Department of Computer Science
Lund Institute of Technology

Master Thesis

Porting and Evaluation of an Embedded Java Virtual Machine on Palm OS

or Japalm - I love the smell of coffee in the morning

Magnus Landqvist
Supervisor: Anders Ive

23rd February 2000

Abstract

The lack of standardized execution formats and communication protocols for embedded devices has been a problem for developers for a long time. These devices range from industrial systems, like robot controllers and sensors to consumer products, like cellular phones and audio/video units. There is also a need for easier and safer applications development environments for these device.

The Java technology is the answer to these problems. Java is a secure language since it prevents memory errors and leaks by using garbage collection instead of manual memory handling. However, the Java Virtual Machine, or JVM, which is the traditional portable platform for Java applications is not well suited for running on small embedded devices. There is a need for a much smaller and more memory efficient virtual machine. The IVM, Infinitesimal Virtual Machine, is a JVM designed from the ground up for small embedded systems and is being developed by Anders Ive at the Department of Computer Science at the Lund Institute of Technology.

The most important goals of this thesis were the porting and integration of IVM to Palm OS and documenting the porting process and all the possible trade-offs and modifications that were necessary for a small device with a limited programming interface, such as the Palm III.

Additional goals were to evaluate and select those parts of the standard Java API that were needed to be transparently implemented to make the IVM as Java compliant as possible, and to create a highly device dependent, but much needed extended API for the IVM.

The final goal was to create a series of test applications that were targeted to run on the Palm port of the IVM. These test applications include interactive graphical, input and performance tests.

All of the goals were met, with the exception that the API could not be as large as wanted because of memory limitations. The included Java language classes lack some functionality, and the extended, Palm specific, part of the API is limited to event handling and drawing and printing to the screen.

Japalm, however, runs well and will be an important factor in the continued development of the IVM. When memory issues are solved the API can be extended and standardized for all IVM ports, taking it one step further towards API conformity.

Contents

1 Introduction

The lack of a standardized execution format for various small embedded computer systems, both in industry, mobile telephony and in handheld computing, has been a problem for developers for a long time. The advent of the Java virtual machine has made it possible to create a standardized format. The creation of such an embedded virtual machine will minimize communications protocol problems because the devices will use the same Java communication classes, and make programming the various devices much simpler and more homogeneous. Another important aspect of the use of Java is safety, by language and API; the probability of a fatal error occurring is minimized with a carefully constructed API. Java is a secure language, ie it prevents memory errors and leaks by using a garbage collector, and is also type safe in contrast to languages like C. If the API provided to the applications developer is properly protected it will work as an "error filter", not allowing a badly written application to overwrite system parts of the device memory, effectively preventing the device execution from crashing. The IVM is also designed to be scaleable, and to make it possible to update, replace and add Java code in runtime.

IVM, Infinitesimal Virtual Machine, is a small virtual machine being developed by Anders Ive at the Department of Computer Science at Lund Institute of Technology. It is designed to be highly portable to any small computer device that supports a C compiler. The IVM is a research project in progress and has not yet been finished.

The purpose of this master thesis is primarily to port the IVM to Palm OS, and to document the development process. The choice of development platform and environment will also be briefly analysed. Documenting the development process does not only mean writing about the actual implementation and port but also documenting the study of the relevant parts of the Palm platform and its API. There are many important differences between the Palm and the "standard" computer of today. Finally I will discuss and analyse what needs to be taken into consideration when implementing an embedded virtual machine for a small device such as the Palm.

The division of the text will be as follows:

In chapter 2, Inside the Palm, I will describe the hardware and the Palm API and their implications for the porting of the IVM. This chapter also describes the various development environments and tools available for a Palm developer. I will give a more in-depth description of the development environment I chose to use, ie CodeWarrior for Palm OS. In order to learn more about the Palm handheld I constructed a small application which uses many of the important parts of the Palm API. The small application, called WhoHas, is also documented in this chapter.

Chapter 3, Porting the IVM, documents the parts of the IVM that needed to be implemented in order to make a complete port. I will also describe how the implementation was done and the limitations the complete IVM port has.

In chapter 4, Implementing Java Classes and Native Interface, I will describe the Java classes and their native interfaces needed for a useful virtual machine on the Palm. I will discuss the solutions to the problems encountered here, such as making a replacement for the functionality of the widely utilized C-library *stdio*. Finally I will present the test programs I constructed in order to try out the Japalm API classes.

The final chapter 5, Conclusion, contains my finishing analysis and conclusion of my work with the Palm handheld and the Japalm application. I will also present and discuss its performance, both regarding execution speed as well as memory usage. Finally, I will speculate about some future fields of application for the IVM.

Both the Japalm application and the test programs described in the report can be downloaded from the World wide web. See Appendix B for details.

Words which need special explanation, like names or special terms, are written in *italics* the first time they appear in the text, and also explained there. If they are subsequently used they are written as normal words as the reader is assumed to be familiar with them then.

2 Inside the Palm

My knowledge about the Palm handheld computer and its operating system has been gathered from reading in Palm Programming - The Developers Guide [1], Palm Programmer's Companion [2] and Palm OS SDK Reference [3]. Being used to developing for "standard" computers, getting acquainted to the Palm handheld's version of a computer was not altogether easy. The first difference I noticed was the user interface. The Palm is controlled almost entirely, with the exception of seven buttons, by the *stylus* (see figure 1), a pen with a plastic point. The stylus is used both as a replacement to the normal computer keyboard, and as a replacement to the computer mouse. The Palm handheld has special character recognition software built into the operating system. When writing in the *Graffiti writing area* (see figure 1) the operating system interprets the movements of the stylus over the *silk screen* (see figure 1) and sends back the corresponding character to the application.

Another difference, maybe less obvious but which has even more implications for the IVM porting process, is the way in which data is shared with other computers. There is no floppy drive in the Palm, and not even a storage unit that behaves like a hard drive, just an internal static RAM. However, there are two communication ports which can be used in order to connect to other devices or a desktop computer. These ports are the serial port and the infrared port. The infrared port is used mainly for beaming applications and data between Palm handhelds, but can also be directly programmed through the Palm API to do other things. One example of what you might do with it is a universal remote control. The serial port has several uses of which the so called *Hotsync* capability is the most important. Hotsync is the name of the Palm's data- and applications synchronisation with the desktop computer it

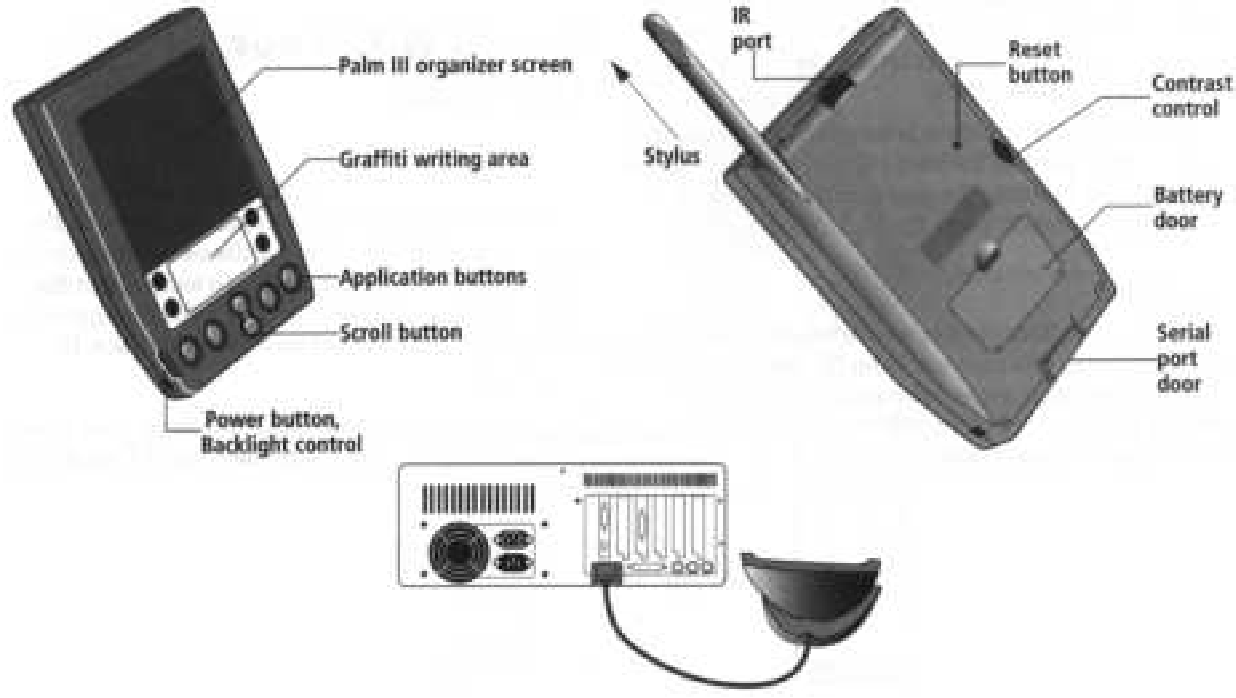

Figure 1: Palm III overview, taken from the Palm III documentation

is connected to through the *cradle* (see figure 1). The cradle is a piece of hardware that can most easily be compared to a serial adaptor cable. When the user wants to synchronize the data on his Palm with the data on his desktop he places the handheld in the Palm cradle and initiates a Hotsync by pressing the *Hotsync button* at the base of the cradle. This opens certain data transferring *conduits*, which are special Hotsync plug-ins. When a Hotsync session is in progress, the Palm and the desktop updates each other's data (or *databases*, as the collection of application data is called on the Palm) with the most recent changes. This procedure might seem very simple, and it certainly is from the ordinary users point of view, but this somewhat uncommon functionality was something I needed to take into account when deciding how to do the IVM port. Naturally the serial port can be used for normal serial communications through the cradle or a special serial cable. The Palm OS also includes support for TCP/IP via PPP on the serial port, using either a direct link or a modem. When developing, the serial port is used for debugging an application directly on the Palm device. It is used for controlling execution and breakpoints in the desktop development environment.

The seven buttons, the infrared port, the serial port and the silk screen defines the physical I/O of the Palm handheld. When I ported the IVM I had to have this I/O-model in mind. First, I had to find out what the implications of not having a keyboard and a mouse were. After careful consideration I realised that the differences were so minimal to the IVM, that this would only affect the native implementation of some classes in a minor way. This was much thanks to the operating system, which provides an API that gives transparency to the input. Since I was going to use the API when implementing the native interface this was not a problem during the design phase of the actual IVM port.

The IVM would not be useful if I didn't implement base classes for communication with an external device or screen. The most immediate need was the need for some kind of debug output to the screen during the development process. Normally, on a standard desktop system, this output would be done using the standard output. However, on the Palm computing platform, there is no standard output (or standard input for that matter). It is true that there exists an additional C module that can be used to implement a simple stdio-application in the Palm SDK. It is intended for development and debug use only, not for the general user, as stated in the Palm OS Programmer's Companion [2]. Since I wanted to have debugging capabilities built into the finished IVM I did not want to use something that limited the graphical interface. Because of this I rejected the special stdio module. To fulfil this immediate need I considered some different solutions. I could either try to implement a replacement for stdio that ran on top of the windowing system, or I could implement classes that interfaced with the windowing system, thus making it simple to pop up message boxes on the screen. Another solution was to use the serial port to log debug outputs on another machine, but since the debugger used the serial port this was not an option in the early development stages.

In the following subchapters I will go more in depth in the different aspects of the Palm handheld that I've briefly explained here. If you are already familiar with the Palm you might want to skip the first chapter 2.1, The Hardware, which serves as an introduction to the internals of the handheld. The second and third chapters 2.2, Developing for Palm, and 2.3, Structure of a Palm Application, can be skipped if you have developed applications for the Palm computing platform before. They serve as basic information providers and point the reader to where he/she can find more in depth information on the subjects.

The two following chapters 2.4, Analysing the I/O, and 2.5, Memory Considerations, contains both information and thoughts about the implications the different aspects of the Palm hardware and its API have on the porting process. These chapters contain the theoretical base I used when deciding how to do the port of the IVM.

The final chapter 2.6, An example program - WhoHas, describes the application I developed as a step in my learning process. This application was never intended to be fully working. It's purpose was merely that I might shed a light on the various aspects of Palm programming that concerned the porting of the IVM.

2.1 The Hardware

The Palm handheld has seven buttons, see 1. They are the power button (with backlight control), the four *application buttons* and the two *scroll buttons*. The application buttons can be used to start the most common applications which are integrated into the Palm OS whereas the functionality of the scroll buttons are defined by the application. Normally these are used for scrolling text on the screen. The main use of the power button should be self explanatory, but if it is depressed for a few seconds the screen backlighting will activate.

The *silk screen* is divided into two areas, the actual LCD-screen and the Graffiti writing area (see figure 1). The Graffiti writing area effectively replaces the computer keyboard with the help of the special character recognition software built into the operating system. To the right and the left of the writing area four tap buttons are located. They are an extension to the application buttons but are activated by tapping on them with the stylus instead.

To input text into the device you simply write the corresponding *Graffiti symbol* of each character in the writing area. In most cases these symbols are very similar to what you would write on paper to produce the character in question.

The actual graphic area of the silk screen has the pixel dimensions 160x160. The screen is entirely monochrome, meaning that a pixel is either on or off.

The "brain" of the Palm handheld is a CPU from the Motorola 68000 series, more specifically either a 68328 (Dragonball) or a 68EZ328 (Dragonball EZ). This means that for developing purposes a cross-compiler generating 68000-code is needed.

2.1.1 The memory model

Table 1 is taken from the book Palm programming - The Developer's Guide [1] and illustrates the dynamic memory layout of Palm OS version 3.0. Palm V and the more recent Palm VII runs version 3.1 and 3.5 respectively. There are also two new Palm III models, the Palm IIIx and the Palm IIIe. However, since I did my development for the regular Palm III I do not find it of importance to document these other devices. Except differences in memory sizes these devices function almost identically to the original.

There is a total of two megabytes of RAM in the Palm device. Most of this is used as data-, resources- and program-storage. This large portion of the memory is called the *storage RAM* whereas the memory available to applications is called the *dynamic RAM*. The storage RAM is write protected and the only way to store data in it for an application is by calling a special Palm OS API-function. The only real way to use this memory as storage for an application is through the database API, which is a part of the Palm OS API.

System resources	**OS 3.0**
System Globals	6 Kb
System dynamic allocation (TCP/IP, IRDA, etc.)	50 Kb
Application stack (call stack and local variables)	4 Kb
Remainder (application globals, dynamic allocation)	36 Kb
Total dynamic memory	96 Kb

Table 1

As can be seen in table 1, dynamic memory can not be taken for granted. I had to take into careful consideration just how to make the IVM port manage to get by on only 36 kilobytes of globals and dynamic allocation.

The storage RAM is similar to the flash ROM/(E)EPROM used by very small embedded systems, eg ARM and AVR. They often have a very limited amount of dynamic RAM available and a ROM area that often is several times larger than the RAM area.

Although there is a 32-bit address space, memory is not accessed using a linear memory model. Physically, the entire RAM memory is contained on a single *memory card*, which is not really a card, but a collection of memory chips whose total capacity is two megabytes. This memory is, as of Palm OS version 3.0, divided into two heaps, the dynamic heap and the storage heap. These areas are in fact the storage RAM and dynamic RAM explained above. In older Palm OS versions the storage RAM was split into many heaps, each at most 64 kilobytes in size. As explained earlier, only the dynamic heap can be written to using direct pointer reference. The storage heap, which contain applications, resources and databases, is write protected and can only be written to using a special API function. The reason for this is to protect the data stored in the storage heap. The storage heap is never cleared (unless the device is powerless for longer than a few minutes). However, the dynamic heap is cleared each time the device is reset.

The operating system uses memory compaction. This is most welcome in the small memory environment of the Palm. Each heap is compacted seperately (only two heaps on Palm OS 3.0 and later versions), and for compaction to work properly a straightforward pointer model is not practical. If a chunk of memory is moved but a pointer to it in the application is not updated, that pointer will refer to the wrong memory. The Palm uses *handles* to overcome this problem. A handle is a pointer to an entry in the *master pointer table*. When memory is compacted, the master pointer table is also updated, making the pointer referred to by the handle still reference the right part of the memory. When a movable chunk of memory is allocated the master pointer table is updated, and the handle of the memory is returned to the caller of the allocation routine. When the application needs to access the memory it first must lock the memory the handle refers to. The API function that locks the handle returns a valid pointer, and the application is guaranteed that this pointer is valid as long as the handle is being locked. When the application no longer needs the pointer it releases the handle with another API function, and that chunk of memory can now be part of the compaction process again. A handle that is locked for too long will cause memory fragmentation. It is therefore advisable only to keep the handle locked while the memory is accessed.

It is, of course, possible to allocate a non-movable chunk of memory with the API, just returning a pointer. This is, however, not advisable as memory fragmentation is almost certain to occur.

2.2 Developing for Palm

An important choice a Palm applications developer has to make is what platform he/she wants to work with. Fortunately, a great variety of development tools[1] exist for all the larger platforms. These include Windows 95/98, NT, Macintosh and UNIX. The official development environment is *Metrowerks CodeWarrior for Palm OS* which exists for Windows and Macintosh. CodeWarrior was originally developed for Macintosh, so a Windows user might find its interface somewhat awkward (I know I did).

There is a free alternative to CodeWarrior, namely a cross-compiling version of gcc[2]. Actually the cross-compiler comes in a special SDK called the GNU PalmPilot SDK[3]. It not only contains gcc but also a resource[4] compiler, *PilRC*, and a graphical resource displayer, *PilrcUI*. The GNU debugger, gdb, also accompanies this SDK. Until recently there was an emulator called Copilot (or XCopilot on Unix). However, this has been superseded on all platforms by *POSE*, the Palm OS Emulator. POSE is a continuation of Copilot and has been taken over by Palm computing, who now is responsible for its development.

The emulator, POSE, can either run an actual Palm handheld ROM image downloaded directly from the device or special debug images with extra debugging functionality and can be helpful when trying out new applications. The emulator can interface with both the CodeWarrior internal debugger and gdb (the latter requires a special piece of intermediate software).

There are some pros and cons for both of the major C development environments. CodeWarrior costs a lot of money whereas the GNU SDK is completely free of charge. It is true that there exists an evaluation version of CodeWarrior that is free to download, but this software has features removed which can only be found in the commercial version. In the revision 5, R5, version you can't debug with the emulator,

[1] All the major development tools can be found from http://www.palmpilot.com/devzone

[2] The GNU C Compiler

[3] Software Development Kit

[4] Besides the actual executable code, a Palm application also contains something called *resources*. These are graphical objects, like forms and menus, or strings. They are created with the corresponding developing environments resource editor/compiler. You could say that the resources describe the way the application looks to the user. A form is like a window; It contains all the elements, like buttons and scrollbars, that the user can interact with. However, unlike the window concept, a form is not movable or resizable by the user.

only directly on a Palm handheld connected the computer through the cradle. The latest version, R6, does not have this limitation, though it has several others:

- A dialog appears when you start your application on the Palm. The user needs to tap it before the execution continues.
- The library-files containing multi-segmented startup code are absent. Since Japalm is to large to fit in one segment this means that it can not be compiled and linked using the evaluation version.
- All of the help files are missing.

With the GNU SDK on the other hand, you can't debug on a real device. This is a feature only CodeWarrior has. Also, if you are not familiar with gdb you will probably find the user interface very limited and awkward. The debugger in CodeWarrior is intuitive, probably thanks to its integration into the IDE[5].

After some consideration I chose to use CodeWarrior since I realized I was going to face some serious debugging in my project. After all, the IVM port was going to be a rather complex piece of Palm software.

Finally I just want to mention that there exist a variety of other development tools for the Palm which enables you to program in different languages (even languages with a graphical representation), but I only mentioned the main C development environments here. The reason for this is of course that this is what I used and considered. The other kinds of environments are not relevant for this thesis.

2.3 Structure of a Palm Application

There exists very good in depth documentation on how a Palm application is constructed, eg Rhodes & McKeehans book [1] and The Palm OS Programmers Companion [2]. The latter is freely available from Palm Computing's web pages. I would recommend it to anyone who wants to get even more familiar with the Palm OS API and programming for Palm.

I will keep things simple and concise in this subchapter. My goal is to describe the structural skeleton, and how a Palm application is driven. It was important for the porting project to get exact knowledge of the application structure in order to make the IVM correctly integrated into a Palm application.

First of all the execution in every Palm application (except specialised stdio applications) starts with the function `PilotMain`. The purpose of this function is to receive and interpret the *launch code* which is passed as a parameter by the operating system. After interpreting this launch code the function decides what kind of execution

[5]Integrated Development Environment

it should do. The launch code generally specifies that a normal application should take place, but others do exist. Examples of other launch codes are whenever a Hotsync takes place, another application executes the application or when a *find*[6] operation has been activated.

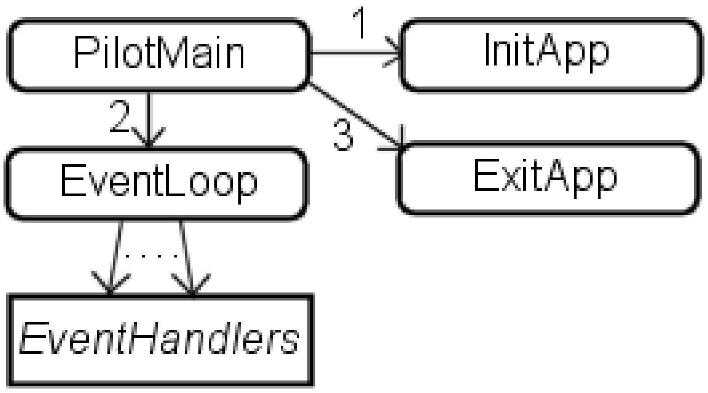

Figure 2: Execution of a Palm application

When the application is started normally, an integer constant called `sysAppLaunchCmd-NormalLaunch` is received, the `PilotMain` function should do three things (see figure 2):

1. Initialize the application, eg by opening databases and opening the initial form, which is the form that is displayed when the application starts. This is preferrably done by a separate function, eg `InitApp`.
2. If the initialisation function does not return an error, start the main event loop function, eg `EventLoop`. This loop should run until the event `appStopEvent` is received. This event is sent to an application by the operating system when the program is finished or aborted by the user. This can happen if the user pushes one of the application buttons or the power button.
3. Clean up after the application and close the used databases. This is also preferrably done by a separate function, eg `ExitApp`.

[6]Find is activated by tapping the special find button. Its purpose is to search an arbitraty applications database for a certain keyword.

The typical event loop, written in C, looks like this:

```
static void EventLoop(void)
{
   Word error;
   EventType event;

   do {
      EvtGetEvent(&event, evtWaitForever);

      PreprocessEvent(&event);

      if (! SysHandleEvent(&event))
         if (! MenuHandleEvent(0, &event, &error))
            if (! MyAppHandleEvent(&event))
               FrmDispatchEvent(&event);

   } while (event.eType != appStopEvent);
}
```

The function `EvtGetEvent` waits until it receives an event from the operating system. The event is then processed by different event handling functions. The functions `SysHandleEvent` and `MenuHandleEvent` are API functions but the function `MyAppHandleEvent` must be written by the programmer. This handler receives the high level events created (or let through) by the previous handlers.

If a handler receives and processes an event, it does not pass it on to the next handler. Instead, it can post a new event into the event queue, which describes what has happened at a higher level. An example of this is when the menu event handler receives an event that says that the user has put the pen down on a menu item. Instead of sending the pen down event to the next handler it creates a new event, notifying the application that a certain menu item was chosen.

The different functions have these tasks:

PreprocessEvent This function is not needed by all applications. It's purpose is to catch events which normally would be handled by `SysHandleEvent`. The reason for this could be that the programmer wants to redefine the functionality of the application buttons for his own application. This is usual in many of the games that exist for the Palm, where the buttons are used to control the gameplay. The event is transformed into another event which is not recognized by the system event handler. It is instead intended for the applications event handler, as described below.

SysHandleEvent This handler takes care of basic things like the application buttons, the power button and Graffiti input. It interprets many low level stylus operations and button presses into more recognizable events. An example of this is the conversion of Graffiti input into characters and translation of taps on the silk screen into its corresponding graphical component tap. The handler creates and sends these new higher-level messages back to the application.

MenuHandleEvent This handler takes care of all menu related operations, like tapping the menu button or tapping on a menu item.

MyAppHandleEvent This handler is not provided by the system. It is instead up to the programmer to write it. Its sole purpose is to catch and handle the `frmLoadEvent` which is sent to the application when the initial form is ready to be shown. First the function recognizes the event, after which it initialises the form and sets it to be the active form. It should then install a special callback handler which is executed each time an event that concerns that form occurs.

FrmDispatchEvent This function is called whenever the event is unhandled by the previous functions. What it does is that it sends the event to the proper callback event handler for the currently active form. These callback handlers are installed by the `MyAppHandleEvent`, and are functions written by the developer. If the callback function ignores the event it is simply discarded.

In CodeWarrior each new application project is created with this kind of skeleton. It even contains a basic form in its resource file and a simple menu. It is recommended to start building an application from this skeleton (called *Starter*).

After gaining this knowledge it was possible to add the functionality I needed. The reference documentation [3] was very helpful in this matter. Since the Palm OS is event driven I had to implement handling for different events like button pushes and list selections in the callback event handler.

Adding resources like forms, buttons, lists and tables are fairly easily done using the Constructor program that comes with CodeWarrior. The new resources can then be accessed through their unique ID:s, automatically defined like constants in the generated resource header.

2.4 Analysing the I/O

This part of the development process was complicated. I had to find out what possibilities for outputting and inputting information existed and what did not. Then I had to decide what kind of output I needed. Finally I had to try to design a solution as good as possible from the available components. As I have mentioned earlier the absence of operating system supported stdio was a big problem. A normal Java interpreter uses stdio a lot. It takes command-line parameters. It can read from the console it runs in and its error messages and normal output goes to that console as well.

It is understandable that an embedded virtual machine might not need (and should not even have) normal stdio, but debug output like error messages from within the virtual machine is altogether a different issue.

After careful consideration I decided that I would limit my debug printouts to a minimum and use non modal dialogs or messageboxes for a start, to display them. This was going to be the first native function I implemented. I used a slimmed version of the standard Java class PrintStream for this purpose. The dialogs were useful in the beginning, when trying to solve hard-to-find bugs and problems, but eventually became annyoing after a while since I had to tap them away every time. When the IVM became stable enough, and I had started to implement graphics I made PrintStream print directly to the screen, making it similar to normal standard output.

Implementing graphical I/O support was another important matter. Making it possible to draw to the screen and to sense stylus taps opened up a much wider range of programming possibilities for the eager programmer. It also served as a coarse performance measure of just how fast the IVM was to respond to stylus movements and drawing operations.

2.5 Memory Considerations

Memory was a big problem. With only 36 kilobytes of dynamic memory available I had to take great care of just how the class preprocessing was done. The garbage collection algorithm currently used in the IVM is a very simple one. It divides the memory in two equally large spaces and then uses the first for allocations. The garbage collector is compacting, and when compaction is initiated the memory elements are copied and compacted to the other half of the memory. The algorithm is described in more detail in Scheduling Garbage Collection in Embedded Systems [4] pp 28-32. Currently the garbage collector has no real time support. When compaction is done the interpretation of the Java application stops until all the memory is copied.

Because of the algorithm, the Japalm IVM application needs approximately twice the memory of the size of the classes. Unfortunately, the memory is even more limited. When a class is being preloaded it uses up more memory than when the loading has finished. Also, a class loads every other class that it needs, unless they are already loaded. This means that a lot more memory is needed when loading classes than in runtime.

It was possible to minimize the effect of the last memory problem described above by statically preloading the base classes needed directly in the Japalm code. This means that the classes could be loaded in an order that didn't need recursive class loading. Also, it would be unwise to compact larger classes last, meaning that classes like String and StringBuffer needed to be loaded first.

When the garbage collector is initialised it preallocates a certain amount of memory. When developing I started out with a heapsize of approximately 20 kilobytes of dynamic RAM, an amount that sufficed well for just a small debugging API. However, once I started adding all the functionality I needed, I quickly had to allocate more memory up to a maximum of 31 kilobytes.

2.6 An example program - WhoHas

The WhoHas application was developed by me during my learning phase of the Palm OS and its API. The reason for spending time on this application was that I should get experience on developing for the Palm so that the IVM port would not be my first application for the device. The intention was to create an application that kept track of things that others borrowed from me. Whenever someone borrowed an item from me I would make a note for that specific item in the application. Each item would thus have a name associated with it, ie the name of the person that had borrowed it. However, an application like this would take a large amount of time to develop fully. Once I had learned enough, and the application worked in a "read only" mode, I decided to move on to the task of porting the IVM.

I tried to make an application which took use of most parts of the Palms features. It included a database which was created and initialised the first time the application ran. This database was then read from by the application as it contained data for the list of things. It was possible to see the status of a marked item by tapping on the edit-button. See figure 3 for a screenshot.

Figure 3: The main screen of the WhoHas Application

3 Porting the IVM

While porting the IVM, I was confronted with several problems. First, in order to do any execution at all, the IVM needed to be able to access the relevant class files. On a "standard" desktop computer this is done simply by reading the files from the filesystem on its hard drive. Since the Palm doesn't have this kind of storage a big problem immediately emerged. How would I get the classes inside the Palm? Second, the IVM was developed using four byte *struct alignment*[7] and *data alignment*[8]. The Palm however uses two byte struct and data alignment by default. This caused quite a lot of trouble during the porting, and solving these problems were a large part of the porting process. Third, quite an effort was put into enabling operating system events to reach the Java level, in order to make an integrated Java API that in the future could be used to program the entire Palm in Java.

The first subchapter 3.1, The filesystem, describes how I solved the storage problem of the Java classes. In essence I coded a file system on top of the Palm database concept.

In subchapter 3.2, Compiler specific problems, I write about all the errors and problems I ran into during the porting process. Problems ranged from struct alignment inconsistency to compiler evaluation order.

Subchapter 3.3, Japalm - a JVM for Palm, describes how the Japalm Palm application integrates and runs the IVM.

In the last subchapter 3.4, Integrating the OS and Java, I describe how I constructed the lowest layers of the Java API. This includes message passing from the operating system and handling system events while interpreting.

3.1 The filesystem

In order to make the IVM run I needed to enable it to access a minimum API consisting of some basic Java classes, like *java/lang/Object*. After studying the Palm reference documentation [3] I found that there did exist a filesystem inside the Palm. Unfortunately there was no easy way to load a file into it from a desktop system. This file system also did not update itself during a Hotsync, as the application databases did. The Hotsync operation is the only way to transfer files from the desktop system to the Palm device. This makes the Palm internal filesystem unusable for anything more than local storage for the applications inside the device.

Another possible solution was to transform the classes into large constant character arrays and compile them into the Japalm application. One of the problems with

[7]Tells how many bytes of padding the compiler should use between the elements of a struct when storing it in memory.

[8]Tells how the start of an allocated chunk of memory is aligned in the address space.

this approach is that this would mean that the application had to be recompiled every time a Java class was changed. I quickly discarded this solution, since better options existed.

I had to find another solution to the problem, one that involved easy adding of files to the file system. The choice fell on the application database. If I could construct a database file (a so called *PDB*[9]-file) and load it into the Palm device, the Japalm application could use it as a file system. The *Palm Install Tool* that comes with the Palm Desktop software pack can be used to install databases as well as applications during a Hotsync operation.

I created a database with the name `JapalmFS-JFIL`, the *creator*[10] `JPM1` and the type `JFIL`. These identificators were used in the Japalm application to open the correct database. The only purpose for this database was to contain the filesystem.

Each record in the database contains exactly one file and its header. The header was constructed by me, and contains the name of the file and the file type used. The file format is described in figure 4 below. The type described in the picture is a byte defining the type of the file in the record. Currently only the type "class file" is used, but there exists support for binary and text files as well in the file system specification.

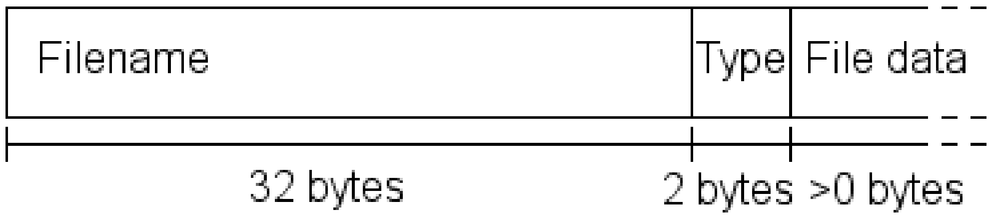

Figure 4: File format specification

Since the Japalm project has a two byte struct alignment, which is standard for Palm applications, the file type entry takes two bytes of space instead if one. It actually only contains one byte of data in the least significant byte.

To be able to create the filesystem I made a small C program called japfs. It uses a configuration file to create the file system. The configuration file consists of several lines, each containing the real filename and the filename in the Japalm file system of a certain file. For example, a classfile which has the path `String.class` and has the name `java/lang/String` in the filesystem. When run, the program takes the name of the output PDB-file and the name of the configuration file as parameters. It then outputs the classes added to the file system and their respective sizes. Finally it prints the size of the entire file system. An example of the output from japfs can

[9]PDB stands for Palm DataBase
[10]Unique identifier of the databases used by a certain application

be found in appendix C.

The japfs program is available from the Japalm webpage, see appendix B.

The only thing that needs to be done after building the PDB-file is to install it into the Palm (or the emulator), using the Palm install tool and Hotsync.

The difficult part was not building the file system though, it was implementing it in the Japalm application. What I had from the beginning was a programming interface specified in a header file in the IVM code. The file operations in that header file needed to be transparently implemented since I otherwise would have to do significant changes to the existing IVM code.

I split the implementation in two layers. The lower layer handled the database management, and opening, locking and reading records. The higher layer contained the file system overlay, ie implementing the required programming interface. I made it possible just to have 32 files open at the same time, in order to save dynamic memory in the Japalm application. I also implemented my own file descriptors and a data structure to keep track of the open files. A maximum of 32 open files might seem like a limitation, but the IVM closes every classfile it has preloaded and if 32 classes would recursively load each other they would run out of memory long before that. Once the memory issues are solved this maximum number is easily altered. The file interface also has support for creating and writing files, but as of now, it is not used by the IVM.

3.2 Compiler specific problems

The garbage collector in the version of the IVM that I started out porting had a peculiar limitation. Its memory needed to be aligned on a four byte boundary. Unfortunately the Palm aligns on two byte boundaries, so this was not always the case. The IVM would halt with a memory error if the memory allocated by the garbage collector was incorrectly aligned. An example of this is when the garbage collector heap is allocated to start at address 0x8002. To avoid this, the garbage collector initializer needs to push the start of the heap forward two bytes to 0x8004. This problem severly limits the portability of the IVM and will be solved in the near future, but as of the printing of this report it still exists.

The IVM also took use of *compiler specific evaluation order*, by using the same macro operation or function several times in an expression. C does not specify in which order they should be evaluated, and CodeWarrior does its evaluation the other way around from gcc or Microsoft Visual C++. These expressions also limited the portability of the code and were rewritten to explicit variants which was not affected by the evaluation order as soon as they were found.

In all the IVM depended and still depends a lot on struct and data alignments. This is not very good for portability. Naturally, uncovering these problems was important and a more portable version is in the works by the time I'm writing this.

3.3 Japalm - a JVM for Palm

Japalm is the shell application for the IVM. See figure 5 for a screenshot of how it looks. It reads the file system and creates a list of all the files within. The user can then choose among the classes and run them by pressing the run-button.

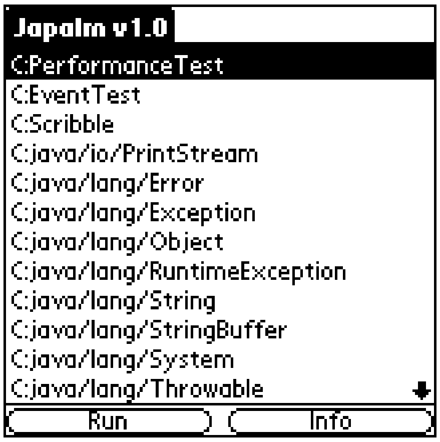

Figure 5: The main screen of the Japalm application

After the Java application has finished running execution returns to the Japalm application and the user can choose another Java application to run. While running a Java application it is possible to interrupt the interpretation as usual by pressing one of the application buttons, the find button or the power button. This is possible since the interpreter examines the operating system event queue on a regular basis. If the interpreter were to ignore the events sent to the application, badly written Java programs (programs that are not exitable) would tend to make the entire Palm device freeze, making a soft reset necessary. This can not happen in the IVM since it always eventually checks the system event queue.

3.4 Integrating the OS and Java

The Palm operating system is event driven. If the user taps on the screen, or writes a character in the Graffiti writing area, an event is passed to the event queue. An important task was to interpret these events and create relevant Java events and insert them into the IVM. This would enable the Java programmer to actually program for the Palm, and not just make simple test programs.

In the final version of my Japalm API I included support for four events that was usable to the Java applications developer:

1. `charEvent` When a character is written in the Graffiti area this event is generated, containing the ascii code of the relevant character.
2. `penDownEvent` When the stylus is brought down on the silk screen this event is generated, containing the x- and y coordinates of the tapped position.
3. `penUpEvent` When the stylus is lifted from the silk screen this event is generated, containing the x- and y coordinates of the tapped position.
4. `penMoveEvent` When the stylus is moved around this event is generated, containing the x- and y coordinates of the tapped position. These events are only generated as fast as the application can retrieve them and are not stacked up in the event queue.

These events are more than enough to make it possible to develop useful Java applications for the Palm device.

The events that are received are translated into Java objects and inserted into a special Java class, called the event receiver. The event checking works as follows:

1. Every N number[11] of *opcodes* [12] the IVM interpreter initiates an event queue polling.
2. The polling is run in a *non-blocking* mode. That is, if there is no event in the event queue the execution does not stop to wait for an event to enter the queue.
3. The polling continues until the queue is empty.
4. The java application might ask for a *blocking* event queue polling. This is a special native call from the Java class that recieves all the events translated into java objects.
5. A blocking call to the polling routine will stop until there is an event in the queue. However, it will only poll once, unlike the non-blocking polling.

4 Implementing Java Classes and Native Interface

After having succesfully ported the IVM and made it work with a very small debugging API consisting of a test class, `java/io/PrintStream` and the basic runtime classes, like `java/lang/Object` and `java/lang/System`, needed for running a test program with output, I started working on the Japalm API currently in use.

My initial intention was to have an as object oriented API as possible. However, memory limitations forced me to simplify the Java event handling and to remove

[11] N is a constant integer compiled into the Japalm/IVM code

[12] An opcode is a virtual machine instruction

functionality from the standard `java/lang` classes, like `String` and `StringBuffer`. In fact, most of the standard classes are simplified to save memory, but the two string classes lack the most functionality. The functionality present is to make it possible to do screen printouts using the concatenation operator, +, eg, `System.out.print("x = "+x+" y = "+y);`, where x and y are integers.

The classes included in the Japalm file system can be found in Appendix C. The first three classes are the test applications and the rest together define the Japalm API.

In chapter 4.1, The Japalm API classes, I describe in more detail what classes were added to the Japalm API and what functionalities were included. I will also write about the native calls used by the various classes.

The final subchapter 4.2, Test applications, describes the applications I created to test the API, and presents results of their execution.

4.1 The Japalm API classes

4.1.1 Base API

The base Java API I started out with was not taken directly from the JDK, but rather a lighter version used in other IVM ports. It contained classes that among other things had support for multi-threading removed. Therefore a lot of native methods were stripped from the source files, making them simpler to implement.

For more information on what classes are used in the standard Java API the reader might want to examine The Java 2 Platform API Specification [5].

The base classes in `java/lang` I included were `Error`, `Exception`, `Object`, `RuntimeException`, `String`, `StringBuffer`, `System` and `Throwable`. The three error- and exception classes were unnecessary to include since error- and exception handling is not yet fully implemented in the IVM. It is, however, possible to throw exceptions. For this reason, the IVM explicitly loads the `Throwable` class.

The Japalm (and IVM) API is based on the minimality principle. Only those methods that are really useful and needed are left in the different API classes. The Java API is much richer in functionality, but is also of a considerable size.

The Japalm API class `Object` is a very basic version of the Java API equivalent. The only method implemented is the `equal`-method. Support for threads, string conversions and a number of other funcionalities are removed. More of these methods will be added when the memory problem has been solved.

Both `String` and `StringBuffer` are stripped down to a minimum. At the moment they are mostly intended to be used together with `System` and `java/io/PrintStream` to create debug print outs. This will also change in the future when more memory is available.

The only class in the base API that has native functions is `java/io/PrintStream`. There are three functions that are intended to print Java strings, integers and characters. These native functions are compiled into the Japalm application and directly executed whenever they are used in any Java application running on it. These native functions use a specially written printing routine, that creates a buffer of written lines and can scroll out the oldest lines at the top of the display when a new line is written, making it look a little like normal stdout-printing.

4.1.2 Extended API

Some classes were added to the API to make it possible to utilize Palm specific from Java programs. The classes are collected under the package `ivm` to distinguish it from the standard Java API.

The most important two of these classes are `ivm/IvmEventReceiver` and `ivm/IvmEvent`. These two classes are explicitly loaded by the Japalm code before a Java application is executed. The event queue polling, regularly executed by the interpreter, needs them to be preloaded when it receives an event that is of one of the four event types that are translated into Java objects.

The `IvmEvent`-class has four attributes:

- `public int eventType` The type of the event, either a character event or a pen up, down or move event.
- `public char theChar` If the event is a character event, this attribute is valid, and contains the character written.
- `public int theX` If the event is not a character event, this attribute is valid, and contains the X position of the stylus.
- `public int theY` If the event is not a character event, this attribute is valid, and contains the Y position of the stylus.

The `IvmEventReceiver`-class is a static class and does not need to be instantiated. All methods and attributes are static. The event receiver contains a ring buffer, currently with 64 slots that can contain events of type `IvmEvent` sent to it. When the buffer is full, and a request to add a new event is being made, the oldest item in the queue is replaced with the new one. Three of the methods in the event receiver are not native. These are `putEvent`, which is executed from the event queue polling loop executed by the interpreter, `getEvent`, which does a non- blocking event retreive and `waitEvent`, which does a blocking event retreive.
The rest of the methods in this class are native, all of which are implemented in the Japalm application:

- `private static native void pollSignalQueue()` This call is used by the `waitEvent` function to do a blocking class to the event queue polling function normally executed by the interpreter (as described in chapter 3.4, Integrating the OS and Java).
- `private static native void togglePollingBlock()` This is also just used internally and is used by both of the non-native methods in the event receiver. The effect of this call is that it prevents the interpreter to initiate an event queue polling while it is toggled on. The first time it is called it sets the polling block toggle to on, and a second call switches it off. Two of these calls encapsulate most of the code in the non-native methods.
- `public static native int getSeconds()` This call actually has not got much to do with the intended functionality of the event receivers. It returns the time in seconds since the last time it was called. Useful for coarse time measurement in the Japalm Java applications.
- `public static native int getTicks()` This call is similar to the previous, except that it returns *ticks* instead of seconds. A tick is a short time interval which is not strictly defined. On a real Palm handheld device a tick is $\frac{1}{100}$ of a second, and on the Palm emulator, POSE, it is also supposed to be $\frac{1}{100}$ of a second, but measurements I conducted while testing the performance test Java application shows that this constant is somewhat uncertain. Running the emulator under Macintosh, the number of seconds per tick is documented to be 60, however I have not been able to verify that since I lack access to a Macintosh computer with CodeWarior installed.

  The ticks count is, however, useful for finer relative timing measurements in Java applications. 4.2, Test applications, under the subsection PerformanceTest.

When the event queue polling operation receives an event that should be translated into Java, it creates a new instance of `ivm/IvmEvent` and executes its corresponding constructor, making sure it gets the right parameters. The `putEvent` method is then called to insert the object into the event receiver queue. All these operations are done in the Japalm event queue polling code, and not in the Java interpretation.

The interface to the screen operations are in a class called `PalmScreen`. It currently supports clearing the screen and drawing lines. Both of these operations consist of native calls:

- Three constants, `public final static int`, `whiteCol`, `blackCol` and `grayCol` which are used as parameters to the `drawLine` method described below.
- `public static native void clearScreen()` This function clears the entire screen.

- `public static native void drawLine(int fromX, int fromY, int toX, int toY, int color)` Draws a line with the color `color` from the coordinate `(fromX, fromY)` to `(toX, toY)`.

4.2 Test applications

4.2.1 PerformanceTest

This application first calls `getSeconds` and `getTicks` to initate the timers. It then executes a bubble sort in interpreted Java code, after which the time the sorting took is presented, using the values returned from yet another call to the two timing functions. After that the `testProgram` native call is done, which executes a small test program compiled into the Japalm application.

The Java and the C version of the sorting routine are identical. They bubble sort 100 numbers that are placed in an inverted sorting order so that every element is swapped every time the sorting loop executes. The program then prints how many ticks the sorting took. For more information about bubble sorting I recommend the reader to look it up in an introductory algorithms textbook, eg Cormen, Leiserson & Rivest [6].

Figure 6 shows a sample session.

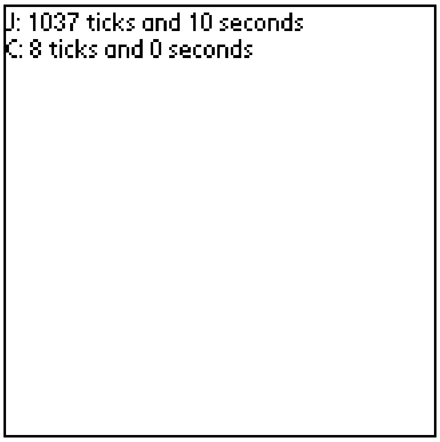

Figure 6: A session of PerformanceTest, run on a real Palm device

The times in figure 6 are measurements of how long the sorting took in Java and in C, when the interpreter polls the event queue every 10th op code being interpreted. This was the standard setting for the IVM, but polling that often severly degrades Java performance, so I decided to change this to one poll every 100th op code. The results are in table 2.

Event queue polling frequency	Java	C
10	3208 ticks	8 ticks
100	1037 ticks	8 ticks

Table 2

Since one tick corresponds to 0.01 seconds the sorting takes 32 or 10 seconds to run in Java and just 0.08 seconds in C.

Because the Java and the C compiler optimize differently it is impossible to calculate a speed difference factor, but it can be said that the bubble sort operation takes approximately 100-400 times longer to do in Java in this particular case. It is reasonable to draw the conclusion that the IVM is not very speed efficient, but not that all Java applications are as much as 100 times slower than their corresponding C programs. The C version can take use of registers and other efficient speed optimizations which the Java version can not. Without these optimizations the C version would take longer to run. In fact, I compiled a special version of Japalm, without any optimization, and the result was 1136 ticks for the Java version and 12 ticks for the C version. The C version became 50 percent slower while the Java was just 10 percent slower.

It is worth mentioning here that the IVM author has not yet entered the optimization phase of the development. It should run much faster in the future.

The Palm emulator presents strange results when timing. According to the system function that returns the number of ticks per second, the emulator should also run 100 ticks per second, but the timing results differ greatly. I will therefore not present any timings done in the emulator, nor will I recommend anybody to use timings produced by the emulator.

4.2.2 EventTest

This application consists of a loop that does blocking waits for system events. Once an event is received the application prints the received event and its data to the screen using the `PrintStream` class.

Figure 7 shows a sample session.

The EventTest application continues to loop until it receives a character event with the character data byte set to the letter q. It then exits the loop and the application quits.

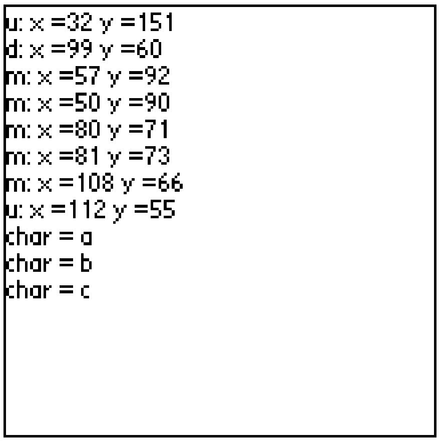

Figure 7: A session of EventTest

4.2.3 Scribble

Scribble is a program which allows the user to draw on the screen with the stylus. Putting the stylus down to the screen starts a line which continually updates itself while the user moves the stylus around on the screen. After lifting the stylus the line ends and the user can start drawing another one.

Figure 8 shows a sample session.

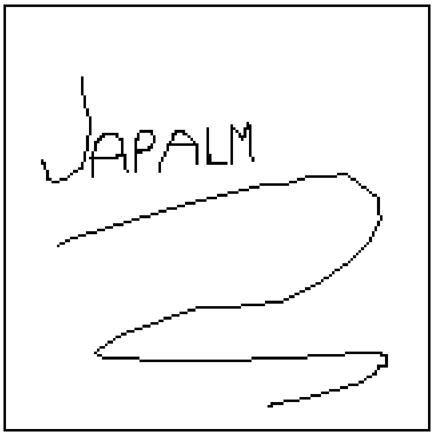

Figure 8: A session of Scribble

This application takes use of three of the four system events, namely penDown-, penUp- and penMove-events. It also utilizes the line drawing functionality of the `PalmScreen`-class as described in subchapter 4.1.

Note that there is no way to end Scribble. You have to exit Japalm and restart it.

5 Conclusion

Now, at the end of my project I have a working port of the IVM running on my Palm III device. It has a filesystem in a single Palm database, which contains the Java API and the test applications used. Basic event handling works, and it is possible to output text to the display in a manner similar to "normal" stdout printouts. The screen is also drawable, however the only graphic primitive supported right now is a line.

The Palm Pilot port of the IVM, Japalm, will prove to be very useful for the IVM research that will continue to run, even if I'm finished with my part. It will serve as both a testing bed for the Java API and applications implemented for the IVM, and will also contribute a great deal to making the IVM even more portable, due to its many peculiarities and its byte ordering, alignment and evaluation order.

5.1 Performance

Japalm performs adequately in execution speed, when taking into account that the IVM code has not yet been optimized. However, memory performance is not yet very good, mainly because of the garbage compaction algorithm used.

The speed measurement made by the application PerformanceTest is not very representative for the speed difference between C and Java programs. The small sorting routine is not an entire application, just a small loop. The C version can take use of registers and other advantages. The scribble application is probably not as much slower than a C equivalent, because of the time it takes to write to the display would be the same, and writing to the display is done very often when scribbling.

5.2 Limitations

As noted before there is a considerable memory limitation. When implementing the API I had to do several trade-offs because of lack of memory. As an example, I had to make just one event class, instead of many different, inheriting from a common superclass as I wanted. That is, however, the only true limitation Japalm has at the moment. Support for multi-threading, smarter memory usage (saving pre-processed classes in the storage heap) and execution optimization is in the works by the IVM author.

A minor limitation is that the class chooser in the Japalm application does not discriminate between runnable classes, containing main-methods, and API classes. If the user tries to run an API class, Japalm will halt with an error.

5.3 Encountered problems

Some of the problems I encountered while porting the IVM has been discussed in previous chapters of this report. Most of these were due to portability issues in the IVM code. Below is a list of the most serious problems encountered, including those already discussed:

- Memory that was not aligned properly to four byte addresses would make the garbage collector cause a memory error that halted the execution. This limitation will be removed in a future IVM version.
- CodeWarrior uses a 2 byte struct alignment as default. The IVM code assumed a 4 byte struct alignment, causing memory errors if no Palm specific changes were made to the IVM code. This limitation will be removed in a future IVM version.
- Some expressions had several calls to the same function. The evaluation order is not specified in C, and CodeWarrior evaluated these differently from gcc and Microsoft Visual C++, which has been used in earlier IVM ports. These expressions have been reprogrammed and are now completely portable.
- Some comparing opcodes gave incorrect results and caused some confusion when running Java in the device. These errors are corrected.
- The event polling and Java queueing gave some unexpected results, due to automatic event polling from the interpreter. The standard setting in the development IVM I ported was to poll the event queue every ten opcodes. This would sometimes cause trouble if a polling was being made when running Java code inside the event receiver class, `IvmEventReceiver`. The Java event queue most times ended up with the wrong number of elements, making some events just disappear. This problem was solved by disabling polling with a special native call every time sensitive information was processed in the event receiver.
- Memory tended to become full even if just a few classes were loaded. This was due to the recursive class loading of the IVM. Every time a class was being loaded into memory it would load all the classes used by that class, causing a lot more memory to be needed, since a class needs more memory when it is being loaded than when it is loaded.

5.4 The future

The IVM and its ports are still in their infancy. Seeing through all the problems and limitations the IVM currently has it is easy to see its future role as an advanced virtual machine for embedded systems. It has been designed, right from the start, to support real-time garbage collection and dynamic loading of Java code.

5.4.1 Future work

Here follow some suggestions on future work with Japalm, not including the internal IVM functionality:

- If the Java classes could be preprocessed and precompacted before putting them in the filesystem, or if they could be processed just once and then written back to the filesystem in the device itself, it would not be necessary to load them into the garbage collector heap. In comparison to the dynamic heap, the storage heap is huge. If the classes could somehow be made to reside in the storage heap, in this case the file system database, it would be possible to have a much larger and more useful API for the IVM. This method would require a rewrite or an extension to the file system layer in Japalm/IVM to support direct access to the memory the files reside in. The technical difficulties of this approach is not at the Palm side, but in the way Java classes work. The way the IVM handles classes and the pointers and tables inside them needs to be rewritten to support this.
- Constructing a loader application that includes the Java application to be executed would add flexibility to Japalm. This would make each Java application visible as a program in the Palm application selector. When executing a Java application, the loader should in turn run Japalm, sending it its own internal database, containing the classes. This database would be temporarily added to the internal Japalm filesystem while the application executes.
- The Japalm class chooser should prevent users from running classes that do not contain a main-method.
- Making it possible to process application button presses in Java would add more flexibility to the Java applications developer.

The following suggestions concern the IVM:

- The garbage collector needs to be replaced with one supporting real time garbage collection. The IVM author is currently planning to use a scheduling mark-sweep algorithm which is described in Scheduling Garbage Collection in Embedded Systems [4] pp 85-88.
- If the memory becomes full the IVM execution is likely to stop. It is strongly recommended that this is detected before an application error can occur. If the memory becomes full it is desirable that an error message appears, and that the IVM terminates gracefully.
- The file system layer needs to be revised. An embedded device almost never has a storage device other than its RAM and ROM. IVM could take use of the fact that classes already are loaded in memory. Currently, the IVM reads the classes byte by byte, as if it was reading a normal file, even if it is actually reading from memory.

5.4.2 Future uses

There are a lot of areas in which the IVM technology will be applicable in the future. I will try to describe some of them here.

An example of a future use is a video and a remote control both containing the IVM but having no other knowledge of how the other device works. They just share a common API for TV/Video-equipment. The first time the remote is used it downloads its control code from the video, while both the video and the remote are running their Java applications. The remote then executes the newly integrated code and can now control the video.

IVM and Java technology could also be used to unify different models of robot controlling and sensor readings by creating standard protocols on top of the existing ones.

The real-time garbage collection is an important factor for industry applications. To be able to use Java in real-time applications the garbage collector must not stop interpretation to clean up the memory. If the collection can be distributed evenly throughout the applications interpretation real-time interpretation is possible. This is much needed in factory systems and robots since they need to be updated continuosly by their sensors. No stop for garbage collection is acceptable.

A handheld computer, which is somewere between desktop computers and small embedded systems in complexity, memory size and processing capability, will prove to be very useful as the "remote" described in the first example, but in a much wider perspective. It will, in the future, be possible to control and measure all devices running Java with a small handheld with a radio- and IR-link.

The Palm device may very well be that handheld, running the IVM.

A References

References

[1] Neil Rhodes & Julie McKeehan, *Palm Programming - The Developer's Guide*, O'Reilly 1999.

[2] *Palm OS Programmer's Companion (Preliminary)*, 3Com Corporation 1999.

[3] *Palm OS SDK Reference (Preliminary)*, 3Com Corporation 1999.

[4] *Scheduling Garbage Collection in Embedded Systems*, Roger Henriksson 1998.

[5] *The Java 2 Platform API Specification*, Sun Microsystems 1999.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, MIT Press 1990.

B Webpage and file availability

All files relevant to this report can be found at my Japalm webpage:
http://www.cs.lth.se/~magnusl/japalm
The Japalm application and the database containing the test programs mentioned as well as the documented API is available for testing. To download the source code for anything else than the file system construction program you have to contact the Department of Computer Science at the Lund Institute of Technology.

C The Japalm filesystem

The table below is a printout of the file system creation application, japfs. It is included to show what classes were included in the final file system described in the report.

The numbers seperated by the slash are the size of the file and the size of the database record created from the file.

```
C:\exjobb\japfs-current>japfs japalm.pdb japfs.cfg
PerformanceTest                 class  1132/1166
EventTest                       class  1157/1191
Scribble                        class  677/711
java/io/PrintStream             class  336/370
java/lang/Error                 class  283/317
java/lang/Exception             class  291/325
java/lang/Object                class  242/276
java/lang/RuntimeException      class  305/339
java/lang/String                class  1360/1394
java/lang/StringBuffer          class  1560/1594
java/lang/System                class  338/372
java/lang/Throwable             class  545/579
ivm/IvmEvent                    class  579/613
ivm/IvmEventReceiver            class  980/1014
ivm/PalmScreen                  class  365/399
Wrote 15 files and a total of 10860 bytes
```