Flash memory in embedded Java programs

Stephan Korsholm
VIA University College
Horsens, Denmark
sek@viauc.dk

ABSTRACT

This paper introduces a Java execution environment with the capability for storing constant heap data in Flash, thus saving valuable RAM. The extension is motivated by the structure of three industrial applications which demonstrate the need for storing constant data in Flash on small embedded devices.

The paper introduces the concept of *host initialization of constant data* to prepare a Flash image of constant data that can be kept outside the heap during runtime.

The concept is implemented in an interpreter based Java execution environment.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Read-only memory (ROM); C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.3.4 [**Processors**]: Interpreters

General Terms

Languages, Design.

Keywords

Java/C Integration, Flash Memory, Embedded Systems, Real-Time.

1. INTRODUCTION

The C programming language together with appropriate runtime libraries is widely used when implementing software for tiny devices. C environments support a range of features such as

1. accessing device registers and raw memory,
2. handling first level interrupts,
3. storing constant data in Flash.

JTRES 2011 September 26-28, 2011, York, UK

These features are often controlled by compiler directives that instruct the compiler to generate code for e.g. (1) accessing hardware through memory mapped device registers (2) marking C-functions as interrupt handlers and generating proper save and restore code stubs and (3) marking a piece of data as constant and placing it in a read-only data segment.

Previous work in [15] and [12] has demonstrated how (1) and (2) can be added to Java execution environments. This paper describes a method to support (3) for Java by marking parts or all of an object as constant and eligible for storing in Flash, thereby freeing valuable RAM storage.

The method presented here shows how to store constant data (e.g. in the form of large arrays) in Flash instead of the heap. The idea is to pre-execute the initial part of the program on a host platform and in that part of the program build the arrays and other data that are considered constant and thus can be kept in Flash. After the pre-execution on the host the constant data is converted into an image that can be kept in Flash on the target. When running on the target the Flash image is accessible for reading by the application while writing to the Flash image is implemented as null operations.

The methods used in the process have been used in popular environments like C (see Section 3) and Ada (see Section 7) for many years in order to solve the same problem. The contribution of this paper is to apply these methods to the Java domain. By doing this we have solved problems from industrial settings. Problems which until now have prevented us from using Java on a certain class of embedded targets. We describe these applications further in Section 1.1.

We consider our proposal a starting point to bridge an important gap many embedded developers have to cross if they consider using a high level, well structured language like Java in the embedded domain. The method can be improved in several ways and we discuss these in Section 8.

1.1 Constant Data

Constant data are data that are initialized at application load time and do not change during the entire lifetime of the application. To find out how much constant data a given C application is using, the unstripped binary file can be inspected. We have looked at three industrial embedded C applications and examined their use of constant data. The example applications are

1. The DECT protocol stack from Polycom. It is used by Polycom, Horsens, DK [14] for implementing wireless communication between devices. It is used on the

CR16 16bit platform from National with 8KB RAM and 768KB Flash

2. The Modbus control and monitoring application from Grundfos. It is used by Grundfos, Bjerringbro, DK [9] for monitoring and controlling pumps. It is used on the NEC v850 32bit platform with 4KB and upwards of RAM and 64-1024KB Flash

3. The HVM Java virtual machine [11]. It is used by the author for programming tiny embedded devices in Java. It is used on the ATMega 8bit platform from AVR with 8KB RAM and 256KB Flash

Table 1 divides the used data memory into two types: variable data and constant data. Variable data may be written during the execution of the program and have to be kept in RAM, whereas constant data are only written once during initialization and can be kept in ROM. For reference, the size of the code itself is included in Table 1 as well.

	Code	Variable Data	Constant Data
DECT	234KB	7KB (8%)	78KB (92%)
Modbus	171KB	207KB (83%)	41KB (17%)
HVM	22KB	4KB (29%)	10KB (71%)

Table 1: Overview of data usage

All three applications use a significant amount of constant data. For the DECT and HVM applications it would not be possible to hold the constant data in RAM together with the variable data as that would require more RAM than is available on the devices.

Further scrutinizing the source code of the applications above reveals the following examples of use of constant data

- DECT. Arrays of bytes used as input to the DSP (Digital Signal Processor) to generate ringing tones

- Modbus. Large tables of approx. 512 entries each. The tables map names to data locations called registers. The indirection through the tables are used to control access to the registers - some tables allow full access to all registers while others allow limited access to only some registers. Section 6 looks into this example in more detail

- HVM. A selection of the class file content is initialized as a graph-like structure for easy access to important information when the code is interpreted

It seems reasonable to expect that other applications for similar types of platforms use constant data to a significant degree as well and it follows that being able to store constant data in Flash is desirable when implementing similar applications in Java for similar types of platforms.

1.2 Why Flash?

Producers of embedded solutions seek to use Flash as opposed to RAM whenever possible. The reason is that the production price per unit is significantly lower by using Flash as much as possible instead of adding more RAM to the design. The Polycom and Grundfos units are produced and sold in large quantities, and even minor savings in unit production cost have a great impact on the profitability of their businesses.

The remainder of this paper will describe a design for using Flash for constant data in Java programs and show an implementation in an interpreter based execution environment.

2. RUNNING EMBEDDED JAVA ON TINY DEVICES

We will use the HVM [11] to illustrate how constant data can be integrated into a Java execution environment. The HVM is a small interpreter based JVM intended for use on tiny devices. An architectural overview of the HVM Java execution environment is shown in Figure 1.

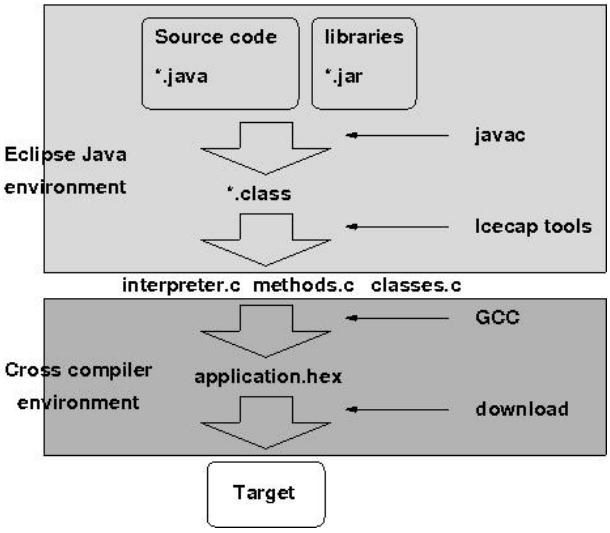

Figure 1: HVM Architecture

The Eclipse IDE integrated with any Java compiler and SDK is used to compile the Java source code into class files.

Next a plugin tool, called icecap [11], analyzes the generated class files and convert them into C code by storing the Java byte codes and other class file content in byte arrays in two auto-generated C files - *methods.c* and *classes.c*.

Finally the developer can use *any C based cross compiler environment* (e.g. the avr-gcc tool chain for the AVR family of processors) or the commercially available IAR compiler tool-chain (for a wide variety of industry standard embedded platforms). The C cross compiler compiles the HVM interpreter itself (*interpreter.c*) and the auto-generated C files into a final executable for the target. A non-cross compiler can be used as well to generate an executable for the host platform (e.g. Linux or Windows) to run the executable in the host environment for testing or debugging.

3. A DESIGN FOR FLASH DATA IN JAVA

Consider the following C example of constant data taken from the DECT application

```
const unsigned char wav_num_0[] =
    { 23, 112, -1, -1 };
```

This is a simplified version of a longer array used by the DSP to generate ringing tones. When the C compiler translates this into assembler it will look like the following

```
.globl wav_num_0
.section .rodata
.type wav_num_0, @object
.size wav_num_0, 4
wav_num_0:
.byte 23
.byte 112
.byte -1
.byte -1
```

Based on this assembler output from the compiler, the linker will create an array of bytes in the section *rodata* which means that the array will eventually be packaged along side the code in Flash. Additionally it will link the use of the variable names to the location of the data in Flash. We notice that preparing the constant data for Flash is done on the host, before the application is started, since it is difficult and usually avoided to write to Flash during runtime. It is only when the application is loaded, that the boot-loader will be able to write code and constant data to the Flash. When control is handed over from the boot-loader to the application, the Flash memory becomes read-only.

In Java the example above looks like this

```
public class ConstantData
{
    final byte wav_num_0[] =
        { 23, 112, -1, -1 };
}
```

And now the assembler output (the Java byte codes) from the compiler is

```
public class ConstantData extends Object{
public ConstantData();
  Code:
   0:  aload_0
   1:  invokespecial #1; //"<init>":()V
   4:  aload_0
   5:  iconstant_4
   6:  newarray byte
   8:  dup
   9:  iconstant_0
   10: bipush 23
   12: bastore
   13: dup
   14: iconstant_1
   15: bipush 112
   17: bastore
   18: dup
   19: iconstant_2
   20: iconstant_m1
   21: bastore
   22: dup
   23: iconstant_3
   24: iconstant_m1
   25: bastore
   26: putfield #2; //wav_num_0:[B
   29: return
}
```

The main difference from the C solution is that the code to set up the constant data is executed on the target during runtime, where as the C solution prepares a read-only section of data on the host. In Java the *wav_num_0* array will be a normal array placed in the heap (in RAM) into which the bytes are written at runtime.

The goal of our design for constant data in Java is to place the wav_num_0 array in Flash rather than on the heap.

3.1 Identifying Constant Data

Byte codes to create and access data in general must know which type of data is accessed, mainly because constant data are not located in the heap (RAM) but located in Flash. To facilitate this, constant data must be *inferred implicitly or marked explicitly.*

For simplicity we will mark constant data at the time of declaration. In the examples below we use the *const* keyword to mark constant data. Since this is not a Java keyword any actual implementation could use Java annotations when marking constant data. Below follows an example of marking a piece of data as constant

```
public class ConstantData
{
  private const byte wav_num_0[] =
    { 23, 112, -1, -1 };

  public static void main(String args[])
  {
    ConstantData cdata = new ConstantData();
  }
}
```

3.2 Creating Constant Data - Host Initialization

Creating and initializing constant data must be done before the application is downloaded and run on the target - this is because writing to Flash at runtime is very difficult and inefficient. Similar to what C environments do, the constant data segments must be built and linked with the final executable on the host. To solve this challenge we suggest to use *host initialization of constant data.*

The idea is to run the initialization part of the application on the JVM on the host platform. While running on the host platform, the soon-to-be read-only segment of constant data can be built in RAM on the host. The start of the initialization phase will be the start of the program. The end of the initialization phase could either be inferred implicitly or marked explicitly. For simplicity we have chosen the latter, as in the following example

```
public class ConstantData
{
  private const byte wav_num_0[] =
    { 23, 112, -1, -1 };

  private const byte wav_num_1[] =
    { 43, 12, -1, -1 };

  private const int NUM = 42;

  public boolean flag;

  public static void main(String args[])
  {
    ConstantData cdata = new ConstantData();
    System.lockROM();
    cdata.flag = true;
  }
}
```

The call to *System.lockROM* signals the end of the initialization phase. During the initialization phase three new objects are created

1. *cdata* is an object with three constant data fields and one non-constant field. We allocate 8 bytes for the two array references and 4 bytes for the int *NUM* - these 12 bytes plus some bookkeeping are allocated in the constant heap. The non constant part of the object is one byte field of size 1 byte. This is allocated in the normal heap. For the HVM the bookkeeping information is a class index and a reference into the normal heap pointing to the non-constant part of the object

2. *wav_num_0* is a constant array of 4 bytes. Since it is marked as *const*, it is not allocated in the standard heap, but in the constant heap. The array uses 4 bytes plus some bookkeeping information. For the HVM an additional 2 bytes for the element count and 2 bytes for the type are needed. Thus 8 bytes are allocated in the constant heap

3. Similarly *wav_num_1* is allocated in the constant heap right after the previous item

When the *System.lockROM()* method is called on the host, the content of the host memory is as depicted in Figure 2.

3.3 Creating Constant Data - Target Initialization

After the host initialization phase, the content of the constant heap is transferred into a C array of bytes and saved in a file *rom.c*. Using the cross compiler environment, the interpreter in *interpreter.c*, the Java application in *classes.c* and *methods.c*, and the constant data Flash image in *rom.c* are linked to form the final executable. The overall architecture is depicted in Figure 3.

So, the initialization part will be run twice - once on the host to produce the constant heap, and once on the target when the executable is downloaded and run. On the target it will repeat the initialization step - it will execute the same code, allocate objects and read/write to them in the same sequence,but *the way the allocation and access are handled is different on the target*:

- When *allocating data* in the constant heap, the same allocation procedures can be carried out, but the constant heap is already initialized and placed in Flash. References to the existing objects are returned to the caller.

- When *writing* to the constant heap no actual write takes place since the value is already there.

- When *reading* from the constant heap a Flash read operation is carried out.

In order for this to work, the initialization phase must be deterministic - when executed on the target, objects must be allocated and written in the exact same order as they were on the host. Additionally the allocation of constant data must only take place during the initialization phase.

3.4 Preparing the Constant Data Heap

The heap of constant data has to be useful on both the host environment during initialization and later on the target environment when the application is running. The values in the constant heap can be viewed as a graph of data, where

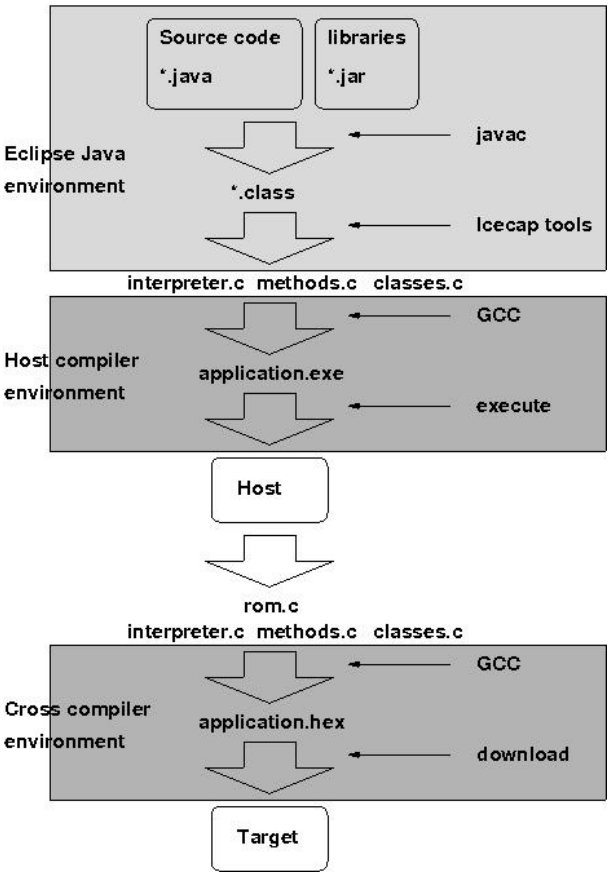

Figure 3: Creating the ROM image

the nodes (objects and arrays) contain basic data like integers and the edge between the nodes are references. This graph is transferred from the host to the target. The consequence of this is

- the edges cannot be actual pointers (addresses), as the addresses are not going to be the same on the host as on the target

- the values in the nodes have to be in the same byte order

Transferring the constant heap is basically a serialization of the data into a platform independent format. In the HVM all references in the constant heap are stored as offsets, either into the constant heap itself, or offsets into the normal heap and all values are stored in little endian format. This incurs an overhead for accessing data in the constant heap, but it does not affect how data are accessed in the normal heap.

3.5 Discussion

By using host initialization of constant data it is now possible to create the Flash image and link it with the final executable - in exactly the same manner as is done in C environments. Reading and writing to the constant heap on the host are done in the same manner as reading and writing to the normal heap, as both heaps are kept in RAM. On the target, accessing Flash from program code is different from

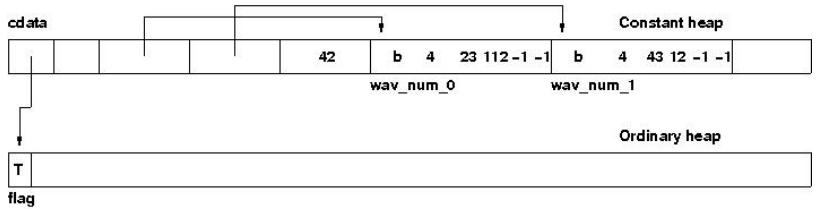

Figure 2: Host memory after initialization

accessing RAM memory. The hardware architecture of embedded devices can be divided into either a Von Neumann architecture or a Harvard Architecture [16]. In the former, both code and data are in the same address space, and instructions for reading and writing to Flash are the same as instructions for reading and writing to RAM. In the latter, architecture code and data are kept in two separate data spaces and accessed using different instructions.

In both cases it is easy and fast reading from Flash at runtime, but it is difficult writing to Flash at runtime.

To support constant data in Java on any embedded architecture, the VM needs to know if it is accessing Flash or normal RAM so that it may use the proper instructions for doing so.

In Section 4 we will show what is required to augment an existing JVM with capabilities for identifying and accessing constant data.

4. IMPLEMENTING CONSTANT DATA IN A VM

All creation and access of constant data take place through byte codes. Constant data can be either arrays or field variables. The byte codes involved in accessing arrays or object fields must be augmented with capabilities to recognize and access constant data as well. The cost of this in terms of execution efficiency is discussed in Section 5.

The byte codes that may be directly involved in accessing constant data are *new*, *newarray*, the array store byte codes - call these *xastore*, the array load byte codes - call these *xaload*, and finally the object field accessors *getfield* and *putfield*. For simplicity we defer the handling of static constant data, but this is a simple extension to the current implementation. In the following we describe the changes that are required to the involved byte codes in order to handle constant data - first on the host, then on the target.

4.1 Host Handling of Constant Data

new contains a reference to the class that is being instantiated. If one of the fields in the class is declared as constant data, then the object is allocated differently than from normal heap objects. Whether the class does indeed contain constant data can either be embedded in the byte code, or looked up at runtime. The HVM checks at runtime if the class contains constant data, and if so the new byte code will allocate two parts of data, one in the constant heap and one in the normal heap as described in Section 3.2.

newarray does not contain enough information to know if the array is constant or not. The array is constant if it is later saved in a constant field, but this is not known when the byte code is executed. To solve this problem, the icecap tools (see Figure 1) perform a static analysis of the code to instrument the *newarray* byte code with a flag indicating if a constant array is being allocated or not.

xastore, xaload retrieve the array to store/load from the stack. On the host it makes no difference to these byte codes if the array is in the constant heap or not.

putfield, getfield As with *new* these byte codes have enough information to decide if the field being written is inside a constant data object or not, and if the field itself is a constant field. This information can either be looked up at runtime or embedded in the byte code. The HVM looks up the information at runtime to properly access the field in either the constant heap or the normal heap.

4.2 Target Handling of Constant Data

new, newarray On the target the *new* byte code should not actually allocate and clear a new piece of memory for constant objects. Rather, if the object being allocated contains constant data, it should return a reference to the data as it is already present in the Flash section. If the allocations of constant objects are proceeding in the same order and with the same sizes as during the initialization phase on the host, the references returned will be correct. Still the non constant part of the object must be allocated in the normal heap.

xastore Whether the array is a constant array or not can be determined by looking at the reference - if it points into the constant heap, it is a constant array. Since we prefer to avoid writing to the Flash on the target, the array store operation can instead check if the value being written is the same as the value already present in Flash.

xaload On Harvard architecture platforms the loading from Flash must be done using a special load instruction - e.g. on the ATMega2560 the instruction to load from Flash is called *lpm* (Load Program Memory) where as the instruction to load from RAM is called *ld*. On many Von Neumann architectures no changes are required to this byte code as loading from Flash is no different than loading from RAM (as is the case with the CR16 platform).

putfield On the target this instruction cannot actually write to constant fields, but should behave in the same manner as the array store instruction. If the target field of the write is a non constant field inside a constant object, then an extra indirection is required.

getfield On the target this instruction works as the array load instructions, so it is necessary to use special load instructions for accessing Flash, but more important, the loading of a non constant field from an object with constant fields requires an extra indirection.

4.3 Checking Constant Data Access

After the initialization phase the program should never write to constant data again. If a new value is written to a constant array after the initialization phase, the interpreter would discover that an illegal write is taking place. In the current implementation on the HVM this will result in a runtime exception.

If constant data access conventions are violated, a static check would be better than introducing new runtime checks as discussed in Section 8.

5. COST OF CONSTANT DATA

The goal of any implementation of constant data for Java should be

1. If a program does not use constant data, its runtime efficiency should not be affected by the option for doing so. Likewise, the resulting size of the executable and its RAM requirements at runtime should not be affected.

2. If a program does use constant data, only the size of constant data or the efficiency by which constant data can be accessed must be affected. In other words - reading and writing to objects or arrays that do not contain any constant data should not be less efficient because of the presence of constant data elsewhere.

To meet these requirements it must be possible to identify and substitute involved byte codes with special purpose byte codes that know about constant data.

In the absence of dynamic class loading, it is statically decidable for the *new*, *putfield*, and *getfield* byte codes if they will be used for objects containing constant data. Creating objects and accessing fields do not involve virtuality in Java - a field cannot be overwritten in a subclass, it can be over shadowed, but it is statically decidable for each *new*, *putfield*, and *getfield* byte code if it will involve constant data fields or not.

For the byte codes used for array creation and access, it is not in general possible to decide statically if they will work on constant arrays or not. Consider the situation where a constant field holding an array is being initialized by calling some virtual method that returns the array. In that case it is generally unknown which *newarray* byte code is being used for creating the array. In the HVM we restrict ourselves to handle array creations as in the following example,

```
public class ConstantData
{
    final byte wav_num_0[] =
        { 23, 112, -1, -1 };
}
```

The code for this can be analyzed and the byte codes annotated so that special purpose byte codes can be used at runtime for the creation and initialization of constant arrays. Even so when reading from an array on Harvard Architecture targets the *xaload* byte code must always check if it is reading from an array in the Flash - this will violate both (1) and (2) above. Currently this problem has not been solved in the HVM implementation.

Other byte codes that are affected by the presence of constant data are the *arraylength*, *instanceof* and *checkcast* byte codes, the first for the same reason as with *xaload*, the latter two because they need to access the object to retrieve its class which again on Harvard Architecture machines must be done differently from accessing normal RAM - hence an extra check is needed in the general case.

To sum up: the cost of supporting constant data in the HVM has been an extra check introduced in the *arraylength*, *instanceof*, *checkcast* and *xaload* byte codes. In Section 6.1 we conclude a worst case cost of approx. 7% increase in running time.

5.1 Impact on GC

Introducing an extra heap (the constant heap) naturally affects the garbage collector (GC). The HVM uses a reference counting GC and to support constant data the following changes have to be made,

- Collecting constant data. Constant data are by nature never garbage. They may become unreachable but they can never be collected since they are placed in Flash which is read-only. The HVM uses 6 bits for the reference count and if the reference count becomes 0x3f the GC will never consider the object garbage. Objects in the constant data heap are given an initial reference count of 0x3f

- Updating reference counts. During the initialization phase it is not needed to update reference counts to objects in the constant heap - they will be marked as immortal anyway, but objects in the normal heap that are referred from the constant heap will have their reference count updated as normal

If a GC strategy uses a marking phase, the constant heap may contain references into the ordinary heap. Such references are additional roots to the marking phase. This set of additional roots never changes after the initialization phase. It seems like a reasonable idea to disable GC during the initialization phase and then after the initialization phase add the constant set of additional roots to the full set of GC roots. Then any mark-sweep collector may safely ignore the presence of constant data.

6. THE MODBUS APPLICATION REWRITTEN FOR JAVA

In the following we will look at the Modbus application from Grundfos. Below is a Java version of some Modbus functionality found in the existing C implementation. This functionality accounts for a large part of the use of constant data in that application. The functionality in the form of constant arrays marks which of a set of 512 registers may be accessed and which may not be accessed. These tables are always consulted before acessing the registers, and depending on the situation, references to different lookup tables are used. If a register may not be accessed, it is marked as *NOACCESS*, otherwise its index into the register array is given. E.g. when a device is configured for the first time by a user with administrator priveleges, it should be possible to access all registers. In this case the *configuratorRegisterMap* is used. When running in the field the software should no longer be able to change everything but only certain parts of the configuration. In that case the *fieldUserRegisterMap*

is used. The register array itself (not shown here) is kept in RAM, but the lookup tables are constant data and can be kept in Flash.

```
class ConstantData {
  public const int[] fieldUserRegisterMap = {
    REG1,
    REG2,
    NOACCESS,
    REG4,
    REG5,
    NOACCESS,
    NOACCESS,
    NOACCESS,
    REG9,
    ...,
    REG512
  };

  public const int[] configuratorRegisterMap = {
    REG1,
    REG2,
    REG3,
    REG4,
    REG5,
    REG6,
    REG7,
    REG8,
    REG9,
    ...,
    REG512
  };
```

The Modbus application contains 7 such lookup tables taking up approx. 3.5 KB of constant memory. In the 4KB version of the target these tables alone would occupy most of the available RAM if translated into Java. Only after adding support for constant data in Flash, is the HVM able to run the code from the example above. Currently, parts of the Modbus application is being ported to Java to gain more experience from using constant data in Flash and from using Java on the Grundfos platforms in general.

6.1 Measuring Cost

When adding a new feature to a Java execution environment, it is reasonable to expect, that programs actually using the feature will have to pay a cost for that in terms of execution time. On the other hand, programs that do not use the feature, should not have to pay for it. In the following we present the result of performing three measurements while executing the same program on the HVM

1. Executing the program while all data are kept and accessed in RAM on a HVM build that does not support constant data in Flash
2. Executing the program while all data are still kept and accessed in RAM but this time on a HVM build that does indeed support constant data in Flash
3. Finally, executing the program while all data are kept and accessed in Flash on a HVM build that does support constant data in Flash

The program we used creates and scans each element in the two lookup tables from the Modbus example described in the previous section. The program was executed on the ATMega2560 and we measured execution time by counting clock cycles.

We would expect a rise in execution time of 3) compared to 1). This corresponds to paying a cost for actually using the feature. But we would like to see as little rise as possible in execution time of 2) as this corresponds to paying a cost for a feature that is not being used.

The results from measuring the cost of adding support for constant data in Flash on the HVM is listed in Table 2.

As can be seen we pay a cost of approximately 7% in scenario 2) above. We consider this a worst case scenario for the following reasons: As described in Section 5 it is difficult to statically decide if array creation and array access op-codes may be used for accessing Flash data, thus they must be able to handle the general case where Flash data may be present. On top of that, array load and array store operations may be executed many times compared to other byte codes, if the scanning of arrays is involved. The test program will execute the byte code *newarray* twice and the byte codes *arraylength*, *iastore* and *iaload* many times corresponding to the length of the arrays.

That there is a cost in actually using the feature and accessing objects in Flash, in 3) above, is not surprising, but of course the implementation should seek to get this cost as low as possible.

6.2 Discussion

The HVM is an interpreter and as such users of it are aware of a degradation in execution efficiency compared to AOT or JIT compiled environments. Such users may be willing to accept an additional degradation of 7.5%. Still, it seems a high price to pay for supporting constant data in Flash. Furthermore the HVM is not optimized and future optimizations of the HVM bringing down the execution time of all byte codes will most likely increase the percentage wise cost of supporting Flash data. A possible option to alleviate this problem is to extend on the static analysis to make it better at deciding which array access byte codes may be used to access Flash and substitute these byte codes for their Flash aware counterparts during compile time. While this is in general an undecidable problem, data flow analysis may in practice be able to bring down the average cost.

7. RELATED WORK

The *final* modifier in the Java language ensures that a variable is not changed after its initialization. For variables of basic type, e.g. *integer*, this works fine but for references it only ensures that the reference itself is not changed - it is still possible to change the target of the reference.

Several suggestions have been made to add “read-only” qualifiers to Java to ensure that both arrays and objects are not changed if marked read-only (e.g. [4] and [2]). The focus for their research is to statically enforce rules about proper use of data and thus be able to catch errors earlier. The focus of this paper is to be able to minimize use of RAM by using Flash instead. These two efforts are synergetic in the sense that any successful suggestion for marking and statically enforcing read-only data in Java would have as a direct side effect that the compiler could generate byte codes to place and access data in Flash.

The suggestion presented in this paper to mark read-only data at the point of declaration does not ensure or enforce that read-only data are indeed read-only, but if the programmer respects the read-only declaration, then we have shown

	Clock Cycles	Cost in %
(1) Without Flash support	179827	0%
(2) With Flash support, not using it	193415	7.5%
(3) With Flash support, using it	240258	33.5%

Table 2: Overview of data usage

how that may carry over to save that data in Flash.

The HVM is similar in its design to the TakaTuka VM [1] and the Darjeeling VM [6]. All three VMs are Flash aware and store byte codes and other runtime structures in Flash. To further reduce RAM use at runtime, this paper describes how runtime items like arrays and objects that are considered read-only can be stored in Flash as well. Since the TakaTuka and Darjeeling VMs are Flash aware interpreter based environments, it is most likely possible to include the design for constant data presented here in those environments as well.

The RTSJ [3] supports accessing physical memory through the physical memory classes and the raw memory access classes. This means that the addressed storage could be of any memory type (RAM, ROM, Flash, etc.). E.g. through the use of the method

```
void getLongs(long off, long[] arr, ...)
```

on the class *RawMemoryAccess* it is possible to read data from Flash and store it in the heap in the *arr* array. As long as this array is in use it will take up heap space. With the design presented in this paper, the array can be referred to directly at its Flash location.

Dividing the execution of a program into several phases is known from e.g. SCJ [17]. A SCJ application is a sequence of mission executions. A mission embodies the life cycle of a safety-critical application. Each mission is comprised of initialization, execution, and cleanup phases. The proper execution of the application is enforced through the user-defined implementation of the *Safelet* interface. The suggestion in this paper for dividing the execution into an initialization phase and a following runtime phase could be greatly improved by adopting similar designs.

The idea of doing a partial execution of the initialization phase is used in other scenarios. An interesting example is the V8 JavaScript Engine used in Google chrome. In order to speed up the instantiation of any additional JavaScript context, the initial setup of built-in objects is done only once to produce what they call a *snapshot*: ".. a snapshot includes a serialized heap which contains already compiled code for the built-in JavaScript code" [8]. The V8 concept of a "snapshot" is very similar to the partially evaluated constant heap described in Section 3.2.

Finally, like C, Ada also supports 'ROM-able objects' [7]. The support stems from the more general concept of 'pre-elaboration', introduced so that real-time programs can start up more quickly. In Ada a certain class of static expressions can be evaluated at compile-time, just like a certain part of expressions in C can. Even though this is introduced to speed up start-up time, the Ada documentation also states that pre-elaborated objects are ROM-able.

8. FUTURE WORK

Explicitly marking constant data (see Section 3.1) and explicitly marking the end of the initialization phase (see Section 3.2) is a reasonable starting point, but this suggestion can be error prone - runtime errors may occur if constant data are marked incorrectly. This invites us to add tool support for identifying constant data automatically and for identifying the end of the initialization phase automatically.

Constant data can informally be described as "data that is written to once, but read from several times". The initialization phase is an execution path that starts from the beginning of the program and until the last constant data have been initialized. Can a static analysis of the application find the maximal subset of the heap that has the "write-once/read-many" propertiy? Can the code point that corrosponds to the transition from the initialization phase to the mission phase be found? If that execution point and subset of data can be found through a static analysis, then the programmer would not have to mark constant heap data, nor have to mark the end of the initialization phase.

The effort described here to use Flash for constant heap data is a small part of a larger effort to make Java an attractive alternative to C for embedded software developers. The overall objective is divided into two parts

1. Make popular features, supported by well known C environments, available to Java developers as well. Prominent examples of such features are (1) device register access (2) 1st level interrupt handling and (3) constant heap data located in Flash.

2. Add tool support to improve on supported features when compared to their support in C environments.

In relation to (1) above, significant improvements have already been made. The support for Hardware Objects [15], 1st level interrupt handling [12] and now constant heap data in Flash may be a prerequisite for many to contemplate making the transition from C environments to Java environments. For example, adding tool support for identifying constant data and the related initialization phase will make the transition from C to Java even more attractive.

The focus of the HVM has not been execution efficiency, but rather to prove that features known from C environments (hardware objects, interrupt handling and constant heap data) - features considered very important by many embedded developers - can indeed be implemented in a Java execution environment. Other Java environments for embedded systems have achieved impressive execution efficiency, e.g.

- Interpretation. The JamVM [10] is considered one of the most efficient interpreters with execution times approximately three times slower than compiled C.

- JIT compilation. The Cacao Java JIT compiler [5] is a highly efficient JIT compiler for embedded systems.

- AOT compilation. The Fiji VM [13], which is an AOT compilation based execution environment for embedded systems delivers "comparable performance to that

of server-class production Java Virtual Machine implementations"

These efficient environments for the embedded target could add support for our features described above, thus taking away an important obstacle for engineers within the embedded industry to consider Java for their next generation development of embedded software.

9. CONCLUSION

This paper motivates why it is important to be able to keep read-only heap data in Flash when executing Java on small embedded devices. Three industrial applications demonstrate the use of constant data in existing C programs - two of these applications will not be able to run in Java if constant data cannot be placed and accessed in Flash. These examples justify the claim that the number of applications that could potentially be implemented in Java on small embedded devices will be significantly greater if constant data in Flash is supported.

Engineers are used to being able to place constant data in Flash in the C environments they use. This paper presents a design for constant data in Java - and an implementation in an interpreter based environment. Engineers will in this respect be able to do in Java what they are used to doing in C. But they will be able to do more than that: the host initialization phase can initialize more complicated structures than is possible in C. As long as data are written just once and the procedure for doing so is deterministic, all usual Java constructs may be used during initialization. In C only a very limited subset of the language can be used to initialize constant data. Thus the design presented here not only adds similar functionality to Java environments, but extends that functionality as well.

10. ACKNOWLEDGEMENT

Thank you to Niels Jørgen Strøm from Grundfos, who stated the original requirement that constant heap data had to be kept in flash, and who formulated the original ideas that formed the basis for our work. Thank you to Dion Nielsen from Polycom for allowing us to use their software. A warm thanks to A.P. Ravn for sharing his great insight and valuable comments on the first drafts of the paper. Also a warm thank you to Hans Søndergaard for reading and commenting on this paper again and again.

Finally, we are grateful for the instructive comments by the reviewers which have helped us clarify several points in this paper.

11. REFERENCES

[1] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing takatuka: a java virtualmachine for motes. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. *SIGPLAN Not.*, 39:35–49, October 2004.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The real-time specification for java 1.0.2. Available at: http://www.rtsj.org/specjavadoc/book_index.html.

[4] J. Boyland. Why we should not add readonly to java (yet). In *In FTfJP*, pages 5–29, 2005.

[5] F. Brandner, T. Thorn, and M. Schoeberl. Embedded jit compilation with cacao on yari. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 63–70, Washington, DC, USA, 2009. IEEE Computer Society.

[6] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[7] T. B. et. al. Ada 9x project report. Revision request report, pages 3.2–3.4, The US Department of Defence, August 1989.

[8] Google. V8 javascript engine, embedder's guide. `http://code.google.com/apis/v8/embed.html`, 2011.

[9] Grundfos. http://www.grundfos.com/. Visited June 2011.

[10] jamvm. http://jamvm.sourceforge.net/. Visited June 2011.

[11] S. Korsholm. Hvm lean java for small devices. `http://www.icelab.dk/`, 2011.

[12] S. Korsholm, M. Schoeberl, and A. P. Ravn. Interrupt handlers in java. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 453–457, Washington, DC, USA, 2008. IEEE Computer Society.

[13] F. Pizlo, L. Ziarek, and J. Vitek. Real time java on resource-constrained platforms with fiji vm. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[14] Polycom. http://www.polycom.dk/. Visited June 2011.

[15] 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.

[16] A. S. Tanenbaum and J. R. Goodman. *Structured Computer Organization, fifth edition*, pages 18, 80. Prentice Hall, Upper Saddle River, NJ, USA, 2010.

[17] TheOpenGroup. Safety-critical java technology specification (jsr-302). Draft Version 0.79, TheOpenGroup, May 2011.