SAND2000 - 2178C

An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software

James A. McCoy
Surety Electronics & Software Department
Sandia National Laboratories
jamccoy@sandia.gov

Abstract

As more complex and functionally diverse requirements are placed on high consequence embedded applications, ensuring safe and secure operation requires an execution environment that is ultra reliable from a system viewpoint. In many cases the safety and security of the system depends upon the reliable cooperation between the hardware and the software to meet real-time system throughput requirements. The selection of a microprocessor and its associated development environment for an embedded application has the most far-reaching effects on the development and production of the system than any other element in the design. The effects of this choice ripple through the remainder of the hardware design and profoundly affect the entire software development process. While state-of-the-art software engineering principles indicate that an object oriented (OO) methodology provides a superior development environment, traditional programming languages available for microprocessors targeted for deeply embedded applications do not directly support OO techniques. Furthermore, the microprocessors themselves do not typically support nor do they enforce an OO environment. This paper describes a system level approach for the design of a microprocessor intended for use in deeply embedded high consequence applications that both supports and enforces an OO execution environment.

Introduction

Deeply embedded computer systems, ones that don't usually interact directly with a human, and that are used in safety critical or security critical applications must meet several strict criteria to be considered for certification. First, these systems must be ultra-reliable. Reliability is a cornerstone for security and safety and the line between reliability and security or reliability and safety can be somewhat blurry. At Sandia National Laboratories the term surety has been used to convey the fact that these three tenants of high consequence systems are inseparable and must be considered together during the development of safety and security critical designs. Reliability plays an essential role in surety systems, not only in maximizing the time between failures, but also in how the system behaves in case of a failure. A reliable system will fail in a provably deterministic manner, the details of which being determined by whether the system is intended for safety or security. In safety critical applications, some minimum functionality may be required, while in a security critical application the best response to a fault may be to shut down completely.

Another major requirement for high consequence systems is that they be analyzable, which implies the system is understandable for the designers and testers as well as the analysts who evaluate the system's response to faults. Analyzability starts with managing the complexity of the design. Clear, complete and accurate documentation is a basic requirement for any system. A design hierarchy built from easily understandable pieces, that are well partitioned and planned out is also critical for understanding a design. For most high consequence applications, the analyzability and reliability of the system take precedence over raw performance, as long as the system throughput requirements are met. In many cases, proper partitioning between hardware and software based on the real-time requirements of the system can make the raw performance of the processor a non-issue. In deeply embedded systems, raw performance is hardly ever the issue; power and volume constraints usually drive the tradeoffs between software and hardware. However, most designers will find a way to take advantage of any unused resources and most designs expand to fill the available design space due to changes and additions to requirements. Any available performance will probably not go wasted.

The Java Execution Environment

The Java programming language is a general purpose, class based, object oriented programming language that is platform independent and was originally intended for embedded consumer electronic applications. The platform independence of Java is achieved through two principle features of the language specification: the class file and the Java Virtual Machine. [1]

Table 1. Contents of a Java class file.

Item	Size	Description
Magic number	4 bytes	Identifies the file as a Java class file.
Minor version	2 bytes	The least significant part of the class file version number.
Major version	2 bytes	The most significant part of the class file version number.
Constant pool count	2 bytes	The number of elements contained in the constant pool.
Constant pool	variable	Contains the constant pool information for the class.
Access flags	2 bytes	Indicates the access permissions and properties of the class.
This class	2 bytes	An index into the constant pool containing a symbolic reference to the class defined by this class file.
Super class	2 bytes	An index into the constant pool containing a symbolic reference to the parent of the class defined by this class file.
Interface count	2 bytes	The number of direct superinterfaces this class implements.
Interfaces	Variable	A table of indexes into the constant pool that contain symbolic references to the interfaces this class implements.
Fields count	2 bytes	The number of fields defined by this class
Fields	Variable	Contains the information about the fields declared by this class.
Methods count	2 bytes	The number of methods defined by this class.
Methods	Variable	Contains the information about the methods declared by this class.
Attributes count	2 bytes	The number of attributes associated with this class.
Attributes	Variable	Contains the class attributes.

Java Class Files

Java class files are produced by a Java compiler from the application source code. Each class definition in the source code produces a class file. A class file is composed of a stream of bytes containing all the information about that class. A Java class file has the structure shown in Table 1. [3]

The Constant Pool

The constant pool in each class file provides a roadmap for that class. Besides containing String, integer, long, float and double constants, the constant pool also contains symbolic references to other classes and their methods and fields, as well as the methods and fields defined by this class. This roadmap is used by objects instantiated from the class to interact with other objects in the application. For example, when one object wants to invoke a method of another object, the first object will use its constant pool to "lookup" the second object's method. Since the constant pool generated by the Java compiler contains symbolic references to the things an object needs to access, the constant pool must be resolved into the actual physical locations of the classes, methods and fields within the application before it can be used. This is accomplished by a component of the Java virtual machine called a class loader. As each new class is

encountered, the class loader will resolve the symbolic references within the class file and then load and link the class file into the application. [4]

Fields and Methods

Fields make up the variables associated with classes and objects. There are two types of fields, static and instance. Static fields are associated with the class and only a single copy of the field ever exists, and is shared by all objects instantiated from the class. On the other hand, instance fields are associated with objects instantiated from the class and each object contains its own copy of the field. Accessibility of an object's fields can be completely restricted (private), to completely unrestricted (public), with some variability in-between (protected and the default, module). [2]

Methods provide the behavior of the classes and objects. Like fields, there are two types of methods, static methods and instance methods. Static methods are not associated with an object while instance methods can only be accessed through an object. The "main" method, which provides the entry point for the application is always a static method. Objects are created and their methods accessed to provide the required behavior for the application. Like fields, the accessibility of an object's methods can be controlled. [2]

The Java Virtual Machine (JVM)

The JVM is an abstract machine that can load class files into its memory, perform integrity checks on the information contained in the class files, and then execute the code contained in the class file methods [3]. The abstract nature of the JVM means that it is not targeted for any one particular microprocessor architecture. Instead, a JVM is typically implemented in software on a particular architecture, which emulates the operation of the abstract machine. Several optimizations are done by many JVM implementations to improve the performance of the machine. An alternative approach is to implement some, or all, of the abstract machine in hardware, thus removing the virtual part of the Java machine. Some examples of Java processors can be found on the Internet [6] & [7].

The Java Virtual Machine Specification [3] provides most of the information necessary to implement an abstract machine. Some decisions about structure and the details of mapping the abstract architecture to a real microprocessor are left to the implementers. The primary features of a JVM are captured in the following list:
It contains a method area for each class loaded. The method area contains information about the class and where it sits in the object hierarchy as well as all the methods implemented by the class. Knowing where its method area is located allows an object to know what its "type" is at runtime. Therefore, the correct behavior for the object is performed regardless of the type of the reference that points to the object.
It contains a constant pool for each class loaded. The resolved constant pool provides a roadmap to the other classes and objects in the application.
It contains one or more class loaders, which are used to load and link the class files that make up the application program. As part of the process of loading class files, the JVM also checks the integrity of the contents of the class files to ensure they have not been corrupted.
It contains at least one stack frame where data is manipulated and local variables are stored.
It contains a heap where objects are created and stored while the application is running.
[4].

The Class Loader

The class loader's main job is to load all the class files that are needed to run the application program. As part of this process it also checks the integrity of the class files as they are loaded, it resolves symbolic references, and links the class files into the JVM's program memory space [3] [4]. In the more common software implementations, the class loader is actually part of the JVM and so this process occurs dynamically, with classes being loaded only when needed. In this case, the concepts of load time and run time are very closely related and can be intermixed, with classes being loaded after the application starts running. In a deeply embedded application where the program is stored in Read-Only Memory (ROM), the load time and run time aspects of the application are considerably more separated. Since load time and run time are separate, the class loader may not be part of the Java machine, especially in low volume applications where minimizing the memory footprint is a requirement. However, they are still very closely related since the machine depends upon the class loader to build the application executable correctly. For an object aware microprocessor implementation of a Java virtual machine, the ROM data structures and their contents provide the "knowledge" to create and access objects. Thus, the class loader and the object aware microprocessor are intimately related.

Why Java?

One might ask why choose Java for use in high consequence embedded applications? There are two main reasons for using Java. The first reason is the benefits the language brings to the programming environment. Besides being a true object oriented language with all the benefits that come with that programming paradigm, Java is also a simple, general purpose language with several features that make it attractive to developers concerned with the security and reliability of their systems [2] [3]:

- No pointers. Java does not support pointer arithmetic the way C and C++ do. Java uses references, which cannot be manipulated the way pointers can in other languages. Pointer errors are notoriously hard to find. By eliminating pointers, Java removes a large class of errors from ever being introduced into the application.
- Arrays are true objects. Java treats arrays as true objects so there is no ambiguity about how to access elements within the array. An even more appealing result is that bounds checking on array accesses is built in so it is impossible to inadvertently read or write beyond the end of the array. This prevents a basic security problem from occurring without any extra effort from the programmers.
- No global variables. In Java, name spaces are important. So variable name clashes are much less likely to occur. This means that programmers can use common, descriptive names for objects and variables without having to worry that it might cause another variable somewhere else in the program to be modified inadvertently. A very difficult bug to track down and fix occurs when one area of a large program modifies a variable being used by another area when it is not supposed to. Finding and fixing these kinds of errors is very time consuming and unproductive.
- No uninitialized variables. Java doesn't allow any variable to be used before it has been initialized and there are default initialization values defined by the JVM specification [3].

Bytecodes support and enforce the security of the execution environment. The Java bytecodes can be thought of as the machine instructions of the JVM. Unlike other processors where the machine language can be used to bypass security or safety features built into the high level language, the bytecodes enforce the same restrictions on the execution environment as the Java language does [1] [3] [4] [5]. Of course if the bytecodes are being emulated on a microprocessor that doesn't enforce the security model, it is entirely possible that the security could be bypassed or compromised for a particular JVM implementation. Java defines a native method interface for extending the language to support hardware interfaces, but as will be shown below a hardware implementation of the JVM can minimize the need for extra machine instructions to just one or two, and still provide a flexible and powerful interface to hardware peripherals.

The second main reason for choosing Java is its platform independence and compact nature. Java is a very efficient language, from the small, compact size of the class files to the use of a stack-based architecture. While the class file structure is complex, once the structure is understood it is easy to extract the information. Therefore, the requirements for a class loader are straight forward and understandable. Taking advantage of a formal strategy to transform the class files into an executable ROM image may provide a provably correct means of generating that executable for high consequence embedded applications.

Since Java is platform independent, implementing a hardware version of the JVM is also fairly straight forward, especially if multithreading and garbage collection are not included. For a high consequence embedded system, the use of multiple threads of execution would make the system much less analyzable. It is rarely needed for deeply embedded systems where interaction with a human is usually unnecessary.

Garbage collection is also more difficult to analyze and the techniques used in software implementations of the JVM are non-deterministic as to when the garbage collection will occur and how long it will take. This is unacceptable for many embedded systems, especially those dealing with high consequence safety and security problems where certain critical actions must take place at precisely prescribed times. The loss of garbage collection may not be a problem for smaller, deeply embedded systems where power is not applied continuously. In these applications, objects are typically created once and last for the duration of a power cycle. This "poor man's" garbage collection is more than sufficient for many small-embedded systems.

An Object Aware Architecture For Embedded Systems

Taking advantage of the platform independence of Java allows a hardware implementation of the JVM to be built that has object awareness built into the microprocessor. This object awareness is possible because of special purpose registers in the hardware, instruction processing that is object aware, and the data structures in the ROM image that support the creation of and access to objects. The special purpose registers provide runtime links to the data structures in the ROM, enabling the microprocessor to "know" what type of object is associated with the method currently being executed. They also provide access to that object's data and methods, and other objects within the application. As each bytecode instruction is fetched from program memory, the instruction decode unit within the microprocessor uses the special purpose registers to access object data and invoke other methods. As new methods are invoked, the current state of the microprocessor is saved on the state stack and a new state, possibly for a different object, is created. Information for the new state is obtained from the data structures stored in the constant pool and method area of the calling method. When the new method begins execution, the special purpose registers now point to the new method area and constant pool. When the new method returns, the state of the old object's calling method is restored and executions proceeds with the state of the old object contained in the special purpose registers.

Based on the JVM

Sandia's Secure Processor (SSP) is based on the JVM described in The Java Virtual Machine Specification by Tim Lindholm and Frank Yellin [3]. Additional details and information are provided by Bill Venners' Inside the Java Virtual Machine [4] and the O'Reilly book Java Virtual Machine by Jon Meyer and Troy Downing [5]. Several features of a traditional JVM are not implemented in the SSP. Multithreading and garbage collection were left out for the reasons specified above. Since garbage collection is not supported in the current version, built in String objects are also not supported. The basic data structures and instructions for handling Strings are defined for the SSP but were not included in this version. String manipulations can create a very large number of String objects on the heap, and those objects take up valuable memory space and are generally not reusable. This is not considered a problem for the first SSP application since it is deeply embedded and does not require interaction directly with a human, the primary use for Strings in most applications that require them. If a true garbage collection capability is added to a future version of the SSP, String support can be easily added. A floating-point arithmetic unit is not included in the current version so the floating-point instructions are not implemented. The first applications targeted for the SSP do not require floating point capabilities. Adding a floating-point unit (FPU) and implementing the floating-point instructions would be straightforward for an enhanced version of the SSP.

The remaining bytecode instructions are completely implemented in the hardware with only two additional non-JVM instructions added for object oriented I/O support. Thus, other than the exceptions stated above, the SSP supports and enforces the object oriented execution environment defined by the JVM and the Java language, including the interaction with external hardware peripherals. This environment is possible because of the support provided by Sandia's formal transformation based class loader. The class loader is implemented using the High Assurance Transformation System (HATS) currently in development at Sandia [9]. Figure 1 illustrates the process of building a ROM image.

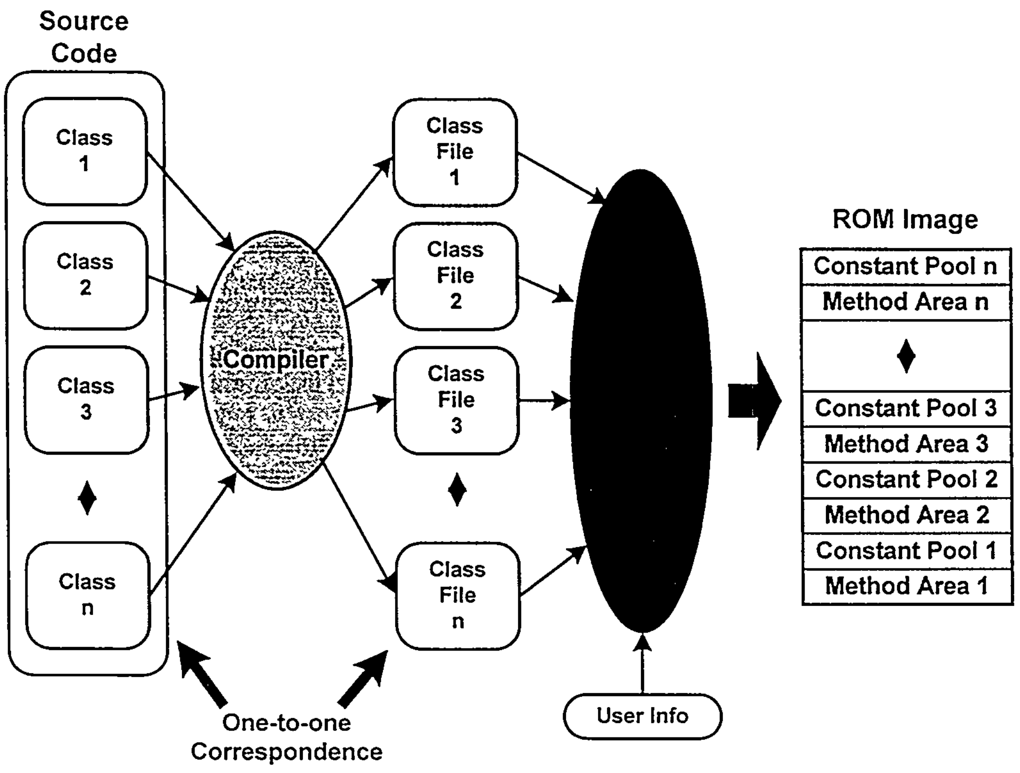

Figure 1.

The source code that makes up the application program is compiled and the compiler generates a class file for each class definition. The class loader is responsible for transforming this collection of class files into an executable ROM image containing the method area and constant pool data structures for each class that are necessary at runtime. The class loader loads, resolves and links all the necessary class files, utilizing user supplied information about the application. This information includes the name of the class file containing the "main" method and the memory locations of the static and instance I/O peripherals. As it loads each class file, it creates a method area and resolved constant pool for that class and records their location. It also records all other classes required by that class and loads them if they are not already loaded. Once all the class files have been loaded, all references to a class are incorporated into the data structures of all other classes that reference that class. The class loader also builds an initialization method in the Startup class. This method initializes all the static fields of all the classes. The class loader also builds the initialization methods for the static and instance I/O objects, copies the bytecodes from the methods in each class file and builds an exception table for each method. As it copies each method's bytecodes to that class' method area, the class loader converts immediate data into the form required by the SSP's instruction execution unit. This conversion preserves the bytecode integrity and relative location within the method so that the branch offsets computed by the compiler remain valid.

Modified Harvard chip architecture

The SSP is based on a modified Harvard architecture. In a traditional Harvard architecture, the program memory and data memory spaces are separate, with separate address and data busses. The SSP goes a step further and separates the main data memory into individual stack, heap and state memories with their own controllers for address and data busses. The program memory has it's own controller also. This partitioning is primarily to improve the understandability and analyzability of the processor design, but it also enables a high degree of parallelism during instruction execution.

As discussed above the program memory contains the method areas and constant pools for all the classes contained in the application program. The program memory controller contains the special purpose registers used to access these data structures as well as the Program Counter register (PC) and the control logic for performing program memory operations. The special purpose registers are composed of a Method Area register (MA), a Constant Pool register (CP), an Exception Table register (ET) and a Super Method Area register (SMA). The ET and SMA registers are used during exception processing.

The stack memory contains each executing method's stack frame. A stack frame is composed of zero or more local variable locations at the start of the frame. Local variables are also known as automatic variables since these variables are only accessible while that stack frame is active. The Frame Pointer register (FP) points to the start of the frame and the local variables are accessed as offsets from the FP register, starting with zero. Following the local variables is the current method's stack, which is used for data manipulation. The Stack Pointer register (SP) keeps track of the top of stack. Data to be operated on is pushed onto the top of the stack by the program. An instruction that operates on the data will pop the data off the stack, perform the operation, and then push the result back onto the top of the stack. As an example we will go through the operations necessary to add the two integers contained in local variables 0 and 1 and store the result back in local variable 2. Assume local variable 0 contains the value 10 and local variable 1 contains the value 22. The program loads the contents of local variable 0 onto the top of stack and the processor automatically advances the SP register to point to the new top of stack. The program then loads the contents of local variable 1 onto the new top of stack, and then issues the integer add instruction. This instruction pops the top two values off the stack, adds them together using integer arithmetic and pushes the result back onto the top of stack. The program then stores the top of stack value into local variable 2. This example is illustrated in Figure 2.

Step 1			Step 2			Step 3			Step 4			Step 5		
						SP	22							
			SP	10			10		SP	32				
SP		2			2			2			2	SP	32	2
	22	1		22	1		22	1		22	1		22	1
FP	10	0	FP	10	0	FP	10	0	FP	10	0	FP	10	0

Figure 2.

When the current method returns, its stack frame is destroyed and the previous method's stack frame becomes active. Once a stack frame has been destroyed, the data that was stored in its local variables and on its stack are unavailable. The SSP will store zeros in these locations as the stack frame is destroyed.

The heap memory is used to store the static fields for all the classes in the application program and the instance fields associated with objects that are created by the program. Objects that are created on the heap contain additional information allowing the object to locate the class used to create it. This information provides the mechanism by which the object's virtual methods are invoked. All normal methods in Java are virtual methods, which means that the type of the object rather than the type of the reference pointing to the object must be used to determine the correct method to invoke. A child object can override its parents methods and a reference declared to be of the parent type can point to the child object. So the type of the object and not of the reference must be used to determine the correct method to invoke in order for the child's correct behavior to occur [6]. Each method area contains a method table that is used to determine the correct method. An entry in the constant pool associated with the invoking instruction contains the index into the method table. A child class inherits its parents method table. If the child has additional virtual methods, they will be added on to the end of the child's method table. If the child overrides one of the parent's virtual methods, that entry in the method table will be replaced by the child's method. Therefore, because the object can access its method table it is able to invoke the correct method regardless of whether or not that method has been overridden. The SSP provides the means to write zeros to all used heap locations upon command.

The state memory provides an efficient means of saving the processor state when invoking a method. The processor state consists of the MA, CP, PC, FP and ET registers. These registers are pushed onto the state stack during the invocation of a method and are popped off the state stack when the method returns, restoring the previous state.

Figure 3 illustrates the runtime configuration of the SSP.

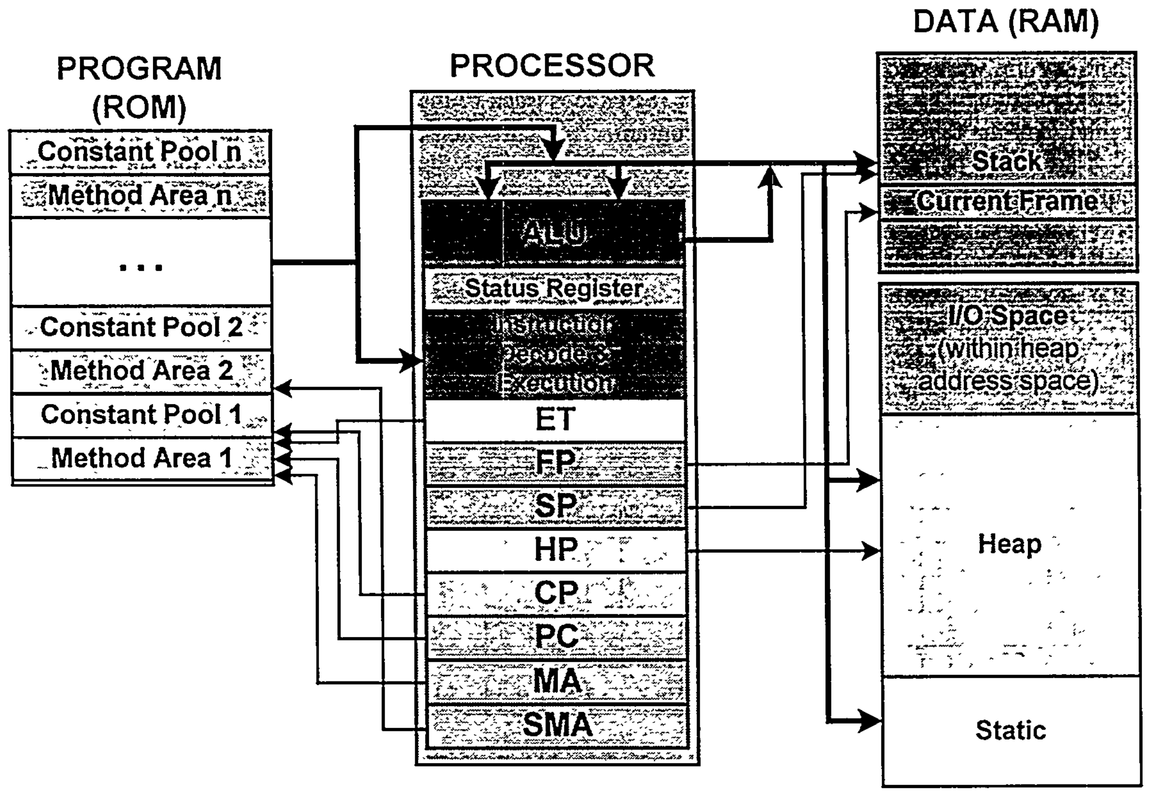

Figure 3.

The SSP I/O Models

The SSP is capable of supporting traditional I/O devices through its static I/O model. Commercial microprocessor peripherals that do not support the SSP's object oriented I/O model are incorporated into an application by providing a static I/O class for each peripheral device. The class loader builds data structures into the constant pool for these classes that support accessing the static fields within these I/O devices. Any commercial device that supports memory mapped I/O can be supported through these static I/O classes.

The SSP also supports an Object Oriented I/O model (OOI/O) that allows I/O devices to be accessed as normal objects, even having multiple instances of the same device (located at different physical addresses, obviously). The class loader plays a very significant role in supporting these devices. OOI/O objects are defined using two class definitions. The first class defines the data fields within the I/O device and the methods for accessing them. The order and type of the fields must match exactly with the registers in the I/O device. The user must supply the class loader information about each OOI/O class. This information includes the number, order, names and types of the device registers corresponding to the fields defined in the class. It also contains the location within the I/O address space of each instance of the OOI/O class and the name of the container class that corresponds to the OOI/O class. Each OOI/O class has a corresponding container class which only contains static references to instances of the OOI/O class. Since these references are static and they are initialized to new instances of the I/O objects, the compiler will generate an initialization method for the container class that creates the objects and initializes them. The class loader will generate startup code that calls these initialization methods before the application program starts so all OOI/O objects will have been created and initialized upon entry into the main method. The OOI/O objects are accessed exactly like ordinary objects allowing the I/O behavior of the system to be expressed at the same level of abstraction as the rest of the application. Figure 4 illustrates the relationship between regular objects and I/O objects. Figure 5 contains an example of a simple OOI/O class and Figure 6 shows its corresponding container class.

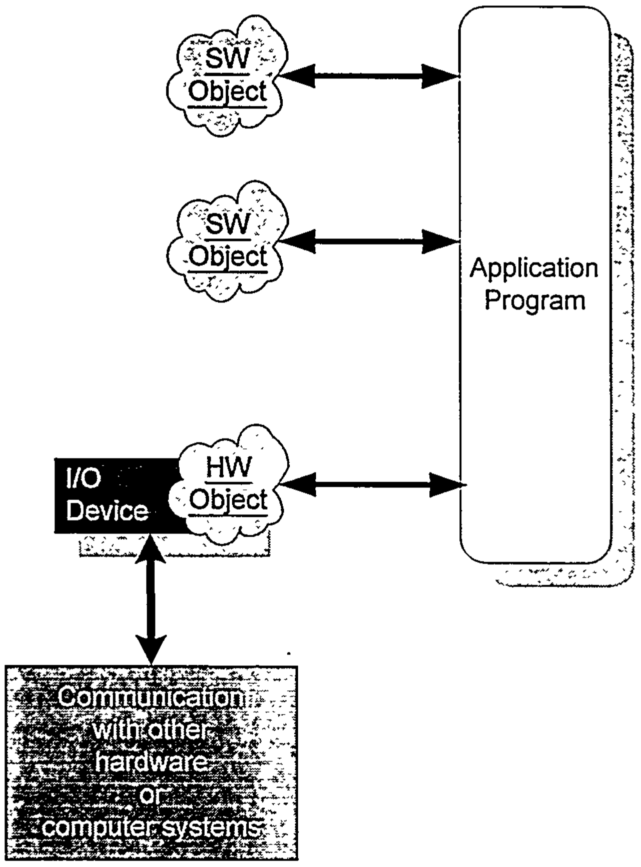

Figure 4.

```
final class SSPsimpleUART
{
  public static final byte TRANSMIT_READY_FLAG = 0x02;
  public static final byte RECEIVER_FULL_FLAG = 0x01;

  private volatile byte dataReg;
  private volatile byte statusReg;

  public boolean sendData(byte data)
  {
    if ((statusReg & TRANSMIT_READY_FLAG) == TRANSMIT_READY_FLAG)
    {
      dataReg = data;
      return true;
    }
    else
      return false;
  }

  public byte getData()
  {
    if ((statusReg & RECEIVER_FULL_FLAG) == RECEIVER_FULL_FLAG)
      return dataReg;
    else
      return -1;
  }

  public boolean dataReceived()
  {
    if ((statusReg & RECEIVER_FULL_FLAG) == RECEIVER_FULL_FLAG)
      return true;
    else
      return false;
  }

  public boolean canSend()
  {
    if ((statusReg & TRANSMIT_READY_FLAG) == TRANSMIT_READY_FLAG)
      return true;
    else
      return false;
  }
}
```

Figure 5.

```
class UARTcontainer
{
  public static SSPsimpleUART ser1 = new SSPsimpleUART();
  public static SSPsimpleUART ser2 = new SSPsimpleUART();
}
```

Figure 6.

Conclusions

In many instances the security and safety of a microprocessor based system relies on the features of the programming language used for implementing the application. In traditional deeply embedded applications, the implementing language is usually C, assembly language or a combination of both. While these languages are sufficient for implementing these systems, they do not provide any significant improvement for understanding and analyzing the resulting design. Higher levels of abstraction are necessary to provide the necessary understanding of the intended operations of the system under development. However, the higher levels of abstraction can add significantly more complexity to the analyst's job, especially when large gaps exist between the abstract model used to describe the operation and the low level details of the implementation.

Another significant disadvantage of traditional embedded environments is the vulnerability of the system to subversion at the machine level due to errors undetected at this level. The machine level provides the means to bypass any security or safety critical features implemented in the high level language because the microprocessor doesn't enforce any of these features. Either special hardware must be included to detect any circumvention, or reliance on administrative procedures such as code reviews and testing are necessary. In either case compliance is difficult and expensive to obtain. At the same time, this level is also the most complex and hardest to understand. Errors can easily be overlooked due to a lack of understanding or the shear complexity of the design.

Object oriented methodologies have proven significantly more helpful in providing the abstraction necessary to work successfully in the application domain. The current object oriented languages like C++ and Java have proven beneficial for use in implementing the abstract models. However, when the design is analyzed at the machine level, there is very little resemblance to the abstract model the implementation is based on. One solution to this dilemma is to embed an object awareness into the hardware, providing a strong cohesion between the abstract model describing the operation of the system and the hardware and software used to implement it. For this object awareness to work, a new microprocessor architecture is necessary, one that directly supports objects. This new microprocessor architecture relies heavily on a provably correct class loader capability. The class loader is responsible for creating the data structures that the hardware relies on to make it object aware. Only a coordinated cooperation between the class loader software and the secure microprocessor hardware can provides a total solution for high consequence embedded systems.

This paper presented an overview of an object aware microprocessor currently in development at Sandia National Laboratories. The architecture is based on the Java Virtual Machine, incorporating additional capabilities that provide an object oriented I/O model. This I/O model enables the application to treat I/O device interfaces the same as other objects in the design. This architecture narrows the gap between the abstract model used to describe the operation and the implementation of that model. It is our belief that by narrowing this gap, a significant improvement in understandability and analyzability of designs incorporating this architecture will occur. This improved understanding and analyzability will in turn improve the security, safety and reliability of those designs.

References

[1] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley. 1996.

[2] K. Arnold and J. Gosling. The Java Programming Language Second Edition. Addison-Wesley. 1998.

[3] T. Lindholm and F. Yellin. The Java Virtual Machine Specification Second Edition. Addison-Wesley. 1999.

[4] B. Venners. Inside The Java Virtual Machine Second Edition. McGraw-Hill. 1999.

[5] J. Meyer & T. Downing. Java Virtual Machine. O'Reilly and Associates, Inc. 1997.

[6] B. Eckel. Thinking In Java. Prentice Hall PTR. 1998.

[7] Sun Microsystems, Inc. *www.sun.com/microelectronics/picoJava*.

[8] Patriot Scientific Corporation. *www.ptsc.com/psc1000*.

[9] V. L. Winter. *Software Construction via Abstraction, Synthesis, and Transformation*. Proceedings High Integrity Software Conference. 1997.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.