VMFoundry™

(Rev 1.2)

Mark Medovich - CTO
Websprocket

Design Automation of Supranet Systems: Benefits for Hardware Design and Bringup

VMFoundry™ automates the process of building an embedded system from libraries that characterize a variety of low level boot, diagnostic, bringup, driver, system, process, and platform behaviors. The libraries are written entirely in Java and may be extended or customized by the designer per the needs of system requirements.

Since Java has many APIs and abstractions for systems and networks, is object oriented, has a large developer base, and is easy to learn, it is an excellent choice for networked, embedded system design automation.

Java Applications Converted to Synthesized Image vs. Java Applications Running on a Java Virtual Machine and OS

Java Applications running on a JVM/OS

Java applications are usually compiled to bytecode. The Java bytecode runs on a statically compiled JVM which has been ported to a target (statically compiled) OS. The software layers of JVM and OS penalize Java application performance because of excessive calls between static, monolithic, procedural software and dynamic, interpreted, object oriented software layers. The amount of software required to run the smallest Java applications tends to be very large because the OS and JVM must be present. There is a much greater problem. Java assumes a garbage collected managed memory environment. *The contract and policy from a JVM's GC and an operating system's memory management policy cannot be standardized.* Java has therefore performed very poorly in embedded system applications that attempt to use the full breadth of the Java programming environment. For example, server applications tend toward large amounts of garbage creation. This action over time will fragment memory. Keeping a separate swap space is mandatory for memory de-fragmentation and compacting algorithms. These problems are addressed via synthesis.

A Synthesized Application

From a programmer's application development perspective, there is little or no difference between writing Java code for which runs on a JVM/OS or for VMFoundry™ synthesized runtimes. Java applications are compiled using a standard off the shelf Java compiler (e.g. javac) in either case. For example, if you write a socket based service on Java's .net API, it runs on any platform that has a JVM. The same Java code can be synthesized and run directly on hardware using VMFoundry™. The synthesized runtime features an object oriented real-time, and deterministic garbage collected memory management system. This solves Java's performance problems for embedded systems with no virtual memory. It does so by exhibiting superior memory management characteristics.

The synthesized application also exhibits superior memory footprint characteristics.

Shortcomings of Java APIs for Hardware Devices and Real-time

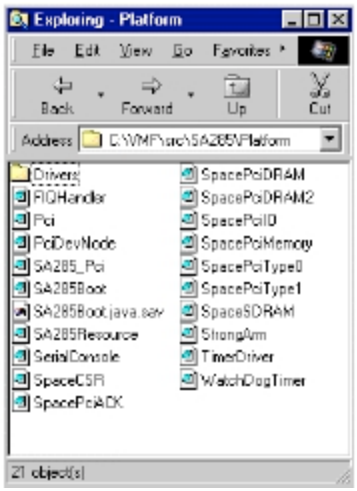
The Java platform lacks APIs for system components, devices, device drivers, boot, target debug, processes, and MP. These APIs are essential for any embedded system.
Java is flexible enough to be used to describe APIs for the aforementioned missing system components. Websprocket has developed APIs for device drivers, messages, interrupts, boot, target debugging and so on, and logically organized them in a design hierarchy which is user friendly to an embedded system designer. If a designer needs to add a component to the design hierarchy, the WebSprocket packages are general purpose enough to accommodate most any need. Once a physical system design has been sufficiently described in a platform design tree, the programmer can build software for said system completely from a top down view, that is, from a Java application itself. Step by step rapid prototyping examples are given in later sections of this document.

An embedded system designer can customize all components necessary to run the desired application, including debug, boot, bringup, devices and device drivers, processes, and user threads. A target platform is selected by the user, from the VMFoundry™ platform libraries.

VMFoundry's™ platform libraries are hardware abstractions which characterize target processor hardware features, and platforms (e.g., development boards, motherboards, etc.) which integrate said processors on a PCB with external peripherals, memory, and bus expansion.

Users may customize platforms with Websprocket's architecture specific porting kits. For example, a custom design using the ARM CPU core will have numerous custom features integrated on an application specific processor. The APIs of an architecture specific porting kit allows the user to define Java classes which characterize the custom features implemented by the designer. Once done, VMFoundry's synthesizer can synthesize for the custom target. VMFoundry™ together with WDKLite™ assume a design hierarchy organized by package name. The main components of a design library are the src folder and the J2ME, J2ME/J2SE Community Source Derivatives, or JemINI folders. The J2ME, J2ME/J2SE Community Source Derivatives, or JemINI folders contain the source or class files for Java foundation libraries used in the design phase. These can be customized as needed by the designer. The src folder contains the WebSprocket packages, supported motherboard or hardware development system boards platform libraries, and code management administrative utilities. For illustrative purposes we will explore some folders in the src folder.

VMFoundry's Platform package for the Intel™ EBSA-285 StrongARM™ development system is illustrated as follows:



Websprocket Platform Library for Intel SA-285

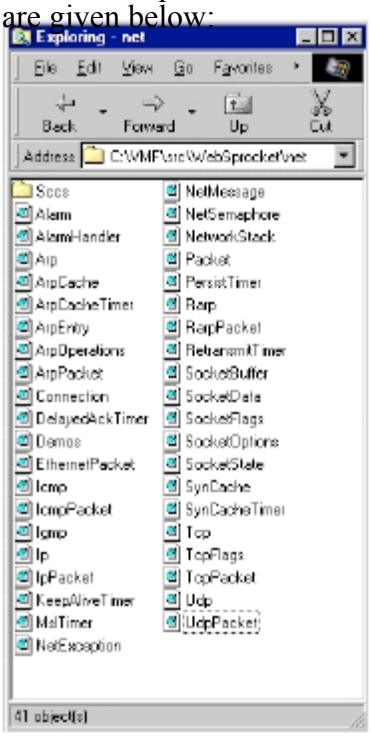
The above files are used to define hardware abstractions for the components and features on a motherboard and it's microprocessor. In the case above, the SA285\Platform folder contains classes relevant specifically to the Intel EBSA-285 board configuration. As an example, the above contains a physical abstraction for defining memory partitions in a physical system, namely, MemoryPartition.java in the Kernel package. The Intel EBSA-285 defines an address range within the overall system for devices. The range is denoted as the "CSR" partition. A designer wishing to support physical accessors into this partition would define a unique class which has fields reflecting the physical system. The SpaceCSR class above, extends MemoryPartition to define the "CSR" address partition of the SA110 peripheral I/O address space in physical memory. More detail on physical system abstraction is provided later in this document.

The ***Kernel*** folder contains numerous synthesizable modules for building boot behavior definitions, and dynamic operating system behaviors:

For example, the Boot class in the above is defines the lowest level of bring up such as processor state and interrupt vector table initialization.
As another example, the DeviceManager object in the above view allows a designer to create a DeviceManager instance which acts as a device repository for a system which needs to support pluggable devices and drivers. VMFoundry™ uses these kernel classes to build a custom object oriented system for an application, based on the application needs and requirements. Websprocket has introduced the idea of Kernel inheritance wherein a Thread inherits it's operating system (patent pending). Naturally, as an entirely OO design, extensibility is a programmer and system requirement option.

The ***WebSprocket net*** folder is an object oriented networking library that contains necessary modules to create custom network stacks. A visual representation

of the WebSprocket.net's stack classes are given below:



The above libraries accommodate Java.net's interfaces to a network stack normally part of an operating system. Inspection of Jemini's java.net indicates java.net's native methods call methods of the above classes.

VMFoundry™ supports designs compiled with J2ME, J2ME/J2SE Community Source Derivatives (www.sun.com), or JemINI™ (www.jemini.org). Sun Community Source licensed libraries and other Open Source libraries are allow end users and industry experts to create custom platforms which are rich enough to accommodate the target application, yet perhaps lighter weight than a large distribution. Customizing libraries (.lang, .io, .util, .net) may be beneficial to achieve an increase in performance and reduce memory footprint for a particular industry's needs.

Dynamic and Distributed Threads

Since Java includes Thread in it's language, the synthesizer uses thread behavioral models to accommodate the Java Thread. VMFoundry™ generates the threads in such a way to accommodate virtual behaviors as well as local implementations. Threads in a given body of code resident on the target may be dispersed, and actually reside on a server. This feature allows the software design of many new possible implementations such as transparent and instant services.

Threaded Interrupts, Devices and other Real-time Considerations

Any embedded system and real time system naturally must include device driver support and interrupts. VMFoundry™ supports Websprocket's Device API which allows a programmer to write device drivers entirely in Java. The Websprocket device API supports the notion that device drivers can be "threaded". Threads which respond to real-time messages sent from a device interrupt service are pre-emptive, hence determinism is achieved using Java threads. Interrupt priority schemes can be user defined. Real-time behavior for Java via Websprocket's paradigm is based on a very simple message architecture. Any real-time system requirement can be user defined by interrupts and threaded device drivers, service threads and real time messages, thread priority, dynamic thread bandwidth allocation, and supervisor threads. The user defined real-time messages can pre-empt the currently running thread and in a few timer ticks, worst case. Timer granularity is

programmable. 2uS pre-emption has been measure on a 228Mhz StrongARM. It is possible to achieve a finer granularity.

In addition to pre-emption , synthetic threads allow a supervisor thread to dynamically assign a thread's bandwidth requirements. Dynamic thread time slicing was designed into the logOS.java Kernel class using well-known digital signal processing techniques. Thus, this feature makes the WebSprocket.Kernel well suited to media applications. Many other interesting scenarios are possible including synthesizing dynamic, non-monolithic, complete operating systems.

Synthesizing a System

Systems of all types can be synthesized with VMFoundry™ and platform libraries. All systems need to be initialized at boot time so we start with an example an explanation of the boot architecture.

1. Boot Levels

The WebSprocket.Kernel and Platform packages include classes which define a system boot rational and design methodology. There are three bringup or "init levels" defined by the same number of classes. They are:

- Boot.java
- "ProcessorBoot".java
- "PlatformBoot".java

Each boot level is defined as a thread. Boot.java's run method creates for the synthesized runtime, an object reference for a system's memory map and the memory map's object memory partitions. Then, Boot initializes very low-level system behavioral models for traps, exceptions, and a provides an object reference processor entity itself. The processor entity exists as an object for the runtime to ensure a multiprocessing extensibility. Vector table, trap handler, system resources, and supervisor threads must be defined in the first boot level. CPU, ASIC, and embedded systems designers can tailor various classes in WebSprocket.Kernel for the specific needs of the system. Those who wish to synthesize an operating system may also customize Boot and add primal supervisor threads. The WebSprocket.Kernel package snapshot illustrates the hardware and software models for system bringup requirements. A variety of classes are provided for debugging, supervisor threads, signals and so forth. The kernel Java class files are expressed purely in the Java language and add no extensions to the Java language. As such, they are highly customizable. InterruptController of the SA285 (Platform) directory is used to characterize the low-level interrupt handling operation of a processor. It is possible for the designer to extend these libraries to incorporate hardware definition primitives. Said primitives would inherently be programmable from the synthesized system.

The "ProcessorBoot" bring up level is a class which extends the Boot thread. A ProcessorBoot class is used to initialize processor hardware which is not part of the core CPU architecture. For example,

SPARC, ARM, MIPs etc. specify a core instruction set architecture definition. Various implementations of a processor which utilizes a core often include a reference MMU, caches, and on chip debug and test features. These

hardware components are initialized by the ProcessorBoot. The ProcessorBoot thread serves as the second initialization level of booting a system. This 2nd level of init is typically reserved for virtual memory layout (e.g. MMU init), and memory protection definition. The "ProcessorBoot.java" always reflects the bringup and configuration specific to the processor implemented on the target platform. Therefore, it's actual name will reflect the microprocessor it's booting. In the case of the SA-285, the on board processor is the Intel SA-110 StrongARM processor, hence there exists a StrongARM.java file in the SA285 directory. StrongARM.java extends Boot.java of the WebSprocket.Kernel package. WebSprocket.Platform.SA285 includes a PlatformBoot file as well, specifically, SA285Boot.java. This file can be considered the blueprint for bringup and configuration of a "motherboard".

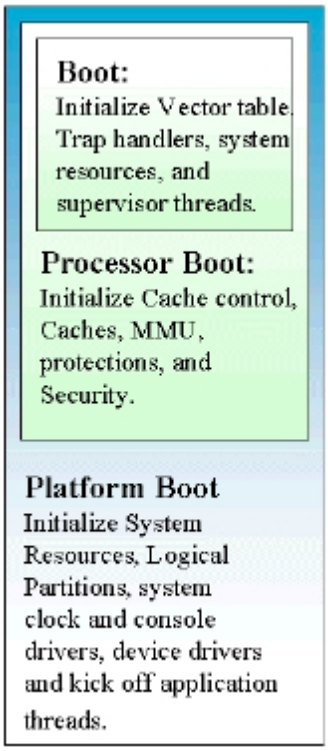
The PlatformBoot Java source is the 3rd and final boot initialization level (init level) and creates logical partitions (MemoryPartition.java) of the system, add devices to a device manager repository, assigns device and system interrupt handlers, starts device driver threads, and kicks off system application threads. Hence, an embedded system designed with the VMFoundry™ synthesizer requires that the designer add application threads, device drivers and devices to the system's "PlatformBoot" main() method. Embedded system designers can easily modify a PlatformBoot of a given hardware platform and incorporate decoded ASICs, peripherals, and other physical changes to the original target platform to accommodate custom designs.

The boot levels are in some ways analogous to a conventional operating system init levels. If a designer's desires to build a specialized operating system, the OS's bring up and boot can be easily defined with the boot classes to accommodate supervisors and processes. Since Java incorporates operating system attributes in the Java language itself, most designers will not have a need to design an OS. Most embedded designers need the rapid prototyping feature of customizing a PlatformBoot.java, then compile, synthesize, test, and deploy.

Summary

Fig. 3,. is a depiction of the system bring up. Synthesis of systems allows the designer to customize boot levels per the needs of hardware and system requirements via the WebSprocket Kernel, and Platform classes. In addition, hardware abstractions for CPU, CPU control, Interrupt controller, bus, devices, device drivers, device management, system resource management, security policy, protection and so on, are defined in Java classes in the Kernel and Platform. These classes are used by VMFoundry™ at synthesis time to create the bare metal extensible machine.

Fig. 3.



Examples

Some code examples will help complete the mental picture of building an embedded system with Java, J2ME, J2ME/J2SE Community Source Derivatives, or JemINI™ source distribution, the WebSprocket.Packages, and VMFoundry™.
As mentioned , the 3 levels of boot are Boot, ProcessorBoot, and PlatformBoot, each representing an "init" level for the system during bring up.

Let's examine the 1st boot level of an embedded system based on the SA285 board, by reading the comments of the source code example, Boot.java:

```
/* A Boot is part of a WebSprocket.Kernel
package. */
package WebSprocket.Kernel;
import WebSprocket.VM.*;

  public class Boot extends Thread {

/* Boot's run() method performs low level
system initialization.  It creates object
references for physical system components
which are needed at runtime*/
   public static void run() {

/* 1st we define a MemoryMap for the System
and make it a SystemResource */
   mp = new MemoryMap();
   SystemResource.setMemoryMap(mp);

/* Define an InterruptController object for
the System and make it a SystemResource */
intrController = new InterruptController();
SystemResource.setIntrController(intrCo
ntroller);

/* Define which processing entity will be the
master for this  system and make it known as
a SystemResource - a processor is an object
*/
      tm = new taskMaster();
      SystemResource.setTaskMaster(tm);

/* Create a Boot object reference for the
system  and initialize the runtime*/
      Boot boot = new Boot();
      boot.initialize();

/* Kick off a thread which fulfills Java's
expectation that a VM actually exists on the
client (e.g., class internals support) */
      VM.start();
    }
/* Boot's initialize method */
      private void initialize(){
/* Create InterruptHandlers for the system.
All interrupt handlers will be of type
InterruptHandler and "registered" in this
array */
      handlers = new
      InterruptHandler[32];


/* Layout the interrupt vector table for the
system*/
      initVectorTable();

/* Define stack area for interrupts*/
      initInterruptStacks();
```

```
/* Create a DeviceManager for adding devices
to the system and make it a SystemResource*/
devManager = new DeviceManager();
SystemResource.setDeviceManager(dev
Manager);

/* Initialize streams */
    System.initIOStreams();

/* Kick off a base level supervisor.  You can
define this if you like */
    suprThrd = new Supervisor(tm);
    suprThrd.start();
  }
```

As can be seen, Boot.java describes the control flow of the startup of a system. Memory layout is not done in Boot because Boot is logically associated with a processor "core" which may not have an MMU. The 1st init level of boot is complete.

The 2nd level of boot is intended to initialize caches, memory topology and protection, and start higher order system threads. The 2nd init is an extension of Boot. A designer implements the methods which will be called to perform the initialization. These methods should be called from the PlatformBoot's main() method.

Therefore, ProcessorBoot serves mostly as a place holder to accommodate fast and flexible portability from one processor to another. For example StrongARM.java extends Boot.java and implements methods used to access coprocessor registers, MMU, and Cache, registers internal to StrongARM. Other ARM variations may implement memory protection and caching uniquely.

Below is a snippet of code from StrongARM.java:

```
/* This method will be called from the
SA285Boot.java (PlatformBoot) to set up the
MMU.  Supervisor threads could be started
from method calls on StrongARM.java. */
   public void StrongArmMMUInit(){

    int accDomainControl = 1;
    int temp;
    int pageTblBase;
    int count;
/* An MMU qualifies as a device.  It needs a
region of memory which will not be GC'ed.
MemoryObj of WebSprocket.Kernel is what
is used for physical memory resources. */
    MemoryObj pageTblObj;
    MemoryMap mp;
 /* Disable MMU */
     temp = 0;
     writeCPControlReg(temp);
/* A MemoryMap object must exist, ..
therefore, go get it */
     mp =
SystemResource.getMemoryMap();

/* A page table base has to have been defined
in a MemoryPartition*/
pageTblBase =
SpaceSDRAM.getpageTableBase();
/* Get a non-GC-able MemoryObj for the
page table*/
pageTblObj =
mp.allocateMemObj("SDRAM",pageTbl
Base,50000,0);


/* Init Page tables, etc.*/
int base = 0;
int pageEntryValue;
pageEntryValue = 0x000C0E;
writeTTBase(pageTblBase);
writeDomainAccessBits(accDomainCon
trol);
  for(int i=0;i < 16;i++){
pageTblObj.write32(base,pageEntryValu
e);
        base+=4;
```

```
        pageEntryValue=
pageEntryValue + SZ_1M;
        }
```

After, the MMU initialization and protected memory definitions are established, supervisor(s) threads can be started as "operating system" processes. It may be desirable for the system designer to have secure threads defined in an inherited ProcessorBoot class for security purposes. For example, an embedded system manufacturers may add supervisors in the 1st or 2nd layer of boot and only expose the PlatformBoot source. This would enable 3rd party developers to add threads only to init level 3 (PlatformBoot) and not expose the security or memory protection model.

Top Down Design

Once a designer or vendor creates a PlatformBoot.java for a target platform or motherboard, an end customer, developer, or user can create custom, networked applications or embedded systems from the "top down". SA285Boot.java is a "PlatformBoot.java" program created for the Intel SA285 StrongARM based development board. The PlatformBoot extends a ProcessorBoot and inherits it's methods. That is, in this case, SA285Boot.java extends StrongARM.java.

If a supplier has already done the work of building a system and defining a Boot.java and a StrongARM.java, a 3rd party developer starts from the top down by adding threads, devices and drivers to SA285Boot.java's main().

Let's now look at the SA285Boot.java layer (the 3rd level of system bring up) and explain with the blue comments:

SA285Boot.java

```
/* A PlatformBoot is part of a
WebSprocket.Platform package. Platform
packages are a hierarchical means for
organization.  For example, the Intel "brutus
board" which is also a StrongARM based
system could appear in the Platform tree in a
Platform.Brutus package.  In the case below
it's the package for the SA285 Platform*/
package WebSprocket.Platform.SA285;
import WebSprocket.net.*;
import WebSprocket.Kernel.*;
import WebSprocket.Application.*;

/* A PlatformBoot extends a Processor Boot
The SA285 development board uses a
StrongARM processor, hence we are
extending "StrongArm*/
public class SA285Boot extends
StrongArm{

   /* We define an SA285Boot and an
   InterruptController for the runtime */
    public static SA285Boot sa285Boot;
    public InterruptController intrCtrl;

/* This is the veritable Java main program.
Those designing applications and systems
based on the SA-285 start here.  The first
thing main must do is create an SA285Boot
reference and kick off it's initialization
method.  This must be done to define the
runtime topology for the system.  For
example, busses, devices, memory partitions,
must be defined per the requirements of the
end design.  In this example, we will build a
simple network appliance which starts a
socket based TCP service on a listener port.
 */
public static void main(String[]argv){

   /* Instantiate an SA285Boot and start it's
      initialization */
        sa285Boot = new SA285Boot();
        sa285Boot.SA285Init();
/* We are inside the main.  A top down
designer can add application threads here.  If
```

```
so desired, they can also kick off application
threads in the SA285Init() method itself.  This
example shows the latter.*/
}

/* The init method we just called from
main().*/
    public void SA285Init(){
/* My embedded appliance will use a serial
console, a pci peripheral (an ethernet
controller), a couple of interrupt driven
devices, drivers, etc.  Therefore, we need an
object for each.*/
        SerialConsole console;
        Pci pci;
        DeviceManager devMgr;
        FIQHandler fiqHandler;
        TimerDriver timer;
        WatchDogTimer watchDog;
/* now we add the MemoryPartions to the
system.  For example, SA-285 has physical
devices mapped to different logical partitions.
PCI, CSR, SDRAM, etc. must each be a
MemoryPartition object in the system */
        addSA285MemPartitions();
/* Once we have the MemoryPartition(s)
added to the system, we can call the
StrongARM's mmu initialization method.
(This method was inherited from
StrongArm.java */
        StrongArmMMUInit();
/* We can also set a debug level for the
system.  This will be used as the "init" level
which catches debug specific traps (e.g.
breakpoints).  */
Debug.setDebugLevel(Debug.LEVEL_2
);
/* We are going to want to used
System.out.print a lot so we need a console.  It
needs to be a SystemResource*/
 console = new SerialConsole();
 SystemResource.setConsole(console);

/* Our embedded system would like to have a
repository for device management (like plug
an play)*/
 devMgr =
SystemResource.getDeviceManager();

/* We want to register our FIQHandler as the
default handler for ARM based FIQ
interrupts.  */
 intrCtrl =
SystemResource.getIntrController();
fiqHandler = new FIQHandler();
intrCtrl.registerHandler(fiqHandler,0x1c
);

/* Our appliance is going to have a threaded
device driver.  When an interrupt occurs for
the specified device, it will send a real time
message to a listener thread.  Priority
permitting, it will pre-empt the currently
running thread and service the interrupt.*/
UsrApplication applThread = new
UsrApplication();
applThread.start();
/* PCI is treated like a device and is a
SystemResource */
        pci = new SA285_Pci();
        devMgr.addDevice(pci);
        SystemResource.setPci(pci);

/* A threaded device driver is instatiated by
passing a listener thread to it's constructor.
We just created the listener thread, so now we
create a device, and pass a listener.  When this
device's interrupt fires, it will send a message
to the listening thread. */
watchDog = new
WatchDogTimer(applThread);
 devMgr.addDevice(watchDog);

 /* Since there may be numerous interrupts in
the system, we have implemented an
FIQHandler interface which invokes the
serviceInterrupt() of a registered handler*/
fiqHandler.addHandler(watchDog,14);

/* When all devices are added to the
DeviceManager, it's init() method */
        devMgr.init();

/* We want to build a network appliance so
let's initialize one. */
related classes
        try {
         NetworkStack.initialize();
        } catch(NetException e) {
          Debug.println(Debug.LEVEL_1,
"Boot: NetworkStack init failed");
        }
```

```
    }

/* This is the addSA285MemPartitions()
object we called a little earlier to lay out the
logical partitions in the MemoryMap  We
assume every system has memory map in
which a number of MemoryPartions exist.*/
private void addSA285MemPartitions(){
MemoryMap mp =
SystemResource.getMemoryMap();
/* Each the classes named with the Space
header are extensions of the MemoryPartition
class. Instances of MemoryPartitions take a
MemoryMap object in the constructor.*/
sdram = new SpaceSDRAM(mp);
csr = new SpaceCSR(mp);
pciType0 = new SpacePciType0(mp);
pciType1 = new SpacePciType1(mp);
pciAck = new SpacePciACK(mp);
pciMemory = new
SpacePciMemory(mp);
pciIO = new SpacePciIO(mp);

/* Once all the memory partitions are defined
they are added to the memory map */
mp.addGroupPartition(sdram);
mp.addGroupPartition(csr);
mp.addGroupPartition(pciType0);
mp.addGroupPartition(pciType1);
mp.addGroupPartition(pciAck);
mp.addGroupPartition(pciMemory);
mp.addGroupPartition(pciIO);
}

private SpaceSDRAM sdram;
private SpaceCSR csr;
private SpacePciType0 pciType0;
private SpacePciType1 pciType1;
private SpacePciACK  pciAck;
private SpacePciMemory  pciMemory;
private SpacePciIO pciIO;
}
```

Summary of Synthesized System

- Hardware components are characterized by Java class files
- Runtime components which utilize hardware are characterized by Java class files
- System bring up consists of three "levels" described as boot threads:
  - **Boot** : Vector Table, Trap Handler, System Resource, and Supervisor thread.
  - **ProcessorBoot**: Cache, Cache Ctrl, MMU, Security class which extends Boot.
  - **PlatformBoot**: used to add SystemResources, create Logical Partitions, instantiate fundamental system drivers, device drivers, and kick off application threads. Extends ProcessorBoot.
- Boot, ProcessorBoot, and PlatformBoot relate to core CPU definition and system initialization, microprocessor definition and system initialization, and motherboard definition and system initialization, respectively.
- Threads may be kicked off in the boot levels to run supervisor and user applications.
- Embedded systems may be designed top down from by adding threads and devices into a PlatformBoot.java main()
- Designs built on the Websprocket packages, compiled a with standard Java compiler using the J2ME,

J2ME/J2SE Community Source Derivatives, or

- J2ME, J2ME/J2SE Community source and JemINI™ Java based designs are synthesizable by VMFoundry™, and require no JVM or operating system. The designer however may design a custom networked operating system in Java which can be synthesized.

- Synthesized systems work with VMServer™, the proxy Java Virtual Machine server, and can classes, threads, and methods from a remote server.

Java Software Abstraction of Physical Systems

package WebSprocket.Kernel

The WebSprocket.Kernel package includes several class definitions required for writing a Java threaded device driver for a synthesized system. Several classes which we will discuss are:

MemoryMap.java
MemoryPartition.java
MemoryObj.java
Device.java
DeviceManager.java
InterruptHandler.java
UsrMessage.java
logOS.java
taskMaster.java

MemoryMap, MemoryPartition

The basis for a programmers model of a programmable platform begins with a definition of physical decodes of hardware of a system. Said decodes definition is commonly referred to as a "memory map". The memory map is needed even for subtle hardware registers on the microprocessor which are not commonly used by the system programmer. The WebSprocket.Kernel package includes a MemoryMap class which defines a programmer's model for accessing a "memory map" at boot or during runtime. An architectural design rule for the Websprocket paradigm is that systems must have a memory map, and hence a MemoryMap object. For a given MemoryMap instance there must exist a collection of physical partitions where hardware can be logically decoded, or "mapped". The "mapped" partitions are defined as user defined instances of MemoryPartition.java which associate a group name with the address range of the physical decodes for said group. Hence, a system's memory map consists of a collection of physical partitions. Since a memory map and it's partitions are so fundamental to programming an the operation of a system, these objects must be instantiated during the boot sequence. By doing so, a system's logical partitions are available as SystemResource(s) (object references) for the program applications, such as device drivers, which need to access low level features of an embedded device. MemoryMap includes methods which are used to define logical decoded partitions, and other methods which return MemoryObj(s) for device driver memory requirements. An example of a MemoryPartition is as follows:

```
public class  SpaceCSR extends
MemoryPartition  {

 /* Every  logical partition in a synthesized
system must extend a MemoryPartition.  The
constructor of a partition should program the
following inherited fields*/
```

```
public SpaceCSR(MemoryMap mp){
/* Every logical partition must be assigned a
name.  This also allows the construction of a
device tree */
 groupName = "CSR";
/* This is the actual range of address space
assigned to “CSR” that is physically
decoded/assigned in the system */
      startAddress = 0x42000000;
      endAddress = 0x420FFFFF;
      size = 0x00100000;
/* this assign MUST be there */
      parent = mp;
      nextFreeAddress = startAddress;
    }
}
```

The MemoryMap instance should have been created in the initial boot level as was seen previously in Boot.java. The physical constants of SpaceCSR are defined by the hardware. In this case, "CSR" is an address range decoded by the SA110 microprocessor and is reserved for system hardware peripherals and logic. See:

http://developer.intel.com/design/strong/quicklist/eval-plat/sa-110.htm

Details regarding the SA285 development board memory map are available from the above.

MemoryObj.java

Device drivers require or implement two types of memory which should not be garbage collected. Certain devices require that a device’s on chip memory be physically mapped into the system. Low level accessors are needed to program a device’s on chip registers with values. These on chip registers must be assigned a MemoryObj reference, that is, they must be accessed with a MemoryObj low level accessor, such as read8(). Additionally, certain devices have on chip memory, but also need system memory for buffer space. In this case, a MemoryObj must be allocated and a reference given to the driver which needs the buffer area.

As an example, an ethernet controller has on chip registers which are used for controller setup, operating mode, etc. These registers are physically decoded via a bus. Additionally, an ethernet controller driver needs system memory for buffer area to transfer data from an on chip FIFO to system memory. The system memory buffer must be allocated and not be garbage collected. A device driver architect may implement by declaring MemoryObj(s) in the Device. Below is a snipet of code from WinBondEthernetDriver.java available from Websprocket’s website.

```
/* registers is a MemoryObj defining the on
chip registers of the Winbond ethernet
controller */
    private MemoryObj registers;
    int phyAddress;          // PHY address

/*  these are memory objects used for transmit
and receive descriptors */
    // memory for tx/rx descriptors
    private MemoryObj txDescriptorMemory;
    private MemoryObj rxDescriptorMemory;

/*  these are memory objects used for transmit
and receive buffer area */
    // memory for tx/rx data buffers
    private MemoryObj txDataBufferMemory;
    private MemoryObj rxDataBufferMemory;
```

Therefore, device drivers implement MemoryObj for a device’s on chip resources (registers), and a device’s device driver buffer memory requirements. MemoryObj(s) will be discussed in further detail in the context of this reference.

Device.java, DeviceManager.java

Devices may be implemented using the device class. Device is an abstract class which contains fields for hierarchical device organization and device initialization. Every device driver which extends Device is required to implement Device's init() method. The init() method is called to initialize the states of hardware control of the device. A device's init() method is "automatically" called by DeviceManager's init(). A DeviceManager instance is used in a system as a convenient device repository. This class can be used for plug and play device management for example. A device can be added to a DeviceManager by invoking DeviceManager's addDevice(Device) and passing the device object. [If a DeviceManager instance exists in the system, it must have been assigned by one of the system's boot levels and set as a system resource with SystemResource.setDeviceManager(DeviceManager dm).] This is usually done in a PlatformBoot's main() because embedded system programmers may be adding devices to a development board which is shipped by a manufacturer. We will come back to DeviceManager in the context of a source code example for building a network appliance. Let's examine a simple interrupt driven, threaded device driver example via blue comments.

WatchDogTimer.java

```
/*  This device is part of a Platform package.
That is because it is a device implemented on
the SA285 "motherboard" a.k.a. Platform.
We will be referring to WebSprocket.Kernel
classes so we import WebSprocket.Kernel*/
package WebSprocket.Platform.SA285;
import WebSprocket.Kernel.*;

/*  We are building a simple threaded device
driver.  A device extends the
WebSprocket.Kernel Device class.  Interrupt
driven devices are required to implement the
InterruptHandler interface. */
 public class WatchDogTimer extends Device
implements InterruptHandler{

/*  We will first define the physical constants
for the system.  The int values below are for
the most part the relative addresses for timer
registers, or for timer interrupt enable and
disable.  The addresses are relative to */
        int FIQEnable  = 0x288;
        int IRQEnable  = 0x188;
        int TIMER2LOAD = 0x320;
        int TIMER2VALUE = 0x324;
        int TIMER2CONTROL = 0x328;
        int TIMER2CLEAR = 0x32c;

    // load for ~1ms  195 ticks
    int  SYS_TIMER_PERIOD = 0xf4240; //
earlier 0x1c3

    // enable timer, periodic, fclk_in/256 prescale
    int TIMER2_CONTROL_MASK = 0xC8;

/*  This timer device has on chip registers
which must be physically accessed.  We
declare a MemoryObj. */
   MemoryObj timerRegs;
/*  This timer device will send a real time
message to a listening thread.  When the
thread is signaled it will pre-empt the running
thread. */
   Thread listnerThread;
/* This device will generate a unique message
which can only be received by a listening
thread.  The message is used as a signal into a
waiting thread Q and  tells the schedular to
promote the waiting thread to running.*/
   UsrMessage msg;

/*  Threaded device driver's constructors
accept a thread which will be used by the
driver as the thread which will handle the
work required of the driver. */
WatchDogTimer(Thread listner){
     super();
     listnerThread = listner;
   }

   /*  Here's the device's init method.  This
will be called directly or by the
```

```
DeviceManager's init() per the user's system definition. */
    public boolean init(){

/*  The device gets a reference to this particular system's MemoryMap  */
MemoryMap memoryMap =
SystemResource.getMemoryMap();

/* Now get a MemoryObj reference for the timer registers that are physically mapped in a given range.  This range is usually fixed by the embedded system designer*/
timerRegs =
memoryMap.allocateMemObj("CSR",0x320,0x36C,0);

/*  Now that we have an instance of a MemoryObj for timer registers, we can use the write32 low level accessor to program timer registers */
timerRegs.write32(TIMER2LOAD,SYS_TIMER_PERIOD);

 // set prescalers, timer modes, and enables

timerRegs.write32(TIMER2CONTROL,TIMER2_CONTROL_MASK);
// enable the interrupt bit in the Enable register
timerRegs.write32(FIQEnable,0x20);

/*  When we create a unique message, we can define an integer code (255) which is used to identify the message.  Websprocket LLC has reserved 0x1000 – 0x3fff and recommends that the user not use values in this range.  A message should also be instantiated with the thread that will use the message*/
      msg = new UsrMessage(255, listnerThread);
      System.out.println("created a message");

/*  Messages can be made real time based on the policy of the processing entity or a supervisor thread.  When this device driver's interrupt occurs, we want it's listener thread to pre-empt the running thread. */
SystemResource.getTaskMaster().makeRTMessage(msg);
      System.out.println("made it RT");
      return true;
   }

/*  Any interrupt driven device must implement a getMask() method which returns the mask value for  masking and unmasking it's interrupt in the system interrupt controller. */
 public int getMask(){
      return 0x20;
}

/*  Our simple threaded handler.  When the interrupt occurs, serviceInterrupt() gets invoked.  It simply sends a message to the listening thread and clears out the interrupt.  The listener wakes and does most of the work.
*/
    public void serviceInterrupt(){
      listnerThread.deliverMessage(msg);
      timerRegs.write32(TIMER2CLEAR,0);
}
} /*  End of WatchDogTimer  */
```

The summary of the structure of a device written on the WebSprocket.Kernel device API is as follows:

- Devices extend Device.java and inherit an init() method which is called to initialize the device mode and physical registers.
- Interrupt driven devices must implement the InterruptHandler interface and override the serviceInterrupt() method with the code that will be executed when a physical interrupt occurs.
- A getMask() method must be implemented which returns a field which used to identify the interrupt source or to enable/disable this device's interrupt (bit) in the interrupt controller register.
- Physical constants of the device are defined as byte, integers, etc. These constants define an "offset address" from a base value associated with a MemoryPartition, or a value with which a register will be programmed.
- One or more MemoryObj(s) are declared for the purpose of providing a device with an object reference which can be used to access physical

registers or memory required by the device.

- A threaded device must be instantiated with a constructor which accepts the listener thread.
- Devices which are threaded should instantiate a relevant UsrMessage which will be used to signal a listener thread.
- If the device is threaded, it's serviceInterrupt() should invoke deliverMessage(UsrMessage um) on the listener thread and pass the relevant message.
- A Device is instantiated in the body of a PlatformBoot's main() method.
- A Device's serviceInterrupt() handler is registered by passing the device as a reference to the system's InterruptController. The system's interrupt controller can be used to implement a user policy for handling hardware interrupts using a single interrupt signal. In the case of ARM and StrongARM devices may share an interrupt signal (e.g. FIQ) and report which device asserted the FIQ via a flag (state) in an interrupt controller register.

InterruptController; InterruptHandler.java

Microprocessors usually have external interrupt signals which are used to interrupt processor program control flow. In the case of the Intel SA-285, all external interrupts assert the processors FIQ signal. External signals which transfer program control usually force the microprocessor to branch to a fixed location in physical memory. The fixed location usually contains an instruction or branch address to code which handles the exception. Most processors define a "vector table" which hard codes a branch address for an external interrupt. The FIQ signal, when asserted on a StrongARM, automatically "vectors" the processor to the instruction located at 0x1C. An InterruptController object provides a means to assign which physical signals cause the processor branch to which "vector" address. The ARM processor defines it's vector table as follows:

Reset	0x00000000
Undefined Instructions	0x00000004
Software Interrupt (SWI)	0x00000008
Prefetch Abort	0x0000000c
Data Abort	0x00000010
IRQ	0x00000018
FIQ	0x0000001c

InterruptController.java provides a mechanism for registering an interrupt service routine with a hardcoded vector. This is done by invoking InterruptController's registerHandler() method and passing an InterruptHandler and the vector for which it will be used. For example:

```
/* fiqHandler's serviceInterrupt() will be
called when an FIQ interrupt occurs. (it is
presumed that an intrCtrl reference exists as
a SystemResource */
fiqHandler = new FIQHandler();
intrCtrl.registerHandler(fiqHandler,0x1c
);
```

When the FIQ interrupt occurs, the trap handler must read an interrupt controller register to determine which external physical component was responsible for the assertion. FIQHandler above implements the InterruptHandler interface which provides a serviceInterrupt() method that can be invoked when a hardware event occurs. In the case of the SA-285, an ASIC is used to OR external device interrupt

request signals. The OR'ed signal is then used to assert the microprocessor FIQ. It is the FIQ handler's job to read the ASIC registers and determine which interrupt is requesting service. A system designer may decide to utilize a varying service schemes and priorities assignments. Therefore, a designer may implement an interrupt handling policy and prioritization on a class which implements InterruptHandler and overrides it's serviceInterrupt() method, then binding said class to an actual physical interrupt source. Let's look at an example implementation for an FIQ handler for the SA-285 board.

FIQHandler.java

```
package WebSprocket.Platform.SA285;
import WebSprocket.Kernel.*;

/* FIQHandler implements the
InterruptHandler interface and inherits it's
serviceInterrupt() method */
public class FIQHandler implements
InterruptHandler{

/* Since there will be multiple sources for
interrupts a range of "priorities" is defined. */
private static int MAX_PRIORITY = 15;
private static int MIN_PRIORITY = 0;


/* FIQSTATUS is the physical offset address
of the interrupt controller status register. */
private static final int FIQSTATUS =
0x280;


/* There is one signal for FIQ interrupts but
there are multiple sources which assert FIQ.
Each source is a unique device with a which
will have a unique driver. Each driver will
implement the InterruptHandler interface to
implement a serviceInterrupt() method for an
interrupt. An array of InterruptHandlers will
be used to register an Device's interrupt
handler */
InterruptHandler fiqHandlers[];

/* An ASIC's interrupt controller registers
are physical and decoded in the system. As
such, they cannot be GC'ed . There must be
accessors which can read and write/mask the
interrupt controller register. We must define
non GC'ed physical resources of a system as a
MemoryObj for the Websprocket paradigm.
*/
private MemoryObj intrRegs;

/* In this example, we want to support up to
16 different handlers */
FIQHandler(){
fiqHandlers = new InterruptHandler[16];
  for(int i=0;i<fiqHandlers.length;i++){
        fiqHandlers[i] = null;
      }
/* The interrupt controller's physical registers
are mapped in the range of offsets of 0x220 to
0x280 on an SA-285 board. */
MemoryMap mp =
SystemResource.getMemoryMap();
intrRegs =
mp.allocateMemObj("CSR",0x220,0x28
0,0);
    }
/* Interrupt driven Devices can add
themselves to an FIQHandler instance by
invoking addHandler and passing their
reference and an index*/
public void addHandler(InterruptHandler
ih, int priority){
     fiqHandlers[priority] = ih;
   }

/* This method may be modified to enforce an
interrupt handling policy. In the example
below, when FIQ is asserted, FIQHandler's
serviceInterrupt() parses an array for a
handler. If the handler exists, it's getMask()
is invoked which returns a bitwise mask used
to determine if it is the source for the current
FIQ assertion. */
public void serviceInterrupt(){
/* When the FIQ comes, read the status */
int intStatus =
intrRegs.read32(FIQSTATUS);
/* Scan for a handler */
```

```
for(int i=MAX_PRIORITY;i>=
MIN_PRIORITY;i--){
        if(fiqHandlers[i]==null)continue;
/* Check to see if this is the handler for the
current interrupt source  */
        if((fiqHandlers[i].getMask() &
intStatus)!=0){
/* If it is, invoke the appropriate service  */
            fiqHandlers[i].serviceInterrupt();
            break;
          }
       }
     }
/* Don't need a mask for FIQHandler itself  */
public int getMask(){
 return 0; }

 }
```

UsrMessage.java; logOS.java

UsrMessage(s) are used in the Websprocket paradigm to provide an exact wait and notification, thread pre-emption scheme for Java. logOS.java is an object oriented "cell" which characterizes various system behaviors in the form of a Java library. Java's wait() and notify() provide a general purpose mechanism for waiting a thread, and notifying waiting threads, however, an invoke of notify() on an object has no deterministic guarantee of notification. Additionally, a programmer has no way of knowing exactly which thread of an object's wait queue is notified. A UsrMessage object when created can be used to directly signal a thread which is in a "waitingForMessage Q" in the runtime. A Thread is placed in a "waitingForMessage Q" by invoking waitForMessage(UsrMessage uM) on the thread. The previous threaded device driver example of "WatchDogTimer.java" was instantiated with a "listener thread" which could be signaled via passing a UsrMessage to said listener thread. It was presumed in the example that the listener thread "exists" in the runtime. We recall in PlatformBoot.java the design example:

```
/* Our appliance is going to have a threaded
device driver.  When an interrupt occurs for
the specified device, it will send a real time
message to a listener thread.  Priority
permitting, it will pre-empt the currently
running thread and service the interrupt.*/
UsrApplication applThread = new
UsrApplication();
applThread.start();
```

The above code, in the body of a PlatformBoot.java was used to create a live reference of a thread which would be a listener for a subsequently instantiated threaded device and driver.

```
/* Here is a listener thread example.  It places
the thread into a waiting for message Queue
by invoking the waitForMessage and passing
a simple message with a programmed integer
code field. */
public class UsrApplication extends
Thread{
/* Hope there is a lot of Mike's out there.*/
String outstring = "Mike is great";
public void run(){
UsrMessage msg = new
UsrMessage(255);
  while(true){
      try{
       sleep(10);
      }catch(InterruptedException e){}
/* Thread goes dormant  and will only get a
come alive if a deliverMessage is invoked on
this thread AND the message matches the msg
*/
       waitForMessage(msg);
      System.out.println(outstring);
     }
   }
 }
```

[waitForMessage(UsrMessage uM) and deliverMessage(UsrMessage uM) methods are provided by extending

Java's Thread with logOS.⌋ A message which is passed to waitForMessage is "stylized" by instantiating the message with constructor parameters that program the message's fields. Let's look back to the WatchDogTimer.java device driver example. The driver accepts a listening thread which will be signaled when an interrupt occurs. The listening thread, when started, must invoke a waitForMessage and pass a message which has been created with a code identical to the code of a signaling message. In this case, the driver creates a UsrMessage with an integer code, and programs it's messenger field with the thread reference that it intends to be signaled via this message. A message can be transformed into a real time message which can pre-empt other running threads. There are two methods in the Websprocket API which allow the creation of the real time message.

taskMaster.java

In the "WatchDogTimer" example, we have invoked the processing entity's makeRTMessage(UsrMessage uM) and passed the UsrMessage we want to make real-time. A processing entity is defined in by an binding an instance of taskMaster.java to a microprocessor. The existence of a microprocessor runtime object enables synthesized machines to be single processor or multi-processor and support concurrency. taskMaster's makeRTMessage can be be invoked only during a boot level as a default implementation but other policies may be implemented. In general, real time embedded systems would not want to allow any third party application to change the real time policy and behavior of said system. In the event that a new or interesting need arise, there is a logOS method which can be used instead of makeRTMessage.

Rapid Prototyping and Design

Much of the information contained herein is primarily for research and the technically curious. A network device hardware supplier normally "inherited" boot libraries from a Platform library which has already been created. In this case, an embedded network appliance can be built very quickly. Let's revisit the SA285 development board from Intel. The Websprocket Platform libraries for the SA285 appear in the WebSprocket.Platform.SA285 directory of a design hierarchy for the SA285 board. In the SA285 directory there exists an SA285Boot.java which contains the main(). Application threads can be added to main() which define the behavior of the embedded system.

Conclusion

Java programmers having access to the Websprocket tool chain may add threads and devices to an existing Platform. The Platform (motherboard) is described by Java source files in a design hierarchy. Bring up, boot, system initialization and definition are Java object instances. Hardware manufacturers may modify various levels of Boot, ProcessorBoot, and PlatformBoot providing a default system configuration for a "motherboard". In general, end application developers do not have to modify a Platform library. To build a network appliance, the application developer simply adds the devices and drivers which are relevant to the desired network device and add the threads of control which will exhibit the behavior of the appliance. You do this entirely in

Java from a single homogeneous design environment, adding application threads to the “PlatformBoot” main method.

The result appliance is a “Java object” which seamlessly interoperates with enterprise applications including the VMServer, a proxy Java Virtual Machine which can attend to 1000's of connected JVM-less clients . The design can be centrally revised, maintained, controlled, and upgraded because it is dynamic.

Performance

VMFoundry™ implements aggressive optimization policies which are a combination of traditional optimizations, and new optimizations made possible via synthesis. Synthesized code is very compact and efficient, and is particularly efficient for object oriented designs.

Optimizations

Memory

A VMFoundry™ provides a *classdicer* which automatically removes unused methods and classes from the target image, thus minimizing memory. Classdicing is particularly useful for network appliance or e-appliance designers. 50% code size reduction or more is possible. A Java webserver with serve-let support, Java file system support, user shell, line editor, and supervisor is compiled with the JemINI™ distribution (over 300 classes) and automatically reduced to ~600K bytes total code and data, and include TCP/IP, ethernet device driver. Considering J2ME's 67 basic libraries, much smaller footprints are achievable with J2ME.

The above example application runs directly on hardware, is fully threaded, classloadable from a proxy server and no additional software is required.

Custom subsets of the APIs can achieve a synthesized network appliance design in ~100K bytes.

Development

VMFoundry™ supports J2ME, J2ME/J2SE Community Source Derivatives, or JemINI™ Java libraries. Websprocket provides an IDE (WDKLite) for project definition, project management, compilation, synthesis, test, and target debug. WdkLite provides a variety of features useful for embedded system debug, such as breakpoint, single step, disassemble in addition to automatic network submission to VMFoundry for synthesis.

See the WdkLite data sheet for information.

Supported Platforms

VMFoundry™ is supported on a variety of popular platforms. Architecture targets currently supported are EBSA-285 StrongARM board, the IQ80310 xScale Development board, and the EVB80200 xscale board . Other ARM7, ARM9 and ARM10 are available upon request. However, since the Platform sources for the above mentioned are open source, it is a relative simple matter of creating libraries to support ARM family boards.

Network Appliance source example:

SA285Boot.java

```
/* A PlatformBoot is part of a
WebSprocket.Platform package. Platform
packages are a hierarchical means for
organization.  For example, the Intel "brutus
board" which is also a StrongARM based
system could appear in the Platform tree in a
Platform.Brutus package.  In the case below
it's the package for the SA285 Platform*/
package WebSprocket.Platform.SA285;
import WebSprocket.net.*;
import WebSprocket.Kernel.*;
import WebSprocket.Application.*;

/* A PlatformBoot extends a Processor Boot
The SA285 development board uses a
StrongARM processor, hence we are
extending "StrongArm*/
public class SA285Boot extends
StrongArm{

    /* We define an SA285Boot and an
   InterruptController for the runtime   This
bit of code was supplied.  You will need an
InterruptController object instance if you
intend to implement any interrupt driven
device drivers.*/
    public static SA285Boot sa285Boot;
    public InterruptController intrCtrl;

/* This is the veritable Java main program.
Those designing applications and systems
based on the SA-285 start here.  The first
thing main must do is create an SA285Boot
reference and kick off it's initialization
method.  This must be done to define the
runtime topology for the system.  For
example, busses, devices, memory partitions,
must be defined per the requirements of the
end design.  In this example, we will build a
simple network appliance which starts a
socket based TCP service on a listener port.
Also we'll build a threaded device driver, add
it to a DeviceManager.
 */
public static void main(String[]argv){

   /* Instantiate an SA285Boot and start it's
      initialization */
        sa285Boot = new SA285Boot();
        sa285Boot.SA285Init();
/* We are inside the main.  A top down
designer can add application threads here.  If
so desired, they can also kick off application
threads in the SA285Init() method itself.  This
example shows the latter.  The SA285Init()
method is pre-written canned code.  If it suits
your needs you don't have to change it.
Again, we'll add code to SA285Init() below so
main consists of just 2 lines of code.*/
}

/* The init method we just called from
main().*/
      public void SA285Init(){
/* My embedded appliance will use a serial
console, a pci peripheral (an ethernet
controller), a couple of interrupt driven
devices, drivers, etc.  Therefore, we need an
object for each.  We want to use
System.out.println so we need a console
instance.  A pure network appliance may not
want one.*/
        SerialConsole console;
        Pci pci;
        DeviceManager devMgr;
        FIQHandler fiqHandler;
        TimerDriver timer;
        WatchDogTimer watchDog;
/* now we add the MemoryPartions to the
system.  For example, SA-285 has physical
devices mapped to different logical partitions.
PCI, CSR, SDRAM, etc. must each be a
MemoryPartition object in the system . A "top
down" developer won't have to change this
method.  */
        addSA285MemPartitions();
/* Once we have the MemoryPartition(s)
added to the system, we can call the
StrongARM's mmu initialization method.
(This method was inherited from
StrongArm.java .  Top down designer doesn't
need to do anything here either because the
default memory management scheme was OK
for our network appliance*/
        StrongArmMMUInit();
/* We can also set a debug level for the
system.  This will be used as the "init" level
which catches debug specific traps (e.g.
breakpoints).  */
Debug.setDebugLevel(Debug.LEVEL_2
);
```

```
/* We are going to want to use
System.out.print a lot so we need a console.  It
needs to be a SystemResource.  SA285Boot
supplied in the Platform directory already
had this code .. we wanted it so we are leaving
it in*/
 console = new SerialConsole();
 SystemResource.setConsole(console);

/* Our embedded system would like to have a
repository for device management (like plug
an play).  We want it .. it was here .. we don't
have to do anything*/
 devMgr =
SystemResource.getDeviceManager();

/* We want to register our FIQHandler as the
default handler for ARM based FIQ
interrupts.  Ditto .. we want interrupt driven
devices so we need an mechanism for
handling interrupts.  Don't change a thing*/
 intrCtrl =
SystemResource.getIntrController();
fiqHandler = new FIQHandler();
intrCtrl.registerHandler(fiqHandler,0x1c
);

/* Our appliance is going to have a threaded
device driver.  When an interrupt occurs for
the specified device, it will send a real time
message to a listener thread.  Priority
permitting, it will pre-empt the currently
running thread and service the interrupt.
Here is the thread instance and start of the
thread we just described above.  So we
actually had to write this code*/
UsrApplication applThread = new
UsrApplication();
applThread.start();
/* PCI is treated like a device and is a
SystemResource .  The SA285 board is a PCI
board that plugs into a PCI bus .. we don't
change a thing here*/
        pci = new SA285_Pci();
        devMgr.addDevice(pci);
        SystemResource.setPci(pci);

/* A threaded device driver is instatiated by
passing a listener thread to it's constructor.
We just created the listener thread, so now we
create a device, and pass a listener.  When this
device's interrupt fires, it will send a message
to the listening thread. This is the device
driver that we wrote and reviewed above in
the WatchDogTimer example. So we had to
write WatchDogTimer.java and insert a few
lines of code here to add the driver*/
watchDog = new
WatchDogTimer(applThread);
 devMgr.addDevice(watchDog);

 /* Since there may be numerous interrupts in
the system, we have implemented an
FIQHandler interface which invokes the
serviceInterrupt() of a registered handler.
WatchDogTimer is interrupt driven so we use
the addHandler of FIQHandler and pass the
device instance*/
fiqHandler.addHandler(watchDog,14);

/* When all devices are added to the
DeviceManager, it's init() method .  Don't
have to do a thing here*/
        devMgr.init();

/* We want to build a network appliance so
let's initialize one.   OK .. we want a network
stack!  We call NetworkStack's initialize()
method to create one.*/
System.out.println("Initializing the
Network Stack");

try {
/* Ethernet Driver should be newed before
initializing network */
NetworkDriver nd = new
RTL8139EthernetDriver();

SystemResource.setLocalHostIntrfc(nd);
Router.addInterface(nd, "10.0.0.70");
Router.addDefaultRouter("10.0.0.1", 1);
NetworkStack.initialize(defaultIpAddres
s);
} catch (Exception e) {
                System.out.println("exception:
" + e.getMessage());
                e.printStackTrace();
            }

//START YOU APPLICATIONS HERE

Websprocket LLC
```

www.websprocket.com
408-530-0631
VMFoundry-info@websprocket.com