The HVM Reference Manual

Stephan E. Korsholm

August 2014

Contents

1 Installation

To produce an executable from Java source two modules are needed (1) The HVM Eclipse plugin and the HVM SDK. These modules are available as two separate jar files from the HVM website [3]. The HVM Eclipse plugin is named `icecaptools_x.y.z.jar` and the HVM SDK is named `icecapSDK.jar`. They can be installed from the jar files or the full source can be checked out and built and installed manually. Both methods are described in the following.

1.1 Installing the Pre-built Binaries

The HVM Eclipse plugin is installed into Eclipse by copying it into the Eclipse plugins folder. The location of this folder may vary depending on how Eclipse is installed. E.g. on Linux, if Eclipse has been installed manually, the location can be found using the command `which` like this,

```
[skr@localhost ~]$ which eclipse
/usr/local/eclipse/eclipse
[skr@localhost ~]$ ls /usr/local/eclipse/plugins/
Display all 483 possibilities? (y or n)
```

In the example above the plugins folder is located at `/usr/local/eclipse/plugins`. So to install the HVM Eclipse plugin on that system the downloaded file `icecaptools_x.y.z.jar` must be placed into that folder.

It is beyond the scope of this manual to explain how to locate the Eclipse plugins folder on all types of OSes and installations. The location varies across installations and over time as well.

1.2 Installing From Full Source

The full source of the HVM Eclipse plugin and the HVM SDK can be checked out over anonymous CVS from the HVM website [3]. These two modules must then be imported into Eclipse as standard Eclipse projects. The HVM Eclipse plugin can only be imported into Eclipse if the Eclipse Plug-in Development Environment (PDE) is installed. The HVM SDK is a basic Java project and only requires the Eclipse Java Development Environment (JDT). When HVM Eclipse plugin and the HVM SDK projects are imported into Eclipse it is now possible to build a version of the Eclipse HVM plugin from scratch. It is beyond the scope of this manual to explain how Eclipse plugins are built and deployed using PDE.

1.3 HVM Eclipse Plugin Dependencies

The HVM Eclipse plugin requires the presence of some other plugins and libraries to function. The plugins required are,

- `org.eclipse.ui`, `org.eclipse.ui.console`, `org.eclipse.ui.ide`, `org.eclipse.text`, `org.eclipse.core.runtime`, `org.eclipse.core.resources`, `org.eclipse.debug.core`, org.eclipse.debug.ui, org.junit. These are standard Eclipse components and it will be hard to find an Eclipse installation without these plugins already present

- org.eclipse.jdt.ui, org.eclipse.jdt.core, org.eclipse.jdt.launching. These plugins belong to the Java development environment. They are part of most Eclipse installations, but it may be possible to find Eclipse installations that does not have JDT installed
- org.eclipse.cdt.core. This plugin is part of the CDT Eclipse environment. This must be installed for the HVM plugin to function. This is usually not part of standard Eclipse installations, so it may be needed to add this. It is used by the HVM plugin to format auto-generated C code
- com.jcraft.jsch. The JSch plugin (Java Secure Channel) is used by the HVM Eclipse plugin to support download and remote debugging of HVM applications over ssh. This plugin is part of most standard Eclipse installations. It should be present in the plugins folder as com.jcraft.jsch_*.jar

Additionally the HVM Eclipse plugin uses the following libraries,

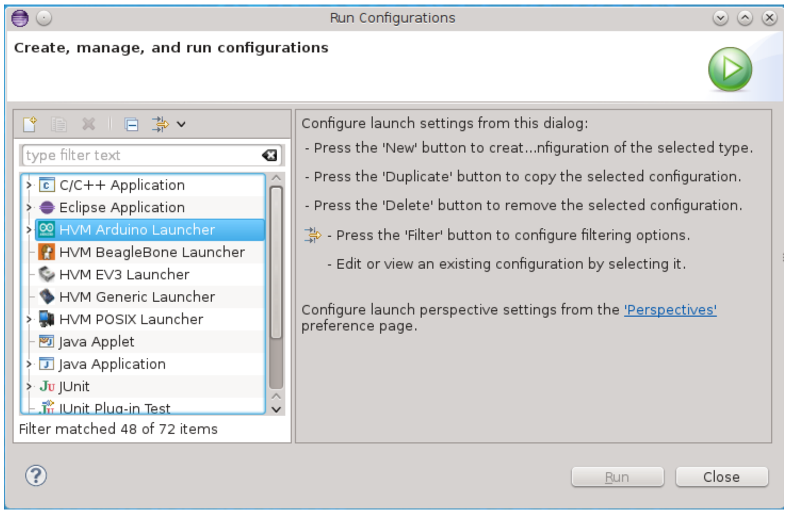
- bcel-6.0.jar. The Byte Code Engineering Library (Apache Commons BCEL) is used by the HVM Eclipse plugin to read Java class files
- commons-net-3.3.jar. Apache Commons Net library implements the client side of many basic Internet protocols. The HVM Eclipse plugin uses the Telnet protocol to deploy HVM apps to the Lego EV3
- RXTXcomm.jar. The RXTX library is a Java cross platform wrapper library for the serial port. It is used by the HVM Eclipse plugin to debug remote HVM applications over the serial port

These libraries are packaged inside the HVM Eclipse plugin and does not need to be installed manually. They are placed inside the HVM Eclipse plugin jar file in the lib folder. Inside that folder are also the sub-folders lib/Linux/, lib/Windows/, lib/Mac/ which contain some native libraries needed by the The RXTX library for the mentioned OSes.

1.4 Checking the Installation

After the HVM Eclipse plugin has been installed into Eclipse a simple way to check if the plugin is activated is to list the available run configurations inside Eclipse. To do this select the 'Run' menu and then select the 'Run Configurations...' sub-menu. A new window appears listing the run configurations known to Eclipse. Figure 1 shows an example.

The HVM Eclipse plugin has been installed correctly and activated if 5 HVM launchers are amongst the available run configurations. If these cannot be found then the plugin installation has failed. The most common cause for a



Filter matched 48 of 72 items

Configure launch settings from this dialog:
- Press the 'New' button to creat...nfiguration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Figure 1: Successful Installation

failed installation is unmet requirements (see Section 1.3). Also after installing a plugin it may be required to run Eclipse from the command-line supplying it the option `-clean`. On Linux this is simply achieved like this,

```
[skr@localhost ~]$ eclipse -clean
```

On Windows it is more difficult. It is beyond the scope of this manual to explain how to locate and start Eclipse from the command line on Windows installations.

If the plugin fails to install please contact the HVM developers to report the problem.

2 Background

The HVM is a Java-to-C compiler with an embedded interpreter. It can translate a Java program into a C program. The input to the translation process is a set of Java source files and the output is a set of C source files. The purpose of the HVM is to enable the Java programming language on low resource embedded devices. It produces self-contained ANSI C code that can be compiled using a cross compiler for the desired target. The minimal resource requirements on the target processor is 10 kB of ROM and 512 bytes of RAM, but in order to execute reasonable sized programs 32 kB of ROM and 2 kB of RAM are required. Figure 2 depicts the full process.

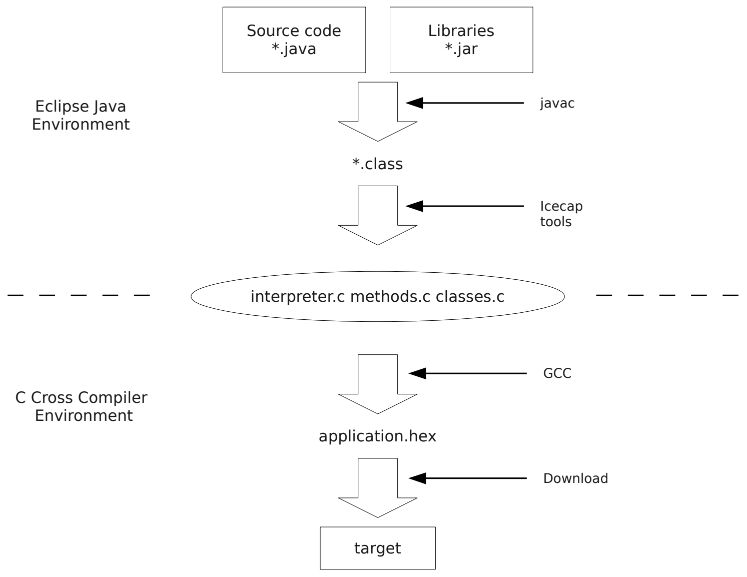


Figure 2: HVM Development environment

2.1 Interpretation vs. Compilation

Consider the following Java method,

```
private static short max(short i, short j) {
        if (i > j)
        {
                return i;
        }
        return j;
}
```

If compiled into C this method will get translated into,

```
int32 Main_TestIntVsComp_max(int32 *fp, int16 i, int16 j) {
        int16 s_val1;
        int16 s_val0;

        s_val1 = (int16) i;
        s_val0 = (int16) j;
        if (s_val1 <= s_val0) {
                goto L9;
        }
        s_val1 = (int16) i;
        return (uint16) s_val1;
        L9: s_val1 = (int16) j;
        return (uint16) s_val1;
}
```

The introduction of the extra local variables `s_val0` and `s_val1` might seem redundant, but when the above code gets compiled using a GCC cross compiler this gets optimized away.

Instead of translating it into C, the HVM can also prepare the method for interpretation. In that case the method gets translated into the following,

```
const unsigned char Main_TestIntVsComp_max[13] PROGMEM = {
        0x1A,0x1B,0xA4,0x00,0x07,0x1A,0xAC,0x01,0x00,0x1B,0xAC,0x01,0x00
};
```

These are just the byte codes of the method embedded into a C `char` array and as such are not directly executable. To execute these bytecodes the interpreter is required. The interpreter is basically a big loop reading one byte code at a time and performing the operation defined by that byte code. The HVM interpreter loop looks like this,

```
loop: while (1) {
      unsigned char code = pgm_read_byte(method_code);
      switch (code) {
#if defined(ICONST_M1_OPCODE_USED) ||
    defined(ICONST_0_OPCODE_USED) ||
    defined(ICONST_1_OPCODE_USED) ||
    defined(ICONST_2_OPCODE_USED) ||
    defined(ICONST_3_OPCODE_USED) ||
    defined(ICONST_4_OPCODE_USED) ||
    defined(ICONST_5_OPCODE_USED)
        case ICONST_M1_OPCODE:
        case ICONST_0_OPCODE:
        case ICONST_1_OPCODE:
        case ICONST_2_OPCODE:
        case ICONST_3_OPCODE:
        case ICONST_4_OPCODE:
        case ICONST_5_OPCODE:
            *sp++ = code - ICONST_0_OPCODE;
            method_code++;
            continue;
#endif
        ....
       }
    }
```

For each byte code there is a case statement. Only very few are shown above.

Only those parts of the interpreter that is used is included in the final executable. The translation phase keeps track of which bytecodes are prepared for interpretation and generates a header file with a set of defines for each byte code used. These defines are used to select which parts of the interpreter get included.

The reason to support both interpretation and compilation is that each execution method has some different strengths than the other: interpretation yields tight code thus saving on ROM usage, but execution is slow; on the other hand compilation requires a bit more ROM but execution is significantly faster. A more detailed comparison between the two execution styles can be found in Section 8. It is possible from the UI to select which methods gets compiled and which gets interpreted. Interpretation is the default execution method.

2.2 The Dependency Extent

Java programs always reference Java libraries, both in the JDK being used but also in third party libraries added to the project as external dependencies. Since the HVM targets low resource embedded platforms it is of importance to only include those library parts that may be used by the application - either directly or indirectly. The first phase of the translation is to compute this set of required classes and methods. This set is called the *Dependency Extent.* The HVM cannot compute the exact dependency extent, so it will always add a bit more than what may get executed at run-time. The HVM cannot handle when the dependency extent gets too large, and this can very easily happen. E.g. the usual Hello World Java program contains the line `System.out.println` and because of the way that the Java standard libraries are organize the System class has a very big dependency extent - too big for the HVM to handle. How the HVM Hello World then looks is described in Section 3.

The dependency extent of `System.out.println` must include functionality to write a string of characters to the console on a standard Host PC - this makes no sense on a low-resource embedded device that usually does not have a console or maybe only a serial line.

If the dependency extent of a program becomes too large for the HVM to handle a *Dependency Leak* has occurred. Many of the `java.util.*` does not leak and can be used in embedded programs. Which classes that leak and which that don't may vary from SDK to SDK. If a dependency leak occurs, the HVM will tell the programmer which line introduced the leak.

2.3 Translation output

The translation process produces a set of C source files. Some of these source files are dependent on the application being translated and some are always the same. The full list of files produced during the translation phase is the following,

- `ostypes.h` Defines some commonly used data types e.g. `uint16` and similar. Contains such a section for both 8/16/32 and 64 bit architectures. Most new embedded platforms will be able to reuse an existing section. See Section 3.1 for how to enable the correct section
- `types.h` Defines data types for objects, classes, methods and similar for when these entities are translated into C. It also contains the two defines `PRE_INITIALIZE_CONSTANTS` and `PRE_INITIALIZE_EXCEPTIONS`. For platforms with very limited amounts of RAM these should not be defined. For other platforms it is recommended to define these constants
- `icecapvm.c` Contains the main function and starts up the execution
- `methodinterpreter.c, methodinterpreter.h` Contains the byte code interpreter
- `gc.c, gc.h, allocation_point.c, allocation_point.h` Contains the memory management system. See more in Section 4.1.
- `print.c` Contains native functionality to print simple messages
- `natives_allOS.c` Is a big utility tool box. It contains a number of helper functions used during run time and common to all architectures. Most of the functions together make up a rudimentary implementation of the standard Java SDK native layer - just enough to run the most used parts of the standard Java SDK classes such as `java.lang.Object` and `java.lang.Class`

- `natives_XXX.c` These files contains helper functions specific to each architecture, e.g. how to allocate the main stack
- `XXX_interrupt.s` If threading is used an implementation of how to perform a context switch is required. These files contain native assembler instructions to perform context switching on a variety of targets. See Section 5.1 for more information
- `native_scj.c` Contains a few functions for running SCJ on POSIX compliant platforms. See Section 5 for more information
- `methods.c, methods.h` All files above are the same after all translations. These file however, change if the Java source changes. They are described in more detail below
- `classes.c, classes.h` These files also change after each translation. They are described in more detail below

2.3.1 The file `methods.c`

This file contains a C version of all Java methods included in the dependency extent. Each method is included either as an actual C function or as an array of bytecodes (see Section 2.1).

Then follows a list of string declarations with the names of each method. These strings are only used during debugging or when printing stack traces in the case of an uncaught exception. They can be excluded from the file in order to save ROM space. For more information please refer to the example in Section 6.1.

Then follows information about each method. This information is stored in an array. Approx 20 bytes is stored per method. It is used by the interpreter and contains information about where the method byte code is stored, if it has any exception handlers and how much stack is required, etc.

This file also contains all the constants being used by the program, e.g. `String` or `long` constants.

Finally it contains information about in which sequence to initialize classes, about exception handlers and about which methods are the default constructors.

With a few exceptions all this information is stored in ROM.

2.3.2 The file `methods.h`

This auto-generated header file contains a list of definitions. First appears a list of defines mapping each method to a unique id. This id is the method's offset into the method information array described above. Finally follows a list of defines enabling each byte code that is used by the application. These defines enable the corresponding case statement in the interpreter that is required to execute the byte code.

2.3.3 The file `classes.c`

At the start of the file is an array placed in RAM holding all the static class variables in all classes in the dependency extent. It follows that all static class data is placed in one consecutive area defined by this array. The file `classes.h` (see below) contains a struct defining the layout of this area. Then follows an array of offsets into the static class data at which places are placed reference values. This is used by various classes in the HVM reflection API (see Section 4.6).

After that follows a sequence of constant strings with the names of each class. These strings are only used during debugging or when printing stack traces in the case of an uncaught exception. They can be excluded from the file in order to save ROM space. For more information please refer to the example in Section 6.1.

After that follows an array of class information containing the super class of a class, the size of object instances, etc.

Finally follows the inheritance matrix used to implement the `instanceof` and `checkcast` bytecodes. This matrix contains information about the subclass relationship between class in compressed format.

Wherever possible all data in this file is placed in ROM.

2.3.4 The file `classes.h`

This file contains a number of defines mapping each class to a unique id. This id is the offset of the class into the array of class information described above. Then follows for each class a struct defining the layout of class instances. This struct is used by both the compiled and interpreted methods to implement the `putfield` and `getfield` bytecodes. Finally follows on larger struct defining the location of static class data in all classes as they appear the static class data memory area described above.

3 Hello World

The HVM plugin can translate Java projects created in Eclipse using ordinary methods, e.g. the JDT New Project wizard. In this example we will create the simplest possible Java project first. Figure 3 shows the Java package explorer in Eclipse how it looks after the project has been created.

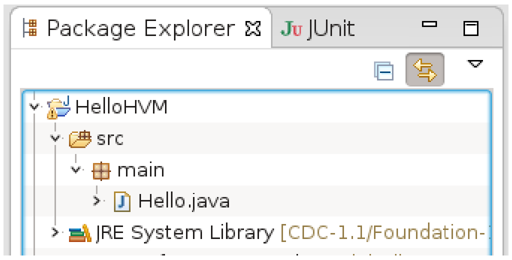


Figure 3: Simplest Possible

The `Hello` class contains a single empty main method:

```
package main;

public class Hello {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
        }
}
```

The Hello class is placed inside its own package. This is important since the HVM translator requires that the main method is in a class that is in a package - the default package does not work. To translate this (empty) Java program into C, first expand the Hello class in the project explorer to reveal the main class. This is shown in Figure 4.

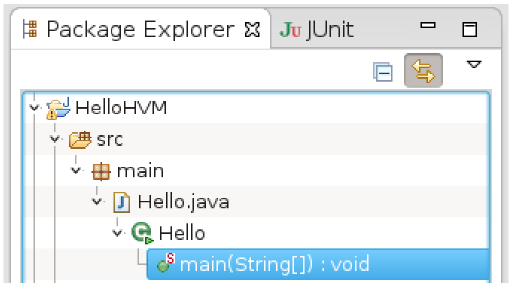
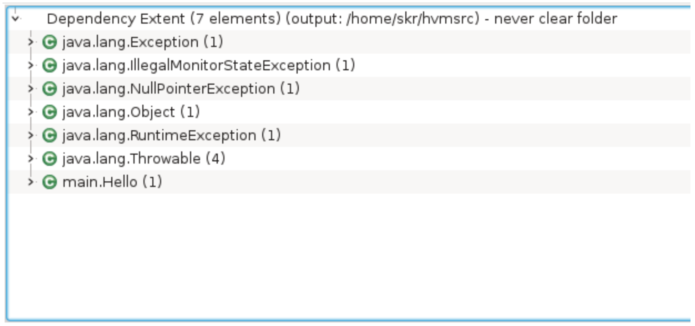


Figure 4: The main method

Then right click on the main class. From the drop down menu select the 'Icecap tools' entry and select the 'Convert to Icecap Application' action. As a result the Dependency View appears as illustrated in Figure 5.



```
Dependency Extent (7 elements) (output: /home/skr/hvmsrc) - never clear folder
  java.lang.Exception (1)
  java.lang.IllegalMonitorStateException (1)
  java.lang.NullPointerException (1)
  java.lang.Object (1)
  java.lang.RuntimeException (1)
  java.lang.Throwable (4)
  main.Hello (1)
```

Figure 5: The Dependency View

This Eclipse View displays the following information:

- The dependency extent is comprised of 7 classes. This may be surprising since the main method in question does not do anything. But the reason is that the empty main method contains a single `return` statement, and this byte code may throw an `IllegalMonitorStateException`, and the constructor of that Exception class may throw a `NullPointerException`. The remaining classes are super classes of said classes and gets included in the extent as well

- The C code produced is placed in `/home/skr/hvmsrc`. This destination folder can be changed by right clicking anywhere inside the dependency view. After changing the destination folder it is necessary to re-translate the application before the files will appear in the new location. This destination folder may be a folder on Eclipse control or it may not, that is irrelevant to the HVM tools

- The destination folder is not cleared before each translation. New files will overwrite existing ones. By right clicking anywhere inside the dependency view the 'Clear output folder' setting can be activated. This works better with some cross compilers

To see which methods are included below each class, the class can be expanded in the tree view. The user may right click any element in the dependency view and mark it for compilation or interpretation. Each element can also be excluded from the extent. E.g. all exception classes in Figure 5 can actually be excluded since we know that this method will never throw an exception at run-time. After changing various properties in the dependency view it is necessary to re-translate the application manually before the changes take effect.

Now a C version of the very simple Java program has been created. In order to run the program it must be compiled using a C cross compiler for the intended target. Using an ATMega1280 8 bit micro controller from AVR the resulting executable takes up approx 6 kB of ROM and 600 bytes of RAM. The ROM space is used for the minimal interpreter configuration. The RAM space is used for a 100 byte minimal Java heap, the main stack and some additional required variables. The ROM offset will increase as methods and classes are added and the RAM offset will increase as the heap is increased.

The details about compiling the produced C code into the final executable is described in the next section.

3.1 Compiling

When the HVM Eclipse Plugin has translated the input Java source into C code, the program is still not ready to execute - the C code also needs to be translated into executable machine code for the target in question. In a sense the HVM uses C code as an intermediate format.

The HVM Eclipse plugin contains a handful of Eclipse Launcher configurations that knows about how to compile the emitted C code, but in general the HVM does not take responsibility for compiling the emitted C code - this is left to the embedded developer, using his favorite embedded development environment and compilers.

The emitted C code can be compiled for any platform using any C compiler. The launcher configurations can be viewed as examples. The following sections describe how to compile the emitted C code for a standard PC host platform (Linux) and for an embedded target (8 bit AVR) - both using the command-line. Then follows a description about how to use the preconfigured launchers.

3.1.1 Compiling Using the Command-line

Linux usually contains an installed GCC tool-chain, or if not one can easily be installed. To use that for compiling a final executable the developer must open a command prompt and change directory into the destination folder where the HVM Eclipse plugin has emitted the generated C code. Using a single command the executable can be compiled like this,

```
gcc -Wall -pedantic -Werror -g -O0 -DPC64 -DREF_OFFSET -DPRINTFSUPPORT
    -DJAVA_HEAP_SIZE=6500 -DJAVA_STACK_SIZE=1024 classes.c  icecapvm.c
    methodinterpreter.c  methods.c gc.c natives_allOS.c natives_i86.c
    allocation_point.c print.c
```

The command-line above has been split into 4 lines so that it is viewable here. The command-line contains the following,

- `-Wall -pedantic -Werror` These options just tells the GCC compiler to only accept standard C code constructs and stop if a warning is encountered
- `-g -O0` These options make the final executable debugable in e.g. Eclipse. The developer will be able to debug and single step the interpreter or compiled methods. This resembles debugging normal C code at the assembler

level, and is not very useful. The HVM also supports Java level debugging (see Section 4.7). For a release build these options should be `-Os` or -O3 - the former produces the smallest possible executable and the latter will produce the fastest possible executable

- `-DPC64` This options is used in `ostypes.h` to define the correct basic data types. If you are using a 32 bit platform use `-DPC32` instead. Other platforms already supported are `WIN32, CR16C, V850ES, SAM7S256, AVR`
- `-DREF_OFFSET` For 64 bit platforms the width of pointers are 8 bytes. These cannot be stored on the HVM Java stack as stack cells are only 4 bytes wide. Using this option the HVM will translate pointers into heap offsets of 4 bytes. It is required to add this option on 64 bit Windows platforms using cygwin. For some reason it does not seem to be required on Linux platforms. It has a negative effect on execution speed. The HVM is not intended for 64 bit platforms, but supports them anyway for debugging purposes
- `-DPRINTFSUPPORT` If the target has libc available this can be enabled
- `-DJAVA_HEAP_SIZE=6500` Defines the size of the Java heap. The default is 64 kB. This example allocates 6500 bytes for the heap
- `-DJAVA_STACK_SIZE=1024` Defines the size of the main stack in 4 byte cells. So this example allocates 4 kB for the main stack. For Linux and Windows this should not be smaller than 4 kB. For embedded platforms it can be as small as 256 bytes, but that depends on the application
- Then follows the list of files making up the application. All these files are generated by the HVM Eclipse plugin (see Section 2.3). Note that the file `natives_i86.c` has been selected since we are on a standard PC host and that the file `x86_64_interrupt.s` has been omitted since we are not using threads
- `-DSUPPORTGC` This is not used above but must be included when compiling applications making use of the reflection system (see Section 4.6)

This will produce an executable in `a.out` that can be immediately executed. If using Cygwin and Windows the procedure is exactly the same.

As the intended targets for the HVM output are low resource embedded devices it is also interesting to see how this gets compiled for e.g. an 8 bit MCU. Here we will demonstrate using a ATMega1280 from AVR. A port of the GCC tool chain exists for the AVR, called `avr-gcc`. Compiling the same source using this compiler looks like this,

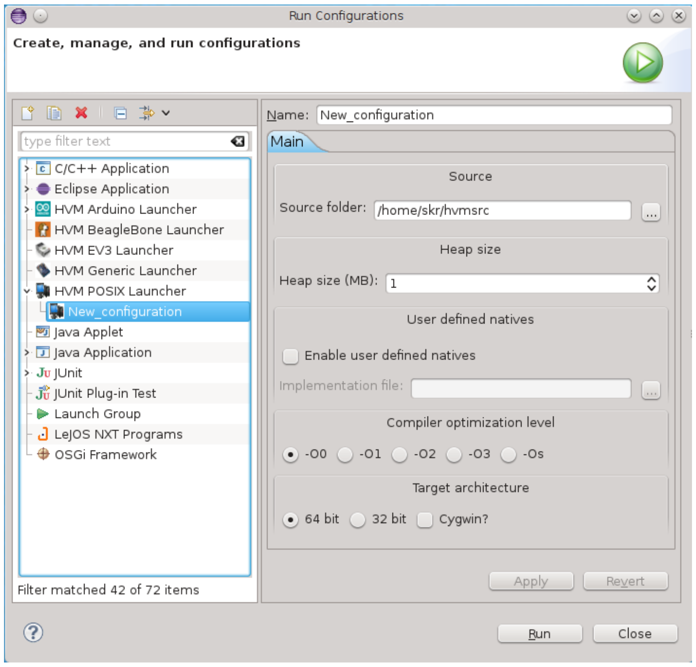
```
[skr@localhost hvmsrc]$ make -f makefile
avr-gcc -c -g -o classes.o classes.c -DF_CPU=7372800L -mmcu=atmega1280 -Wall
        -pedantic -Os -DJAVA_HEAP_SIZE=100 -DJAVA_STACK_SIZE=96
avr-gcc -c -g -o avr_interrupt.o avr_interrupt.s ... ...
avr-gcc -c -g -o icecapvm.o icecapvm.c ...
avr-gcc -c -g -o methodinterpreter.o methodinterpreter.c ...
avr-gcc -c -g -o methods.o methods.c ...
avr-gcc -c -g -o gc.o gc.c ...
avr-gcc -c -g -o print.o print.c ...
avr-gcc -c -g -o natives_allOS.o natives_allOS.c ...
avr-gcc -c -g -o allocation_point.o allocation_point.c ...
avr-gcc -c -g -o natives_avr.o natives_avr.c ...
avr-gcc -c -g -o ATMega1280_natives.o ATMega1280_natives.c ...
avr-gcc -o main classes.o avr_interrupt.o icecapvm.o methodinterpreter.o
                methods.o gc.o print.o natives_allOS.o allocation_point.o
                natives_avr.o ATMega1280_natives.o ...
avr-size -C --mcu=atmega1280 main
AVR Memory Usage
----------------
Device: atmega1280

Program:    6784 bytes (5.2% Full)
(.text + .data + .bootloader)

Data:        631 bytes (7.7% Full)
(.data + .bss + .noinit)

avr-strip main
avr-objcopy -O ihex -R .eeprom main main.hex
```

Only the first line includes all the options. On the following lines these are substituted for ... for easier viewing here. After linking is done the tools `avr-size,` `avr-strip` and `avr-objcopy` are used to turn the application into downloadable format. The `hex` file can now be downloaded to the target using e.g. `avrdude`.

The HVM Eclipse plugin knows nothing about what the emitted C code is going to be deployed upon and the AVR tools used above are not part of the HVM Eclipse plugin. This is just one example of how the emitted C code may be deployed to an embedded target. For other targets it will likely be completely different. The only thing the HVM Eclipse plugin does to facilitate this is to produce self contained C code that is specifically designed to be as portable as possible.

3.1.2 Compiling Using the Launchers

The HVM Eclipse plugin contains a selection of Eclipse Launcher configurations that support the building and deployment of the emitted C code to a limited set of well known targets. To use a launcher select the 'Run Configurations...' sub menu from the 'Run' menu in Eclipse. This will open up a new window listing the available launchers. The HVM contributes 5 launcher configurations (See Figure 1).


Run Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- Eclipse Application
- HVM Arduino Launcher
- HVM BeagleBone Launcher
- HVM EV3 Launcher
- HVM Generic Launcher
- HVM POSIX Launcher
  - New_configuration
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- LeJOS NXT Programs
- OSGi Framework

Filter matched 42 of 72 items

Name: New_configuration

Main

Source

Source folder: /home/skr/hvmsrc

Heap size

Heap size (MB): 1

User defined natives

[ ] Enable user defined natives

Implementation file:

Compiler optimization level

[✓ -O0] -O1 -O2 -O3 -Os

Target architecture

[✓ 64 bit] 32 bit [ ] Cygwin?

Figure 6: The POSIX Launcher

Figure 6 shows the launcher used to compile for and run on a PC host. The launcher supports Linux 32 or 64 bit and Windows XP, 7 and 8 32 or 64 bit using Cygwin. Before launching the developer must select the folder where the HVM plugin has emitted the C code. Then set the heap size. If the program contains native methods (see Section 4.2) the developer may enable the inclusion of a native implementation file. Then the optimization level can be selected. Then the developer must chose whether the PC host is 32 or 64 bit and finally if cygwin are used for compilation. the HVM Generic launcher is not implemented yet but the EV3, Beaglebone and Arduino launchers are fully functional.

The EV3 and Beaglebone launchers require an IP address of the target. Thus the launcher has an additional field where the developer should but the IP address of the device. The EV3 can be accessed over TCP/IP using a wireless USB dongle. The Beaglebone already has an ethernet link over USB.

The Arduino launcher deploys over the serial line, so the developer must enter which COM port to use.

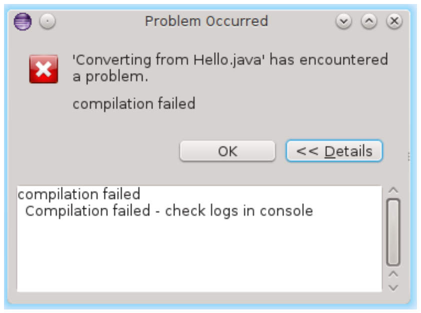
The EV3 and Beaglebone launchers use the same cross compiler. Before using these launchers the developer must install the proper cross compiler and ensure that these are in the path environment variable. If launching to the Arduino the proper cross compiler must be installed manually as well.

3.1.3 Completing the Hello World Example

The Hello World example listed in Section 3 was just an empty main method. To make it a real Hello World application we must print out a message. To illustrate a common mistake we will try to execute the following:

```
package main;

public class Hello {

        public static void main(String[] args) {
                System.out.println("Hello World");
        }
}
```

Figure 7 shows that an error occurs.



'Converting from Hello.java' has encountered a problem.

compilation failed

```
compilation failed
 Compilation failed - check logs in console
```

Figure 7: Failed translation

If the logs in the console are visited, the cause of the error is listed:

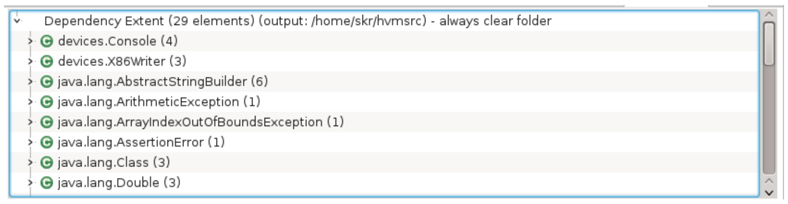
```
    ...
        at sun.reflect.Reflection.verifyMemberAccess(Reflection.java:126)
        at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:96)
        at sun.reflect.misc.ReflectUtil.ensureMemberAccess(ReflectUtil.java:103)
        at java.util.concurrent.atomic.AtomicReferenceFieldUpdater$AtomicReferenceFieldUpdaterImpl.<ir
        at java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater(AtomicReferenceFieldUpda
        at java.io.BufferedInputStream.<clinit>(BufferedInputStream.java:79)
        at java.io.BufferedInputStream.<init>(BufferedInputStream.java:183)
        at java.lang.System.initializeSystemClass(System.java:1188)
        at java.lang.System.setProperty(System.java:793)
        at sun.misc.Version.init(Version.java:52)
        at sun.misc.Version.<clinit>(Version.java:48)
        at java.lang.System.initializeSystemClass(System.java:1183)
        at main.Hello.main(Hello.java:6)
Dependency leak, failed to compile Hello
```

This output can be found by switching to the Eclipse Console View and select the `Icecap tools messages` console. The way to find this console is to locate the console icon (resembling a computer display) to the right in the Eclipse Console View and click on the drop down button.

What is apparent from the console output above is that a dependency leak has occurred (see Section 2.2). And the reason is that the dependency extent of `System.out.println` is too large for the HVM analyzer to handle. Instead we should use `devices.Console.println` like this:

```
package main;

public class Hello {

        public static void main(String[] args) {
                devices.Console.println("Hello World");
        }
}
```

The `devices` package is part of the HVM SDK downloaded as the library `icecapSDK.jar` (see Section 1). This library must now be added to the build path of the source Java project. This can be achieved by adding a folder to the project called e.g. `lib`, placing `icecapSDK.jar` inside that folder and adding the library to the project build path. If we attempt to translate again the process will finish with a result similar to what is illustrated in Figure 8.



```
Dependency Extent (29 elements) (output: /home/skr/hvmsrc) - always clear folder
  devices.Console (4)
  devices.X86Writer (3)
  java.lang.AbstractStringBuilder (6)
  java.lang.ArithmeticException (1)
  java.lang.ArrayIndexOutOfBoundsException (1)
  java.lang.AssertionError (1)
  java.lang.Class (3)
  java.lang.Double (3)
```

Figure 8: HelloWorld Dependencies

The dependency extent has now grown from 7 to 29 classes. The additional classes are used to print out text strings on the console. All the exception classes can be excluded from the extent to minimize it. This application can now be executed on a PC host using the HVM POSIX launcher (see Figure 6).

4 Features

The HVM contains some features not known from standard Java and some features of standard Java works differently in the HVM. The following sections cover these features.

4.1 Memory Management

The HVM does not support standard Java garbage collection. Instead it relies on the SCJ scoped memory model for managing allocation and deallocation of memory resources [11]. The SCJ profile is covered in more detail in Section 5. For non-SCJ programs it means that data gets allocated consecutively in memory areas and released in chunks by releasing whole memory areas at once. As an example consider this extension of the Hello World application,

```
package main;

public class Hello {

        public static void main(String[] args) {
                int count = 0;
                while (true) {
                        devices.Console.println("Hello World [" + count + "]");
                        count++;
                }
        }
}
```

If we run this program on a standard PC host and give it 3 kB of heap space the result is,

```
[skr@localhost hvmsrc]$ ./a.out
Hello World [0]
Hello World [1]
Hello World [2]
Hello World [3]
Hello World [4]
Exception in thread "" java.lang.OutOfMemoryError
   at java.lang.AbstractStringBuilder.<init>(:0x0009)
   at java.lang.StringBuffer.<init>(:0x0009)
   at main.Hello.main(:0x0009)
```

The reason is that as new strings are being created in the heap through the use of the '+' operator, they never get released or garbage collected and the heap quickly runs out of space. The can be fixed using memory areas in the following manner,

```
package main;

import vm.Memory;

public class Hello {

        public static void main(String[] args) {
        Memory mainArea = Memory.getHeapArea();

        int waterMark = mainArea.consumedMemory();
        while (true) {
            devices.Console.println("Hello World [" + count + "]");
            count++;
            mainArea.reset(waterMark);
        }
    }
}
```

Now the program runs continuously without running out of memory.

Instead of allocating in the default heap area, new areas can be created, and the program can switch to using that area for allocations. This is illustrated in the following,

```
Memory mainArea = Memory.getHeapArea();
int start = Memory.allocateBackingStore(SCRATCHPADSTORESIZE);
Memory scratchPadStore = new Memory(start, SCRATCHPADSTORESIZE, "scratchPadStore");
mainArea.switchToArea(scratchPadStore);
...
```

The use of memory areas require more care than the use of a standard garbage collector. All the usual problems one can get when using explicit memory management (e.g. dangling pointers) can occur here as well. The memory area concept for Java is a breach of Java security that the HVM developer should understand and use with care.

In practice many embedded applications for low resource systems have very simple memory allocation requirements. In such scenarios a full garbage collection system is not required and the memory area concept will suffice.

4.2 Native Methods

Native methods written in C can be called from Java space, and parameters of any type can be passed between Java and C space. Consider the following native method declaration,

```
package main;

public class TestNative {

        public static void main(String[] args) {
                foo();
        }

        private static native void foo();
}
```

Attempting to launch this application using the HVM POSIX Launcher (see Section 3.1.2) will yield an error,

```
/tmp/ccP9krLn.o:(.rodata+0x3ea): undefined reference to `n_main_TestNative_foo'
```

The reason is that the program calls a native method, but this method is not implemented anywhere. To get rid of the error the developer must implement the native method in C and add the source of the implementation to the build command. The name of the native method that must be implemented is built of the name of the package, the name of the containing class and the Java name of the native method it self - this name is then prepended with `n_`. Thus the name of the native method above becomes `n_main_TestNative_foo`. When the HVM interpreter calls the native method it supplies one argument to the call which is the Java stack pointer. When the native method returns it must return the value `-1` to indicate successful completion. Thus the full signature for the native method above becomes

```
extern int16 n_main_TestNative_foo(int32 *sp);
```

No matter how many parameters there are to the native method call and no matter the return type, the signature of the native implementation will always be the same: One parameter of type `int32` and the return type of `int16`. These types are defined in `ostypes.h`. This header file must be included in the native methods implementation file.

The developer can now implement this method in an additional C source file and add this source file to the build in the HVM POSIX Launcher.

4.2.1 Native Methods - Passing Values

In the following example the native method is defined to take a single integer parameter,

```
package main;

public class TestNative {

        public static void main(String[] args) {
                foo(42);
        }

        private static native void foo(int x);
}
```

The signature of this native method will still be,

```
extern int16 n_main_TestNative_foo(int32 *sp);
```

The actual parameter 42 can be retrieved from the stack pointer (`sp`). In this case it will be at the first index on the stack because it is the first parameter. A C implementation which uses this value looks like,

```
int16 n_main_TestNative_foo(int32 *sp)
{
    int32 x = sp[0];
    ...
}
```

Other basic types are passed in the following manner,

- `boolean, byte and short` These are mapped to the C types `uint8, int8` and `int16`
- `char` A HVM char is 4 bytes. It is mapped to `int32`
- `float` If supported by the C run time this is mapped to `float`
- `long` A long value takes up two consecutive 4 byte stack cells. The first is the most significant 4 bytes of long value and the next is the least significant 4 bytes
- `double` A double takes up two consecutive 4 byte stack cells. The first is the most significant 4 bytes of long value and the next is the least significant 4 bytes. If supported by the C run time these 8 bytes can be concatenated and cast two a `double`

References to arrays and objects can be passed to native functions as well. How these can be used from C space is described below

4.2.2 Native Methods - Passing Objects

Object references are passed in a single 4 byte stack cell. For 32 bit or smaller architectures this value is the actual pointer into the heap where the object resides. HVM objects contain a 16 bit header (the class index). The object layout depends on the class type. C structs are automatically generated for each class type in the file `classes.h` (see Section 2.3.4). For 64 bit architectures this value is not an actual reference if the application has been compiled with option `-DREF_OFFSET`. It can be converted into an actual reference using the following statement,

```
Object* obj = HEAP_REF(Object*, (pointer)sp[0]);
```

4.2.3 Native Methods - Passing Arrays

Array references are equivalent to object references. This means that the macro `HEAP_REF` has to be used to turn the value into an actual pointer on 64 bit architectures. HVM arrays contain the same 16 bit header as normal objects. On the next 16 bits follows the number of elements in the array (or length of the array). Then follows the elements of the array in consecutive order.

4.3 Hardware Objects

Hardware Objects according to [9] are supported in the HVM, both for interpreted and compiled methods. Hardware Objects is a facility for accessing raw memory from Java space, but it's often used to control IO devices through access to device registers of the underlying micro controller. On many embedded targets (e.g. the ATMega2560 from Atmel) the access to device registers has to take place using special purpose machine instructions, a read or write through a load/store instruction will not have the desired effect. For this reason the HVM cannot make an implementation of hardware objects that simply accesses memory using standard C idioms. The HVM delegates the actual access to native C functions implemented in a thin hardware specific interface layer. The function for writing a bit to IO has the following signature,

```
    void writeBitToIO(int32 address,
                      unsigned short offset,
                      unsigned char bit);
```

An implementation of this function now has to be given for each target. When access to fields of Hardware Objects is done from Java space, the interpreter or Java-to-C compiler will make sure that appropriate functions are called. If enabling inlining in the C compiler, the function will be inlined and executed very efficiently. This function is implemented in the target specific file `natives_XX.c` (see Section 2.3).

```
package ATMega1280.HardwareObjects;

import vm.Address32Bit;
import vm.HardwareObject;

/**
 * Hardware Object for I/O ports on the AVR ATMEGA 1280.
 *
 */
public class Port extends HardwareObject {
        // page 411-414
        public static final int PORTA = 0x020;
        public static final int PORTB = 0x023;
        public static final int PORTC = 0x026;
        public static final int PORTD = 0x029;
        public static final int PORTE = 0x02c;
        public static final int PORTF = 0x02f;
        public static final int PORTG = 0x032;
        public static final int PORTH = 0x100;
        public static final int PORTJ = 0x103;
        public static final int PORTK = 0x106;
        public static final int PORTL = 0x109;

        public static final byte PIN0 = 0x0;  // 0x1
        public static final byte PIN1 = 0x1;  // 0x2
        public static final byte PIN2 = 0x2;  // 0x4
        public static final byte PIN3 = 0x3;  // 0x8
        public static final byte PIN4 = 0x4;  // 0x10
        public static final byte PIN5 = 0x5;  // 0x20
        public static final byte PIN6 = 0x6;  // 0x40
        public static final byte PIN7 = 0x7;  // 0x80

        public byte PIN;
        public byte DDR;
        public byte PORT;

        /**
         * Initialization of a I/O port. Addresses are defined as PORTA - PORTL
         * @param address The address of the port (PORTA- PORTL).
         */
        public Port(int address) {
                super(new Address32Bit(address));
        }
}
```

Figure 9: ATMega1280 Hardware Object

Figure 9 shows an example of the use of hardware objects for accessing device registers on an ATMega1280. The class `Port` can now be instantiated e.g. to control the LED in the following manner,

```
public class LED
{
   private static Port p;

   /**
    * Initiate LEDs on EasyAVR M1280 SK board.
    */
   public static void initLEDs()
   {
      p = new Port(Port.PORTH);
      p.DDR = (byte)0xFF;
   }

   /**
    * Toggle a LED.
    * LEDs numbers are in reversed order.
    * @param byte ledNo 1 - 8
    */
   public static void toggleLED(byte ledNo)
   {
      p.PORT ^= (1<<(ledNo-1));
   }
}
```

A call to `initLEDs` will create a new hardware object of type `Port`. This object is not placed in the heap put at the address supplied in the constructor (`Port.PORTH` which is 0x100). The statement `p.DDR = (byte)0xFF;` now writes the value 0xFF to the `DDR` instance variable. Since the object is placed at address 0x100 and the `DDR` field variable is the second byte field variable in the class `Port` the value is actually written to address 0x101 - this will on a ATMega1280 set port H as being an output port.

4.4 Native Variables

Native variables is a way to map static Java class variables to C variables and have the Java-to-C compiler produce code that reads and writes to the corresponding C variable instead of the class variable as would be the usual case. This mapping can be controlled by the developer through the `IcecapCVar` annotation, the definition included here:

```
public @interface IcecapCVar {
        String expression() default "";
        String requiredIncludes() default "";
}
```

As an example of its use, assume that on a particular device parts of the software is implemented in C. In this example C space contains a global variable `uint32 systemTick;`. This variable is continuously updated every 10 ms and used to measure system time. Using native variables this counter can be accessed directly from Java space in the following manner:

```
public class XXX {
    @IcecapCVar(expression ="systemTick",
                requiredIncludes = "extern uint32 systemTick;")
    static int tick;

    public void test()
    {
        if (tick % 100 == 0)
        {
            ...
        }
    }
}
```

The optional `expression` attribute defines to which C expression the static variable is mapped and the optional `requiredIncludes` attribute allows the developer to add includes or external declarations required to compile the `expression`. Native variables can only be accessed from compiled code.

Another use of native variables is to read/write to device registers (see Section 4.3). The following code snippet shows how native variables can be used as an alternative to hardware objects to access device registers on an ATMega1280,

```
package Main;

import icecaptools.IcecapCVar;
import icecaptools.IcecapCompileMe;

public class ArduinoHWRegs {

        @IcecapCVar(expression = "PINB", requiredIncludes = "#include \"avr/io.h\"\n")
        static byte PINB;

        @IcecapCVar(expression = "PORTB", requiredIncludes = "#include \"avr/io.h\"\n")
        static byte PORTB;

        @IcecapCVar(expression = "DDRB", requiredIncludes = "#include \"avr/io.h\"\n")
        static byte DDRB;

        public static void main(String[] args) {
                testPort();
                while (true) {
                        ;
                }
        }

        @IcecapCompileMe
        private static void testPort() {
                DDRB = (byte) 0xFF;
                PORTB = (byte) 0xAA;
        }
}
```

The method using the native variables (`testPort`) has been annotated with the `@IcecapCompileMe` annotation. This is important since native variables can only be accessed from compiled methods (see Section 2.1). The feature of native variables is also implemented in the HaikuVM [10].

4.5 Interrupt Handlers

The HVM SDK (see Section 1) contains classes and infrastructure for handling interrupts in Java space. The class `vm.InterruptDispatcher` defines how to register interrupt handlers. To initialize the interrupt handling functionality the following must be done,

```
InterruptDispatcher.numberOfInterrupts = 57;
InterruptDispatcher.handlers = new InterruptHandler[numberOfInterrupts];
InterruptDispatcher.init();
```

The above will work for an ATMega1280. The following example shows how to register an interrupt handler for the ATMega1280 debounced keys,

```
InterruptDispatcher.registerHandler(new DebouncedKey_1_Handler(), 6);
```

The debounced key interrupt has vector number 6. The handler can now be implemented like,

```
public static class DebouncedKey_1_Handler implements InterruptHandler
{
      @Override
      @IcecapCompileMe
      public void handle()
      {
         LED.display((short) 0x0f);
      }
      ....
}
```

This handler uses a hardware object for turning on/of the LED (see Section 4.3). Normally the developer will chose to always compile interrupthandlers rather than interpret them. This completes the part of the setup that can be done in Java space. To finalize, the developer must create a thin C stub delegating the interrupt from C space to Java space. For the ATMega1280 this looks like,

```
int32_t debouncedKeysStack[64];
/* Interrupt vector for Debounced Key 1 */
ISR (INT4_vect)
{
        vm_InterruptDispatcher_interrupt(&debouncedKeysStack[0],6);
}
```

This defines an interrupt service routine that immediately calls Java space. The function `vm_InterruptDispatcher_interrupt` is part of the interrupt handling infrastructure implemented in Java. This gets translated into C by the HVM Eclipse plugin. The interrupt service routine can be implemented in the natives implementation file (see Section 4.2) or any file included in the build. The first parameter to the interrupt handler is the stack to use while handling the interrupt.

4.6 Reflection

The HVM SDK contains a limited reflection API. The reflection classes are used by the SCJ infrastructure 5 and by a garbage collector currently under development.

4.6.1 Scanning Classes

The file `classes.c` contains the translated version of the application dependency extent in C (see Section 2.3. This file contain an array of method class information. This information can be scanned using `reflect.ClassInfo` like illustrated in Figure 10.

```
short numberOfClasses = ClassInfo.getNumberOfClasses();

devices.Console.println("dumping " + numberOfClasses + " classes:");

for (short index = 0; index < numberOfClasses; index++) {
    ClassInfo cInfo = ClassInfo.getClassInfo(index);

    StringBuffer buffer = new StringBuffer();
    buffer.append(cInfo.superClass);
    buffer.append(", ");
    buffer.append(cInfo.dimension);
    buffer.append(", ");
    buffer.append(cInfo.hasLock);
    buffer.append(", ");
    buffer.append(cInfo.dobjectSize);
    buffer.append(", ");
    buffer.append(cInfo.pobjectSize);
    buffer.append(", ");
    buffer.append(cInfo.getName());
    devices.Console.println(buffer.toString());
}
```

Figure 10: Scanning Classes

4.6.2 Scanning Objects

Apart from scanning the static class data it is also possible to scan the references originating in an individual object (if any) and to get the absolute address of the object as a Java int. Figure 11 illustrates this. This is implemented using Hardware Objects (see Section 4.3). It is also possible to scan arrays as illustrated in Figure 12.

```
import reflect.ObjectInfo;
import util.ReferenceIterator;
import vm.HVMHeap;
import vm.Heap;

public class TestObjectTraversal {

    private static class A
    {
        @SuppressWarnings("unused")
        A ref1;
        @SuppressWarnings("unused")
        A ref2;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
       boolean failed = test();
       ....
    }

    public static boolean test() {
        Heap heap = HVMHeap.getHeap();

        A a = new A();
        A b = new A();

        a.ref1 = a;
        a.ref2 = b;

        int aAddress = ObjectInfo.getAddress(a);

        ReferenceIterator references = heap.getRefFromObj(aAddress);

        if (references.hasNext())
        {
            int ref1 = references.next();
            if (ref1 == aAddress)
            {
                if (references.hasNext())
                {
                    int ref2 = references.next();
                    if (ref2 == ObjectInfo.getAddress(b))
                    {
                        return false;
                    }
                }
            }
        }
        return true;
    }
}
```

Figure 11: Scanning Objects

```
import reflect.ObjectInfo;
import util.ReferenceIterator;
import vm.HVMHeap;
import vm.Heap;

public class TestArrayTraversal {

    private static String[] testArray;

    public static void main(String[] args) {
        Heap heap = HVMHeap.getHeap();
        boolean failed = test1(heap);
        ....
    }

    public static boolean test1(Heap heap) {
        String obj1 = new String("hej");
        String obj2 = new String("med");
        String obj3 = new String("dig");

        testArray = new String[3];

        testArray[0] = obj1;
        testArray[1] = obj2;
        testArray[2] = obj3;

        int aAddress = ObjectInfo.getAddress(testArray);

        ReferenceIterator references = heap.getRefFromObj(aAddress);

        if (references.hasNext()) {
            while (references.hasNext()) {
                int nextReference = references.next();
                devices.Console.println("Object at offset " + nextReference);
                if (ObjectInfo.getAddress(obj1) == nextReference) {
                    devices.Console.println(obj1);
                } else if (ObjectInfo.getAddress(obj2) == nextReference) {
                    devices.Console.println(obj2);
                } else if (ObjectInfo.getAddress(obj3) == nextReference) {
                    devices.Console.println(obj3);
                } else
                    return true;
            }
            return false;
        }
        else
        {
            devices.Console.println("No references in array?");
        }
        return true;
    }
}
```

Figure 12: Scanning Arrays

4.6.3 Loading Classes

The HVM does not support dynamic class loading, but classes that are part of the static dependency extent can be programmatically instantiated at run time. This is illustrated below,

```
package test.icecapvm.minitests;

public class TestReflectForName {

    private static class Number
    {
        private int x;

        @SuppressWarnings("unused")
        public Number()
        {
            x = 42;
        }

        public int getX()
        {
            return x;
        }
    }

    public static void main(String[] args) throws ClassNotFoundException,
                                                  InstantiationException,
                                                  IllegalAccessException {
        @SuppressWarnings("rawtypes")
        Class cls = Class.forName("test.icecapvm.minitests.TestReflectForName$Number");

        Number n = (Number) cls.newInstance();
        if (n.getX() == 42)
        {
            /* SUCCESS */
        }
    }
}
```

4.6.4 Reflective Method Invocation

Finally the HVM reflection system contains support for invoking methods programmatically as illustrated here,

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class TestReflectMethod2 {

    public int foo(int x) {
        return x + 10;
    }

    public static void main(String[] args) throws NoSuchMethodException,
                                                  SecurityException,
                                                  IllegalAccessException,
                                                  IllegalArgumentException,
                                                  InvocationTargetException {
        TestReflectMethod2 trm = new TestReflectMethod2();
        Class<? extends TestReflectMethod2> cl = trm.getClass();
        Class<Integer> type = int.class;
        @SuppressWarnings("rawtypes")
        Class[] types = new Class[1];
        types[0] = type;
        Method m = cl.getMethod("foo", types);
        Integer x = (Integer) m.invoke(trm, new Integer(32));
        if (x == trm.foo(32)) {
            try {
                m = cl.getMethod("nonexistant", types);
            } catch (NoSuchMethodException e) {
                /* SUCCESS */
            }
        }
    }
}
```

4.7 Java Level Debugging

The HVM Eclipse plugin supports Java level debugging using Eclipse on all platforms for which a launcher exists. A debug sessions is started by launching an application in debug mode using one of the HVM Eclipse Launcher configurations. The following debugging features are supported,

- Setting and deleting breakpoints - only one breakpoint pr. method is currently supported
- Step over and step into
- The values of simple type local variables - so far objects and arrays cannot be inspected
- Execution stack inspection
- Debugging for interpreted methods only

The remotely executing application is controlled by the debugger over a communications link. For POSIX platforms this link is TCP/IP for AVR platforms it is over the serial line (UART). When run in debug mode the application will communicate with the debugger over the communications link in order to update breakpoints, report that a breakpoint has been hit and send variable and stack data to the debugger. The debugger is implemented in Eclipse using the Eclipse debugging framework (`org.eclipse.debug.*`). The client side of the debugging link is implemented mostly in `natives_allOS.c` for all platforms and in `natives_XX.c` for a handful of short platform specific debugging functions (see Section 2.3).

5 SCJ

To support applications based on the SCJ profile on HVM, a small hardware abstraction layer has been constructed. It introduces primitives for preemptive task scheduling, memory management, device access through hardware objects, first level interrupt handling, a monitor, and a real-time clock. Basically this is the well known exercise of writing a micro kernel - an exercise performed by many software developers over the years. A micro kernel is very small, just a few hundred bytes. A distinguishing feature of the HVM Java HAL is that it is implemented almost entirely in pure Java. This has two significant benefits,

1. Portability. All parts of the Java HAL that are written in pure Java will execute on HVM.
2. Program specialization. If a program uses parts of the Java HAL only, program specialization will exclude unused parts from the final executable. This tailoring of the HAL to a given application comes with no further effort for the parts of the HAL written in Java.

For a particular 16 bit RISC architecture the non-Java parts of the HAL are isolated into 32 assembler instructions and 60 lines of C code.

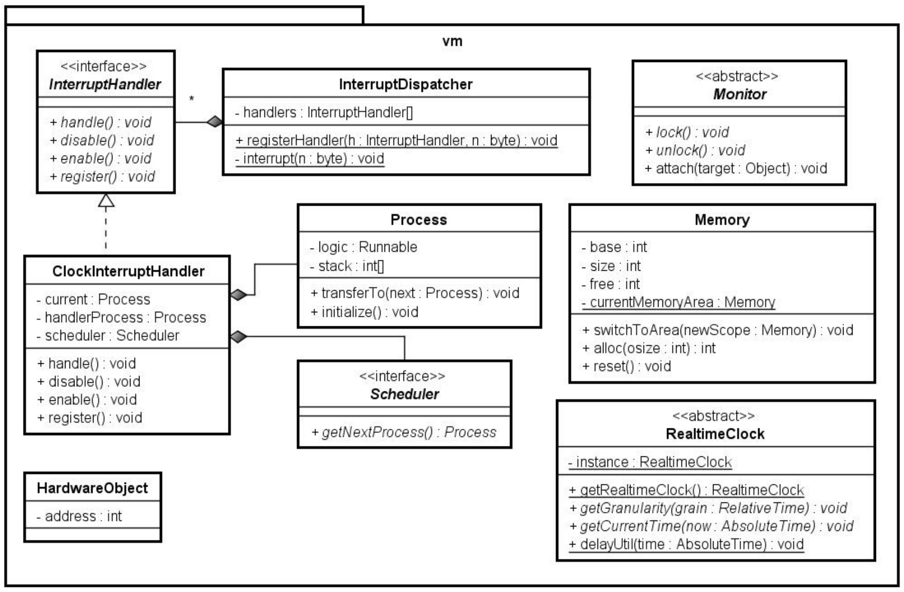

vm
<<interface>>
InterruptHandler
+ handle() : void
+ disable() : void
+ enable() : void
+ register() : void
*
InterruptDispatcher
- handlers : InterruptHandler[]
+ registerHandler(h : InterruptHandler, n : byte) : void
- interrupt(n : byte) : void
<<abstract>>
Monitor
+ lock() : void
+ unlock() : void
+ attach(target : Object) : void
Process
- logic : Runnable
- stack : int[]
+ transferTo(next : Process) : void
+ initialize() : void
Memory
- base : int
- size : int
- free : int
- currentMemoryArea : Memory
+ switchToArea(newScope : Memory) : void
+ alloc(osize : int) : int
+ reset() : void
ClockInterruptHandler
- current : Process
- handlerProcess : Process
- scheduler : Scheduler
+ handle() : void
+ disable() : void
+ enable() : void
+ register() : void
<<interface>>
Scheduler
+ getNextProcess() : Process
<<abstract>>
RealtimeClock
- instance : RealtimeClock
+ getRealtimeClock() : RealtimeClock
+ getGranularity(grain : RelativeTime) : void
+ getCurrentTime(now : AbsoluteTime) : void
+ delayUtil(time : AbsoluteTime) : void
HardwareObject
- address : int

Figure 13: Java HAL overview in package `vm`

Figure 13 presents the Java HAL and the following subsections explain each component.

5.1 Processes and Stacks

The `Process` class implements the Modula 2 process concept [12]. It can be used to start the concurrent execution of the `run` method of a `Runnable` instance with a stack supplied in the form of a Java integer array. The simple part is that of switching from one process to another. To accomplish this, a currently running process calls `transferTo` with the next process to execute. The call to `transferTo` enters native code space and through a short sequence of assembler instructions the current state of the CPU is pushed onto the stack, the stack pointer is stored in the currently executing `Process` object, a new stack pointer retrieved from the `next` process object and the new state popped from the new stack. For a 32 bit Intel platform this involves just a handful of assembler instructions. The effect that is achieved is that when `transferTo` returns, it does not return to the calling process but rather to the `next` process at the point when it called `transferTo` previously. This implements a simple co-routine switching mechanism.

```
import icecaptools.IcecapCompileMe;
import vm.Memory;
import vm.Process;
import vm.ProcessLogic;
import ATMega1280.Drivers.LED;

public class TestProcess {

        private static Process p1;
        private static Process p2;
        private static Process mainProcess;

        private static class P1 implements ProcessLogic {
                @Override
                @IcecapCompileMe
                public void run() {
                        LED.display((short) 0xAA);
                        p1.transferTo(p2);
                }

                @Override
                public void catchError(Throwable t) {
                        LED.display((short) 0xF0);
                }
        }

        private static class P2 implements ProcessLogic {
                @Override
                @IcecapCompileMe
                public void run() {
                        LED.display((short) 0xA0);
                        p2.transferTo(mainProcess);
                }

                @Override
                public void catchError(Throwable t) {
                        LED.display((short) 0xF1);
                }
        }

        public static void main(String[] args) {
                LED.initLEDs();

                p1 = new vm.Process(new P1(), new int[0x3f]);
                p2 = new vm.Process(new P2(), new int[0x3f]);
                mainProcess = new vm.Process(null, null);
                p1.initialize();
                p2.initialize();
                mainProcess.transferTo(p1);
                LED.display((short) 0xFF);
                while (true) {
                        ;
                }
        }
}
```

Figure 14: Process Transfer

An example of performing a Process transfer is given in Figure 14. The native machine code required to perform the stack switch is implemented in the files `XX_interrupt.s`. Such files exist for 32/64 bit Intel (Windows and Linux) and AVR, but can be written in a similar manner for other platforms.

5.2 Scheduling

The HVM allows for implementing schedulers in Java space. The scheduler will find the next Process to execute and perform a transfer to that process as described in Section 5.1. The scheduler gets called by the infrastructure at reschedule points. Currently the interpreter (or compiled code) (see Section 2.1) calls a native function `yieldToScheduler` at regular intervals. This function (implemented in `natives_allOS.c`) checks a global variable `systemTick` in order not to reschedule too often. The `systemTick` variable should be updated at reasonable intervals - but not too often - to control the amount of scheduling going on. If `systemTick` has been updated an interrupt is given to the interrupt dispatcher (see Section 4.5). If the developer has registered a handler for this interrupt and a scheduler to use the scheduler will be called. The following code snippet illustrates how to set up the scheduling,

```
vm.ClockInterruptHandler.initialize(scheduler, sequencerStack);
vm.ClockInterruptHandler clockHandler = vm.ClockInterruptHandler.instance;

clockHandler.register();
clockHandler.enable();
clockHandler.startClockHandler(mainProcess);
clockHandler.yield();
```

The HVM SDK contains three types of schedulers: (1) a PriorityScheduler (2) a CyclicScheduler and (3) a scheduler simulating the standard Java `Thread` package scheduling. It is very easy for the developer to write his own specific scheduler.

The full SCJ infrastructure implementation is described in [4].

6 Full Example

This section describes how to construct, configure and deploy a full SCJ application to an ATMega1280 8 bit micro controller. The HVM does not require the developer use SCJ. For a simpler Hello World type of example refer to Section 3.

The SCJ application is comprised of one mission sequencer, one mission and one periodic event handler. The handlers are schedulable objects, in this case scheduled by the SCJ Level 1 priority scheduler. For in-depth discussion of SCJ core concepts refer to [4, 11].

The event handler period is one second and after it has fired 5 times it requests a mission termination to terminate the application.

The following types of memory resources will be required,

- Immortal memory area. All SCJ applications uses immortal memory
- The Mission sequencer private memory area and execution stack. A SCJ mission sequencer is also an event handler therefor the need for an execution stack
- The mission memory
- The handler private memory area and handler execution stack. In SCJ handlers private memory area gets cleared in between each release of the handler
- The priority scheduler execution stack. The priority scheduler gets invoked at reschedule points. It runs using its own execution stack even though it is not an actual handler on its own

For Linux and Windows stack sizes should not be smaller than 4 kB. For embedded platforms it can be as small as 256 bytes (or smaller), but that depends on the application. The size of the memory areas (immortal memory, mission memory, mission sequencer and handler private memory) is hard to predict. The preferred way to find these sizes is to allocate too much and then profile the application on a PC host environment to find the actual use. The memory area sizes can be set in `main`.

```
        private static final int APP_STACK_SIZE = 2048;


        public static void main(String[] args) {
                 Const.BACKING_STORE_SIZE = 100000;
                 Const.IMMORTAL_MEM_SIZE = 40000;
                 Const.MISSION_MEM_SIZE = 40000;
                 Const.PRIVATE_MEM_SIZE = 2 * 10000;
                 Const.PRIORITY_SCHEDULER_STACK_SIZE = APP_STACK_SIZE;
                 Const.HANDLER_STACK_SIZE = APP_STACK_SIZE;
                 Const.IDLE_PROCESS_STACK_SIZE = APP_STACK_SIZE;
                new Level1Launcher(new MyApp());
        }
```

Figure 15: Conservative Memory Sizes

Figure 15 shows an example of setting some conservative memory sizes (the full source code is available in the Appendix). If compiled with a Java heap of size 200000 kB this will run on a windows host PC. The next step is to include memory profiling to discover what is actually used.

```
        private static final int APP_STACK_SIZE = 2048;

        public static void main(String[] args) {
                Const.BACKING_STORE_SIZE = 100000;
                Const.IMMORTAL_MEM_SIZE = 40000;
                Const.MISSION_MEM_SIZE = 40000;
                Const.PRIVATE_MEM_SIZE = 2 * 10000;
                Const.PRIORITY_SCHEDULER_STACK_SIZE = APP_STACK_SIZE;
                Const.HANDLER_STACK_SIZE = APP_STACK_SIZE;
                Const.IDLE_PROCESS_STACK_SIZE = APP_STACK_SIZE;
                Memory.startMemoryAreaTracking();
                new Level1Launcher(new MyApp());
                Memory.reportMemoryUsage();
        }
```

Figure 16: SCJ Memory Profiling

This is shown in Figure 16. The profile is printed to the console:

```
HEAP[1]: size = 200000, max used = 105118
MemTrk[1]: size = 15000, max used = 2244
javax.safetycritical.ManagedMemory$ImmortalMemory[1]: size = 40000, max used = 25682
PvtMem[3]: size = 2000, max used = 34
javax.safetycritical.MissionMemory[1]: size = 40000, max used = 8778
Max backing store usage = 97000
```

The 5 memory areas created are,

- `HEAP` This area is the full Java heap. All SCJ memories are allocated here. All allocations taking place before the SCJ infrastructure starts up will also be made in here. If the defines `PRE_INITIALIZE_CONSTANTS` and `PRE_INITIALIZE_EXCEPTIONS` are set in the file `types.h` constants and exceptions will also be allocated in here

- `MemTrk` This area is only used by the memory profiling. It will be removed when running without the profiler enabled

- `javax.safetycritical.ManagedMemory$ImmortalMemory` This is the immortal memory. It is used for various static SCJ infrastructure memory and for all SCJ memory areas

- `PvtMem` This is the private memory used by the handler

- `javax.safetycritical.MissionMemory`

It is also reported that `97000` bytes of backing store is used. Next section explains how these memory areas can be shrinked.

6.1 Reducing Memory Requirements

After creating a conservative configuration of SCJ memory area sizes, the first step to reduce memory requirements is to shrink the stacks, since these make up for a significant amount of memory used by our example application (see Section 6). In the example allocation, the stack size (`APP_STACK_SIZE`) can be reduced to `256`. After retranslating and rerunning the result now becomes,

```
Created 5 memory area types:
HEAP[1]: size = 200000, max used = 105118
MemTrk[1]: size = 15000, max used = 2244
javax.safetycritical.ManagedMemory$ImmortalMemory[1]: size = 40000, max used = 4178
PvtMem[3]: size = 2000, max used = 34
javax.safetycritical.MissionMemory[1]: size = 40000, max used = 1610
Max backing store usage = 97000
```

To run the application with this stack size it must be compiled with option `-Os`. Immortal memory and mission memory has now shrinked significantly. The next step is to reduce the size of the `PvtMem` memory area. This is set in the `MySequencer` constructor. We reduce it to `50`. The mission memory area can clearly be reduced as well, and so can the immortal memory area. After a redeploy the result is,

```
Created 5 memory area types:
HEAP[1]: size = 200000, max used = 105118
MemTrk[1]: size = 15000, max used = 2244
javax.safetycritical.ManagedMemory$ImmortalMemory[1]: size = 4200, max used = 4178
PvtMem[3]: size = 50, max used = 34
javax.safetycritical.MissionMemory[1]: size = 1650, max used = 1610
Max backing store usage = 20900
```

The backing store can now be reduced and we arrive at the following configuration,

```
Created 5 memory area types:
HEAP[1]: size = 200000, max used = 26118
MemTrk[1]: size = 15000, max used = 2236
javax.safetycritical.ManagedMemory$ImmortalMemory[1]: size = 4200, max used = 4178
PvtMem[3]: size = 50, max used = 34
javax.safetycritical.MissionMemory[1]: size = 1650, max used = 1610
Max backing store usage = 20900
```

Of the SCJ backing store size of now `20900` bytes `15000` are used for the `MemTrk` area for the memory tracking. When memory profiling is disabled a backing store of `5900` will be enough to hold the SCJ data areas. The `HEAP` memory seems to use approx `5000` bytes more than that. This is mostly for preallocated exceptions and preloaded constants. To avoid this the developer can undefine the `PRE_INITIALIZE_EXCEPTIONS` and `PRE_INITIALIZE_CONSTANTS` macros. After applying these settings and tweaking a bit more the application can run with,

- an immortal memory area of `3500` bytes
- a mission memory area of `1650` bytes
- `50` bytes for the handler private memory area
- a backing store size of `5210` bytes (approx sum of the above)
- a heap size of `6500` bytes

With these configurations in place the SCJ application can now run on e.g. a ATMega1280 with 8 kB of RAM. Non SCJ programs will not use that much RAM memory but only what is allocated by the application. Java objects and arrays are packed and do not take up any more space than C structs and arrays.

In order to run a SCJ application on a embedded architecture the files `natives_XXX.c` and `XXX_interrupt.s` have to be ported to the architecture. This has already been done for the ATMega1280 (and other AVR targets). The next section about porting gives more details about how to port the HVM to a new target.

7 Porting

The smallest platform that the HVM has been ported to is the Arduino UNO (AVR based). The Arduino UNO is an 8 bit micro-controller with 32 kB ROM and 2 kB RAM. On this configuration there is room for non-trivial Java programs controlling devices using e.g. hardware objects or native variables. It is also possible to support process switching using the HVM Process concept, but full SCJ requires more resources than is available on the Arduino. The smallest platform that supports HVM + SCJ is the ATMega1280 (128 kB ROM + 8 kB RAM).

The HVM generates platform independent ANSI C code. Only a small part of the infrastructure is platform specific. The following sections describe the recommended way to port the HVM to a new embedded platform.

7.1 Establishing a C Environment

The first step in the porting is to obtain a C development environment for the target platform. This includes most importantly a C cross compiler. It can be GCC based, but it can also be any other C cross compiler. The HVM has been successfully compiled using the IAR compiler from NOHAU [5, 6]. The HVM does not rely on the presence of a C run time library, but can take advantage of a POSIX compliant run time library if available. The next step is to develop a Hello World type application using the C development environment (not Java or HVM). For small platforms this could be a simple program to blink the LED, or for lager platforms it could be a simple program printing a message to the UART. It is very important to not skip this step - the developer must be confident that the C tools are working and that it is possible to compile, link and deploy a simple C application before attempting to port the HVM.

7.2 Compiling the HVM files

The next step is to construct a compiler command to compile and link the HVM generated C files. A good starting point is,

```
gcc -Wall -pedantic -Werror -g -Os -DJAVA_HEAP_SIZE=1000 -DJAVA_STACK_SIZE=1024
    classes.c icecapvm.c methodinterpreter.c methods.c gc.c natives_allOS.c
    allocation_point.c print.c
```

Before executing the compiler command the developer must specify the basic data types of the target and implement some platform specific functions. The next Sections explains how.

7.2.1 Defining Basic Data Types

The HVM infrastructure is dependent on the definition of some basic data types and macros. These are defined in the file `ostypes.h`. This file already contains some sections that can be reused in some cases. If not the developer has too add a new section. The data types are,

- `uint8`, `int8`, `uint16`, `int16` These byte and short data types defaults to:

  ```
  typedef unsigned short uint16;
  typedef signed short int16;
  typedef unsigned char uint8;
  typedef signed char int8;
  ```

- `uint32`, `int32` These define the data type for 32 bit signed and unsigned integers. This will vary between 8, 16 and 32 bit platforms. The requirement for the correct data type is that `sizeof(type))` must be 4
- `pointer` This data type must define a number data type that can hold a reference. The requirement for the correct data type is that `sizeof(type))` is the width in bytes of references on the platform. E.g. for an ATMega1280 - which can only address up 16 bit address space - this data type has been defined as `unsigned short`

Apart from these basic data types the following macros has to be defined as well,

- `pgm_read_byte(x)`
- `pgm_read_word(x)`
- `pgm_read_pointer(x, typeofx)`
- `pgm_read_dword(x)`

They define how to read data from ROM memory. On some platforms there is no distinction between pointers into ROM and into RAM. In that case they could be defined as,

```
#define pgm_read_byte(x) *((unsigned char*)x)
#define pgm_read_word(x) *((unsigned short*)x)
#define pgm_read_pointer(x, typeofx) *((typeofx)x)
#define pgm_read_dword(x) pgm_read_pointer(x, uint32*)
```

On other platforms e.g. the AVR platforms special instructions must be used to access ROM. On the AVR these macros are defined in the file `<avr/pgmspace.h>`.

7.2.2 Defining Platform Specific Functions

If compared with the compiler commands generated by the launchers or suggested in Section 3.1 some files are missing from the suggested command in Section 7.2:

- `natives_XXX.c` This file contains C helper functions specific to each architecture, e.g. how to allocate the main stack and access IO device registers
- `XXX_interrupt.s` This file is only relevant if running SCJ programs or using the HVM Process concept. The file contain native assembler instructions to perform context switching. See Section 5.1 for more information

If running the compiler without these files a number of functions are unresolved. These functions must be implemented for the new target to complete the porting. The functions are treated individually in the following.

- `void init_compiler_specifics(void)` This function gets called as the first thing upon entry into `main`. It has been used on some platforms to copy initialized data into the correct segments. It can be left empty. It is only called once

- `void initNatives(void)` This function gets called before starting the VM. It is recalled if the VM is restarted. It can be left empty

- `int32* get_java_stack_base(int16 size)` This function gets called before entering the VM. It should return a pointer to a RAM memory area that is used as the Java stack

- `void start_system_tick(void)`, `void stop_system_tick(void)` Refer to Section 5.2. These functions start a timer updating the `volatile uint8 systemTick` global variable. It prevents to eager scheduling if a scheduler has been started. Only pertains to SCJ programs

- `void mark_error(void)`, `void mark_success(void)`. Only used by the regression testing system. Can be left empty

- `void writeByteToIO(pointer address, unsigned short offset, unsigned char lsb)` This function is used for implementing hardware objects. It must be implemented to write `lsb` to `address` + `offset`. The `offset` is in bits. On most architectures this can be very easily implemented as a normal pointer deference. But on some architectures special purpose instructions must be executed to read/write to IO space. The other `read/writeXXXToIO` are similar but for other data types

- `int16 n_vm_RealtimeClock_awaitNextTick(int32 *sp)` Should block until the `volatile uint8 systemTick` global variable gets updated. Can either be a busy loop or more intelligently implemented using some special purpose power saving instructions

- `int16 n_vm_RealtimeClock_getNativeResolution(int32 *sp)` Must return the resolution of the system tick timer as started in the function `void start_system_tick(void)`. The number is returned as an `uint32` containing the number of nanoseconds between two system ticks. Refer to Section 4.2 on how to implement and return values from native methods

- `int16 n_vm_RealtimeClock_getNativeTime(int32 *sp)` Refer to the PC host implementation in `native_scj.c`

- `init_memory_lock`, `lock_memory`, `unlock_memory`. If interrupts can occur while allocating memory using `new` these functions must implement a mutex around memory allocation. Can be left empty for programs not using interrupts or if none of the interrupt handlers allocate memory (which they are not supposed to do)

- `_yield` This function must be implemented in assembler to save all register to the stack, save the stack pointer in the global variable `stackPointer`, call the infrastructure function `_transfer`. Upon return from transfer it must move the value of the global variable `stackPointer` to the stack pointer, restore all registers and performing a return. Refer to existing implementations

- `pointer* get_stack_pointer(void)` Must return the value of the stack pointer as it was when calling the function. Refer to existing implementations

- `void set_stack_pointer(void)` Must set the value of the stack pointer to the value of the global variable `stackPointer` and return to the caller. Refer to existing implementations

- `void sendbyte(unsigned char byte)` Prints a byte. Used to print messages the console. Can be left empty. If a UART is available this can be used for printing. On platforms supporting `printf` this can simply print the byte as a char on `stdout`

8 Performance

This Section shows measurements comparing the execution efficiency of the HVM to other similar environments. Even though the HVM can be used to program Java for embedded systems it is also very important to engineers that the efficiency by which Java can run is close to the efficiency they are accustomed to for their current C environments. As the following measurements will show functionality written in Java and executed on the HVM is 2-3 times slower than handcoded C. Interpreted Java code take up less space than compiled C code, but an offset price of approximately 8 kB ROM space must be payed. Compiled Java code takes up approximately twice the space of handcoded C. The proper selection of which methods to compile and which methods to interpret can yield an efficient and tight executable.
For high-end embedded platforms results already exists regarding execution speeds of Java programs compared to the same program written in C. In their paper [8] the authors show that their Java-to-C AOT compiler achieves a throughput to within 40% of C code on a high-end embedded platform. This claim is thoroughly substantiated with detailed and elaborate measurements using the CDj and CDc benchmarks[2].

Here we introduce a small range of additional benchmarks. The idea behind these benchmarks are the same as from CDj/CDc: To compare a program written in Java with the same program written in C.

8.1 Method

The 4 benchmark programs are written in both Java and C. The guiding principles of the programs are,

- *Small*. The benchmarks are small. They don't require much ROM nor RAM memory to run. The reason why this principle has been followed is that it increases the probability that they will run on a particular low-end embedded platform
- *Self-contained*. The benchmarks are self-contained, in that they do not require external Java nor C libraries to run. They don't even require the `java.util.*` packages. The reason is that most embedded JVMs offer their own JDKs of varying completeness, and not relying on any particular Java API will increase the chance of the benchmark running out-of-the-box on any given execution environment
- *Non-configurable*. The benchmarks are finished and ready to run as is. There is no need to configure the benchmarks or prepare them for execution on a particular platform. They are ready to run as is. This will make it easier to accurately compare the outcome from running the benchmarks on other platforms, and allow other JVM vendors to compare their results

- *Simple.* The behavior of each benchmark is simple to understand by a quick scrutinizing of the source code. This makes it easier to understand the outcome of running the benchmark and to asses the result.

The benchmark suite of only 4 benchmarks is not complete and the quality and relevance of the suite will grow as new benchmarks are added. The guiding principles of the benchmarks are very important, especially the principle of being self-contained, since this is a principle most important for being successful at running a benchmark on a new embedded platform.

The current benchmarks are:

1. *Quicksort.* The `TestQuicksort` benchmark creates an array of 20 integers initialized with values from 0 to 20 in reverse order. Then a simple implementation of the quicksort method sorts the numbers in place. This benchmark applies recursion and frequent access to arrays

2. *TestTrie.* The `TestTrie` benchmark implements a tree like structure of characters - similar to a hash table - and inserts a small number of words into the structure. This benchmark is focusing on traversing tree like structures by following references

3. *TestDeterminant.* The `TestDeterminant` benchmark models the concept of vectors and matrices using the Java concepts of classes and arrays. Then the Cramer formula for calculating the determinant of a given 3x3 matrix is applied

4. *TestWordReader.* The `TestWordReader` benchmark randomly generates 17 words and inserts them into a sorted list of words, checking the list before each insert to see if it is not there already. Only non duplicates are inserted.

The full source of the benchmarks can be found in the HVM SDK jar, along with approx 200 other small test programs used for regression test of the HVM. The nature of these benchmarks are not exhausting all aspects of the Java language, but they still reveal interesting information about the efficiency of any given JVM for embedded systems. The purpose of the benchmarks are to reveal how efficiently Java can be executed in terms of clock cycles as compared to C and how much code space and RAM are required. The benchmarks are not intended to test garbage collection, and non of the benchmarks require a functioning GC to run. Nor do they give any information about the real-time behavior of the system under test. To test GC efficiency and/or real-time behavior of a given JVM the CDj/CDc benchmarks are available.

In Section 8.2 compares the results from running these benchmarks on GCC, FijiVM, KESO, HVM, GCJ, JamVM, CACAO and HotSpot. This will give us valuable information about the efficiency with which these environments can execute Java code as compared to each other and as compared to C based execution environments.

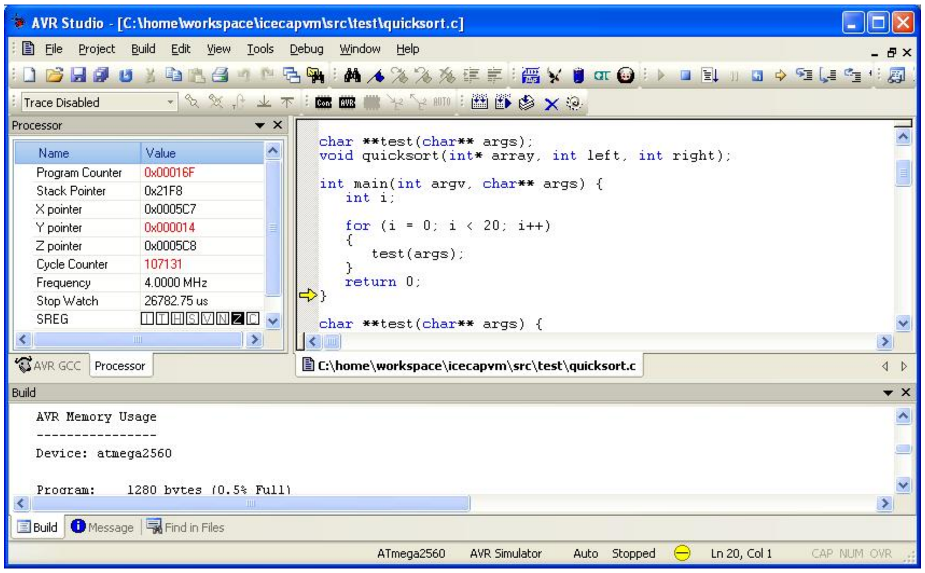
8.1.1 Benchmark execution - High-end Platforms

Since only three of the tested execution environments (GCC, KESO and HVM) are capable of running these benchmarks on low-end embedded systems, they were first run on a 32 bit Linux PC. On this platform all JVMs under test could execute the benchmarks. The number of instructions required to run the benchmarks was measured using the **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) [1, 7]. The reason for measuring the instruction count and not the number of clock cycles is that the instruction count is a deterministic value for the benchmarks, but the clock cycle count is not on advanced processors. This first run of all the benchmarks on a 32 bit Linux PC will not by it self give us the desired results for low-end embedded platforms, but it will allow us to compare the JVMs under test against each other and against C on high-end platforms. To achieve the desired results for low-end embedded platforms the benchmarks will be run on a particular low-end embedded environment as well using GCC, HVM and KESO. This will give the desired results for these two JVMs, but compared with the results for high-end environments one can make statements about what could have been expected had it been possible to run all JVMs on a low-end embedded environment.

For all execution environments the native instruction count was measured by calling the PAPI API before and after each test run. The tests was run several times until the measured value stabilized - this was important for the JIT compilers especially, but also for the other environments. E.g. calling `malloc` for the first time takes more time that calling `malloc` on subsequent runs. All in all the measurements reported are for *hot* runs of the benchmarks.

8.1.2 Benchmark execution - Low-end Platforms

To obtain a result for low-end embedded platforms the benchmarks was run using GCC, HVM and KESO on a ATMega2560 AVR micro-controller. This is an 8 bit micro-controller with 8 kB of RAM and 256 kB ROM. On this simple platform there is a linear, deterministic correspondence between number of instructions executed and clock cycles. The AVR Studio 4 simulator was used to run the benchmarks and accurately measured the clock cycles required to execute each benchmark. Figure 17 shows an example of running the `TestQuicksort` benchmark using GCC. To produce the executable the avr-gcc cross compiler (configured to optimize for size) was used.


AVR Studio - [C:\home\workspace\icecapvm\src\test\quicksort.c]
File Project Build Edit View Tools Debug Window Help
Trace Disabled
Processor
AVR GCC
Processor
C:\home\workspace\icecapvm\src\test\quicksort.c
Build
Build
Message
Find in Files
ATmega2560
AVR Simulator
Auto
Stopped
Ln 20, Col 1
CAP NUM OVR

Name	Value
Program Counter	0x00016F
Stack Pointer	0x21F8
X pointer	0x0005C7
Y pointer	0x000014
Z pointer	0x0005C8
Cycle Counter	107131
Frequency	4.0000 MHz
Stop Watch	26782.75 us
SREG	I T H S V N Z C

```
char **test(char** args);
void quicksort(int* array, int left, int right);

int main(int argv, char** args) {
    int i;

    for (i = 0; i < 20; i++)
    {
        test(args);
    }
    return 0;
}

char **test(char** args) {
```

```
AVR Memory Usage
----------------
Device: atmega2560

Program:    1280 bytes (0.5% Full)
```

Figure 17: AVR Studio 4

In this test run the clock cycles spent to get to the for-loop was measured (in this case 125 clock cycles), and this number was subtracted from the time taken to perform the benchmark. Then the test was run 20 times, in this case yielding a clock cycle count of 107131. GCC takes (107131 - 125) / 20 = 5350 clock cycles to perform the benchmark.

To obtain similar results for KESO, the C source produced by the KESO Java-to-C compiler was compiled using the avr-gcc cross compiler. An AVR Studio 4 project was created to enable the measurement of clock cycles as above. Again the start up time was measured and each benchmark run a number of times to arrive at an average time taken for KESO to execute the benchmark. Similarly for HVM. All projects configured to optimize for size.

These measurements are directly relevant for low-end embedded platforms and allow us to validate how the HVM compares to GCC and KESO. Since these three environments also appear in the high-end platform measurements, where they can be related to results from the other environments, they offer a chance in Section 8.3 to predict how these other high-end environments would have performed had they been able to run on the ATMega2560.

8.2 Results

The measurements performed using the PAPI API on a 32 bit Linux PC platform are listed in Figure 18 and 19.

The instruction count taken for the C version to execute is defined as 100. The instruction count taken for the other environments is listed relatively to C above. E.g. the HVM uses 36% more instructions to execute the Trie benchmark

	C	KESO	FijiVM	HVM	GCJ
Quicksort	100	101	136	111	172
Trie	100	93	54	136	245
Determinant	100	59	37	96	171
WordReader	100	251	218	177	328
Total	100	126	111	130	229

Figure 18: Instruction Count Comparison - Part 1

	C	JamVM	HVMi	CACAO	HotSpot
Quicksort	100	697	4761	147	156
Trie	100	772	1982	294	234
Determinant	100	544	1664	294	48
WordReader	100	975	4979	263	142
Total	100	747	3346	250	145

Figure 19: Instruction Count Comparison - Part 2

than native C.

The results from comparing HVM and KESO on the ATMega2560 are listed in Figure 20.

	C	KESO	HVM
Quicksort	100	108	130
Trie	100	223	486
Determinant	100	190	408
WordReader	100	331	362
Total	100	213	347

Figure 20: Cycle count comparison

This is an accurate cycle count comparison for KESO and HVM.

8.3 Discussion

The most interesting results are contained in Figure 20. This whows that for the benchmarks tested,*KESO is approximately 2 times slower than C and the HVM is approximately 3 times slower than C.*

There are several observations that should be taken into account when considering the above experiment:

- KESO supports GC, the HVM does not but relies on SCJ memory management. Even though GC is not in effect above, the KESO VM probably pays a price in terms of execution efficiency for the presence of GC

- The HVM supports Java exceptions, KESO does not. Very rudimentary experiments not shown here indicate that the cost of exception support is an approx 25% decrease in performance for the HVM
- Scrutinizing the C code produced by KESO shows that the Java type `short` is used in places where this is not correct. E.g. code had to be manually fixed for the WordReader benchmark to reintroduce the correct data type `int` in various places. Using `short` where `int` is required might be reasonable in several cases, and this will have a significant impact on performance, especially on 8 bit platforms as the ATMega2560.

The following substantiated observations for low-end embedded platforms can be made,

- Java-to-C compilers are a little slower than native C, but not by an order of magnitude. It is likely that they can be approximately half as fast as native C
- KESO is faster than HVM. The HVM achieves a throughput of approx 50% that of KESO.

Appendix

Full SCJ Example

```
package Main;

import javax.realtime.Clock;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.safetycritical.Launcher;
import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.Safelet;
import javax.safetycritical.StorageParameters;
import javax.scj.util.Const;
import javax.scj.util.Priorities;

import vm.Address32Bit;
import vm.HardwareObject;

public class TestSCJSimpleLowMemory {

        private static final int APP_STACK_SIZE = 2048;

        private static class Light extends HardwareObject {
                @SuppressWarnings("unused")
                byte ledPort;

                public Light(int address) {
                        super(new Address32Bit(address));
                }

                public void on(int colour) // 0 = green; 1 = red
                {
                        if (colour == 0) {
                                ledPort = 1;
                        } else
                                ledPort = 2;
                }

                public void off(int colour) {
                        if (colour == 0) {
                                ledPort = 3;
                        } else
                                ledPort = 4;
                }
        }

        private static class MyPeriodicEvh extends PeriodicEventHandler {
                Light light;
                int count = 0;
                private MissionSequencer<MyMission> missSeq;

                protected MyPeriodicEvh(PriorityParameters priority,
                                PeriodicParameters periodic, Light light,
                                MissionSequencer<MyMission> missSeq) {
                        super(priority, periodic, new StorageParameters(0,
                                        new long[] { APP_STACK_SIZE }, 0, 0, 0));
                        this.light = light;
                        this.missSeq = missSeq;
                }

                public void handleAsyncEvent() {
                        if (count % 2 == 0) {
                                light.on(0);
                        } else {
                                light.off(0);
                        }

                        count++;

                        if (count == 5) {
                                missSeq.requestSequenceTermination();
                        }
                }
        }

        private static class MyMission extends Mission {
                MissionSequencer<MyMission> missSeq;

                public MyMission(MissionSequencer<MyMission> missSeq) {
                        this.missSeq = missSeq;
                }

                public void initialize() {
                        int address = 123456;
                        Light light = new Light(address);

                        PeriodicEventHandler pevh1 = new MyPeriodicEvh(
                                        new PriorityParameters(Priorities.PR97),
                                        new PeriodicParameters(null, new RelativeTime(1000, 0,
                                                        Clock.getRealtimeClock())), // period
                                        light, missSeq); // used in pevh.handleAsyncEvent
                        pevh1.register();
                }

                public long missionMemorySize() {
                        return Const.MISSION_MEM_SIZE;
                }

        }

        private static class MyApp implements Safelet<MyMission> {
                public MissionSequencer<MyMission> getSequencer() {
                        return new MySequencer();
                }

                public long immortalMemorySize() {
                        return Const.IMMORTAL_MEM_SIZE;
                }

                private static class MySequencer extends MissionSequencer<MyMission> {
                        private MyMission mission;

                        MySequencer() {
                                super(new PriorityParameters(Priorities.PR95),
                                                new StorageParameters(0, null, 2000, 0,
                                                                Const.MISSION_MEM_SIZE));

                                mission = new MyMission(this);
                        }

                        public MyMission getNextMission() {
                                if (mission.terminationPending()) {
                                        return null;
                                } else {
                                        return mission;
                                }
                        }
                }
        }

        private static class Level1Launcher extends Launcher {
                public Level1Launcher(MyApp myApp) {
                        super(myApp, 1);
                }
```

```
                @Override
                public void run() {
                        startLevel1();
                }
        }

        /*
         * gcc -Wall -pedantic -g -Werror -Os -DPC64 -DREF_OFFSET -DPRINTFSUPPORT
         * -DSUPPORTGC -DJAVA_HEAP_SIZE=6500 -DJAVA_STACK_SIZE=96 -L/usr/lib64
         * classes.c icecapvm.c methodinterpreter.c methods.c gc.c natives_allOS.c
         * natives_i86.c rom_heap.c allocation_point.c rom_access.c native_scj.c
         * print.c x86_64_interrupt.s -lpthread -lrt -lm
         */

        public static void main(String[] args) {
                Const.BACKING_STORE_SIZE = 5210;
                Const.IMMORTAL_MEM_SIZE = 3500; // 3320 will do
                Const.MISSION_MEM_SIZE = 1650;
                Const.PRIVATE_MEM_SIZE = 2 * 1000;
                Const.PRIORITY_SCHEDULER_STACK_SIZE = APP_STACK_SIZE; // 256 will do at

                Const.HANDLER_STACK_SIZE = APP_STACK_SIZE; // 256 will do at -Os
                Const.IDLE_PROCESS_STACK_SIZE = 64; // 64 will do at -Os
                // Memory.startMemoryAreaTracking(); // Add 15000 bytes to
                // BACKING_STORE_SIZE and JAVA_HEAP_SIZE to include this
                // executes in heap memory
                new Level1Launcher(new MyApp());
                // Memory.reportMemoryUsage();
                args = null;
        }
}
```

References

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[2] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time java benchmarks. *Concurr. Comput. : Pract. Exper.*, 23(14):1679–1700, Sept. 2011.

[3] S. Korsholm. HVM Lean Java for Small Devices. `http://www.icelab.dk/`, 2014.

[4] S. Korsholm, H. Søndergaard, and A. Ravn. A real-time java tool chain for resource constrained platforms. *Concurrency and Computation: Practice & Experience*, 2013:1–25, September 2013.

[5] NOHAU. `http://www.nohau.se/iar`. Visited January 2012.

[6] NOHAU. `http://www.iar.com/en/Products/IAR-Embedded-Workbench/`. Visited February 2012.

[7] PAPI. Papi - the Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/index.html`, 2012.

[8] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. *SIGPLAN Not.*, 45(6):146–159, June 2010.

[9] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[10] sourceforge. HaikuVM - A Java VM for ARDUINO and other micros using the leJOS runtime. `http://haiku-vm.sourceforge.net/`, 2014.

[11] TheOpenGroup. Safety-Critical Java Technology Specification (JSR-302). Draft Version 0.79, TheOpenGroup, May 2011.

[12] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1985.