A Java Compatible Virtual Machine for Wireless Sensor Networks

Niels Brouwers

Delft University of Technology

CSIRO

A Java Compatible Virtual Machine for Wireless Sensor Networks

Master's Thesis in Computer Science

Embedded Software group
Faculty of Electrical Engineering, Mathmatics, and Computer Science
Delft University of Technology

Niels Brouwers

May 15, 2009

Author
Niels Brouwers
Title
A Java Compatible Virtual Machine for Wireless Sensor Networks
Msc presentation
May 29, 2009

Graduation committee

dr. K.G.Langendoen	Delft University of Technology
ir. dr. D. H. J. Epema	Delft University of Technology
dr. M. Pinzger	Delft University of Technology
dr. P. Corke	CSIRO

Preface

When I decided to go back to University, almost six years ago, I promised myself I'd do it properly this time and find a 'cool' thesis project to work on. I consider myself privileged to have been able to work on something that combines two long-standing personal interests, virtual machine design, and embedded systems. The project took much longer than it probably should have, but I enjoyed every moment of it.

I would like to thank my supervisor at Delft University, Koen Langendoen, for his patience and guidance. His comments have greatly improved my technical writing skills and his insights more than once set me back on the right track.

I would also like to thank my supervisor at the CSIRO, Peter Corke, for having me during the first seven months of my research. He has made it possible for me to write my first publication as a first author, and even provided a proper sleeping mattress when my back was aching from the one my land lady gave me.

Both my supervisors have given me the freedom to develop my research according to my own views, even when they might not have believed it could be done. They have given me a lot of support during the formation of my final thesis and even co-authored two publications, one of which is the basis for the work you are reading now.

I would of course like to thank my mother who always was there to re-fill the war chest when I was once again out of money, my friends who had to hear me rant about garbage collection schemes and byte code analysis frameworks, and finally I would like to thank my girlfriend who had to put up with me leaving her for seven months. I love you more than you know.

Niels Brouwers
May 15, 2009

Abstract

Writing software for Wireless Sensor Networks (WSN) is hard, as programmers have to write robust, distributed, highly concurrent applications on extremely resource limited devices. Virtual machines offer among other things support for high-level object-oriented languages, dynamic memory management and protection, hardware abstraction, and efficient code distribution. The main challenge is to ensure good programming tools and a minimal footprint for the virtual machine to match the limited amounts of memory available on typical WSN platforms.

This thesis describes the design and implementation of Darjeeling, a virtual machine modelled after the Java VM and capable of executing a substantial subset of the Java language, but designed specifically to run on 8- and 16-bit microcontrollers with 2-10kB of RAM.

The Darjeeling VM uses a 16- rather than a 32-bit architecture, which is more efficient on the targeted platforms. Darjeeling features a novel memory organisation with strict separation of reference from non-reference types that eliminates the need for run-time type analysis in the underlying precise garbage collector. Darjeeling also includes a linked stack model that provides light-weight threads, compacting garbage collection, and synchronization.

The VM has been implemented on three different platforms, and was evaluated with micro benchmarks as well as a real-world monitoring application. The latter includes a pure Java implementation of the Collection Tree Protocol (CTP) conveniently programmed as a set of cooperating threads, and a reimplementation of an existing environmental monitoring application. The results show that Darjeeling is a viable solution for deploying large-scale, heterogeneous sensor networks.

Table of contents

1 INTRODUCTION

A Wireless Sensor Network (WSN) is a collection of inexpensive, low-power, spatially distributed computers, called 'nodes', communicating wirelessly through an ad-hoc network and cooperating towards some common goal, usually the aggregation of sensor information. Such networks can be used for a wide array of applications such as environmental monitoring, security, cattle management, and battlefield surveillance to name a few.

Experience shows that writing software for sensor networks is quite hard. Programmers have to write modular, distributed software, often with high levels of concurrency, on extremely resource-constrained devices. Operating in the intersection of these four problem domains with tools that offer little in the areas of debugging, error detection and recovery, and dynamic memory management makes the task even more daunting.

It has been proposed that virtual machines (VM) may help to alleviate these problems. Virtual machines are a well known and powerful means of abstracting underlying computer hardware from an application, allowing portability across platforms without recompilation. Virtual machines also allow the implementation in software of certain desirable features, such as memory protection, that the hardware does not provide.

The Java virtual machine is an attractive candidate for use in sensor networks because it provides a solid concurrency model, dynamic memory management with garbage collection, memory protection through type safety, and support for object-oriented software engineering. Additionally the Java language enjoys widespread popularity and familiarity among programmers.

Unfortunately virtual machines introduce a non-trivial overhead in both space and time. The run-time interpretation of instructions causes programs to run slower by at least an order of magnitude. The VM itself requires a certain fixed amount of memory to operate, and interpreted programs are generally more memory demanding due to limitations on memory layout.

The primary goal of this thesis is to examine the impact, both positive and negative, of using a high-level language (Java) for wireless sensor networks software development. A secondary goal is to improve on existing work regarding virtual machine design for constrained devices. Both these goals are approached by designing, implementing, and evaluating an experimental virtual machine.

This thesis describes Darjeeling, a virtual machine capable of executing a large subset of the Java language on micro controllers. Its key features are efficient multi threading using ad-hoc stack space allocation, a custom 16-bit architecture to increase performance and decrease stack space usage, a novel approach for precise compacting garbage collection, and support for on-the-fly loading and unloading of modules. A real-world application in the form of a well known many-to-one routing algorithm imlemented in Java is demonstrated and evaluated.

The remainder of this thesis is structured as follows. Some background on wireless sensor networks, virtual machines, and previous work is presented in Section 2, followed by the design consideration of the Darjeeling VM in Section 3. Next, the implementation details regarding the linking model, memory organisation, byte code analysis, and execution model are presented in

Section 4. Section 5 presents the micro-benchmark performance evaluation for the Darjeeling VM on three target platforms, and Section 6 presents the evaluation of a well-known routing protocol and the re-implementation of a real-world monitoring application that uses it. Conclusions are presented in Section 7.

The source code to Darjeeling, together with documentation and publications are made available at `http://darjeeling.sourceforge.net`.

2 BACKGROUND

The work presented in this thesis combines the fields of wireless sensor networks and virtual machine design. Section 2.1 provides some background information on WSNs, and Section 2.2 motivates the use of virtual machines in sensor networks. Section 2.3 discusses reported VMs relevant to WSNs and embedded devices. Finally conclusions are drawn in Section 2.4.

2.1 Wireless sensor networks

Wireless sensor networks have been deployed to adress a wide range of problems. At Great Duck Island in Maine, USA, a network was deployed to monitor the nesting behavior of sea birds [19]. This is an example typical for environmental monitoring applications where individual nodes gather local sensor data, nesting burrow usage in this case, and relay the information back to an internet gateway via an ad-hoc wireless network.

Other deployments such as the virtual fencing application described in [7] use mobile nodes. To control the grazing patterns of cows physical barriers (fences) are required to restrict their movement. To allow for a more dynamic, and in the long run potentially more cost-effective management, a sensor network was used to influence the movement of the animals. The individual cows were equipped with nodes that track their position using GPS and apply a negative stimulus in the form of an electric shock when some virtual line is crossed. The network also relays this tracking information back to the researchers for further study by behavioral scientists.

These and other applications require networks that are both low cost and low power. The low cost aspect is important because it allows for more sampling points in space, and therefore more data, at a given hardware budget. They should also be low power so as to maximise the network lifetime when a finite energy source (such as battery) is used. In line with these goals most sensor node hardware platforms use cheap, low-power micro controller units (MCU) [8, 20]. These devices are severely limited in terms of processing power and memory capacity, typically having between 2-10kB RAM. An example of a popular hardware platform is the TMote Sky [20], shown in Figure 2.1.

Communication between nodes happens via a wireless link, wich consumes a significant amount of energy when transmitting or receiving. Protocols have been developed that are optimised for extremely low power consumption at very low data rates. Media Access Control (MAC) layers use duty-cycling to reduce energy consumption due to idle listening [35, 36]. This means that the radio and MCU are in deep sleep most of the time and wake up only periodically to exchange information. The range of the on-board radios is such that most networks require multi-hop routing protocols. Because the spatial distribution and network topology is usually unknown these protocols must be able to establish links and routes ad-hoc.

With a data rate of a single measurement every few minutes most of the power consumption of a network is due to the deep sleep requirements of the MCU and radio, and the cost of packet

Figure 2.1: The TMote Sky sensor node platform

transmission and reception. The MCU is 'on' only a fraction of the time, so even though it consumes in the order of 10 times more power in this state the 'on time' is not a major contributor to the total energy consumption. This means that real-time considerations aside, execution speed is not a hard constraint when designing WSN software. This allows for a trade off of execution speed for the benefits of a virtual machine.

A WSN software stack is made up several components that operate seperately and concurrently. The communication module contains a MAC layer and one or more routing layers, on top of which run one or more applications. Even in the simplest scenario, that of environmental monitoring, two routing layers are required. First, sensor measurements must be trafficed efficiently to one or more so-called 'sink nodes' and second, code updates must be pushed out into the network for bug fixes or retasking. Both these tasks require separate applications to run on top of the routing protocols as well.

Various WSN middleware solutions address the need for concurrency in different ways. cooperative multi-threading model is used by Fleck OS (FOS) [7]. The drawback of this model however is that each thread requires stack space to be pre-allocated to accomodate the worst-case usage. When many threads are needed this memory overhead may become prohibitively large. TinyOS [16] uses an event-based concurrency model aided by an extension on the C programming language, and Contiki [9] proposes a form of stackless continuations called 'proto threads'. Neither of these models is very intuitive however and especially event-based programming has received much criticism for its complexity and opaqueness.

2.2 Motivation for virtual machines

Despite the resource limitations of the microcontrollers used in typical sensor nodes it is an attractive idea to run a Virtual Machine (VM) to take advantage of the portability and flexibility that it provides. A virtual machine abstracts the underlying platform, offering a standard programming interface across a range of target platforms. This is important in the WSN context as we are moving towards heterogeneous networks, by design (i.e. two-tier architectures), or simply as a consequence of long-running systems evolving over time (e.g., adding new nodes after some years deployment). The support for dynamic loading of VM byte code provides programmers with a great deal of flexibility as it allows for updating and extending a running application. Again, the long lifetime of sensor networks make this a crucial feature as bug fixing and application-level retasking are inevitable when applications are deployed for more than a few weeks.

Another type of benefit that a virtual machine provides is the ease of programming that comes with it. The Java Virtual Machine (JVM) for example provides programmers with the luxuries of an object oriented language, dynamic memory management and protection, threads, and exception handling, none of which are provided by the underlying hardware. In addition, the JVM provides a safe execution environment ensuring that errors are handled gracefully and cannot take the entire software stack down as in the case for execution on bare hardware. This enhances the robustness of the software executing in the field, and more importantly reduces the code development and testing efforts considerably. Java generally provides quicker deployment and increased maintainability over C/C++ [6]. This helps bring down the total cost of ownership (TCO) of a sensor network not only during initial development, but also post-deployment.

Virtual machines provide a means of overcoming the challenges of fault tolerance, cost and heterogeneity. Low-cost embedded systems often have no user interface and are deployed in remote or dangerous areas so must run autonomously throughout their complete lifetime, i.e. for several years. Microcontrollers lack advanced features such as a memory management unit and a single faulty process can potentially take down the entire software system. Virtual machines help to alleviate these problems by providing strong checking, memory management and error handling services that improve robustness and allow software faults to be handled appropriately before they become failures.

The benefits of virtual machines for sensor networks have been recognised by the research community [14, 15, 26, 28] and the general consensus is that the price to be paid, that is the loss in execution speed, is of minor importance for the majority of WSN applications performing simple monitoring tasks. However despite this, today most applications are still coded in low-level programming languages, like NesC for TinyOS. The reasons are, firstly, most implementations of virtual machines have compromised on the supported functionality to reduce the memory footprint so that it can run within a few kB of RAM provided by typical sensor node platforms. Secondly, the state-of-the-art virtual machines are all proprietary implementations, which burdens the acceptance by the research community as it hampers the integration of experimental sensors, protocols, and algorithms.

2.3 Existing virtual machines

Several VM implementations have been reported in the wireless sensor networks research community, and different VMs strike a different balance between flexibility, supported features, and resource (memory) usage. Some of the VMs are open source projects (e.g., Maté [15] and leJOS [29]), while others are completely proprietary (e.g., Sentilla [26] and Java Card [32]. In general we can distinguish two competing philosophies: application-specific vs. generic VMs, which we will discuss next.

Application-Specific Virtual Machines (ASVM) [16] are optimised for a specific problem domain and abstract common operations as instructions in a virtual machine. Programs tend to be very small, in the order of ten to a hundred bytes, which makes reprogramming nodes in a network very energy efficient. Examples of ASVMs are Maté [15] and VMSCRIPT [22]. VM* [14] is a Java VM project that advocates synthesis of virtual machines tailored for specific applications, and is in that sense a form of ASVM. It supports incremental linking to allow the VM to grow as new features are needed, which overcomes the limited flexibility associated with most ASVMs. Unfortunately this project is closed source.

The class of generic virtual machines for sensor networks provides execution of some higher-level programming language, usually Java or a subset. This approach provides greater flexibility, for example to support application-level retasking, at a cost of generally larger program sizes and more complex interpreters. In this context the Connected Limited Devices Configuration (CLDC) specification [30] is very relevant as it describes a minimal standard Java platform for small, resource-constrained devices that have some form of wireless connectivity. Although often understood as describing cell phones, lately the specification has been increasingly mentioned in the context of wireless sensor networks. The specification is for minimal hardware requirements defined as a 16- or 32-bit processor with 160kB to 512kB of memory available to the JVM, which in terms of memory is almost two orders of magnitude greater than what typical sensor nodes provide.

Sun Microsystems has introduced several technologies related to Java on embedded devices. The Squawk project [28] provides an open-source, CLDC compatible virtual machine, largely written in the Java language. The project introduces the concept of a *split VM architecture* where class loading and verification is done off-line, resulting in more compact executables and lower memory footprint. The Squawk VM runs on top of the Sun SPOT platform, and ARM based sensor node with 512kB of RAM and 4MB of flash. This makes Squawk too large for our targeted platforms.

The Java Card [32] virtual machine targets 16-bit microcontrollers with approximately 2kB of RAM. Although it does not support features such as multi threading or even garbage collection, it is significant for proposing a 16-bit architecture and modified instruction set to execute Java programs more efficiently on micro controller platforms.

A recent generic VM is the leJOS [29] project, which features an open-source JVM that can execute on memory constrained devices, and has been demonstrated for wireless sensor networks in [10]. Unfortunately it does not support garbage collection amongst other essential features, so it is not suitable for more complex applications.

2.4 Conclusions

Wireless sensor network software has to be robust and preferably modular, is highly concurrent, and runs on extremely resource-constrained devices. Middleware should help programmers achieve these goals. Existing operating systems allow for memory-efficient concurrency but fail to provide an easy, intuitive, and robust programming interface.

Virtual machines provide a host of features that increase ease of programming, modularity, and robustness. They provide an intuitive programming model that abstracts away the details of hardware implementations, thus dealing with the heterogeneity problem. The size of compiled programs also tends to be small, allowing for more efficient code updates. These VMs have to run on resource-limited devices however, so the overhead that they impose should not be prohibitively large.

Reported virtual machines are either ASVMs that severly limit the programmer's flexibility, closed source VMs that are not available to the research community, or severely limited in terms of features. As a result little is known about how using a Java virtual machine for instance affects the properties of a deployment in terms of power consumption, memory footprint, and ease of development. Although the benefits of virtual machines are clear and generally accepted the cost involved remains unquantified.

3 DESIGN

Having identified the void for an open-source feature-rich virtual machine for resource-poor platforms like sensor nodes, we have designed a new virtual machine called the Darjeeling Virtual Machine (DVM) that is similar to the JVM and can execute a considerable subset of the Java language. Our virtual machine is designed from the ground up for devices with small heaps and minimises memory consumption wherever possible.

3.1 Requirements

The primary design goal for Darjeeling is to provide execution of complex Java applications on a range of different sensor network architectures. This requires memory efficiency, portability, and some means to load and unload libraries and applications.

3.1.1 Memory efficiency

In order to allow execution of meaningful applications Darjeeling should be memory efficient, as the small heap sizes of our target platforms are the main factor in constraining program complexity.

Threads especially should have as little overhead as possible, so that preemptive threading can be used freely by application programmers. Since we also want to allow multiple applications (each comprising a number of threads) running concurrently on a single node there must be little per-application overhead.

3.1.2 Portability

A virtual machine is middleware that sits in between the operating system (if any) and running applications. Many such operating systems exist for wireless sensor networks that provide different concurrency models such as event-driven or thread oriented. There are also different MCUs to deal with, most notably the ATmega and MSP430 series, with different memory types and sizes, and different addressing schemes. Therefore Darjeeling must be portable across all these MCUs and operating systems.

3.1.3 Application loading

There are many cases in which it is useful to reprogram nodes after they have been deployed in the field. This can be to fix bugs, introduce new functionality, or completely retask the network. While some testbeds allow reprogramming directly through wired means it is usually more practical to support loading applications over the air.

A virtual machine can allow multiple applications to coexist safely on a single node. Replacing separate applications is much more efficient than reprogramming an entire image. Darjeeling must therefore allow the loading and unloading, and starting and stopping of applications without having to reset the node or otherwise affect its running state.

3.2 Motivation for a new VM

The Java Virtual Machine (JVM) [17] is designed around 32-bit architectures with typically megabytes of memory. Certain design decisions that make sense for such platforms might not be ideal in the context of sensor networks where common MCUs use either 8- or 16-bit architectures and memory is measured in kilobytes.

3.2.1 Linking model

The first issue we address is that of the Java class file format and the dynamic linking model. Java classes are linked dynamically with a per-class granularity. This allows for great flexibility, but comes at a significant cost. Java class files are generally large as they contain linking information in the form of string literals. A second problem with the dynamic loading technique is that it requires some linking information to be kept in RAM at run-time, introducing a non-trivial memory overhead for each loaded class.

Other embedded JVM efforts have implemented conversion tools that perform static linking between groups of class files to either reduce or completely remove the need for string literals in their respective file formats, greatly reducing code footprint and eliminating the per-class memory overhead [2, 14, 28, 31]. The trade off is the loss of reflection and some flexibility in the linking model. This seems reasonable, especially as it allows for an efficient implementation where the byte code and class definitions can be kept in flash memory. Therefore Darjeeling uses a tool called the *infuser* that performs static linking of groups of class files.

3.2.2 Stack width

In traditional Java VMs values are stored in 32-bit slots. This is true for the operand stack, local- and global variables, and inside objects. It allows for quick access on 32-bit architectures, but is impractical for memory-constrained 8- and 16-bit platforms. On these platforms references are typically 16 bits wide so storing them in 32-bit slots results in a 100% memory overhead. This applies also to smaller integer types such as `byte`, `boolean` and `short`.

The Java Card virtual machine [31] addresses the memory overheads for narrow hardware platforms by using a more suitable 16-bit slot width. This requires a modified instruction set because Java automatically widens smaller integer types to a 32-bit `int` and does not contain instructions to modify 16-bit values directly. The 16-bit architecture also requires byte code analysis to optimise `int` based arithmetic to `short` arithmetic where this is possible. We have chosen to follow Java Card and use a 16-bit architecture. Arithmetic expressions are optimised by the Infuser tool.

3.2.3 Garbage compaction

Frequent allocation and deallocation of objects causes the heap to become fragmented with holes that are too small to accommodate common allocation requests, essentially wasting space. This is highly undesirable on our memory-constrained target platforms, so a compacting garbage collection algorithm is required.

Compacting garbage collectors first mark all objects in use, then *slide* them to one side of the heap, eliminating any holes. After compaction, references to relocated objects have to be updated in the running state of the program. In order for this to work it must be possible to identify which elements of the running states are references, and which ones are not so that no integer values are modified by mistake. It is possible to include type information about class fields and global variables in the executable file. Types of values on the operand stack and in local variables, however, may change dynamically so some mechanism must be in place to determine the types of these elements at run time.

One option is to simply type the stack and local variables as the program runs. Additional type information is kept on the stack and book-keeping is done when slots are accessed with `push`, `pop`, `load`, and `store` primitives. This method, called `type tagging', is elegant in neither space nor time. An improvement is to do type analysis as the mark phase starts. The byte code is annotated post-compile with typing information called *stack maps* [1] at selected addresses. The VM can then infer the types of the stack elements at the current address in relatively few iterations from the closest precalculated state. This method is widely used but has the drawbacks that extra typing information must be added, which increases the code footprint, and that the type inference mechanism increases code complexity and reduces performance.

We have instead chosen an unconventional approach that strictly separates reference and non-reference types on the operand stack and in local variables. This solution comes at a cost of a single byte per activation record (see Section 4.3.2) and several extra instructions to manage the operand stack, but avoids any runtime handling of type information. We also separate references from non-references in objects, which allows for uniform treatment of stack and heap and simplifies the garbage collector even further.

4 IMPLEMENTATION

Above we have shown that meeting the portability, memory-efficiency, and application-level retasking goals required us to let go of the JVM specification, because they are not compatible with the capabilities of the targeted, resource-poor processing platforms. Consequently we have chosen an approach where we designed our Darjeeling VM (DVM) to be similar enough so as to allow execution of post-processed Java programs, but with unique features that contribute to a low memory footprint. In particular we adopted a 16-bit architecture with corresponding instruction set, a static linking model, a linked stack architecture for memory-efficient multi threading, and a novel memory layout that separates reference types from others simplifying garbage collection. In this section we detail the implementation aspects of the Darjeeling VM, which has been ported to three target platforms with different microcontrollers (ATmega128, MSP430), operating systems (TinyOS, Contiki, FOS), and radios (CC1000, CC2420, nRF905).

4.1 Linking model

The Darjeeling runtime, the virtual machine, is responsible for executing Java programs. These programs comprise components such as class- and method definitions, executable byte code blocks, and string literals. These building blocks, which we shall call *entities*, are traditionally stored in Java `.class` files.

In the Java world, individual `.class` files are treated much like small libraries that are loaded on demand at run time. Classes contain a so-called *constant pool* that contains linking information. Any entity that is referenced by the class has an entry in the constant pool. A `new` instruction for instance, carries an index into the constant pool where information about the class to be instantiated can be found.

Darjeeling uses a *split VM architecture* [28] where class loading, byte code verification, and transformation is done off-line by a tool called the *Infuser*. Multiple class files are linked statically into loadable modules called *infusions*, which cooperate to form running programs. Named references are replaced with a numbering scheme so that keeping a run-time constant pool in memory is no longer necessary. Infusions are typically libraries such as the base infusion containing the `java.lang` package, or applications such as the CTP routing protocol application discussed in Section 6. Infusions `flatten' a hierarchy of Java classes into lists of entities, and can import other infusions and reference the entities therein.

The process is shown in Figure 4.1. A series of Java source files is fed into a standard Java compiler producing a corresponding set of class files. These class files are then input to the Infuser tool along with one or more infusion header files. A call to the infuser typically produces two files: a Darjeeling Infusion (`.di`) file, and a Darjeeling Infusion Header file (`.dih`). Informally speaking, the infusion file contains the class definitions and byte code, and the headers contain linking information. Together these two files form the infusion.

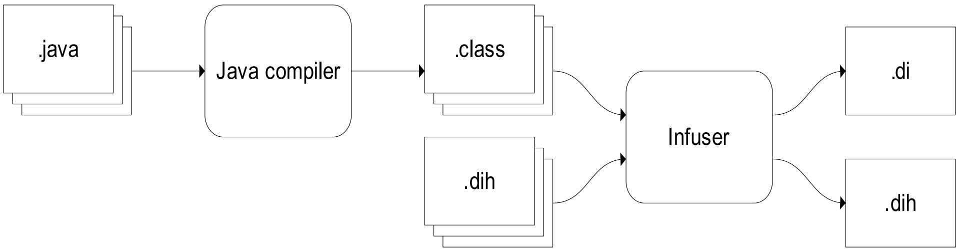


Figure 4.1: Infusion process.

The identifiers that are found in the `.di` files are called *local IDs* and consist of two parts, a *local infusion ID* and an *entity ID*. The first element refers to an item in the import list of an infusion. The second element refers to an entity within that imported infusion. Local IDs are stored as a two-byte tuple. Figure 4.2 shows how a method is resolved at runtime. In this example a method inside the `motor' infusion is called from the `car' infusion. First, the local ID is partially resolved into a *global ID* by looking up the infusion in the import list. A global ID is a tuple of a pointer to a loaded infusion, and an entity ID. The method itself can now be retrieved from the infusion's method list.

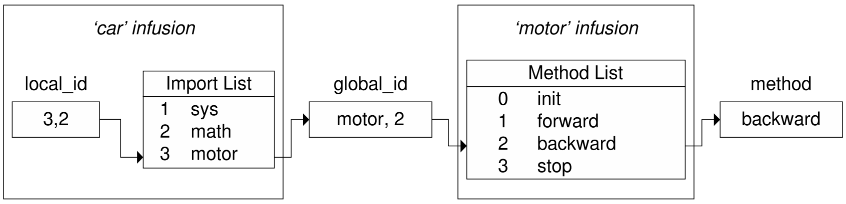


Figure 4.2: Local ID resolution.

4.2 The Infuser tool

The linking scheme described in the previous section, as well as our 16-bit architecture and strict separation of reference from non-reference types requires an off-line tool that transforms java `.class` files into infusions. The infuser is around 15,500 source lines of Java code, about two-thirds of the entire Darjeeling code base. It can be used from the command line or as an Ant (Java build system) task.

The Infuser reads `.class` and `.dih` files and creates an internal tree representing the different classes, methods and other entities. After the loading phase a number of processing steps are performed on this tree, each implemented using the visitor design pattern, ensuring code modularity and extensibility. Operations include a verification step to make sure no unsupported functionality is used, a linking step, byte code transformation (described more in depth in section 4.6), and finally output generation.

Byte code is represented internally as a linked list of *instruction handles*, each containing a single instruction. Handles have links to direct predecessors and successors and store inferred type information. An analysis framework is in place that performs type inference using an algorithm similar to the one described in [31], and can do live analysis on local variables.

The `.di` file format is organised as a flattened tree with relative pointers from parent to child nodes, so that the contents can be read easily from both program flash or ram. Each block has a type ID so that new block types can be added easily, making the format extensible.

4.3 Memory organisation

Memory management in Darjeeling was implemented with two goals in mind; memory efficiency and separation of types. Our linked stack architecture described in Section 4.3.1 enables threads with low fixed overhead. Sections 4.3.2 and 4.3.3 describe how reference and non-reference types are separated on the operand stack and in heap objects, respectively.

4.3.1 Linked stack

Each method call produces a new stack frame (activation record) as a context for the execution of that method. Java stack frames contain a local variable section, various bookkeeping values, and an operand stack. Bookkeeping includes a return address, stack pointer, and various context variables such as a pointer to the method that is being executed.

Traditionally stack frames are allocated on a single, pre-allocated space. The local variable section of a frame is located at the start of a frame, so that the frame of the caller can be overlapped with the callee. The last *n* active slots (parameters) on the operand stack of the caller overlap with the the first *n* slots (method arguments) of the callee. This allows for efficient passing of parameters between the two methods.

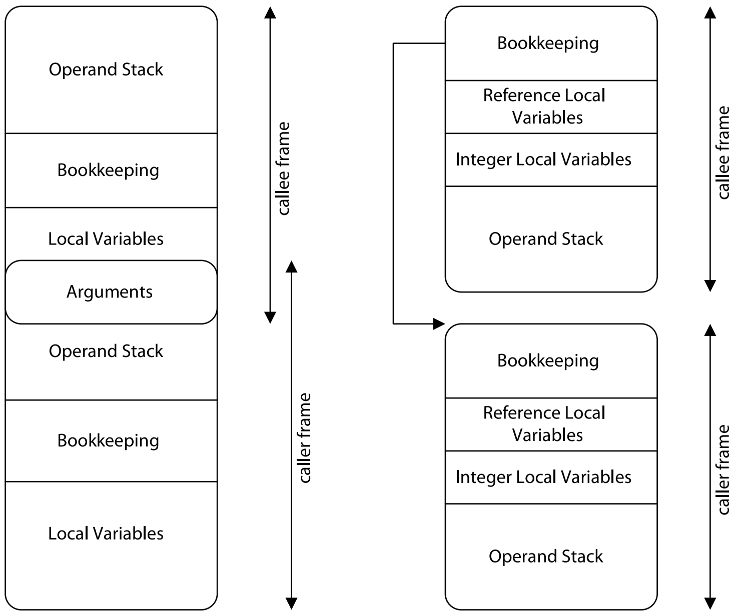
The drawback of the traditional method is that the size of this pre-allocated stack is equal to the worst-case requirement, introducing a severe memory overhead for each running thread. For Darjeeling we chose to implement *linked stack management* [3] where stack frames are allocated ad-hoc from the heap. This allows threads to grow and shrink as the program runs, allowing for light-weight threads. An obvious drawback is that each stack frame requires a heap chunk header, but we note that the benefits of dynamic stack allocation greatly outweigh this overhead (see Section 5).

A comparison between the two layouts is shown in Figure 4.3. A typical JVM stack frame organisation is shown on the left, our linked stack model is shown on the right. Our stack frames consist of the same elements, but the local variable section is split into two parts for reference and non-reference types to help with garbage collection and compaction (see below).

Because stack frames are allocated on the heap, and because the local variables of our stack frames are split into two separate sections, it is impossible to directly overlap the operand stack of the caller with the the local variables of the callee. One option is to copy these values from operand stack of the caller to the local variables of the callee, but this is not efficient because it would duplicate values and waste space. Instead, we added specialised instructions to let the caller access the operand stack of the callee directly for retrieving arguments.

4.3.2 Double-ended stack

Darjeeling uses two operand stacks instead of one. One of the stacks holds the reference types, the other is for non-reference types. The stacks are allocated within the same memory space, the size of which can be easily obtained by byte code analysis. Two stack pointers are used; one is initialised to the lower bound of the stack space, the other to the upper bound. The first grows upwards, the other downwards. This is shown in Figure 4.4. When the root set needs to be marked, the collector can simply traverse the reference stack and directly mark each heap object it encounters. This reduces the complexity of the root set marking phase to $O(n)$ and eliminates any



(a) JVM (b) DVM

Figure 4.3: Stack layout comparison.

false positives. During compaction, every time an object is moved, the collector can traverse all the reference stacks and update the pointers in place.

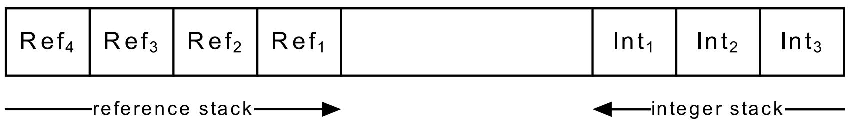


Figure 4.4: Double-ended stack.

The cost of this technique is that each stack frame now contains two stack pointers instead of one. In our stack frames these are stored using a single byte indicating the number of elements on each of the stacks, so that the overhead is just a single byte per stack frame.

In order for this technique to work some modification must be made to the byte code instruction set to replace stack manipulation instructions like `pop` with `ipop` and `apop` for integer and reference pop respectively. How this is accomplished is described in Section 4.6.

4.3.3 Objects

Strict separation of integer and reference types has also been applied to the layout of objects in memory. Figure 4.5 shows three objects allocated on the heap. The dotted line indicates the partition between the integer and reference fields. Darjeeling packs the integer fields of objects on the heap so that fields of type `byte` or `short` occupy only 1 or 2 bytes respectively instead of the standard 4.

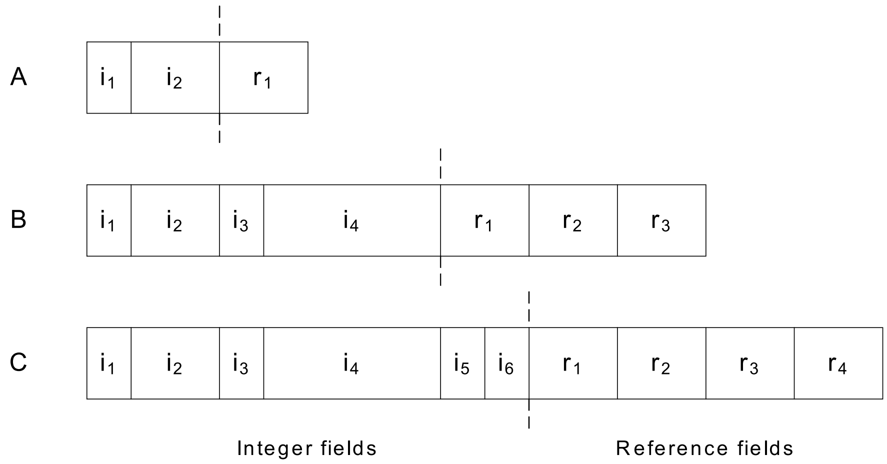


Figure 4.5: Object layout.

The instructions `getfield` and `setfield` have been replaced by the new instructions `getfield_<T>` and `setfield_<T>`, where *T* is one of `int`, `short`, `byte` or `ref`. The offset of the field inside the integer- or reference block is stored as an immediate value in the instruction. Since there are different `getfield` and `setfield` instructions for each data type the VM knows how many bytes to read at the given offset. Indexing into the reference fields is slightly more involved, as the offset of the reference field block must first be obtained. This is achieved by retrieving the size of the integer field block from the class definition which is stored in flash.

In our example *B* is a child class of *A*. It is always possible to substitute an instance of class *B* for an instance of its parent class *A*, as inheritance dictates, because the integer- and reference blocks are handled separately.

4.4 Execution

One of the benefits of Java programming over C-based middleware is that it provides an intuitive, preemptive multithreading concurrency model. Darjeeling implements preemptive multithreading in a very simple and straightforward way, allowing it to run on top of event-based, thread-based, and protothread-based concurrency models and even on systems that do not provide concurrency at all. In essence the running virtual machine is a polling loop that alternates between a `dj_vm_schedule()` and `dj_exec_run()` call. The former decides which thread should be run next, and the latter executes the specified number of atomic JVM instructions.

The time-slicing is not timer-based, although it is possible to call `dj_exec_run()` with a large number of instructions to execute and call `dj_exec_breakExecution()` on a timer to interrupt the thread after a fixed interval. Listing 4.1 shows how Darjeeling runs on top of FOS. The `dj_vm_getSleepTime()` method returns the number of milliseconds until one of the threads has to be woken up. If there are one or more threads currently in the running state this method returns zero.

In case of the TinyOS port, if there are no threads to schedule immediately, a single-shot timer is used to renew the virtual machine task in the future. On FOS we use the API call

```
while (dj_vm_countLiveThreads(vm)>0)
{
        dj_vm_schedule(vm);
        if (vm->currentThread!=NULL) dj_exec_run(RUNSIZE);
        sleep = dj_vm_getSleepTime(vm);
        if (sleep==0)
                fos_thread_yield();
        else
                fos_thread_sleep(sleep);
}
```

Listing 4.1: The Darjeeling main loop on FOS.

```
// javax.fos.Leds.setLed()
void javax_fos_Leds_void_setLed_byte_bool()
{
        // pop arguments off the stack
        // in reverse order
        int16_t on = dj_exec_stackPopShort();
        int16_t id = dj_exec_stackPopShort();
        // set the appropriate leds
        if (id==0) fos_leds_blue(on);
        if (id==1) fos_leds_green(on);
        if (id==2) fos_leds_red(on);
}
```

Listing 4.2: A native method implementation.

`fos_thread_sleep()`. Both methods allow the underlying OS to put the MCU into low-power mode.

4.4.1 Native code

In order for Java programs to access the node hardware, native libraries, and the virtual machine state, some mechanism for calling native methods is required. The Java language provides a `native` keyword for methods to signal that the implementation of that method is implemented natively rather than in Java. The infuser tool can generate C stubs from these Java method declarations for the application programmer to implement. These are then linked into the final image and programmed into the node.

Listing 4.2 shows the native implementation of the `javax.fleck.Leds.setLed()` method. Parameter passing is done through the operand stack, in reverse order. A native method may optionally return a value by pushing it onto the operand stack.

Some interaction with native hardware requires a Java thread to block until some event has occurred. A common example is waiting for a message from the MAC layer. On FOS the MAC `send()` and `receive()` methods simply block the calling thread. TinyOS provides the SplitControl mechanism that generates an event when the send operation has been completed, and receive is an event that may occur at any time.

```java
private static Object receiveLock = new Object();

// blocks until a message arrives
public static native byte[] _receive();

public static byte[] receive()
{
        synchronized(receiveLock)
        {
                return _receive();
        }
}
```

Listing 4.3: Locking on the Java side.

We expose these mechanisms using two stage locking. On the Java side a `synchronized` block is used to ensure that only one thread at a time may be blocked for an event from the underlying OS. If multiple threads call the same method they will be blocked waiting for the first thread rather than for the OS event. Listing 4.3 shows how this is implemented for the `receive()` method on the MAC layer.

The `receiveLock` monitor prevents multiple Java threads trying to process the same radio packet when a receive event occurs. A thread calling the native method `_receive` will be blocked immediately and another thread is scheduled for execution. When a receive event occurs the previously blocked thread is reactivated.

4.5 Infusion management

Applications and libraries executed by Darjeeling are stored as infusions. Infusion files (.di) typically reside in program flash. When an infusion is loaded by the VM it will execute its class initialisers, and if the infusion has an entry point method, it will create a new thread and execute that method in the new thread. Although in traditional JVMs the class loading is lazy and determines the order in which class initialisers are run, Darjeeling runs them in the order in which they appear in the file.

While loading new infusions is fairly trivial, unloading them is more involved since the VM must be left in a correct state after the infusion has been removed. Any references to elements from the infusion to be unloaded in the running state of the various threads may cause faulty behaviour when those elements can no longer be accessed.

Before an infusion may be unloaded any threads that are executing method implementations from that infusion must first be killed. This still leaves objects on the heap that are instances of classes defined in the unloaded infusion. These objects cannot be kept as they can no longer be accessed, their access methods have been unloaded, but they also can not be deallocated because there might be references to them from running threads. For example, when an application, passes an event handler to a routing protocol and later gets unloaded, the event handler object persists because the routing protocol still holds a reference to it.

We considered killing all threads that have direct or indirect references to such `bad' objects but decided against it because it would not only involve implementing another marking algorithm, but also because in our example it would cause the routing protocol to be killed as well. We chose a solution that allows the application programmer to decide how to deal with such cases.

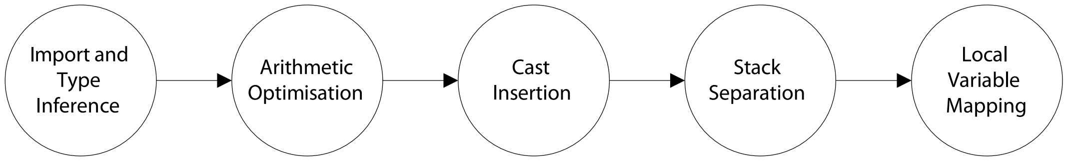


Figure 4.6: byte code transformation pipeline.

Instances of classes that are unloaded are marked `invalid' by the unloading mechanism. Accessing these objects causes the VM to throw a `ClassUnloadedException`. This exception can be caught, allowing the application to remove the reference. If it is not caught the thread will terminate due to an uncaught exception. Either way, the VM is left in a correct and predictable state.

4.6 Byte code transformation

Darjeeling has a custom byte code instruction set that is optimised for 16-bit arithmetic, supports a double-ended stack architecture, and has typed versions of field get and set methods. It is the job of the infuser tool to map Java byte code to Darjeeling byte code. There is not always a one-to-one mapping between instructions and the transformation may be context sensitive. In these cases the generated Darjeeling instruction depends on the input types of the source Java instruction. To allow for these context sensitive transformations type inference has to be performed to calculate the input types for each instruction. The transformation pipeline is shown in Figure 4.6. The next sections discuss its various stages.

4.6.1 Import and type inference

First the Java byte code is imported and translated into corresponding Darjeeling byte code. Any instructions that cannot be translated right away, such as stack manipulation operations, are replaced with place holders.

A type inference algorithm determines the states of the local variables and operand stack before and after each instruction. These states are called a handle's pre- and post state. The post state can be calculated from the pre state by simulating the effect the instruction has on the operand stack and local variables. The pre state of an instruction handle can be calculated by merging the post states of all its incoming handles.

Usually type inference is used for byte code verification and only the types of values on the operand stack are inferred. The infuser instead tracks which instruction handle produced each value and infers the type from there. This allows for multi-pass optimisation with type information being updated automatically when instructions are replaced.

```
public static short max(short a, short b)
{
        return a>b?a:b;
}
```

Listing 4.4: The max() function.

pc	in	out	instruction	pre stack	post stack
0		1	iload_0		short
1	0	2	iload_1	short	short, short
2	1	3, 5	if_icmple(5)	short, short	
3	2	4	iload_0		short
4	3	6	goto(6)	short	short
5	2	6	iload_1		short
6	4, 5		ireturn	short	

Table 4.1: Java byte code for max().

pc	in	out	instruction	pre stack	post stack
0		1	sload_0		short
1	0	2	sload_1	short	short, short
2	1	3, 5	if_icmple(5)	short, short	
3	2	4	sload_0		short
4	3	6	goto(6)	short	short
5	2	6	sload_1		short
6	4, 5		sreturn	short	

Table 4.2: Darjeeling byte code for max() after import

Consider Listing 4.4, a function from the `Math` class that calculates the maximum of two shorts. The corresponding annotated Java byte code is shown in Table 4.1. The first column shows the offset of each instruction, the set of incoming and outgoing offsets (direct predecessors and direct successors) are shown in columns two and three. The pre- and post-stack columns show the *logical* types of the operand stack. We distinguish between logical types, the `actual' type of the value, and the *physical* type. In the JVM `short` values are automatically widened to `int`, so although only shorts are used in our example their physical types are actually `int`.

The resulting Darjeeling byte code, after this phase, is shown in Table 4.2. The `iload` and `istore` instructions have been replaced by their short-sized counterparts, and since the method returns a value of type `short` the `ireturn` instruction has been replaced with `sreturn`. The `if_icmple` instruction, which compares two integer values, has not been optimised to its short form yet. This is done in the next phase.

4.6.2 Arithmetic optimisation

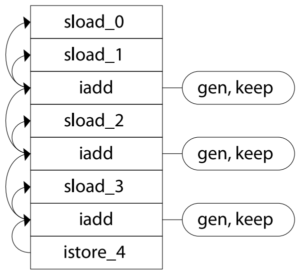
The JVM automatically widens small integer types to `int`, and all arithmetic is done using 32-bit intermediate values and instructions. To match the native integer size of most micro controller platforms Darjeeling can work with the `short` data type directly, only widening values of type `boolean` and `byte`. The infuser optimises Java byte code to use short-typed instructions where this is possible, but must be careful when optimising arithmetic to make sure that expression semantics are preserved. The arithmetic optimisation stage performs this transformation.

Many Java instructions that operate on `int` types have short-typed counterparts in the DVM instruction set. The `iadd` instruction pops two 32-bit integer values off the operand stack, adds them, and pushes the result back onto the stack. It is important to realise that the result is implicitly truncated to 32-bit. Darjeeling has a short-sized version called `sadd` that performs the same calculation, only using 16-bit values. Because `sadd` truncates the result it cannot always be substituted for `iadd` even if both input types are `short`, because the addition might generate overflow that is relevant for later computation.

Our algorithm starts by annotating the input byte code with two flags. The *gen* flag is set for every instruction handle that might generate overflow. The *keep* flag is set for instructions whose overflow should be preserved. An instruction that has both flags set cannot be safely optimised. The keep flag is generated by looking for instructions that require overflow to be preserved for one or more of their input values, and setting the keep flag on the instruction(s) that generate them.

```java
short a, b, c, d;
int e = a + b + c + d;
```

(a) Java source.



(b) Annotated byte code.

Figure 4.7: Addition in Java and corresponding byte code.

Consider the Java fragment in Listing 4.7(a). The corresponding annotated byte code is shown in Figure 4.7(b). All the addition instructions have the gen flag set because they take short values as input, and this can generate overflow. The backwards pointing arrows show which instructions cause the gen flag to be set on which of their inputs. The `istore_4` instruction for instance requires any overflow generated by the last `iadd` to be preserved. In this state the algorithm is done immediately because there are no instructions that can be optimised.

```java
short a, b, c, d;
short e = (short)(a + b + c + d);
```

Figure 4.8: Truncated addition.

Now consider a slightly altered version of the addition chain as shown in Listing 4.8. In this case the result of the addition is explicitly casted to `short`. The corresponding code is shown in Figure 4.9(a). Because the `i2s` instruction removes any overflow from the last `iadd` it no longer has the keep flag set. The algorithm can now safely replace it with an `sadd` because both input types are short and overflow is irrelevant.

Whenever an instruction is optimised the change may affect the keep flag on its inputs and gen flags on the instructions that use its output. The result of optimising the `iadd` instruction is shown in Figure 4.9(b). By applying this mechanism iteratively the entire expression can be optimised to use only short instructions, as shown in Figure 4.9(c) and Figure 4.9(d).

4.6.3 Cast insertion

The arithmetic optimisation stage cares only about logical types of values, not about the physical types in which they are wrapped. Recall the example in Figure 4.7(b), where the `iadd` instructions

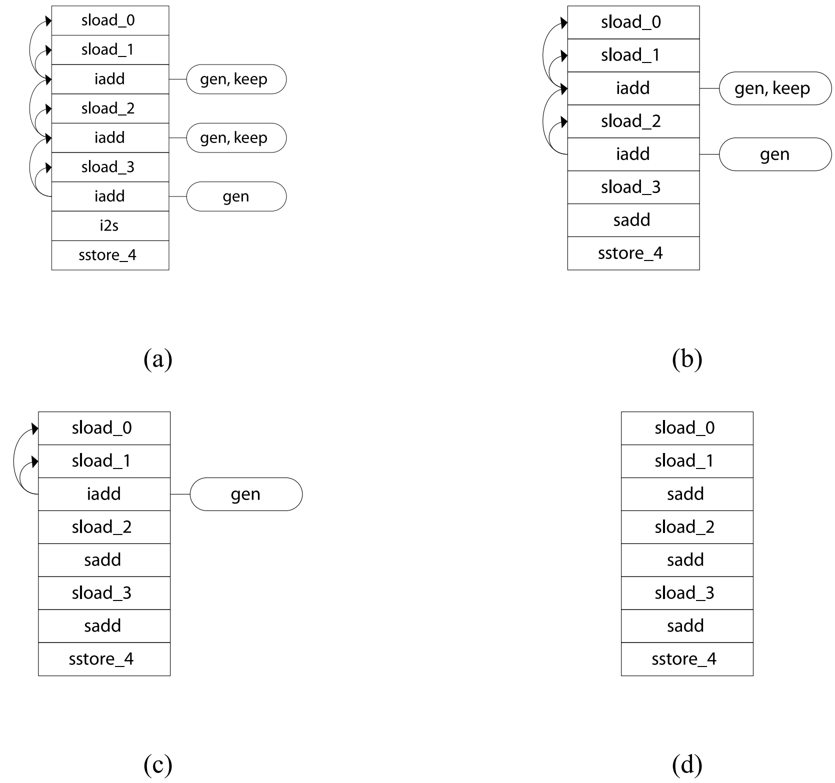


Figure 4.9: Arithmetic optimisation for the truncated addtion example.

require `int` typed inputs. The `sload` instructions produce short values on the 16-bit stack, so these must be widened to 32-bit `int`s using an `s2i` instruction. The byte code listings before and after the transformation are shown in Figure 4.10(a) and 4.10(a), respectively.

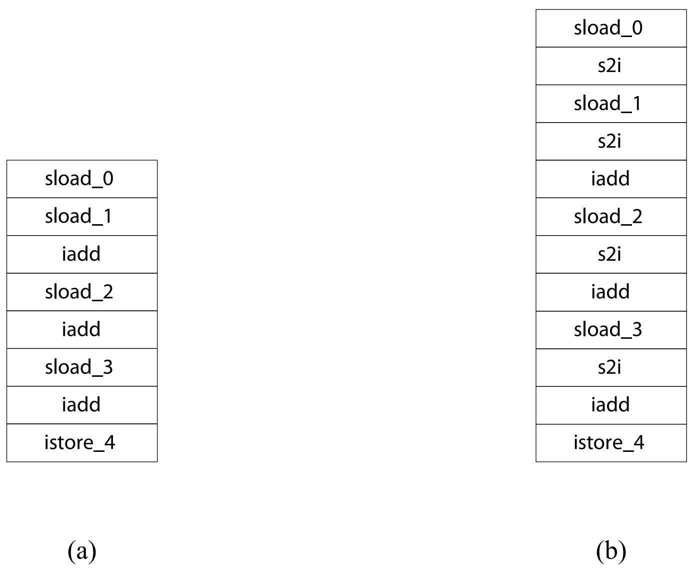


Figure 4.10: Cast insertion.

Unified	Reference stack	Integer stack
v_2:short, v_1:short	...	v_2:short, v_1:short
v_2:short, v_1:ref	v_1:ref	v_2:short
v_2:ref, v_1:short	v_2:ref	v_1:short
v_2:ref, v_1:ref	v_2:ref, v_1:ref	...

Table 4.3: Stack contents at dup_x1.

Unified	DVM instruction
v_2:short, v_1:short	`idup_x1`
v_2:short, v_1:ref	`adup`
v_2:ref, v_1:short	`idup`
v_2:ref, v_1:ref	`adup_x1`

Table 4.4: Replacement instructions dup_x1.

4.6.4 Stack Separation

As discussed in sections 3.2.3 and 4.3.2, Darjeeling uses a double-ended operand stack to eliminate the need for run-time type analysis. Logically speaking there are two stacks, one for integers, and one reference types. For most instructions it is immediately clear which stacks they mutate, but for some instructions this is context sensitive and type analysis is required for correct translation.

We illustrate the context sensitivity for the `dup_x1` instruction, whose effect is to 'duplicate the top element on the stack and insert it one place down'. This can be written as $..., v_2, v_1 \rightarrow ..., v_1, v_2, v_1$. The correct replacement for `dup_x1` depends on the values of v_1 and v_2 . These can be any of `short`, `int`, or `ref`, yielding 9 different possible combinations. In the interest of simplicity we shall omit the `int` type, but it should be clear that the case can be easily extended to include other data types.

The possible combinations of input types for the instruction are shown in Table 4.3. The three columns show the contents of the unified (Java) stack, the reference stack, and the integer stack respectively. From this table we can see how the transformation $..., v_2, v_1 \rightarrow ..., v_1, v_2, v_2$ would apply to each case. Table 4.4 shows the correct replacement instruction for `dup_x1` for each of the four cases.

If both v_1 and v_2 are of the same type, `dup_x1` can simply be replaced with a dup_x1 that operates on the appropriate stack, `idup_x1` for integers and `adup_x1` for references. If they are of different types, the top element and the element below it are located on different stacks, so the instruction can be replaced by an appropriate duplicate instruction.

4.6.5 Local Variable Mapping

Java compilers such as `javac` are usually not very efficient when it comes to assigning local variables to local variable slots because for most virtual machine implementations memory consumption due to stack space is not a major issue, and the assumption is made that frequently executed code will be jitted (compiled) anyway. Usually one or more slots are allocated for each local variable, leading to memory wastage when two or more variables can be mapped onto the same space. The infuser tool performs live range analysis to determine which variables can be mapped onto the same slot(s) and remaps the local variables accordingly. Darjeeling does not require the Java compiler to supply type information about local variables. Instead this information is inferred from byte code analysis so that even when a variable is declared `int` it may still be optimised to the `short` type.

4.7 Limitations

We have chosen to support a subset of the Java language. Like Java Card [31] we do not support floating point or 64-bit datatypes. We are looking at supporting the `long` type in the future to increase compatibility with CLDC 1.0.

The DVM also does not support reflection since the type information required is not stored in our executable file format. Support for the `synchronized` modifier on methods is not supported because there is no loaded class to synchronise on, although this is easily simulated using `synchronized` blocks.

5 BENCHMARKS

To assess the performance of the Darjeeling VM we have created a set of benchmark programs that exercise specific parts of the implementation in terms of execution speed and code size. Performance in a real-world environmental monitoring application is discussed in Section 6.

5.1 Performance

To get a feel for the execution overhead of our interpreter we constructed three micro benchmarks. First we ran a bubble sort algorithm that sorts 256 32-bit integer values, initialised to strictly decreasing values. The second test is an 8x8 vector convolution that is performed 10,000 times, also using 32-bit values. Finally we hand-optimised an implementation of the well known MD5 hashing algorithm, generating MD5 hashes for the empty string, `a', `abc', `darjeeling', and `message digest'. These hashes are generated 1,000 times.

Each of the benchmark tests has both a C and a Java version and is written in such a way that the difference between both versions is almost completely syntactical. We were careful to eliminate any memory allocation after the test initialisation to make sure the garbage collector would not be triggered and skew the results.

Test	C	Java	Java/C	Instructions/s
Bubblesort	0.3s	23.3s	77.7	60,618
Vector conv.	9.1s	496.7s	54.47	65,454
MD5	13.1s	399.7s	30.4	52,294

Table 5.1: Performance benchmarks.

We ran the tests on an ATMega128 microcontroller at 8MHz. The results are shown in Table 5.1. The C and Java columns show the run times for the bubble sort, vector convolution, and MD5 tests respectively, and the Java/C column shows their ratio. The final column shows how many Java instructions per second were executed during each of the tests.

In these particular tests the performance overhead of the Darjeeling VM was about 30-78x. The measured byte code instruction throughput was between 52-65k instructions per second. Note that although the VM was executing more instructions per second on the bubble sort test than on the MD5 test, the first is much slower compared to its native counterpart than the latter. We can explain this by looking at how VM instructions are interpreted.

Interpreting a VM instruction has a fetch, decode, and execute phase. While the fetch and decode components are more or less the same for each instruction, there are large time differences for the execution phase. Method invocation and type checking instructions take longer to execute than simple arithmetic, and instructions that operate on 32-bit values are generally slower than their 16-bit counterparts. During the MD5 test the interpreter is executing less, but more complex instructions per second. The bubble sort test uses more trivial instructions, causing a greater relative overhead due to fetching and decoding.

Test	C	Java	Java/C	Instructions/ss
Bubblesort	0.2s	19.3s	81.5	71,526
Vector conv.	4.6s	520.9s	113.2	71,512

Table 5.2: Performance benchmarks, 16-bit.

We illustrate this effect by re-running the bubble sort and vector convolution tests, but this time altered to only use 16-bit data types (the MD5 algorithm is a 32-bit algorithm so could not be rewritten). The results are shown in Table 5.2.

The interpreter has a higher instruction throughput in both tests due to the simpler instructions, but the fixed overhead makes it unable to take advantage of the `short`-typed arithmetic as much as the native C implementations. The 16-bit Java version of the vector convolution test has even become slightly slower compared to the 32-bit one, which is due to the infuser not being able to completely optimise the arithmetic and having to insert explicit conversion instructions between the `int` and `short` data types (see Section 4.6.2). As expected, the interpreter is performing much worse on these new tests compared to native C, with measured performance overheads of 82x and 113x for the bubble sort and vector convolution tests respectively.

These results illustrate that the performance overhead of Darjeeling is difficult to measure because it is very much dependent on the application under test. A high instruction throughput does not necessarily mean a good performance, especially when compared to equivalent native code. We found that the cost of interpreting Java versus executing native C code, in terms of execution time, ranges from a factor of 30-113x in our tests. This translates to roughly two orders of magnitude. Section 6.2 examines how this number affects the network life time in a real deployment, and shows that the impact is less than 1% due to the CPU only contributing a very small amount to the total energy consumption.

5.2 Code size

As discussed in Section 4.1, Darjeeling uses a split-VM architecture. This allows for considerably smaller executables. Table 5.3 compares the size of several infusions when stored as Java .jar files, which are zip compressed archives that contain .class files, and our .di file format which uses no compression but eliminates all string literals.

	jar	di	reduction
Blink	856	146	83%
CTP	27,349	7,889	71%
Test Suite	42,834	19,023	55%
Base library	14,795	3,591	75%

Table 5.3: File format comparison (sizes in bytes).

Size reduction is related to the ratio of string literals versus byte code in the input .class files. The base library for instance is mostly made up of class and interface definitions with little to no actual

	JVM	DVM	Reduction
Blink	10	6	40%
CTP	11.4	6.3	45%
Test Suite	13.7	9.3	32%
Base library	7.1	4.1	42%

Table 5.4: Local variables and operand stack size (bytes).

	JVM	DVM	Reduction
Blink	18	14	22%
CTP	19.4	14.3	27%
Test Suite	21.7	17.3	20%
Base library	15.1	12.1	20%

Table 5.5: Total stack frame size (bytes).

byte code, which accounts for the significant reduction in code size (75%). The same holds for the exemplary TinyOS application, blink, which only has a single method implementation. The CTP routing protocol library (see Section 6), is also reduced considerably in size because most methods are short, just a few lines of code. This was done to improve modularity and readability in line with good software engineering practice. In contrast the test suite has relatively long methods each performing a series of short tests, which is why it was reduced less than the others.

The conclusion is that although writing modular code with many small methods, classes, and interfaces will still add to the code size, the cost is not nearly as high as with the traditional .jar file format because Darjeeling does not retain the string literals in the infusion file.

5.3 Stack space

In order to measure the benefits of our 16-bit stack architecture in terms of memory usage during execution, we measured the average size of stack frames for methods in the four benchmark infusions. The size required for a single stack frame is the sum of the operand stack size, the size required for the local variables, and some fixed overhead for bookkeeping such as return address, stack pointers, and so forth. We calculated this number both for the 32-bit JVM case and our 16-bit DVM.

Table 5.4 shows the size of the variable part of the stack frame, the operand stack and local variables, for each of the test cases. During the development of the CTP library we used short types for loops counters and intermediate values wherever possible, resulting in a reduction of 45%. The test suite on the other hand uses integer types in many places, but still stack usage is reduced by 32%. This is mainly due to reference types being reduced from 32-bit to 16-bit. Table 5.5 shows the average *total* stack frame size, including the fixed overhead (8 bytes) due to the bookkeeping section. It shows that the total reduction is between 20-27% for our benchmark infusions.

5.4 Multithreading

Our linked stack architecture allocates stack space on demand, allowing threads to grow and shrink their memory usage as they run. To illustrate how this affects memory consumption we ran a garbage collection test from our test suite. It sorts 20 numbers by inserting them into a binary tree, causing not only a large number of small linked objects to be created, but also generating a lot of nested method calls. The test calls the algorithm three times, in three concurrent threads.

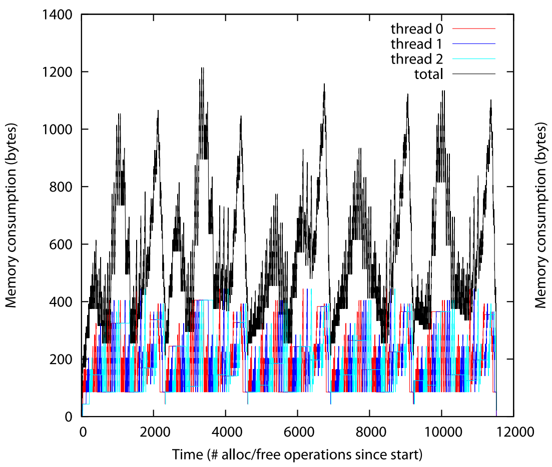
Figure 5.1(a) shows the amount of stack space for each thread as the test ran, including the thread object and heap manager headers. Each thread uses up to 445 bytes, which makes the total peak at 1335. This means that the worst-case total stack usage for this test is equal to the sum of the worst-case values of the individual threads.

We can improve on this performance by allowing only one of the threads to perform the tree sort algorithm at a time. We ran the test again, but this time we added a `synchronized` block around the call to the algorithm, essentially serialising individual runs. The trace for this second run is shown in Figure 5.1(b). This time the total peaks at a mere 535 bytes, which is the peak value of a single thread, 445 bytes, plus the overhead of the two blocked threads at 45 bytes each.

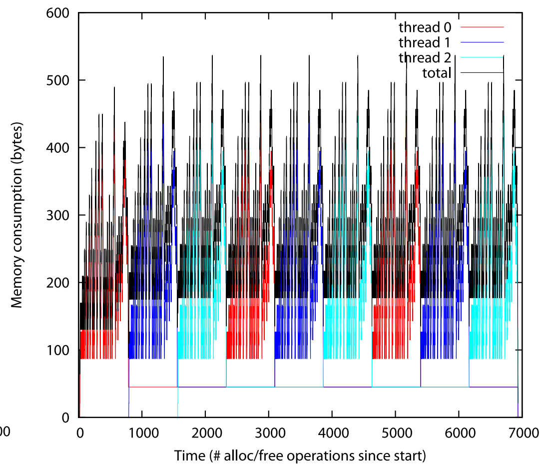
Using a linked stack architecture in conjunction with synchronisation we can make programs that would otherwise use large amounts of stack space run in much tighter memory spaces, while leaving full control with the application programmer.

5.5 Portability

We evaluated the portability of Darjeeling by running our CTP implementation on three different test beds. Table 5.6 shows the different platforms and their properties. We have chosen to run Darjeeling on three different operating systems: TinyOS [16], Contiki [9] and FOS [8], each implementing a different form of concurrency.



(a) Garbage collection, unsynchronised.



(b) Garbage collection, unsynchronised.

	TNode	TMote Sky	Fleck3/Fleck3B
Main	225	359	86
Radio	69	92	69
Sensors	184	157	110
Total	478	608	265

Table 5.7: SLOCCount for native code modules.

	TNOde	TMote Sky	Fleck3/Fleck3B
MCU	Atmega128	MSP430	Atmega128(1)
Architecture	8-bit	16-bit	8-bit
RAM	4kB	10kB	4/8kB
Radio	CC1000	CC2420	nRF905
OS	TinyOS	Contiki	FOS
Concurrency	events	protothreads	threads

Table 5.6: Platforms.

We implemented native functions for the `javax.radio.Radio` class which has broadcast, unicast send, and receive primitives. Table 5.7 shows how many source lines of C code each of the ports required. The main component is the entry point of the application that bootstraps the virtual machine. The radio and sensors components are the native code required to access the MAC layer and sensor nodes from Java respectively.

The TNOde and Sky testbeds are indoor with a great amount of overlap, so that each node can hear a large number of neighbouring nodes. On each of the platforms we allocated the same amount of RAM (2kB) to the virtual machine leaving the rest for the operating system. The Java application was programmed into the nodes via wired means.

6 A REAL-WORLD APPLICATION

In this section we evaluate the performance of Darjeeling for a real-world environmental monitoring application. The node, shown in Figure 6.1, is part of a small network that monitors micro-climate variation in an area of forest regeneration. The variables measured include soil moisture and leaf wetness, wind speed and direction, temperature and humidity, as well as engineering data regarding solar power generation, battery voltage and link quality. This application is typical of the majority of real-world sensors networks - the node is asleep most of the time, waking periodically (on the scale of minutes) to make measurements, perform some computation, then return results via a collection tree protocol.

In this section we describe the implementation of a collection tree protocol written entirely in Java, and also the re-implementation in Java of the sensor network application, originally written in C.

6.1 The routing protocol

Demonstrations exist where Java is used to simply glue C components together, but this does not show the true advantage of Java compared to an ASVM. To see how well Darjeeling copes with more complex multithreading applications we implemented a pure Java version of the well known Collection Tree Protocol (CTP), the standard routing protocol for TinyOS 2.x. This is also a powerful demonstration of how higher-level languages like Java can be used to quickly and conveniently prototype algorithms. The implementation uses many advanced features of the Java language such as inheritance, multiple threads, synchronisation, generics, exception handling, and dynamic memory management.

The CTP library comprises three main components, the packet handler, routing engine, and data engine. The packet handler provides a packet sending and receiving interface to the other two components and supports reliable unicast with retries. It queues outgoing packets to maintain packet order. A queue of only a single incoming packet is kept on the native side of the implementation. While this could potentially cause incoming packets to be dropped if the virtual machine is not fast enough to process them, in practice we have not observed this problem. The packet handler provides an event interface that notifies listeners on successful or unsuccessful packet delivery, and packet reception.

Packets are stored as byte arrays wrapped in an instance of the `Packet` class, which has get- and set operations that directly mutate the data. We use inheritance to construct different packet types such as `CtpRoutingFrame` and `CtpDataFrame`. We use the factory design pattern to allow pluggable decoding of incoming packets

The routing engine beacons a node's estimated transmission count to the sink (ETX) and keeps a list of neighbour nodes and their ETX values. It estimates the link quality to each neighbour from the number of retries that, on average, are needed to deliver a unicast packet to the node. Neighbours are pinged periodically in a round-robin fashion. The neighbour with the lowest total

Figure 6.1: The environmental monitoring node which measures soil moisture, leaf wetness, wind speed and direction, temperature and humidity.

cost is elected as the node's parent in the collection tree. The data engine forwards and periodically generates data packets.

Our Java implementation uses five threads, two in the routing engine, two in the packet handler, and one in the data engine. The size of the code, as counted with SLOCCount, is 1,163 source lines of Java, divided over 20 source files. In comparison, the NesC implementation uses 1,613 source lines of code. The infused CTP library as a .di file is less than 8kB (cf. Table 5.3).

6.2 Environmental monitoring application

The application was originally coded for the threaded operating system environment, FOS, with a CTP-like routing module implemented in C. The original C-code application contains 2 application threads and 6 system threads for CTP. The application comprises 43,852 bytes of program flash, 1,219 bytes of RAM (data + bss) and another 1,600 bytes of stack space, easily fitting within the resources of an ATmega128 processor.

It has been reimplemented in Darjeeling using the Java-based CTP. Access to sensors is via native methods in the `org.xxx.sensors` infusion. The software stack is shown in Figure 6.2. The Darjeeling run-time requires 75,412 bytes of program flash, 1,090 bytes of RAM and another 300 bytes for FOS thread stacks. It does not rely on the FOS CTP implementation, which reduces the total number of threads. This leaves well over 2kB of RAM available for the Java heap. The Java version has around half the source lines of the C version, mainly due to far fewer module initialisations required, and is a single Java thread implementing a wait, measure, send cycle. The infused application as a .di file is 1,564 bytes, which is clearly small enough to conveniently load over the air. The run-time, CTP and application infusion combined easily fit within the resources of an ATmega128 processor.

Each version of the application was run on the same node and the current consumption measured using a digital scope and a current shunt resistor. Data was downloaded from the scope for analysis, which allows for some precision in execution time measurement, and the trace for

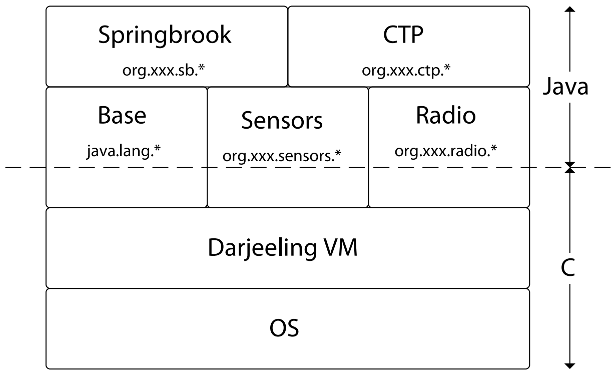


Figure 6.2: Software stack.

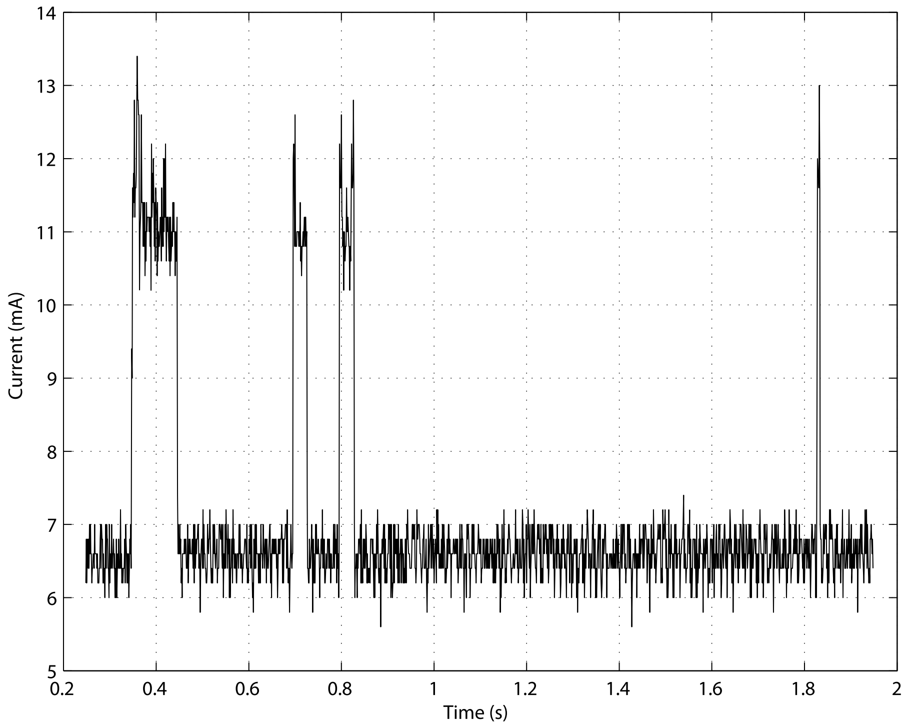


Figure 6.3: Current consumption versus time for the environmental monitoring node implemented in Darjeeling. The radio transceiver's standby draw of 12mA has been subtracted from the raw data to produce these plots.

Darjeeling is shown in Figure 6.3. To simplify the comparison we have disabled all LEDs as well as the CTP networking to eliminate energy consumption due to beaconing and packet forwarding - we wish to focus on the application performance not CTP.

From the figure we can see that the measurement cycle lasts a bit less than 2 seconds and includes various delays for sensor settling time and wind-speed pulse accumulation. During the delays the processor is put into sleep state by FOS. The ATmega128 processor is only entering a light sleep mode due to the use of non-low power timers and the serial port for debugging within the currently non power-optimised Darjeeling run-time. The total ontime for the cycle is 168 ms. By comparison the C version of the application is active for 114 ms. This indicates that Darjeeling is 50% slower than C, but over the measurement cycle of 5 minutes, this represents just 54 ms extra computation every 300s, which is an additional 0.018% overhead.

7 CONCLUSIONS

Software development for wireless sensor networks could benefit greatly from the use of virtual machines and higher-level languages such as Java. Error handling through exceptions, dynamic memory management and protection, object-oriented design, and an intuitive concurrency model are some of the features that make them an attractive concept. The feeling among many researchers however is that the performance overhead of interpreting VM instructions is prohibitively large. This computational overhead also translates directly into increased power consumption as the CPU has to be kept under power for longer.

The goals of this thesis were firstly to evaluate the use of the Java programming language in the context of wireless sensor networks and quantify the overhead in terms of execution speed and memory usage, and second to contribute to the field of virtual machine design for resource-constrained platforms. These goals were approached by designing and implementing Darjeeling, a virtual machine built from the ground up for severly memory constrained devices.

Darjeeling allows for the execution of a large subset of the Java language on micro controllers with typically 2-10kB of RAM. Through performance micro benchmarks we have shown that the overhead of interpreting Java is roughly two orders of magnitude compared to native C code. We demonstrated a data gathering application with a routing layer written in Java, which takes less than 8kB of code space and less than 2kB of RAM. The nature of WSN applications is such that the CPU is put into a power-saving deep sleep mode most of the time, so that the contribution of actual processing to the total energy consumption is very small. We observed that our Java application, although slower, decreased the network life time with less than 1% compared to an equivalent native implementation.

Darjeeling contributes to the field of VM design in several ways. We introduce a novel approach to garbage collection, using strict separation of reference and non-reference types. This allows for an efficient *O(n)* marking phase and precise compacting collection without the need for expensive run-time type inference or type tagging. We have shown that efficient preemptive multi-threading can be achieved by allocating stack space ad-hoc, with idle threads using as little as 45 bytes, providing a feasible alternative to event-based and protothreading concurrency models. Finally we show how our 16-bit instruction set reduces stack space requirements with as much as 27% in our real-world example, and describe an algorithm for optimising 32-bit integer arithmetic to equivalent 16-bit arithmetic where this is possible.

Our toolchain includes a static linking tool, called the infuser, that reduces code footprint by 55-83%, at the cost of sacrificing reflection. Darjeeling was demonstrated to be highly portable across MCUs and operating systems, with platform-specific code for the VM ranging from 265-608 source lines of code. Darjeeling is, to the best of our knowledge, the first full-featured open source and Java capable VM for wireless sensor networks.

BIBLIOGRAPHY

[1] Ole Agesen and David Detlefs and J. Eliot and B. Moss (1998) *Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines,* SIGPLAN Conf. on Prog. Lang. Design and Impl

[2] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom (2008) *Introducing TakaTuka: a Java virtualmachine for motes,* Proceedings of the 6th ACM conference on Embedded network sensor systems

[3] von Behren, Rob and Condit, Jeremy and Zhou, Feng and Necula, George C. and Brewer, Eric (2003) *Capriccio: Scalable Threads for Internet Services,* SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles

[4] Bhatti, S. et al. (2005) *MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms,* Mob. Netw. Appl.

[5] Brouwers, N. et al. (2008) *Darjeeling, a Java Compatible Virtual Machine for Wireless Sensor Networks,* Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion

[6] Butters, A. M. (2007) *Total Cost of Ownership: A Comparison of C/C++ and Java*

[7] Bishop-Hurley, G. J. and Swain, D. L. and Anderson, D. M. and Sikka, P. and Crossman, C. and Corke, P. (2007) *Virtual fencing applications: Implementing and testing an automated cattle control system,* Comput. Electron. Agric.

[8] Corke, P. (2008) *FOS: A New Operating System for Sensor Networks,* Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN08)

[9] Dunkels, A. and Gronvall, B. and Voigt, T. (2004) *Contiki: A Lightweight and Flexible Operating System for Tiny Networked Sensors,* Local Computer Networks, 2004. 29th Annual IEEE International Conference on,

[10] Dunkels, A. et al. (2006) *Run-time dynamic linking for reprogramming wireless sensor networks,* Proceedings of the 4th international conference on Embedded networked sensor systems

[11] Dunkels, A. et al. (2006) *Protothreads: simplifying event-driven programming of memory-constrained embedded systems,* Proceedings of the 4th international conference on Embedded networked sensor systems

[12] Fonseca, R. and Gnawali, O. and Jamieson, K. and Kim, S. and Levis, P. and Woo, A. (2006) *The Collection Tree Protocol (CTP),* TEP 123 Draft

[13] Hill J. et al (2000) *System Architecture Directions for Networked Sensors.,* Architectural Support for Programming Languages and Operating Systems

[14] Koshy, J., Pandey, R. (2005) *VMSTAR: synthesizing scalable runtime environments for sensor networks,* In SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems

[15] Levis, P., Culler, D. (2002) *Maté: a tiny virtual machine for sensor networks,* SIGOPS Oper. Syst. Rev.

[16] Levis, Philip and Gay, David and Culler, David (2005) *Active Sensor Networks,* NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation

[17] Lindholm, Tim and Yellin, Frank (1999) *The Java Virtual Machine Specification (2nd Edition),* , Prentice Hall PTR

[18] Lindholm, T., Yellin, F. (2006) *Virtual Machine Specification, Java CardTM Platform, Version 2.2.2.,* Sun Microsystems Inc

[19] Mainwaring, Alan and Culler, David and Polastre, Joseph and Szewczyk, Robert and Anderson, John (2002) *Wireless sensor networks for habitat monitoring,* WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications

[20] Moteiv (2006) *Ultra low power IEEE 802.15.4 compliant wireless sensor module (datasheet),* http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf

[21] http://www.harbaum.org/till/nanovm

[22] D. Palmer, et al. (2005) *An Optimising Compiler for Generated Tiny Virtual Machines,* EmNetS-II. The Second IEEE Workshop

[23] Porthouse, C., Butcher, D. (2004) *Multitasking JavaTM on ARM platforms,* Whitepaper, ARM Limited, http://www.arm.com/pdfs/MVMWhitePaper.pdf

[24] B. Saballus et. al. *Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing*

[25] Sen, S. & Oliver, C. R. (2006) *A Rule-Based Language for Programming Wireless Sensor Actuator Networks using Frequency and Communication.,* EmNetS-III. The Third IEEE Workshop on Embedded Networked Sensors

[26] Sentilla http://www.sentilla.com

[27] Shi, Y. et al. (2005) *Virtual machine showdown: stack versus registers,* VEE'05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments

[28] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White (2006) *Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine,* ACM VEE

[29] *leJOS: a Java based OS for Lego RCX* Solorzano, J., , http://lejos.sourceforge.net

[30] Sun Microsystems, Inc. (2000) *Connected, Limited Device Configuration Specification Version 1.0a*

[31] Sun Microsystems (2002) *Java Card™ 2.2 Off-Card Verifier. Whitepaper*

[32] Sun Microsystems (2006) *Virtual Machine Specification, Java Card Platform, v2.2.2,* march

[33] http://tinyvm.sourceforge.net/

[34] http://www.tiobe.com

[35] Tijs van Dam, Koen Langendoen (2003) *An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks*

[36] Wei Ye, John Heidemann, Deborah Estrin (2003) *Medium Access Control with Coordinated, Adaptive Sleeping for Wireless Sensor Networks*

A INSTRUCTION SET

AALOAD

Opcode: 56 (0x38)
Format: `aaload`
Integer stack: ..., *index:*`short` ⇒ ...
Reference stack: ..., *arrayref* ⇒ ..., *value:*`short`

Description: Retrieves a `reference` from an array at index *index* and pushes it onto the reference stack. The *index* and *arrayref* are popped from the integer- and reference stack respectively.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

AASTORE

Opcode: 61 (0x3d)
Format: `iastore`
Integer stack: ..., *index:*`short` ⇒ ...
Reference stack: ..., *arrayref*, *value* ⇒ ...

Description: Pops a `reference` value from the stack and stores it in array *arrayref* at index *index*. The *index* is popped from the integer stack, the *arrayref* and *value* are popped from the reference stack.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

ACONST_NULL

Opcode: 15 (0x0f)
Format: `aconst_null`
Reference stack: ..., ⇒ ..., `null`

Description: Push `null` onto the reference stack

ADUP

Opcode: 70 (0x46)
Format: `adup`
Reference stack: ..., $value_1$ ⇒ ..., $value_1$, $value_1$

Description: Duplicates the top value on the refrence stack.

ADUP2

Opcode: 71 (0x47)
Format: `adup2`
Reference stack: ..., $value_2$, $value_1$ ⇒ ..., $value_2$, $value_1$, $value_2$, $value_1$

Description: Duplicates the top two values on the refrence stack.

ADUP_X1

Opcode: 72 (0x48)
Format: `adup_x1`
Reference stack: ..., $value_2$, $value_1$ ⇒ ..., $value_1$, $value_2$, $value_1$

Description: Duplicates the top value on the reference stack and inserts it one place down.

ADUP_X2

Opcode: 73 (0x49)
Format: `adup_x2`
Reference stack: ..., $value_3$, $value_2$, $value_1$ ⇒ ..., $value_1$, $value_3$, $value_2$, $value_1$

Description: Duplicates the top value on the reference stack and inserts it two places down.

ALOAD

Opcode: 32 (0x20)
Format: `aload`, *slot_nr*
Reference stack: ..., ⇒ ..., value

Description: Loads a `reference` value from the reference local variable pool at slot index *slot_nr* and pushes it onto the stack

ALOAD_0

Opcode: 33 (0x21)
Format: `aload_0`
Reference stack: ..., ⇒ ..., value

Description: Loads a `reference` value from the reference local variable pool at slot index 0 and pushes it onto the stack

ALOAD_1

Opcode: 34 (0x22)
Format: `aload_1`
Reference stack: ..., ⇒ ..., value

Description: Loads a `reference` value from the reference local variable pool at slot index 1 and pushes it onto the stack

ALOAD_2

Opcode: 35 (0x23)
Format: `aload_2`
Reference stack: ..., ⇒ ..., value

Description: Loads a `reference` value from the reference local variable pool at slot index 2 and pushes it onto the stack

ALOAD_3

Opcode: 36 (0x24)
Format: `aload_3`
Reference stack: ..., ⇒ ..., value

Description: Loads a `reference` value from the reference local variable pool at slot index 3 and pushes it onto the stack

ANEWARRAY

Opcode: 160 (0xa0)
Format: `anewarray`, *infusion_id*, *entity_id*
Reference stack: ..., ⇒ ..., *array*

Description: Creates a new array of class pointed to by the local ID (*infusion_id*, *entity_id*).

APOP

Opcode: 68 (0x44)
Format: `apop`
Reference stack: ..., $value_1$ ⇒ ...

Description: Pops the top value off the refrence stack.

APOP2

Opcode: 69 (0x45)
Format: `apop2`
Reference stack: ..., $value_2$, $value_1$ ⇒ ...

Description: Pops the top two values off the refrence stack.

ARETURN

Opcode: 152 (0x98)
Format: `areturn`
Reference stack: ..., *value* ⇒ ...

Description: Exists the current method, returning a value of type `ref`.

ASTORE

Opcode: 47 (0x2f)
Format: `astore`, *slot_nr*
Reference stack: ..., value ⇒ ...

Description: Pops a `reference` value from the reference stack and stores it in the reference local variable pool at index *slot_nr*

ASTORE_0

Opcode: 48 (0x30)
Format: `astore_0`
Reference stack: ..., value ⇒ ...

Description: Pops a `reference` value from the reference stack and stores it in the reference local variable pool at index 0

ASTORE_1

Opcode: 49 (0x31)
Format: `astore_1`
Reference stack: ..., value ⇒ ...

Description: Pops a `reference` value from the reference stack and stores it in the reference local variable pool at index 1

ASTORE_2

Opcode: 50 (0x32)
Format: `astore_2`
Reference stack: ..., value ⇒ ...

Description: Pops a `reference` value from the reference stack and stores it in the reference local variable pool at index 2

ASTORE_3

Opcode: 51 (0x33)
Format: `astore_3`
Reference stack: ..., value ⇒ ...

Description: Pops a `reference` value from the reference stack and stores it in the reference local variable pool at index 3

ASWAP

Opcode: 74 (0x4a)
Format: `aswap`
Reference stack: ..., $value_2$, $value_1$ ⇒ ..., $value_1$, $value_2$

Description: Swaps the two top elements on the reference stack.

ATHROW

Opcode: 161 (0xa1)
Format: `athrow`
Reference stack: ..., *throwable* ⇒ ...

Description: Throws *throwable*.

BALOAD

Opcode: 52 (0x34)
Format: `baload`
Integer stack: ..., *index:*`short` ⇒ ..., *value:*`short`
Reference stack: ..., *arrayref* ⇒ ...

Description: Retrieves a `byte` or `boolean` from an array at index *index*, widens it to type `short` and pushes it onto the integer stack. The *index* and *arrayref* are popped from the integer- and reference stack respectively.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

BASTORE

Opcode: 57 (0x39)
Format: `bastore`
Integer stack: ..., *index:*`short` , *value:*`short` ⇒ ...
Reference stack: ..., *arrayref* ⇒ ...

Description: Pops a `byte` or `boolean` value from the stack (`byte` types are automatically widened to `short`) and stores it in array *arrayref* at index *index*. The *index* and *value* are popped from the integer stack, the *arrayref* is popped from the reference stack.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

BINC

Opcode: 119 (0x77)
Format: `binc`, *slot_nr*, *increase*:`byte`

Description: Increases local variable of type `byte` at index *slot_nr* by *increase*

BIPUSH

Opcode: 17 (0x11)
Format: `bipush`, *valuebyte1*
Integer stack: ..., ⇒ ..., *valuebyte1*:`int`

Description: Widen immediate `byte` value *valuebyte1* to type `int` and push onto the integer stack

BSPUSH

Opcode: 16 (0x10)
Format: `bspush`, *valuebyte1*
Integer stack: ..., ⇒ ..., *valuebyte1*:`short`

Description: Widen immediate `byte` value *valuebyte1* to type `short` and push onto the integer stack

CALOAD

Opcode: 53 (0x35)
Format: `caload`
Integer stack: ..., *index:*`short` ⇒ ..., *value:*`short`
Reference stack: ..., *arrayref* ⇒ ...

Description: Retrieves a `char` from an array at index *index*, widens it to type `short` and pushes it onto the integer stack. The *index* and *arrayref* are popped from the integer- and reference stack respectively.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

CASTORE

Opcode: 58 (0x3a)
Format: `castore`
Integer stack: ..., *index:*`short` , *value:*`short` ⇒ ...
Reference stack: ..., *arrayref* ⇒ ...

Description: Pops a `char` value from the stack (`char` types are automatically widened to `short`) and stores it in array *arrayref* at index *index*. The *index* and *value* are popped from the integer stack, the *arrayref* is popped from the reference stack.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

CHECKCAST

Opcode: 162 (0xa2)
Format: `checkcast`, *infusion_id*, *entity_id*
Reference stack: ..., *object* ⇒ ..., *object*

Description: If *object* is not an instance of the class pointed to by the local ID (*infusion_id*, *entity_id*), throws a `ClassCastException`.

GETFIELD_A

Opcode: 79 (0x4f)
Format: `getfield_a`, *indexbyte*
Reference stack: ..., *objectref* ⇒ ..., *value*

Description: Gets the value of the `reference` field in object *objectref* at immediate index *indexbyte* and pushes it onto the reference stack

GETFIELD_B

Opcode: 75 (0x4b)
Format: `getfield_b`, *offsetbyte*
Integer stack: ... ⇒ ..., *value*:`short`
Reference stack: ..., *objectref* ⇒ ...

Description: Gets the value of the `byte` field in object *objectref* at immediate offset *offsetbyte*, widens it to type `short` and pushes it onto the integer stack

GETFIELD_C

Opcode: 76 (0x4c)
Format: `getfield_c`, *offsetbyte*
Integer stack: ... ⇒ ..., *value*:`short`
Reference stack: ..., *objectref* ⇒ ...

Description: Gets the value of the `char` field in object *objectref* at immediate offset *offsetbyte*, widens it to type `short` and pushes it onto the integer stack

GETFIELD_I

Opcode: 78 (0x4e)
Format: `getfield_i`, *offsetbyte*
Integer stack: ... ⇒ ..., *value*:`int`
Reference stack: ..., *objectref* ⇒ ...

Description: Gets the value of the `int` field in object *objectref* at immediate offset *offsetbyte* and pushes it onto the integer stack

GETFIELD_S

Opcode: 77 (0x4d)
Format: `getfield_s`, *offsetbyte*
Integer stack: ... ⇒ ..., *value*:`short`
Reference stack: ..., *objectref* ⇒ ...

Description: Gets the value of the `short` field in object *objectref* at immediate offset *offsetbyte* and pushes it onto the integer stack

GETSTATIC_A

Opcode: 89 (0x59)
Format: `getstatic_a`, *infusion_id*, *indexbyte*
Reference stack: ..., ⇒ ..., *value*

Description: Gets the value of a `reference` static variable in the infusion indicated by *infusion_id* at index *indexbyte* and pushes it onto the reference stack.

GETSTATIC_B

Opcode: 85 (0x55)
Format: `getstatic_b`, *infusion_id*, *indexbyte*
Integer stack: ..., ⇒ ..., *value*:`short`

Description: Gets the value of a `byte` static variable in the infusion indicated by *infusion_id* at index *indexbyte*, widens it to type `short`, and pushes it onto the integer stack.

GETSTATIC_C

Opcode: 86 (0x56)
Format: `getstatic_c`, *infusion_id*, *indexbyte*
Integer stack: ..., ⇒ ..., *value*:`short`

Description: Gets the value of a `char` static variable in the infusion indicated by *infusion_id* at index *indexbyte*, widens it to type `short`, and pushes it onto the integer stack.

GETSTATIC_I

Opcode: 88 (0x58)
Format: `getstatic_i`, *infusion_id*, *indexbyte*
Integer stack: ..., ⇒ ..., *value*:`int`

Description: Gets the value of a `int` static variable in the infusion indicated by *infusion_id* at index *indexbyte*, and pushes it onto the integer stack.

GETSTATIC_S

Opcode: 87 (0x57)
Format: `getstatic_s`, *infusion_id*, *indexbyte*
Integer stack: ..., ⇒ ..., *value*:`short`

Description: Gets the value of a `short` static variable in the infusion indicated by *infusion_id* at index *indexbyte*, and pushes it onto the integer stack.

GOTO

Opcode: 146 (0x92)
Format: `goto`, *branch_adress*

Description: Branch unconditionally.

GOTO_W

Opcode: 147 (0x93)
Format: `goto_w`, *branch_adress*

Description: Branch unconditionally, wide address.

I2B

Opcode: 124 (0x7c)
Format: `i2b`
Integer stack: ..., *value*:`int` ⇒ ..., *value*:`byte`

Description: Narrows a value of type `int` to `byte`.

I2C

Opcode: 169 (0xa9)
Format: `i2c`
Integer stack: ..., *value*:`int` ⇒ ..., *value*:`char`

Description: Narrows a value of type `int` to `char`.

I2S

Opcode: 125 (0x7d)
Format: `i2s`
Integer stack: ..., *value*:`int` ⇒ ..., *value*:`short`

Description: Narrows a value of type `int` to `short`.

IADD

Opcode: 107 (0x6b)
Format: `iadd`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ + $value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IALOAD

Opcode: 55 (0x37)
Format: `iaload`
Integer stack: ..., *index:*`short` ⇒ ..., *value:*`int`
Reference stack: ..., *arrayref* ⇒ ...

Description: Retrieves an `int` from an array at index *index*, and pushes it onto the integer stack. The *index* and *arrayref* are popped from the integer- and reference stack respectively.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

IAND

Opcode: 116 (0x74)
Format: `iand`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ & $value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IASTORE

Opcode: 60 (0x3c)
Format: `iastore`
Integer stack: ..., *index:*`short` , *value:*`int` ⇒ ...
Reference stack: ..., *arrayref* ⇒ ...

Description: Pops an `int` value from the stack and stores it in array *arrayref* at index *index*. The *index* and *value* are popped from the integer stack, the *arrayref* is popped from the reference stack.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

ICMPEQ

Opcode: 140 (0x8c)
Format: `icmpeq`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ equals $value_1$.

ICMPGE

Opcode: 143 (0x8f)
Format: `icmpge`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ greater than or equals $value_1$.

ICMPGT

Opcode: 144 (0x90)
Format: `icmpgt`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ greater than $value_1$.

ICMPLE

Opcode: 145 (0x91)
Format: `icmple`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ less than or equals $value_1$.

ICMPLT

Opcode: 142 (0x8e)
Format: `icmplt`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ less than $value_1$.

ICMPNE

Opcode: 141 (0x8d)
Format: `icmpne`, *branch_adress*
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ...

Description: Branch if $value_2$ not equals $value_1$.

ICONST_0

Opcode: 9 (0x09)
Format: `iconst_0`
Integer stack: ..., ⇒ ..., *0:*`int`

Description: Push `int` constant [V_2;V_1] = 0 onto the integer stack

ICONST_1

Opcode: 10 (0x0a)
Format: `iconst_1`
Integer stack: ..., ⇒ ..., *1:*`int`

Description: Push `int` constant $[V_2;V_1]$ = 1 onto the integer stack

ICONST_2

Opcode: 11 (0x0b)
Format: `iconst_2`
Integer stack: ..., ⇒ ..., *2:*`int`

Description: Push `int` constant $[V_2;V_1]$ = 2 onto the integer stack

ICONST_3

Opcode: 12 (0x0c)
Format: `iconst_3`
No Change

Description: No operation

Integer stack: ..., ⇒ ..., *3:*`int`

Description: Push `int` constant $[V_2;V_1]$ = 3 onto the integer stack

ICONST_4

Opcode: 13 (0x0d)
Format: `iconst_4`
Integer stack: ..., ⇒ ..., *4:*`int`

Description: Push `int` constant $[V_2;V_1]$ = 4 onto the integer stack

ICONST_5

Opcode: 14 (0x0e)
Format: `iconst_5`
Integer stack: ..., ⇒ ..., *5:*`int`

Description: Push `int` constant $[V_2;V_1]$ = 5 onto the integer stack

ICONST_M1

Opcode: 8 (0x08)
Format: `iconst_m1`
Integer stack: ..., ⇒ ..., *-1:*`int`

Description: Push `int` constant $[V_2;V_1]$ = -1 onto the integer stack

IDIV

Opcode: 110 (0x6e)
Format: `idiv`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ / $value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IDUP

Opcode: 64 (0x40)
Format: `idup`
Integer stack: ..., $value_1$`:short` ⇒ ..., $value_1$`:short` , $value_1$`:short`

Description: Duplicates the top value of type `short` on the integer stack.

IDUP2

Opcode: 65 (0x41)
Format: `idup2`
Integer stack: ..., $value_1$`:int` ⇒ ..., $value_1$`:int` , $value_1$`:int`
Integer stack: ..., $value_2$`:short` , $value_1$`:short` ⇒ ..., $value_2$`:short` , $value_1$`:short` , $value_2$`:short` , $value_1$`:short`

Description: Depending on the state of the stack, either duplicates one value of type `int` or duplicates two values of type `short`.

IDUP_X1

Opcode: 66 (0x42)
Format: `idup_x1`
Integer stack: ..., $value_2$`:short` , $value_1$`:short` ⇒ ..., $value_1$`:short` , $value_2$`:short` , $value_1$`:short`

Description: Duplicates the top value of type `short` on the integer stack and inserts it one place down.

IDUP_X2

Opcode: 166 (0xa6)
Format: `idup_x2`
Integer stack: ..., $value_3$`:short` , $value_2$`:short` , $value_1$`:short` ⇒ ..., $value_1$`:short` , $value_3$`:short` , $value_2$`:short` , $value_1$`:short`

Description: Duplicates the top value of type `short` on the integer stack and inserts it two places down.

IFNULL

Opcode: 133 (0x85)
Format: `ifnull`, *branch_adress*
Reference stack: ..., *value* ⇒ ...

Description: Branch if *value* not equals `null`.

IIFEQ

Opcode: 126 (0x7e)
Format: `iifeq`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* equals zero.

IIFGE

Opcode: 129 (0x81)
Format: `iifge`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* greater than or equals zero.

IIFGT

Opcode: 130 (0x82)
Format: `iifgt`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* greater than zero.

IIFLE

Opcode: 131 (0x83)
Format: `iifle`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* less than or equals zero.

IIFLT

Opcode: 128 (0x80)
Format: `iiflt`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* less than zero.

IIFNE

Opcode: 127 (0x7f)
Format: `iifne`, *branch_adress*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Branch if *value* not equals zero.

IINC

Opcode: 121 (0x79)
Format: `iinc`, *slot_nr*, *increase*:`byte`

Description: Increases local variable of type `int` at index *slot_nr* by *increase*

IINC_W

Opcode: 167 (0xa7)
Format: `iinc_w`, *slot_nr*, *increase*:`short`

Description: Increases local variable of type `int` at index *slot_nr* by *increase*

IIPUSH

Opcode: 20 (0x14)
Format: `bspush`, $valuebyte_4$, $valuebyte_3$, $valuebyte_2$, $valuebyte_1$
Integer stack: ..., ⇒ ..., ($valuebyte_4$ <<24 + $valuebyte_3$ <<16 + $valuebyte_2$ <<8 + $valuebyte_1$):`int`

Description: Push immediate `int` value ($valuebyte_4$ <<24 + $valuebyte_3$ <<16 + $valuebyte_2$ <<8 + $valuebyte_1$) onto the integer stack

ILOAD

Opcode: 27 (0x1b)
Format: `iload`, *slot_nr*
Integer stack: ..., ⇒ ..., value:`int`

Description: Loads an `int` value from the integer local variable pool at slot index *slot_nr* and pushes it onto the stack

ILOAD_0

Opcode: 28 (0x1c)
Format: `iload_0`
Integer stack: ..., ⇒ ..., value:`int`

Description: Loads an `int` value from the integer local variable pool at slot index 0 and pushes it onto the stack

ILOAD_1

Opcode: 29 (0x1d)
Format: `iload_1`
Integer stack: ..., ⇒ ..., value:`int`

Description: Loads an `int` value from the integer local variable pool at slot index 1 and pushes it onto the stack

ILOAD_2

Opcode: 30 (0x1e)
Format: `iload_2`
Integer stack: ..., ⇒ ..., value:`int`

Description: Loads an `int` value from the integer local variable pool at slot index 2 and pushes it onto the stack

ILOAD_3

Opcode: 31 (0x1f)
Format: `iload_3`
Integer stack: ..., ⇒ ..., value:`int`

Description: Loads an `int` value from the integer local variable pool at slot index 3 and pushes it onto the stack

IMUL

Opcode: 109 (0x6d)
Format: `imul`
Integer stack: ..., *value*$_2$:`int`, *value*$_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ * *value*$_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

INEG

Opcode: 112 (0x70)
Format: `ineg`
Integer stack: ..., *value*$_1$:`int` ⇒ ..., *result*:`int`

Description: Negates the top `int` element on the integer stack. Note that any potential overflow is discarded.

INSTANCEOF

Opcode: 163 (0xa3)
Format: `instanceof`, *infusion_id*, *entity_id*
Integer stack: ... ⇒ ..., *result*:`boolean`
Reference stack: ..., *object* ⇒ ...

Description: Evaluates whether *object* is an instance of the class pointed to by the local ID (*infusion_id*, *entity_id*).

INVOKEINTERFACE

Opcode: 157 (0x9d)
Format: `invokeinterface`, *infusion_id*, *entity_id*
Integer stack: ..., [*arg*]* ⇒ ...
Reference stack: ..., [*arg*]* ⇒ ...

Description: Invokes an interface method, pointed to by the local ID (*infusion_id*, *entity_id*), on *object*.

INVOKESPECIAL

Opcode: 155 (0x9b)
Format: `invokespecial`, *infusion_id*, *entity_id*
Integer stack: ..., [*arg*]* ⇒ ...
Reference stack: ..., *object*, [*arg*]* ⇒ ...

Description: Invokes a static method, pointed to by the local ID (*infusion_id*, *entity_id*), with *object* as a first reference argument. Used for calling constructors mostly.

INVOKESTATIC

Opcode: 156 (0x9c)
Format: `invokestatic`, *infusion_id*, *entity_id*
Integer stack: ..., [*arg*]* ⇒ ...
Reference stack: ..., [*arg*]* ⇒ ...

Description: Invokes a static method, pointed to by the local ID (*infusion_id*, *entity_id*)

INVOKEVIRTUAL

Opcode: 154 (0x9a)
Format: `invokevirtual`, *infusion_id*, *entity_id*
Integer stack: ..., [*arg*]* ⇒ ...
Reference stack: ..., *object*, [*arg*]* ⇒ ...

Description: Invokes a virtual method, pointed to by the local ID (*infusion_id*, *entity_id*), on *object*.

IOR

Opcode: 117 (0x75)
Format: `ior`
Integer stack: ..., *value*$_2$:`int`, *value*$_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ | *value*$_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IPOP

Opcode: 62 (0x3e)
Format: `ipop`
Integer stack: ..., *value*$_1$*:*`short` ⇒ ...

Description: Pops a value of type `short` off the integer stack.

IPOP2

Opcode: 63 (0x3f)
Format: `ipop2`
Integer stack: ..., *value*$_1$*:*`int` ⇒ ...
Integer stack: ..., *value*$_2$*:*`short` , *value*$_1$*:*`short` ⇒ ...

Description: Depending on the state of the stack, either pops a value of type `int` off the integer stack, or pops two values of type `short` off the integer stack.

IREM

Opcode: 111 (0x6f)
Format: `irem`
Integer stack: ..., *value*$_2$:`int`, *value*$_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ % *value*$_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IRETURN

Opcode: 151 (0x97)
Format: `ireturn`
Integer stack: ..., *value*:`short` ⇒ ...

Description: Exists the current method, returning a value of type `int`.

ISHL

Opcode: 113 (0x71)
Format: `ishl`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1 << value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

ISHR

Opcode: 114 (0x72)
Format: `ishr`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1 >> value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

ISTORE

Opcode: 42 (0x2a)
Format: `istore`, *slot_nr*
Integer stack: ..., value:`int` ⇒ ...

Description: Pops an `int` value from the integer stack and stores it in the integer local variable pool at index *slot_nr*

ISTORE_0

Opcode: 43 (0x2b)
Format: `istore_0`
Integer stack: ..., value:`int` ⇒ ...

Description: Pops an `int` value from the integer stack and stores it in the integer local variable pool at index 0

ISTORE_1

Opcode: 44 (0x2c)
Format: `istore_1`
Integer stack: ..., value:`int` ⇒ ...

Description: Pops an `int` value from the integer stack and stores it in the integer local variable pool at index 1

ISTORE_2

Opcode: 45 (0x2d)
Format: `istore_2`
Integer stack: ..., value:`int` ⇒ ...

Description: Pops an `int` value from the integer stack and stores it in the integer local variable pool at index 2

ISTORE_3

Opcode: 46 (0x2e)
Format: `istore_3`
Integer stack: ..., value:`int` ⇒ ...

Description: Pops an `int` value from the integer stack and stores it in the integer local variable pool at index 3

ISUB

Opcode: 108 (0x6c)
Format: `isub`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ - $value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

ISWAP_X

Opcode: 67 (0x43)
Format: `iswap_x`, *indexbyte*
Integer stack: Depends on the values of *m* and *n*

Description: Two 4-bit parameters *m* and *n* are encoded in *indexbyte*. The parameter *m* is stored in the high nibble, *n* is stored in the low nibble. The instruction takes the top *m* slots on the integer stack and swaps them with the *n* underlying slots.

IUSHR

Opcode: 115 (0x73)
Format: `iushr`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1 >>> value_2$ (logical shift right) is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

IXOR

Opcode: 118 (0x76)
Format: `ixor`
Integer stack: ..., $value_2$:`int`, $value_1$:`int` ⇒ ..., *result*:`int`

Description: Pops two `int` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ ^ $value_2$ is pushed onto the integer stack as a `int`. Note that any potential overflow is discarded.

LDS

Opcode: 21 (0x15)
Format: `lds`, *infusion_id*, *entity_id*
Reference stack: ..., ⇒ ..., *stringref*

Description: The local id (*infusion_id*, *entity_id*) points to a string table entry which is loaded and pushed onto the reference stack.

LOOKUPSWITCH

Opcode: 149 (0x95)
Format: `lookupswitch`, *default address*, *num_pairs*:`short`, *jump table...*
Integer stack: ..., *value*:`int` ⇒ ...

Description: The jump table consists of key/value pairs in the form of an *key*:`int` value and a branch target. If *value* matches one of the keys, this instruction branches to the corresponding branch target. Otherwise the instruction branches to the default address.

MONITORENTER

Opcode: 164 (0xa4)
Format: `monitorenter`
Reference stack: ..., *object* ⇒ ...

Description: Enters a critical section with *object* as a monitor.

MONITOREXIT

Opcode: 165 (0xa5)
Format: `monitorexit`
Reference stack: ..., *object* ⇒ ...

Description: Exits a critical section with *object* as a monitor.

NEW

Opcode: 158 (0x9e)
Format: `new`, *infusion_id*, *entity_id*
Reference stack: ..., ⇒ ..., *object*

Description: Creates a new instance of the class pointed to by the local ID (*infusion_id*, *entity_id*).

NEWARRAY

Opcode: 159 (0x9f)
Format: `anewarray`, *array_type*
Reference stack: ..., ⇒ ..., *array*

Description: Creates a new array of type *array_type* (byte, char, boolean, short, int, etc).

NOP

Opcode: 0 (0x00)
Format: `nop`
Integer stack: No Change
Reference stack: No Change

Description: No operation

PUTFIELD_A

Opcode: 84 (0x54)
Format: `putfield_a`, *indexbyte*
Reference stack: ..., *objectref*, *value* ⇒ ...

Description: Sets the value of the `byte` field in object *objectref* at immediate index *offsetbyte*

PUTFIELD_B

Opcode: 80 (0x50)
Format: `putfield_b`, *offsetbyte*
Integer stack: ..., *value*:`short` ⇒ ...
Reference stack: ..., *objectref* ⇒ ...

Description: Sets the value of the `byte` field in object *objectref* at immediate offset *offsetbyte*

PUTFIELD_C

Opcode: 81 (0x51)
Format: `putfield_c`, *offsetbyte*
Integer stack: ..., *value*:`short` ⇒ ...
Reference stack: ..., *objectref* ⇒ ...

Description: Sets the value of the `char` field in object *objectref* at immediate offset *offsetbyte*

PUTFIELD_I

Opcode: 83 (0x53)
Format: `putfield_i`, *offsetbyte*
Integer stack: ..., *value*:`int` ⇒ ...
Reference stack: ..., *objectref* ⇒ ...

Description: Sets the value of the `int` field in object *objectref* at immediate offset *offsetbyte*

PUTFIELD_S

Opcode: 82 (0x52)
Format: `putfield_s`, *offsetbyte*
Integer stack: ..., *value*:`short` ⇒ ...
Reference stack: ..., *objectref* ⇒ ...

Description: Sets the value of the `short` field in object *objectref* at immediate offset *offsetbyte*

PUTSTATIC_A

Opcode: 94 (0x5e)
Format: `putstatic_a`, *infusion_id*, *indexbyte*
Reference stack: ..., *value*:`reference` ⇒ ...

Description: Sets the value of `reference` static variable in the infusion indicated by *infusion_id* at index *indexbyte*

PUTSTATIC_B

Opcode: 90 (0x5a)
Format: `putstatic_b`, *infusion_id*, *indexbyte*
Integer stack: ..., *value*:`short` ⇒ ...

Description: Sets the value of `byte` static variable in the infusion indicated by *infusion_id* at index *indexbyte*

PUTSTATIC_C

Opcode: 91 (0x5b)
Format: `putstatic_c`, *infusion_id*, *indexbyte*
Integer stack: ..., *value*:`short` ⇒ ...

Description: Sets the value of `char` static variable in the infusion indicated by *infusion_id* at index *indexbyte*

PUTSTATIC_I

Opcode: 93 (0x5d)
Format: `putstatic_i`, *infusion_id*, *indexbyte*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Sets the value of `int` static variable in the infusion indicated by *infusion_id* at index *indexbyte*

PUTSTATIC_S

Opcode: 92 (0x5c)
Format: `putstatic_s`, *infusion_id*, *indexbyte*
Integer stack: ..., *value*:`short` ⇒ ...

Description: Sets the value of `short` static variable in the infusion indicated by *infusion_id* at index *indexbyte*

RETURN

Opcode: 153 (0x99)
Format: `return`

Description: Exists the current method.

S2B

Opcode: 122 (0x7a)
Format: `s2b`
Integer stack: ..., *value*:`short` ⇒ ..., *value*:`byte`

Description: Narrows a value of type `short` to `byte`.

S2C

Opcode: 170 (0xaa)
Format: `s2c`
Integer stack: ..., *value*:`short` ⇒ ..., *value*:`char`

Description: Narrows a value of type `short` to `char`.

S2I

Opcode: 123 (0x7b)
Format: `s2i`
Integer stack: ..., *value*:`short` ⇒ ..., *value*:`int`

Description: Widens a value of type `short` to `int`.

SADD

Opcode: 95 (0x5f)
Format: `sadd`
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as $value_1$ and $value_2$. The result $value_1 + value_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SALOAD

Opcode: 54 (0x36)
Format: `saload`
Integer stack: ..., *index:*`short` ⇒ ..., *value:*`short`
Reference stack: ..., *arrayref* ⇒ ...

Description: Retrieves a `short` from an array at index *index* and pushes it onto the integer stack. The *index* and *arrayref* are popped from the integer- and reference stack respectively.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

SAND

Opcode: 104 (0x68)
Format: `sand`
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ & $value_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SASTORE

Opcode: 59 (0x3b)
Format: `sastore`
Integer stack: ..., *index:*`short` , *value:*`short` ⇒ ...
Reference stack: ..., *arrayref* ⇒ ...

Description: Pops a `short` value from the stack and stores it in array *arrayref* at index *index*. The *index* and *value* are popped from the integer stack, the *arrayref* is popped from the reference stack.

Exceptions: Throws `NullPointerException` if *arrayref* is `null`. Throws `IndexOutOfBoundsException` if *index* is not a valid index.

SCMPEQ

Opcode: 134 (0x86)
Format: `scmpeq`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ equals $value_1$.

SCMPGE

Opcode: 137 (0x89)
Format: `scmpge`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ greater than or equals $value_1$.

SCMPGT

Opcode: 138 (0x8a)
Format: `scmpgt`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ greater than $value_1$.

SCMPLE

Opcode: 139 (0x8b)
Format: `scmple`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ less than or equals $value_1$.

SCMPLT

Opcode: 136 (0x88)
Format: `scmplt`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ less than $value_1$.

SCMPNE

Opcode: 135 (0x87)
Format: `scmpne`, *branch_adress*
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ...

Description: Branch if $value_2$ not equals $value_1$.

SCONST_0

Opcode: 2 (0x02)
Format: `sconst_0`
Integer stack: ..., ⇒ ..., *0:*`short`

Description: Push `short` constant 0 onto the integer stack

SCONST_1

Opcode: 3 (0x03)
Format: `sconst_1`
Integer stack: ..., ⇒ ..., *1:*`short`

Description: Push `short` constant 1 onto the integer stack

SCONST_2

Opcode: 4 (0x04)
Format: `sconst_2`
Integer stack: ..., ⇒ ..., *2:*`short`

Description: Push `short` constant 2 onto the integer stack

SCONST_3

Opcode: 5 (0x05)
Format: `sconst_3`
No Change

Description: No operation

Integer stack: ..., ⇒ ..., *3:*`short`

Description: Push `short` constant 3 onto the integer stack

SCONST_4

Opcode: 6 (0x06)
Format: `sconst_4`
Integer stack: ..., ⇒ ..., *4:*`short`

Description: Push `short` constant 4 onto the integer stack

SCONST_5

Opcode: 7 (0x07)
Format: `sconst_5`
Integer stack: ..., ⇒ ..., *5:*`short`

Description: Push `short` constant 5 onto the integer stack

SCONST_M1

Opcode: 1 (0x01)
Format: `sconst_m1`
Integer stack: ..., ⇒ ..., *-1:short*

Description: Push `short` constant -1 onto the integer stack

SDIV

Opcode: 98 (0x62)
Format: `sdiv`
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ / $value_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SINC

Opcode: 120 (0x78)
Format: `sinc`, *slot_nr*, *increase*:`byte`

Description: Increases local variable of type `short` at index *slot_nr* by *increase*

SINC_W

Opcode: 168 (0xa8)
Format: `sinc_w`, *slot_nr*, *increase*:`short`

Description: Increases local variable of type `int` at index *slot_nr* by *increase*

SIPUSH

Opcode: 19 (0x13)
Format: `bspush`, $valuebyte_1$
Integer stack: ..., ⇒ ..., $(valuebyte_2<<8 + valuebyte_1)$:`int`

Description: Widen immediate `short` value ($valuebyte_2$ <<8 + $valuebyte_1$) to type `int` and push onto the integer stack

SLOAD

Opcode: 22 (0x16)
Format: `sload`, *slot_nr*
Integer stack: ..., ⇒ ..., value:`short`

Description: Loads a `short` value from the integer local variable pool at slot index *slot_nr* and pushes it onto the stack

SLOAD_0

Opcode: 23 (0x17)
Format: `sload_0`
Integer stack: ..., ⇒ ..., value:`short`

Description: Loads a `short` value from the integer local variable pool at slot index 0 and pushes it onto the stack

SLOAD_1

Opcode: 24 (0x18)
Format: `sload_1`
Integer stack: ..., ⇒ ..., value:`short`

Description: Loads a `short` value from the integer local variable pool at slot index 1 and pushes it onto the stack

SLOAD_2

Opcode: 25 (0x19)
Format: `sload_2`
Integer stack: ..., ⇒ ..., value:`short`

Description: Loads a `short` value from the integer local variable pool at slot index 2 and pushes it onto the stack

SLOAD_3

Opcode: 26 (0x1a)
Format: `sload_3`
Integer stack: ..., ⇒ ..., value:`short`

Description: Loads a `short` value from the integer local variable pool at slot index 3 and pushes it onto the stack

SMUL

Opcode: 97 (0x61)
Format: `smul`
Integer stack: ..., $value_2$:`short`, $value_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as $value_1$ and $value_2$. The result $value_1$ * $value_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SNEG

Opcode: 100 (0x64)
Format: `sneg`
Integer stack: ..., $value_1$:`short` ⇒ ..., *result*:`short`

Description: Negates the top `short` element on the integer stack. Note that any potential overflow is discarded.

SOR

Opcode: 105 (0x69)
Format: `sor`
Integer stack: ..., *value*$_2$:`short`, *value*$_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ | *value*$_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SREM

Opcode: 99 (0x63)
Format: `srem`
Integer stack: ..., *value*$_2$:`short`, *value*$_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ % *value*$_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SRETURN

Opcode: 150 (0x96)
Format: `sreturn`
Integer stack: ..., *value*:`short` ⇒ ...

Description: Exists the current method, returning a value of type `short`.

SSHL

Opcode: 101 (0x65)
Format: `sshl`
Integer stack: ..., *value*$_2$:`short`, *value*$_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ << *value*$_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SSHR

Opcode: 102 (0x66)
Format: `sshr`
Integer stack: ..., *value*$_2$:`short`, *value*$_1$:`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value*$_1$ and *value*$_2$. The result *value*$_1$ >> *value*$_2$ is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SSPUSH

Opcode: 18 (0x12)
Format: `sspush`, *valuebyte*$_2$, *valuebyte*$_1$
Integer stack: ..., ⇒ ..., *(valuebyte*$_2$*<<8 + valuebyte*$_1$*):*`short`

Description: Push immediate `short` value (*valuebyte*$_2$ <<8 + *valuebyte*$_1$) onto the integer stack

SSTORE

Opcode: 37 (0x25)
Format: `sstore`, *slot_nr*
Integer stack: ..., value:`short` ⇒ ...

Description: Pops a `short` value from the integer stack and stores it in the integer local variable pool at index *slot_nr*

SSTORE_0

Opcode: 38 (0x26)
Format: `sstore_0`
Integer stack: ..., value:`short` ⇒ ...

Description: Pops a `short` value from the integer stack and stores it in the integer local variable pool at index 0

SSTORE_1

Opcode: 39 (0x27)
Format: `sstore_1`
Integer stack: ..., value:`short` ⇒ ...

Description: Pops a `short` value from the integer stack and stores it in the integer local variable pool at index 1

SSTORE_2

Opcode: 40 (0x28)
Format: `sstore_2`
Integer stack: ..., value:`short` ⇒ ...

Description: Pops a `short` value from the integer stack and stores it in the integer local variable pool at index 2

SSTORE_3

Opcode: 41 (0x29)
Format: `sstore_3`
Integer stack: ..., value:`short` ⇒ ...

Description: Pops a `short` value from the integer stack and stores it in the integer local variable pool at index 3

SSUB

Opcode: 96 (0x60)
Format: `ssub`
Integer stack: ..., *value2* :`short`, *value1* :`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value1* and *value2* . The result *value1* - *value2* is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SUSHR

Opcode: 103 (0x67)
Format: `sushr`
Integer stack: ..., *value2* :`short`, *value1* :`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value1* and *value2* . The result *value1* >>> *value2* (logical shift right) is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

SXOR

Opcode: 106 (0x6a)
Format: `sxor`
Integer stack: ..., *value2* :`short`, *value1* :`short` ⇒ ..., *result*:`short`

Description: Pops two `short` values from the operand stack as *value1* and *value2* . The result *value1* ^ *value2* is pushed onto the integer stack as a `short`. Note that any potential overflow is discarded.

TABLESWITCH

Opcode: 148 (0x94)
Format: `tableswitch`, *default address*, *low*:`int`, *high*:`int`, *jump table...*
Integer stack: ..., *value*:`int` ⇒ ...

Description: Jump table switch. The jump table contains *high-low* branch targets, plus one 'default' branch target. When *value* is in [*low-high*] this instruction branches to the corresponding address, and branches to the 'default' address otherwise.