obtain type information of every variable at runtime and to treat value types as objects. These runtime capabilities are achieved by keeping minimal information on types in order to provide efficient and flexible boxing/unboxing conversions.

These conversions are an elegant mechanism inspired from C# and adapted for our needs. Consider the use of boxing/unboxing in the B# example below:

```
class Test {
    static void Main() {
        int    i = 123;
        object o = i;      // Boxing
        int    j = (int)o; // Unboxing
    }
}
```

An `int` value is converted to object and back again to `int`. Boxing happens when a valuetype variable is converted to a reference type. Unlike C#, an object box is already allocated and therefore, the value does

not need to be copied into the box. Unboxing is the complementary process, but contrary to C#. B# avoids copying a value into and out of a box by providing a pre-allocated wrapping mechanism within the B# stack. The operators of B#, their precedence and associativity, are identical to those in C. Only the pointer operators *, &, and -> are excluded and only the operators, new for object creation and is for type testing, are also included. Expressions are formed by combining literal values and variables with B# operators.

B# also borrows several sequential statements from C including statement lists, block statements, expression statements, return, if, while, and break. In addition, the lock and start statements of B# are inspired from Edison.

In B#, classes are the most fundamental types. A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). A class provides a specification for dynamically created instances of the class, also known as objects.

Classes support inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes. New classes are created using class declarations. A class declaration starts with a header that specifies the modifiers of the class, the name of the class, the base class (if any), and the interfaces implemented by the class.

In B#, the members of a class are either static members or instance members. Static members belong to classes and instance members belong to objects (instances of classes). The following table provides an overview of the kinds of members a class can contain.

An interrupt handler (also called an interrupt service routine) is a special method which is executed when an external hardware interrupt is triggered. Each handler is specified by adding the interrupt keyword in front of a method, making it implicitly static and internal with no return value.

2.5 Forth

2.5.1 Brief description

Forth was invented in the 1960s, and is a stack-based, interpreted language. The embedded systems programming language, Forth, is best introduced by its inventor, Charles Moore: "Forth is unique among programming languages in that its development and proliferation has been a grassroots effort unsupported by any major corporate or academic sponsors. It is an interpreted language which means that the source code is not compiled and then linked as in the C language. Interpreted languages are fed directly into an interpreter at runtime and because of the greater processing overhead to interpret the human readable source code they are slower than compiled languages. Interpreting a single line of source code can take hundreds of thousands of machine code lines when using an interpreted language such as BASIC. Originally, it used a single stack to pass data. In the 1970s, Forth added a dictionary, which provided pointers to other Forth code segments or assembly code. This is an indirect, threaded code implementation. Forth is intended to replace the "vast hierarchy" of languages between the programmer and the machine with a single layer consisting of two elements: a programmer-to-Forth interface and the Forth-to-machine interface. The "vast hierarchy" is described by Mr. Moore as being "The software provided with large computers supplies a hierarchy of languages: the assembler defines the language for describing the compiler and supervisor; the supervisor, the language for job control; the compiler, the language for application programs; the application program, the language for its input.

Forth allows embedded systems programmers to easily create bug free, efficient, real-time code. This simplicity, and Mr. Moore's fervent desire to port the interpreter to new machines, allowed Forth early entry into embedded systems development. The resulting source code will also be small, efficient and powerful. Forth is designed, and has been used, to create many application-specific programming environments. Mr. Moore desired to eliminate systems programming and focus only on the specific application at hand. For this reason, Forth is very flexible in the sense that the core Forth interpreter is designed to be used with application-specific libraries created during the implementation phase of a specific application. This bottom up design allows the resulting specialized Forth interpreter to be as small and simple as possible.

Programming with Forth requires creating application specific libraries and this means additional implementation time and cost to produce them. This further aggravates the time requirements for getting an application running when using Forth. The application specific library affords the Forth programmer an efficient and high speed execution of the interpreted source code because the library is not bloated by any code or programming constructs that are not useful to the task at hand. The up side is that Forth allows a specific application to be created on the smallest and least expensive hardware possible. Forth has been standardized, and the standard is named ANS Forth. By creating Forth CPUs the distinction between the two types of languages is blurred and the apparent loss in efficiency by using an interpreted language is

significantly reduced. Forth CPUs allowed designers to simply “drop in” the CPU onto their specific systems and then simply start coding. These Forth-specific embedded CPUs are very fast and efficient because they are a physical implementation of the Forth virtual machine interpreter. Mr. Moore continues to work on the Forth interpreter to this day through his own company called Forth, Inc. Forth is limited by the fact that it is designed to have the interpreter recreated for each specific application instance.

2.6 Embedded Eiffel

2.6.1 Brief description

Embedded Eiffel was created by Hewlett-Packard, Inc. and ISE, Inc. to allow for the implementation of sophisticated, tiny and error free printer firmware. Embedded Eiffel has an advanced garbage collection engine designed to make it extremely difficult to have a memory leak. Embedded Eiffel supports multithreading and object-oriented programming. It was intended to be the perfect embedded language and design methodology. It is designed to be small and, in fact, the current Embedded Eiffel runtime environment only uses 110 K of ROM and 70 K of RAM. Embedded Eiffel is a compiled embedded systems programming language. It is designed to execute as quickly as comparable, compiled languages such as C. The object-oriented features of Embedded Eiffel empower the programmer to create complex parallel real-time embedded systems. The Embedded Eiffel language provides the freedom to create safe and error-free code at the higher level of objects. Embedded Eiffel provides a robust multithreaded programming environment with garbage collection. Hewlett-Packard, Inc. describes the motivation behind the design choice very well in this quote from their Web site: “The printer software had to be multithreaded, taking advantage of the natural concurrency of printer tasks to divide the processing into a number of parallel threads.” Garbage collection is a design goal of Embedded Eiffel. Its garbage collection engine is robust and each thread instance has its own garbage collection instance. The Hewlett-Packard Web site says “The dynamic model is designed so that memory reclamation, in a supporting environment, can be automatic rather than programmer-controlled.” The Embedded Eiffel language’s garbage collection facilitates an object oriented programming development methodology by allowing programmers to detach themselves from the rudiments of memory management. The standardized library empowers programmers who program on several different embedded systems architectures. The standardized library is intended to reduce bugs and implementation time. Some controversial features which are specific to Embedded Eiffel are: no go-to statements, no global variables, no low-level pointers and no pointer arithmetic. Disallowing the use of these features is intended to raise the level of abstraction in programming Embedded Eiffel. It forces programmers to use higher level tools to accomplish their goals. Embedded Eiffel’s designers were interested in raising the level of abstraction during application program specification and implementation. The goal was to speed up the development of a specific embedded application by hiding static low-level details from the programmer. The intention of imposing these constraints was to enhance program robustness, for they prevent the programmer from using error prone programming idioms. An interesting facet of Embedded Eiffel is that it is not only intended to be an embedded systems programming language; it is also a software engineering design methodology. When a team is programming an embedded product with Embedded Eiffel, they are using an entire design methodology tailored to quickly produce a working, safe and error-free product (according to its creators). The Embedded Eiffel design methodology, termed concurrent engineering, breaks a project into concurrent “clusters.” Then each cluster is developed using a conventional waterfall design methodology. Embedded Eiffel is limited by the fact that it is a proprietary language. Programmers must buy, or arrange to buy, the compiler, training seminars, books, and other programming essentials. Embedded Eiffel is also a relatively new language.

3 Conclusion

Even considered within the narrow scope of embedded systems, the decision of what language to use to implement the solution to a given programming problem is a difficult one. Many factors must be considered and different weights given to each of them.

The factors relevant to a language decision probably include at least:

- Efficiency of compiled code
- Source code portability
- Program maintainability
- Typical bug rates (say, per thousand lines of code)
- The amount of time it will take to develop the solution
- The availability and cost of the compilers and other development tools
- Your personal experience (or that of the developers on your team) with specific languages or tools

All of these factors are important. Yet the majority of embedded programmers just seem to assume that <insert your favourite language here> (almost invariably assembly or C when you're talking to embedded folks) is always the right choice. The fact that some of them always pick assembly and others C is one clue that the language decision is more complicated than they would have the rest of us believe.

We're absolutely convinced that matching your choice of programming language to the computing task at hand is essential. If you choose correctly, you will be rewarded with a straightforward design and an even easier implementation. If you choose wrong, you may run into problems at one of those points or, worse, during the subsequent integration and/or testing phase. A bad language choice may not keep you from finishing, but it could cause a lot of headaches along the way.

Unfortunately, selecting the correct language is a combination of experience, talent, and luck that can't be translated into a formal decision-making process. Too many variables and too many project-specific issues must be considered.

4 Bibliography

- (28. 12 2006). <http://www.embedded.com/showArticle.jhtml?articleID=196800175>
- (29. 12 2006). <http://www.embedded.com/showArticle.jhtml?articleID=192503587>
- (29. 12 2006). http://en.wikipedia.org/wiki/Programming_language_abgerufen
- (29. 12 2006). http://en.wikipedia.org/wiki/Ada_%28programming_language%29
- (28. 12 2006). http://en.wikipedia.org/wiki/C_programming
- (5. 1 2007). <http://www.embedded.com/showArticle.jhtml?articleID=183700818>
- (8. 1 2007). http://en.wikipedia.org/wiki/Java_%28programming_language%29
- (8. 1 2007). <http://www.wirelessdevnet.com/channels/java/features/j2me.html>
- (8. 1 2007). <http://en.wikipedia.org/wiki/J2me>
- Maurer, S. S. (3. 1 2007). A survey of embedded systems programming languages.