IST-002057 **PalCom**

# Palpable Computing:

***A new perspective on Ambient Computing***

## Deliverable 40 (2.3.2) Runtime Environment

Due date of deliverable: m 36
Actual submission date: m 36

Start date of project: 01.01.04
Duration: 4 years

University of Aarhus

Revision: 1

| Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006) | | |
|---|---|---|
| **Dissemination Level** | | |
| **PU** | Public | **PU** |
| **PP** | Restricted to other programme participants (including the Commission Services) | |
| **RE** | Restricted to a group specified by the consortium (including the Commission Services) | |
| **CO** | Confidential, only for members of the consortium (including the Commission Services) | |

**Integrated Project**
**Information Society Technologies**

# Contents

# 1 Executive Summary

This deliverable describes progress made in a number of key areas during months 25 to 36 of the project in work package 3 (WP3) dealing with the specification and reference implementation of the PalCom Runtime Environment.

This report along with its accompanying implementation prototype constitutes an update of Deliverable 22 [23].

The objectives for the period were described in the updated Description of Work [32] for the project, and this report documents, that substantial progress has been made with respect to these objectives.

# 2 Contributors

The following people have contributed to this deliverable:

- Peter Andersen, University of Aarhus,
- Jeppe Brønsted, University of Aarhus,
- Erik Corry, OOVM/University of Aarhus,
- Henrik Gammelmark, University of Aarhus,
- Michael Lassen, University of Aarhus,
- Ulrik Pagh Schultz, University of Southern Denmark/University of Aarhus,
- Jesper Honig Spring, Ecole Polytechnique Fédérale de Lausanne (EPFL),

Please direct comments to *palcom2-wp3* **at** *ist-palcom.org* or directly to selected authors.

# 3 Notational Conventions

In what follows, the PalCom Virtual Machine is denoted `pal-vm`[1], "PalVM" or simply the "VM".

The code examples used in this document are written in Java [7] and Smalltalk [16]. When referring to the Java method `m` of the class `C` we write `C.m`, whereas we write `C>>m` for the Smalltalk method `m` of the class `C`.

[1]The VM used to be called `pre-vm-c`, but was renamed to `pal-vm` as part of a general renaming of the tools in PalCom to have `pal` prefix

# 4 Introduction

This Deliverable constitutes an update of Deliverable 22 [23]. This report along with its accompanying implementation prototype updates the specification of and describes the implementation of the PalCom Virtual Machine - PalVM.

The deliverable describes progress made in a number of key areas during months 25 to 36 of the project in work package 3 (WP3) dealing with the runtime environment for PalCom. The purpose if this report is twofold: On the one hand to demonstrate the progress with respect to the objectives set out for WP3 for the last 12 months, and on the other hand to function as a (partial) reference for users of the `pal-vm` and the accompanying tools.

The objectives for this deliverable were derived from the outstanding open issues listed in the previous WP3 Deliverable 22 [23, Section 14] and the feedback from the second PalCom Review.

The objectives are included in the updated Description of Work [32]. For convenience relevant parts are repeated below.

**Toolbox Contributions**

1. *Further specification of palpable runtime environment.*
   This will include defining what is to be handled at VM level, and what is to be handled as libraries or services on a higher level. This task also focuses on defining what is to be part of the base class-libraries of a minimum PalCom platform. This work relates to the scalability challenge, by determining the minimum devices supported.
   Furthermore clarify what parts of the VM are platform independent and what parts need special attention on specific devices. This work relates to the heterogeneity/coherence challenge in the focus on different basic execution platforms.
   *Improved Virtual Machine Reference implementation & supporting libraries*
   The reference implementation of the runtime architecture in the embedded PalCom VM will be refined in a number of ways. This includes addressing a number of the still open issues listed in Section 14 of Deliverable 22, like improved process handling, improved support for preemptive scheduling (both these address construction/deconstruction at the service level as well as understandability for the programmer), improved exception handling (necessary mechanism for a number of the challenges addressed by contingency handling, see below), improved garbage collector, reduced memory consumption, optimisations (all three relate to scalability, especially when using very small devices).
   The `pal-vm` further address heterogeneity vs. coherence in abstracting away different hardware platforms for the programmer and allowing access to third-party (legacy) code (e.g. C code).
2. *Support for Resource- and contingency management*
   A prototype implementation of the primitive mechanisms in the Runtime Environment for Resource- and Contingency management Specification will be developed in cooperation with WP5. As can be seen from the WP5 work plan, this contribution provide basic mechanisms that address a number of the PalCom challenges, for example visibility and inspection (of available resources and of erroneous situations), (resource-constrained) construction and re-construction, automatic as well as user-controlled handling of errors, stability in the event of changes caused by errors, understandability of what is wrong in contingency situations. See also item 5 below.
3. *Continued support for networking*
   Continued low level VM support for the essential discovery- and communication components specified and developed in WP4, including BlueTooth. As can be seen from the WP4 work plan, this contribution provide basic mechanisms that address a number of the PalCom challenges, e.g. construction (by discovering what services and resources are available, and establishing connections), visibility and inspection (of discovered devices, services, connections), sense-making & negotiation (automation) in dealing with heterogeneous and changing network topologies.
4. *Support for introspection and reflection*
   A reflection mechanism, possibly based on the ”reflective channel” principle known from OSVM (from previous PalCom partner OOVM) will be added to the VM specification and implementation. Furthermore, support for HMAPs in the VM will be improved.

**Toolbox Contributions, continued**

5. *Support for program construction, analysis and supervision*
   Continued support for program construction including improved compilers `pal-j`, `pal-st` and `pal-beta` as well as the underlying `pal-asm` bytecode assembler. These tools address the challenge of construction (of palpable programs) Support for program de-construction, analysis and supervision in the form of the `pal-dis` bytecode disassembler and a (possibly remote) debugger for `pal-vm`. The language interoperability support provided by the compilers and the VM address the heterogeneity/coherence PalCom challenge on the programmatic level, as well as supporting increased code reuse and understandability (allows you to program in a more familiar programming language, depending on your programming background).
6. *Investigate possibility for offering palpable qualities on top of JVM*
   It will be investigated to what extent it is possible to add palpable qualities on top of JVM for those parts of a palpable system, that do not run on the `pal-vm`. All above-mentioned challenges are potential candidates for this, but most likely topics are means for resource visibility and inspection built as a library on top of JVM.

**Tasks**
*Task 1: Support for resource and contingency management and code base for open source*
Leading up to milestone 1, month 33, this task focuses on improving the VM and preparing the VM and the base class libraries for the internal release of the open source code base. The improvements will focus on toolbox contributions 1-4 above. If time allows, toolbox contributions 5 and 6 may also be addressed. (IM33)
*Task 2: Further specification of runtime and improved code base for application prototypes*
Leading up to milestone 2, month 35, this task is twofold: One objective is dealing with issues arising from the internal use of the open-source code base in order to mature this for the external announcement at IST-Event 2006. Another objective is to provide input to WP2 with an updated specification of the Palcom runtime environment model. (IM35)
*Task 3: Deliverable 2.3.2: Revised VM with resource and contingency support*
This task deals with collecting the results of tasks 1 and 2. The result of this task will be the report that together with the reference implementation of the VM constitutes Deliverable 2.3.2, end of month 36. (D2.3.2)

**Deliverables**
*Month 36: Further specification of and reference implementation of PalCom runtime environment (report and prototype) (2.3.2).*
As described in task 3 above, the purpose of this deliverable is to catch up on the development of the Palcom Runtime model and reference implementation since the previous WP3 deliverable 22 in month 22. The report will describe new parts of the model and implementation in detail but refer to deliverable 22 for the basic ideas. As described in task 1, the major new toolbox contributions expected deal with runtime support for resource and contingency handling. Almost all dimensions of the PalCom challenges are addressed here, cf. toolbox contributions 1-4 listed above: scalability/understandability, heterogeneity/coherence, construction/deconstruction, visibility, sense-making & negotiation, change/stability (each explained in detail in the toolbox contribution list).

Below it is outlined how the above objectives have been addressed for the last 12 months. The work in WP3 has focused on these objectives, but has also been driven by needs from other PalCom work packages that arose throughout year 3 of the project.

# 5 Structure of this Deliverable

The rest of this document is structured as follows:

Firstly, in Section 6 it is described how the `pal-vm` implementation of the PalCom component model has been updated and concretised.

In Section 7 the core concepts of the `pal-vm` and the execution environment of the bytecodes are described. The details of the individual bytecodes supported are described in Appendix B.

In Section 8 the notion of a process is explained, and how processes allow for “sandboxing” of execution. This is followed by Section 9 where threads and schedulers are documented, including how to work with hierarchical schedulers and preemptive scheduling, hereby addressing this Toolbox Contribution 1 element.

Section 10 explains the mechanism for calling native code, e.g., C functions. This addresses the Toolbox Contribution 1 element on third-party (legacy) code interfacing, and also relates to Toolbox Contribution 3, since most support for networking is now done using this native interface.

Immediately following this, the related issue of VM internal interoperability between languages is described in Section 11. This addresses the mechanisms needed to handle calls from one language to another, and subclassing of classes written in another language.

Section 12 address the Toolbox Contribution 1 element on improved exception handling mechanisms now supported by the VM. This also relates to Toolbox Contribution 2.

Section 13 address the resource handling element of Toolbox Contribution 2 by detailing the primitive mechanisms added to the VM to support first order resource handling.

Following this, Section 14 addresses Toolbox Contribution 4 by describing a design proposal for a split of the VM into a non-reflective and a reflective part. Furthermore the support for HMAPs in the VM is documented.

Section 15 describes a possible serialisation design for VM processes, and how this might be used for migration and persistence mechanisms.

In Section 16 it is argued why performance is important for the practical use of the VM and how this is addressed. This relates to the Toolbox Contribution 1 elements on improved garbage collection, reduced memory consumption and optimisation in general.

Toolbox Contribution 5 on program construction, analysis and supervision is addressed in the next two sections, 17 and 18, which describe the compilers targeting `pal-vm`, the language restrictions imposed by these, and the (plans for) debugging facilities on the VM.

In Section 19, new usages of the VM within the project are described. The focus is on the Tiles application prototype from Work Package 11.

Section 20 describes the programming model used when programming for the `pal-vm`.

Section 21 lists the execution platforms, that `pal-vm` currently runs on, and which are planned for the near future, and specific platform dependencies.

Finally, Section 22 lists the major outstanding open issues, whereafter Section 23 summarizes the work done, and compares it with the objectives.

A number of appendices are available from page 83 and onwards. These are meant as references for programmers and compiler constructors targeting `pal-vm`:

Appendix A documents the code base contributions relating to `pal-vm`, and how to obtain them.

Appendix B details each bytecode, that the interpreter understands. This list was also included in the previous WP3 deliverable [23], but has been updated to reflect the current VM.

Appendix C documents the textual format, that is understood by the PalCom bytecode assembler `pal-asm`.

Appendix D documents the textual Palcom Component Specification files understood by the source language compilers, and Appendix E contains a grammar for the binary structure of component files.

Appendix F describes in detail the reflection data, that can be embedded into a PalCom component.

Finally, Appendix G gives an overview of the minimal base class library. This relates to the Toolbox Contribution 1 element of identifying a minimal base class library.

# 6 Pal-VM Component Model

We start this deliverable by describing in more detail the `pal-vm` implementation of the PalCom Component Model previously given in Deliverable 22 [23].

A `pal-vm` component is a binary deployment unit for classes and data. A component contains executable code in the form of classes (and methods), and metainformation. The persistent components are located as component files on disk or accessible from the network in a format described in Appendix E. The `pal-vm` loads the persistent component into its runtime equivalent (the runtime component), when creating a process.

The metainformation (also called reflection data) is used for a number for purposes:

- Guiding the loading process: Required metainformation informs the `pal-vm` as to how to load components into a process. See below.
- Type checking: Interface Description Annotations (IDA) give type information on the classes and methods in the component, which is currently used by the `pal-j` for type checking and name mangling. See sections 11.3 and 11.4.1.
- Debugging: Name information and line number information can be used to implement debugging and inspection functionality. See Appendix F.

The `pal-j` and the `pal-st` compilers generate binary components from a component specification file and a collection of source files for the classes. The syntax of the component specification file is common for `pal-j` and `pal-st`. See Section 6.1 below and Appendix D for further details.

The required metainformation is as follows:

- **name**: The name of the component.
- **requires**: A list of dependencies to other components. When a component requires another component, the required component is given a local name in the name space of the requiring component.
- **main class**: The name of the main class in the component.

When a component is loaded into a process, first all required components are also loaded transitively, then a singular instance of the main class is instantiated. Executing the component consist of calling the `run` method on the singular instance of the main class.

PalCom components are specified using a Palcom Component Specification file (PCS), described below.

## 6.1 Component Specification Files

The Palcom Component Specification file (PCS) is used as input to compilers that compile class source files into binary components. The component specification file contains the list of classes to be compiled into the binary component, the name of the main class, and a list of components required by the component being specified. The component specification files are currently used by the `pal-j` and the `pal-st` compilers.

A language may have its own constructs that are more natural for describing components than the component specification files, and choose to use these constructs instead. In future versions of `pal-j`, the use of component specification files may be replaced by constructs from the Java programming language. The Smalltalk language does not have any construct for describing components, and therefore the component specification files will continue to be used for `pal-st`.

Examples of use can be found in the code base (see Appendix A) at `palcom/doc/tutorials`.

One such example from `palcom/doc/tutorials/interop/basename/src/jbank/jbank.pcs` is:

```
component jbank {
  class JBank
  class JHistoryAccount
  mainclass JBank
  requires "stbankaccount.prc" as stbankaccount
  requires "java.lang.prc" as java.lang
}
```

This file specifies a component named, `jbank`, containing the classes `JBank` and `JHistoryAccount`. The `jbank` component requires the Smalltalk component `stbankaccount.prc` and the standard Java component `java.lang.prc`. The `mainclass` declaration specifies the class to execute an instance of, when creating a process using this component, see Section 8. The `mainclass` declaration is optional for Java, since the `pal-j` compiler can infer the main class automatically by finding a class with a `static void main` method. The `requires` declarations assign a local name to the required components, which is the name used for these components in the scope of the `jbank` component.

PCS files used as input to `pal-j` contain only classes described in Java source files and the `.java` extension is automatically added to the class names. So in this example, compilation of `jbank.pcs` will automatically compile the files `JBank.java` and `JHistoryAccount.java`. Similarly, PCS files for the `pal-st` compiler reference only classes written in Smalltalk. All PCS files can reference components generated by any compiler (`pal-j` or `pal-st`).

The PCS file syntax is defined in Appendix D.

# 7 Bytecodes

The virtual machine executes a set of binary bytecodes. A list of these bytecodes can be found in Appendix B. These bytecodes are evaluated in an environment as described in Section 6 of Deliverable 22 [23], which includes the self reference, locals and arguments, globals, literals, and static and dynamic links. Since then there have been some changes in the organisation of the stack. The current organisation of the self reference, locals and arguments in the stack frame is shown in Figure 1.

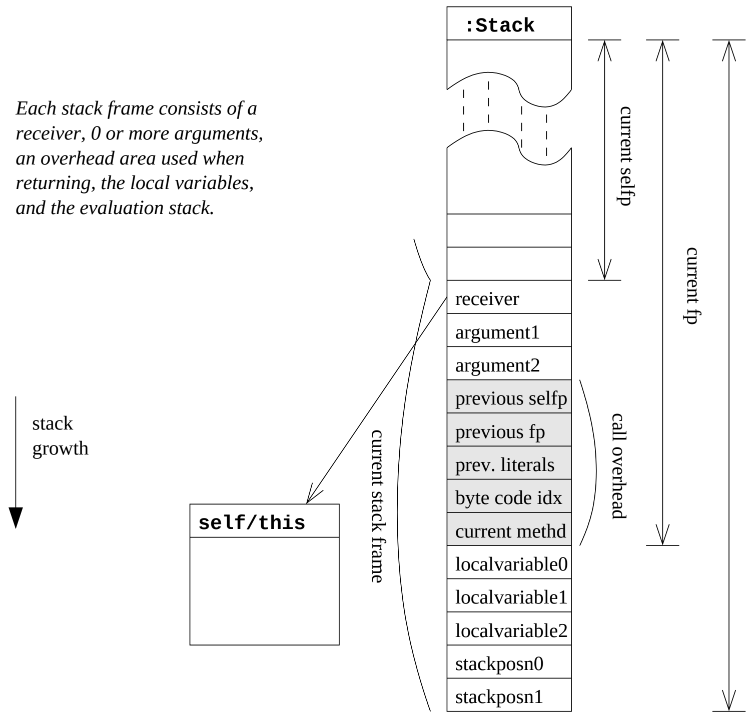


Figure 1: Structure of the stack

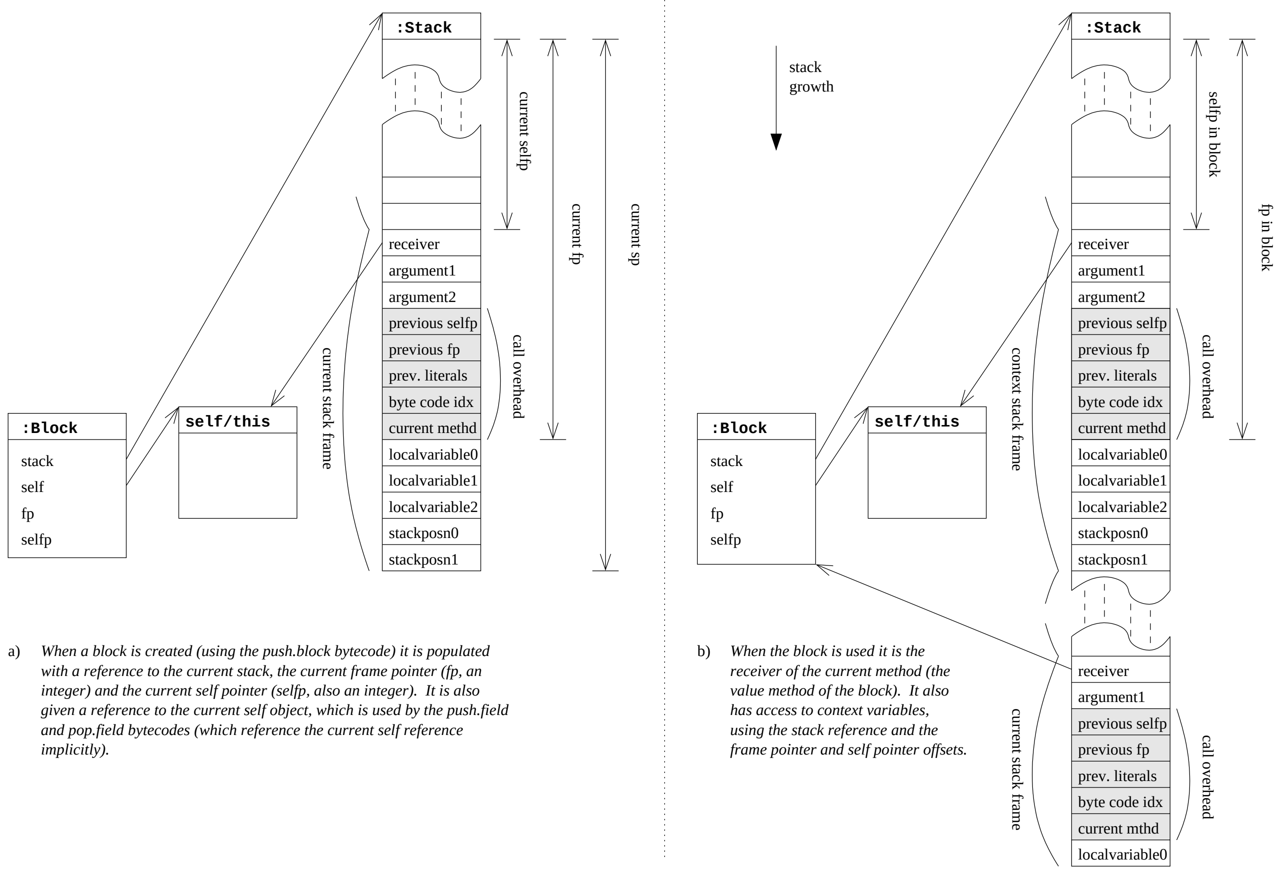
Each method invocation results in the creation of a stack frame on the current stack as shown in Figure 1. There are several stacks in the system (associated with different threads and coroutines). The stack is automatically resized on method invocation if it is too small for the new stack frame. The stack object contains offsets that describe the current frame pointer, `fp`, the position of the current receiver object, `selfp`, and the current stack pointer, `sp`.

## 7.1 Contexts

The bytecode set described in Deliverable 22 included several bytecodes designed to handle local variables not on the stack. These bytecodes, which handled 'contexts' were intended mainly for use in Smalltalk code. Since that time, the main thrust of development has moved to Java code and so the context-oriented bytecodes were removed from the specification.

Blocks are now implemented with a reference to the stack and the offset of the frame where local variables and arguments from the context are available. This is illustrated in Figure 2a, where the block object is created, and in Figure 2b, where the block object is used.

In the future it may be found more efficient to allocate the blocks on the stack itself, in which case they need only an intra-stack offset to the context variables. In both cases it is a requirement that a block is not used (its method is not executed) after the stack frame which is its context has been popped from the stack (returned from).



a) *When a block is created (using the push.block bytecode) it is populated with a reference to the current stack, the current frame pointer (fp, an integer) and the current self pointer (selfp, also an integer). It is also given a reference to the current self object, which is used by the push.field and pop.field bytecodes (which reference the current self reference implicitly).*

b) *When the block is used it is the receiver of the current method (the value method of the block). It also has access to context variables, using the stack reference and the frame pointer and self pointer offsets.*

Figure 2: Implementation of blocks

Violations of this rule would lead to system instability. In order to ensure that this cannot happen, there are certain operations that are not permitted for blocks: They cannot be returned from methods, they cannot be thrown (as exceptions) and they cannot be written into objects (including arrays and hash maps). In addition, they cannot be written into local variables and arguments accessed through the context of a block. The VM contains checks to ensure that these rules are not violated, and throws an exception if they threaten to be violated. In the OSVM virtual machine[6] there were similar restrictions, but they were enforced by static typing in the compiler and name mangling of methods that took blocks as arguments. The solution with static typing may be faster than the runtime checks currently performed by `pal-vm`.

## 7.2 Internal bytecodes

In addition to the bytecodes documented in Appendix B, the current implementation of `pal-vm` has some internal bytecodes. These are generated by the VM itself by bytecode rewriting, and are used for two main purposes:

Firstly, those primitives whose principal effects are on the state of the interpreter are simpler to implement as byte codes than as conventional primitives (which use a function pointer lookup to call implementation code external to the interpreter). The `call` bytecodes for these primitives are therefore overwritten with special internal bytecodes when the method is loaded into the VM.

Secondly, some performance improvements have been obtained by bytecode rewriting. Common operations that can be identified in the bytecode stream have been accelerated using special bytecodes.

These internal bytecodes are optional aspects of the current concrete implementation of the VM. They are not available to the compilers, nor do the compilers need to have support for them. If the VM implementation changes (as it has many times during the optimisation phase described in Section 16) then the compilers do not need to be updated, as long as the external bytecodes remain constant. There are currently 48 internal byte codes.

## 7.3 Primitives vs. bytecodes

When designing the PalCom virtual machine specification, decisions had to be made whether to implement certain instructions with a new bytecode or with a new primitive. Certain rules of thumb were followed:

- Primitives have a predetermined effect on the expression stack of the interpreter. (All arguments are popped, the result is pushed.) If an instruction is to have a different effect on the expression stack then it must be implemented as a bytecode.
- Instructions that are very rare, for example the instructions used to implement the `name` method in the `Component` class will normally be implemented as primitives. This helps to keep the size of the interpreter down, by not cluttering the bytecode set with rarely used bytecodes.
- The Smalltalk compiler `pal-st` is a very simple one, and there is a straightforward correspondence between source code constructs and bytecodes. Instructions for which Smalltalk syntax exist are normally generated as bytecodes, whereas other instructions are generated as primitives.
- If there is doubt as to whether an instruction should be implemented as a bytecode or a primitive then it is normally implemented as a primitive. The VM can convert the primitive to a bytecode when the method is loaded if this is convenient, whereas the opposite conversion is more difficult: Some bytecodes require less space than a primitive call and do not use an entry in the literal table of the method. Thus, preferring primitives to bytecodes gives the VM implementer most flexibility in designing the VM implementation.
- Once an instruction has been implemented in one way it normally stays implemented that way. This preserves backwards compatibility and avoids the inconvenience of having to modify several compilers, an assembler and a VM at the same time.

Most of the available primitives are used only in one place, in a particular method of a particular class from the base library. This library, `ist.palcom.base`, is written in Smalltalk. The primitives can be used from other languages by calling the methods of the base class. The exception to this rule is the `Object throw` primitive, which throws an exception. This is emitted by the Java compiler when the Java `throw` keyword is used.

## 7.4 Reading and writing fields of objects

There are no bytecodes for reading and writing the fields of objects other than the current (this/self) object. See Section 10 in Deliverable 22 [23] for the motivation for this.

# 8 Processes

The `pal-vm` executes one or more concurrently running *processes*. A process is an independent execution of a PalCom runtime component, and contains the runtime component it is executing along with any required component. A process is associated with a *coroutine*, which is an independent execution sequence using one of multiple stacks in the VM.

Each process is isolated from other processes, meaning that the object graph of one process only shares immutable objects with other processes. Therefore a process cannot make direct message-sends to objects in another process or inspect the state of objects belonging to other processes. Instead, communication between processes uses PalCom communications protocols or exchanges of data using the global HMAP. The communication protocols are asynchronous in nature, being message based and not remote-procedure-call based. The PalCom communication protocols are described in more detail in Deliverable 41 [29].

A process contains the following information:

- **Main component**: The component used to create the process.
- **Required components**: All components required by the main component including the main component itself.
- **Main class**: The main class of the main component, see Section 6.
- **Instance**: A singular instance of the main class.
- **Coroutine**: A coroutine that executes the `run` method of the singular instance.
- **Global variables**: A map containing all class variables of the process.
- **File descriptors**: A list of file descriptors on which the process is waiting for a communication event.
- **Timeout**: A time interval after which the process wants to be re-scheduled.

A process is created using its main component in the following steps:

- All required components are loaded by loading required components transitively starting with the main component. The required components are found using the `requires` meta-information. See Section 6.
- The main class is fetched from the main component, using the `mainclass` meta-information.
- A coroutine for running the process is created and subsequently scheduled in the main scheduler.
- The coroutine initialises the global variables of the process by sending an `initialize` message to all components.
- The coroutine creates an instance of the main class.
- The coroutine executes the `run` method on the instance of the main class.

Classes are shared between processes, therefore the mutable parts of the classes must be implemented as variables in the process. The mutable part of a class consists of *class variables* called "static fields" in Java and "class variables" in Smalltalk.

The runtime component contains an interface that allows it to be scheduled by the main scheduler. The list of file descriptors and the timeout value are information needed by the scheduler.

# 9 Scheduling

While the `pal-vm` executes one or more *processes*, each process schedules one or more *threads*. A thread is part of a system of coroutines that are scheduled together according to some common mechanism. The difference between a thread and a process is that the threads within a process share data and can communicate directly, whereas processes cannot share data and have to communicate using communication protocols (or the global HMAP, which supports limited data sharing).

Schedulers are responsible for giving CPU time to processes and threads and for taking care of the situation where none of the threads and processes has anything to do.

Scheduling happens at two levels: the process level and the thread level. The `pal-vm` contains a scheduler for scheduling processes at process level. This is a global scheduler that does not know anything about the threads within the processes. Each process contains its own scheduler for handling the threads within the process at the thread level. The application programmer has the freedom to implement his own scheduler at the thread level, but it is expected that the usual case will be to use the default scheduler implementation provided in the `palcomthreads` core library. This library contains a scheduler for handling `PalcomThreads` and implements a common priority system as well as a number of common synchronization mechanisms.

The schedulers at both levels are implemented in libraries using common scheduling mechanisms in the `pal-vm`.

The scheduling mechanisms allow both *cooperative* scheduling as well as *preemptive* scheduling. In the following, the scheduling mechanisms will be described by first describing the mechanisms for cooperative scheduling, and then the mechanisms for preemptive scheduling.

## 9.1 Cooperative Scheduling

A process, as well as a thread, is associated with a coroutine that contains the actual execution stack of the process and thread. Therefore scheduling processes and threads is performed by attaching and suspending the associated coroutines using the `attach()` and `suspend()` methods. In the following, “attach process” or “attach thread” means to attach the associated coroutine of the process or thread.

Simple cooperative scheduling attaches the coroutines in the scheduler’s queue in a round-robin fashion:

```
while not Q.isEmpty()
  active = Q.next()
  active.attach()
```

This scheduler attaches the coroutines in order and waits for the active coroutine to either complete its execution (terminate) or suspend itself voluntarily. This is the basic form of cooperative scheduling, since a process/thread will keep running exclusively until it suspends or terminates. This system can be unresponsive and lead to busy polling. If a process is waiting for data on some file descriptor, it will either have to block all other processes until data arrives or keep suspending itself until data is available.

To improve on this situation a *select* mechanism has been implemented. The `select` method, defined in the `pal-vm` system class, takes as its parameters a list of file-descriptors and a timeout value. When a process calls `System>>select` it thereby informs the system of the file descriptors on which it is awaiting input, and the maximum time it wants to be delayed. When the scheduler is to attach a new process, it will choose a process that either has a timeout value that makes it eligible for execution or is waiting for data on a file-descriptor on which data is ready. If no such process exists in the queue, the scheduler calls the standard C library `select` function with the union of all the file-descriptors of the processes and the minimum of the timeout values. When the C library `select` function returns, at least one process will be eligible for execution. The C library `select` function also suspends the calling operating system (OS) process and therefore serves as a mechanism for scheduling between OS processes. If the VM is running on a system with no (or minimal) OS, then there is no need to yield control to other OS processes. If the VM finds itself with no runnable threads or processes then it can save power by halting the CPU. The CPU will go into a power-saving sleep mode from which it will be woken by an interrupt. The interrupt may provide input to a thread or indicate a timeout.

Use of the `System>>select` method requires the `PalcomThreads` library to keep track of which file descriptors the various threads in a process are currently waiting on. The list of file descriptors is maintained in the `PalcomThreads` library by application threads making calls to the `PalcomScheduler.enableIOEvent( Channel c, int timeout)` method.

This is done with the `enableIOEvent` method.

The use of native futures, described in Section 10.3, represents an alternative to the use of the `System>>select` method. Native futures have the advantage of being usable for waiting on events other than the presence of data on a file descriptor. In their current form they have the disadvantage of having no support for timeouts.

## 9.2 Preemptive Scheduling

The select mechanism enables communication on file descriptors without blocking other processes or performing busy polling. There can still be unacceptable response times in the system however: A process that performs a long calculation can block other processes that may need a guaranteed response-time. Therefore, an interrupt mechanism has been implemented to support preemptive scheduling. The interrupt mechanism allows a scheduler to set a timeout when attaching a process. After the supplied timeout has passed, the process is forced to suspend. This is implemented by an OS level (or CPU-provided) interrupt-handler that sets a global flag. The interpreter checks the flag whenever it handles a message-send or a backward-branch and can therefore force an active coroutine to suspend. Since all code must perform a message-send or a backward-branch at finite intervals during execution, this will ensure that the coroutine cannot run forever. It will take some time from the timeout occurs till the interpreter gets a chance to force the suspend, depending on the length of the *forwards instruction sequences* (sequences of byte codes that do not perform message-send or backward branches). This delay is usually not a problem in practice, but if it becomes a problem, the compiler can insert special check-for-timeout instructions in the forwards instruction sequences to ensure guaranteed response-times.

Preemptive scheduling is currently only implemented at the process level. This is due to a current lack of VM-level synchronisation mechanisms. Since processes do not share data, synchronisation is not needed between processes. This is however an issue with threads – they share data and therefore synchronisation is needed. The common synchronisation mechanisms implemented in the `palcomthreads` library only work in the cooperative scheduling scenario. VM-level synchronisation can be implemented using the wellknown test-and-set instruction that can be used as the basis for a wide range of synchronisation mechanisms.

# 10 Native Calls

Native calls are a mechanism for calling functions supplied by native libraries. Thus they are the interface to all code written in languages that cannot be compiled to the `pal-vm` bytecode set.

On a modern operating system there are C library calls for determining the address of a native function, given its name and the name of the library in which it is found. In POSIX these calls are called `dlopen` and `dlsym`. By making use of these calls, the `pal-vm` can make native calls to any native function specified in the bytecodes of an application, without the VM needing to have prior knowledge of that function.

Unfortunately, on simple operating systems the support for looking up the addresses of named functions may be missing. Some small Linux variants (for example the one on the UNC20, see Sections 19 and 21) fall into this category. For these platforms, the VM must be augmented with a list of the native functions that we wish to support with native calls. When a new native call is added, the VM must be recompiled on these platforms.

The native interface contains the following elements:

- Memory objects: An object that allows allocation and freeing of memory external to the `pal-vm` heap.
- Native calls: A mechanism for calling native functions.
- Native future calls: A mechanism for calling native functions that immediately returns a *future* that will hold the actual return value of the function when it becomes available.

## 10.1 Memory objects

Memory objects encapsulate an area of physical (native) memory that can be read from and written to. A memory object can be used to store a C structure or a C++ object. They can also be used for memory buffers when using native calls that require such buffers.

The underlying physical block of memory of a memory object is normally created by calling the native function `malloc` and `calloc`. In this case it is the responsibility of the programmer to ensure that the native `free` function is called once and only once when the memory object is not needed (using the `Memory>>free` method). Since the `pal-vm` system does not have finalizers, this cannot be automated.

Though the garbage collector may move a Memory object, the memory that it refers to cannot be moved. This is in keeping with what a C function expects from a memory area.

## 10.2 Native calls

Native calls use a correspondence between `pal-vm` types and C types for parameters and return values. The conversions between these types are made very simple in order not to complicate the VM. It is not possible to give a native function the address of an internal object in the VM. This means that there is no need to prevent the garbage collector in the VM from moving or destroying objects that are referenced from native code or in native data structures. In addition, there is never a need to synchronise the use of structures that are shared between native code and Smalltalk/Java code. The following simple correspondences are supported:

- Integer corresponds to int
- String corresponds to char array

- Symbol corresponds to char array
- Memory is converted to and from any struct or allocated memory area

Any more complicated native 'objects' or data structures must be handled by allocating native memory (using `malloc` or `calloc` and `free` or similar) and then copying data in and out of these allocated areas.

Native calls are invoked by the methods:

- `System>>nativeMemoryCall:with:resultSize:`
  for calling a function that returns a pointer to a memory area of known size, e.g., `malloc`. The return value of the method call is a Memory object.
- `System>>nativeIntegerCall:with:`
  for calling a function that returns an integer, e.g., `time`. The return value of the method call is an Integer or Integer32 object.
- `System>>nativeStringCall:with:`
  for calling a function that returns a null terminated char pointer, e.g., `strerror`. The return value of the method call is a String object.
- `System>>nativeVoidCall:with:`
  for calling a function that does not return a value, e.g., `free`.

The `with:` parameter is an array of arguments to the function call. The number of elements in the array must match the number of arguments that the native function expects to receive. The arguments are converted using the simple correspondences above.

## 10.3 Native Futures

Normally, the VM will be suspended when a native function is called, with no byte codes able to run until the native function returns. Therefore the native call mechanism presented above is only usable for native calls that return almost immediately. If the calls are being used to perform input/output then this means that only non-blocking function calls can be used. These are calls that always return immediately, whether or not there is input ready to be read, or buffer space available for output.

If a potentially long-running native method is to be called, for example one that performs a lookup of an Internet host name in order to find an Internet address (IP-number), then a *native future* may be used. This has the advantage that the VM is not suspended while the native function call runs.

The following methods:

- `System>>nativeLongRunningIntegerCall:with:`
- `System>>nativeLongRunningMemoryCall:with:`
- `System>>nativeLongRunningVoidCall:with:`

are similar to their non-long-running namesakes, but will perform the native call in a different operating system thread. The VM will continue to run in the original operating system thread. While the native call is taking place, they will `yield`, causing the scheduler to schedule other threads or processes in the VM. The effect of this is that the native call can make blocking calls to the operating system. In the example of an Internet name lookup, the native call is likely to send some UDP packets to a name server, then block the operating system thread, waiting for responses to arrive on its UDP socket.

This works by using a `NativeFuture` object. This is a special object that has three methods. The method `NativeFuture>>available` tells the scheduler whether or not the native call has completed. The methods `NativeFuture>>integerValue` and `NativeFuture>>memoryValue:size:` may only be called after the `available` method has returned true, and are used to retrieve the return value from the native function call. At the moment, the `NativeFuture` object is polled, whenever the scheduler is scheduling threads, but a more sophisticated system, whereby the scheduler is informed asynchronously of the completion of the native method could be implemented. Giving the scheduler more information about the progress of the native call would improve the efficiency of the scheduler and work better with preemption, both described in Section 9.

Native futures require operating system support for threads. This is very widespread. If a version of the `pal-vm` is developed that runs directly on the hardware without an intervening operating system (OS), then the `pal-vm` would have to be augmented with support for threads sufficient to implement native futures. The support needed is very simple: A similar addition to the OSVM virtual machine caused an increase of only 2kbytes in VM size[6].

# 11 Language Interoperability

As mentioned in the introduction, The language interoperability support provided by the compilers and the VM address the heterogeneity/coherence PalCom challenge on the programmatic level. The `pal-vm` supports a broader range of languages by:

- Being a dynamically typed VM, allowing direct mapping for both dynamically typed languages like Smalltalk and statically typed languages like Java
- Requiring much less conformance of the high level languages than, e.g. the .NET Common Language Specification. E.g. `pal-vm` languages are not required to support method overloading.
- Leaving static type checking and handling of constructs such as method overloading to the high level language compilers.

As was demonstrated in Deliverable 21 [22], if a broad transparent language interoperability is wished for, a common type system can be defined externally to the VM, and supported by the VM with very few changes.

The VM currently supports the programming languages Smalltalk [16] and Java [7] (BETA [18] is currently not supported – this may change in the future).

## 11.1 Interoperability Issues

Code generated by the Java compiler (`pal-j`) runs together with code generated by the Smalltalk compiler (`pal-st`) on the `pal-vm`. This raises a number of issues:

- **Interlanguage calls**: dealing with name-mangling and static type checking. The Java compiler generates mangled names and the Smalltalk programmer needs a way to call Java code without knowing the mangling scheme. On the Java side, the compiler needs type information on the Smalltalk code in order to do type-checking. Since type-information is not supplied by the Smalltalk code, some mechanism is needed to supply this. Interlanguage calls are described in the following sections.
- **Interlanguage inheritance**: Not only interlanguage *calls*, but also interlanguage *inheritance* is possible. This is also further elaborated in the sections below, and summarized in Section 11.7.
- **Java arrays**: dealing with runtime type checking of Java arrays. Extra type information is needed in arrays at runtime in order to enable runtime type checking when storing values in Java arrays. Interoperability with Java arrays is described in Section 14.1.
- **Everything-is-an-object**: In Smalltalk and in the `pal-vm` everything is an object, but in Java, null and simple types like integers are not objects. This interoperability issue is further discussed in Section 17.2.
- **Exception handling**: Some exceptions originating from Smalltalk code should be mapped to an exception more appropriate for Java. For example, the Smalltalk `#NilDoesNotUnderstand` exception should be mapped to a Java `java.lang.NullPointerException`. The handling of exception interoperability is postponed until Section 12.1, after the description of the general exception handling mechanisms in Section 12.

The following will describe the mechanisms implemented to support interlanguage calls. Other issues around supporting multiple languages on one VM are discussed in Section 17. The handling of exceptions is discussed in Section 12.

Three mechanisms are currently in use for interlanguage calls:

- **Basename dispatch**: An automatic mechanism in the `pal-vm` for allowing a call issued in one language to be dispatched to a method defined in another language without requiring the compiler or programmer to know the name-mangling schemes.
- **External class**: A mechanism for describing the interface of a Smalltalk class to the Java compiler, which can be used by the programmer to supply type-checking information and name-mangling information about Smalltalk classes to the Java compiler.
- **Explicit name-mangling**: A mechanism for using mangled names directly in Smalltalk.

The general "basename dispatch" mechanism is the latest implemented of these mechanisms, and it is intended to replace the Java specific "external class" mechanism. The mechanism for explicit name-mangling is needed to allow Smalltalk code to override/call overloaded Java methods.

## 11.2 Code Example

For the purpose of the description, a concrete code example will be used in the form of a simple bank account implementation that involves both Java and Smalltalk classes to illustrate interlanguage calls. The example will be used to first describe the name-mangling used by the compilers, and then the three interlanguage call mechanisms.

While the bank account example is not directly related to Palcom it does have the advantage of being simple and familiar by virtue of its similarity to examples found in the object-oriented literature. The relationships between the classes in the example are illustrated in Figure 3.

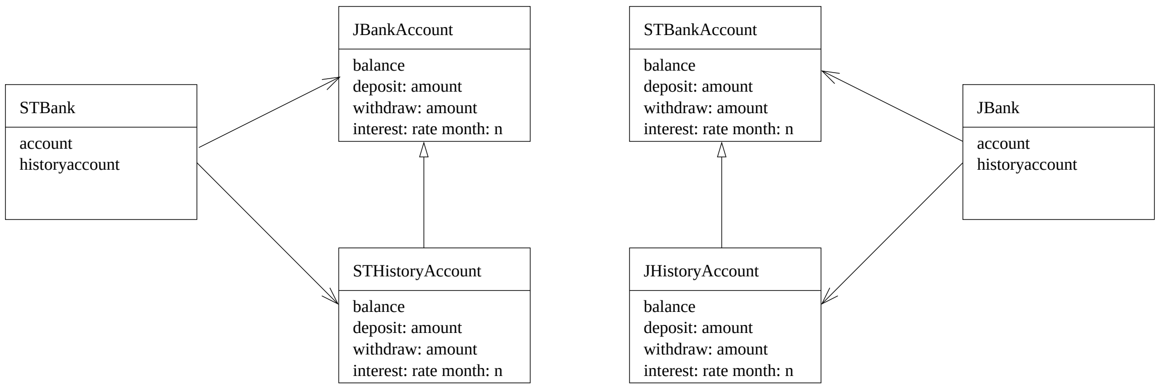


Figure 3: Class Diagram for an Example Illustrating Language Interoperability

The bank account example contains the following classes implemented in Smalltalk and Java:

- **JBankAccount**: Java implementation of a bank account class (see Figure 4).
- **STBankAccount**: Smalltalk implementation of a bank account class (see Figure 5).
- **JHistoryAccount**: Java extension of the Smalltalk bank account class (see Figure 6).
- **STHistoryAccount**: Smalltalk extension of the Java bank account class (see Figure 7).
- **JBank**: Java application that uses STBankAccount and JHistoryAccount (see Figure 8).
- **STBank**: Smalltalk application that uses JBankAccount and STHistoryAccount (see Figure 9).

```
package jbankaccount;
class JBankAccount {
    private int balance;
    public int balance() {
        return balance;
    }
    public void deposit(int amount) {
        balance += amount;
    }
    public void withdraw(int amount) {
        balance -= amount;
    }
    public int interest(int rate, int n) { ... }
}
```

Figure 4: Java Bank Account Example

```
STBankAccount = (
  | balance |
  initialize = (
    balance := 0.
  )
  balance = ( ˆbalance )
  deposit: amount = (
    balance := balance + amount.
  )
  withdraw: amount = (
    balance := balance - amount.
  )
  interest: rate month: n = ( ... )
----
  new = (ˆsuper new initialize )
)
```

Figure 5: Smalltalk Bank Account Example

```
package jbank;
import stbankaccount.STBankAccount;
class JHistoryAccount extends STBankAccount {
    private int count;
    public void deposit(int amount) {
        super.deposit(amount);
        count++;
    }
}
```

Figure 6: Java Extension of the Smalltalk Bank Account Class

```
STHistoryAccount = JBankAccount (
  | count |
  deposit: amount = (
    super deposit: amount.
    count := count + 1.
  )
)
```

Figure 7: Smalltalk Extension of the Java Bank Account Class

```
package jbank;
import stbankaccount.STBankAccount;
public class JBank {
  public static void main(String[] args) {
    STBankAccount account = new STBankAccount();
    account.deposit(100);
    JHistoryAccount historyaccount = new JHistoryAccount();
    historyaccount.deposit(200);
  }
}
```

Figure 8: Java Application

```
STBank = (
  run = (
    | account  historyaccount |
    account := JBankAccount new.
    account deposit: 100.
    historyaccount := STHistoryAccount new.
    historyaccount deposit: 200.
  )
)
```

Figure 9: Smalltalk Application

## 11.3 Method Name Mangling

This section will describe the actual method name-mangling, used by the compilers, to aid the understanding of the three interlanguage call mechanisms.

The compilers prepend a letter to the method names to show the origins of the methods. The Java compiler (`pal-j`) prepends a 'J' and the Smalltalk compiler (`pal-st`) prepends a 'S'. The number of colons in the generated names must be equal to the number of arguments of the methods (required by the `pal-vm`).

Smalltalk method names are of the form:

- `balance` becomes `Sbalance`
- `deposit:` becomes `Sdeposit:`
- `interest:month:` becomes `Sinterest:month:`

The Java compiler mangles the names in order to support Java's overloaded method names. Therefore the parameter-types and return-type becomes part of the mangled name.

- `int balance()` becomes `Jbalance_Integer`
- `void deposit(int amount)` becomes `Jdeposit_V_Integer:`
- `int interest(int rate, int n)` becomes `Jinterest_Integer_Integer:Integer:`

The Java compiler places a number of colons in the mangled name corresponding to the number of parameters.

## 11.4 Basename Dispatch

In the basename dispatch mechanism, the `pal-vm` dispatches a method by truncating the method-names using language specific rules to obtain a basename. These rules are currently hard coded in the `pal-vm` but the plan is to implement a mechanism for supplying the language specific rules dynamically using either Smalltalk (preferable) or a regular expression based description language.

When searching a class for a method to invoke given a method-name, the `pal-vm` will first do a local search in the class for an exact match of the method-name. This search will only be performed in the class itself and not in any super-classes to the class. If an exact match is not found, the method name is truncated to the basename and a basename search is performed locally, comparing the basename to similarly truncated names of the available methods, originating from a different language than that of the original method name. If no match is found locally, the whole search is performed recursively on the super-class.

The Java names are truncated by removing the first 'J' and stripping everything after the first underscore. Then a number of colons are appended corresponding to the number of arguments.

- `Jbalance_Integer` becomes `balance`
- `Jdeposit_V_Integer:` becomes `deposit:`
- `Jinterest_Integer_Integer:Integer:` becomes `interest::`

The Smalltalk names are truncated by removing the first 'S' and stripping everything after the first ':'. Likewise a number of colons are appended to match the parameter count:

- `Sbalance` becomes `balance`
- `Sdeposit:` becomes `deposit:`
- `Sinterest:month:` becomes `interest::`

This scheme ensures that the truncated Java names matches the truncated Smalltalk names thus making interlanguage dispatch work.

In the `JBank` example, the `deposit(100)` call is translated to the name-mangled call, `Jdeposit_V_Integer:`. Since this method is not defined in the `STBankAccount` class (where it is called `Sdeposit:`), the `pal-vm` will do a base-name dispatch for `deposit:` instead, and correctly find the `Sdeposit:` method.

In the `STBank` example, `deposit: 100` is translated to a name-mangled call, `Sdeposit:`, which is not found in the `JBankAccount` class (here it is called `Jdeposit_V_Integer:`). A base-name dispatch for `deposit:` will correctly find the `Jdeposit_V_Integer:` method.

The advantage of the basename mechanism is that basic non-overloaded interlanguage calls are trivially supported, without breaking the support for overloaded methods within Java. Java overloading of methods is supported internally in Java, since methods with the exact name from the same language are searched for first.

The limitation is that is not possible to call overloaded Java methods from Smalltalk. For this the mechanism called *Explicit Name Mangling* is used, see below.

### 11.4.1 Type Checking with IDA

The basename dispatch mechanism alone is not sufficient to allow Java code to call Smalltalk code. The `pal-j` compiler must be able to type-check code that makes calls to Smalltalk code. When type-checking the `JBank` class, the compiler would like to verify that the STBankAccount class exists and contains a method `deposit()` with the correct interface. The *Interface Description Annotations* (IDA) in the binary component file supply that information to the `pal-j` compiler. The `pal-st` compiler generates these annotations as meta-information in the component. Since Smalltalk is a dynamically typed language, the compiler cannot generate the type annotations directly from the Smalltalk source. Instead the type information is generated from special comments in the Smalltalk source that must be written by the programmer. The format of the comments is very similar to Javadoc [13] comments and these comments can be used both for generated the type information, and for generating documentation. The comment for `deposit:` looks as follows.

```
"{ Deposits the specified amount to this account
   @param amount int the amount to deposit
 }"
deposit: amount = (
  balance := balance + amount.
)
```

The formal part of the comment is "`@param amount int`", which tells the compiler that the parameter, `amount`, has the type, `int`. The type annotation comment is compiled into an interface description annotation by the `pal-st` compiler. The rest of the comment is not required, but is just there to serve as documentation.

In effect IDA introduces a limited version of the in Deliverable 15 [21, Section 5.2.3] proposed Common Palcom Type System (CPTS). The currently supported types, that can be used when annotating Smalltalk methods are the primitive types `int`, `boolean`, and class names. As IDA matures, this CPTS is expected to be augmented.

## 11.5 External Class in Java

The external class mechanism makes use of an extended Java syntax to explicitly make the interface of a Smalltalk class known to the `pal-j` compiler. The external class declaration for the `STBankAccount` class is listed below.

```
package stbankaccount;
external class STBankAccount {
  STBankAccount() "initialize";
  int balance() "balance";
  void deposit(int amount) "deposit:";
  void withdraw(int amount) "withdraw:";
  int interest(int rate, int n) "interest:month:";
}
```

When `pal-j` encounters this declaration, it does not generate any code. Instead it uses the information to type-check any uses of the `STBankAccount` class within the Java program. The external class declaration also contains information on the actual names of the Smalltalk methods, which allows the compiler to use the actual Smalltalk names instead of the name-mangled Java names. In the `JBank` example, the `deposit(100)` call is thus translated to `send Sdeposit:` instead of `send Jdeposit_V_Integer:`.

This mechanism can also be used to map Smalltalk names to some completely unrelated Java name that fits better with the design of Java libraries. The `at:put:` operation typical of Smalltalk lists can be mapped to `void set(int i, Object value)`, which is a more natural name for this operation in Java:

```
external class ArrayList {
  ...
  Object set(int i, Object value) "at:put:";
}
```

The rename mechanism cannot be used to rearrange the order of the arguments – it can only be used to give the method a different name.

The external class mechanism is exclusively used to make Smalltalk classes accessible to programs written in Java. As mentioned above, the mechanism is expected to become unnecessary over time, when all Smalltalk methods have been IDA annotated. The aliasing mechanism described above is also expected to be possible to support in the IDA syntax.

## 11.6 Explicit Name-Mangling in Smalltalk

The explicit name-mangling mechanism is a syntax in Smalltalk that makes it possible to use a pre-mangled name. This is a name that is not changed by the compiler, but used directly as-is in the generated byte-code. A pre-mangled name is a name preceded by the "@" symbol. Such a name is used literally (stripped of the "@") without an "S" prepended.

The use of pre-mangled names makes it possible to call or override overloaded Java methods that cannot otherwise be called or overridden using the basename dispatch method. The following `PrintStream` class contains the overloaded method `print(...)`, which is defined in two versions: a version taking an int argument and a version taking a String argument.

```
class PrintStream {
  void print(int value) { ... }
  void print(String value) { ... }
}
```

The mangled names of these two print methods are:

- `void print(int value)` becomes `Jprint_V_Integer:`
- `void print(String value)` becomes `Jprint_V_java_lang_String:`

However the *basename* of both print methods is `print:`, which makes it impossible to distinguish between them in Smalltalk. Calling the `print:` method will match both methods and the result would be to call a random print method. Definining a `print:` in a Smalltalk subclass to `PrintStream` would have effect of overriding both print methods.

### 11.6.1 Calling an Overloaded Method

To call an overloaded Java method from Smalltalk, the explicit name-mangling mechanism is needed. To call `print(int)`, we would like the compiler to call `Jprint_V_Integer:`. This is done by:

```
...
| stream |
stream := ...
stream @Jprint_V_Integer: 17.
...
```

It is also possible to call Java methods that takes more than one argument. Calling `int interest(int rate, int n)`, which has the mangled name `Jinterest_Integer_Integer:Integer:` is done by:

```
...
| account |
account := ...
account @Jinterest_Integer_Integer: 7 Integer: 12.
...
```

### 11.6.2 Overriding an Overloaded Method

To override an overloaded Java method in Smalltalk, the explicit name-mangling can be used similar to when calling an overloaded method. A Smalltalk extension to the `PrintStream` class would need to define two different `print` methods. This is done by:

```
StOutputStream = PrintStream (
  @Jprint_V_Integer: value = ( ... )
  @Jprint_V_java_lang_String: value = ( ... )
)
```

This will define methods that correctly correspond to the mangled names generated by the Java compiler.

## 11.7 Interlanguage Inheritance

As can be seen from the examples above, interlanguage inheritance is straightforward: There are no special conventions for *class names*, and the VM does not care in which language a superclass is written.

The specification of a superclass is done by using the name of the superclass as declared, regardless of the declaring language.

Special care must be taken, though, when overriding a *method* in a superclass written in another language, as the preceeding sections have shown. Specifically Java method overloading may call for special mechanisms, as detailed in section 11.6 above.

# 12 Exception Mechanism

In order to support the PalCom goals of resilience and contingency management it is important to have a way to handle unexpected events. A well-established low level mechanism for signalling and handling unexpected events in a program is provided by *exceptions*, as known from Java[7] and some dialects of Smalltalk[16].

The `pal-vm` supports throwing and catching of exceptions via a generic and highly flexible exception mechanism. The mechanism allows any object to be thrown as an exception. On top of the support provided by the runtime environment, it is possible to build abstractions in library code which extend the exception mechanism to satisfy the requirements of specific languages.

The PalCom runtime environment implements the support for throwing and catching exceptions using variants of two methods: `Block>>catch:` and `Object>>throw`. The semantics of the two operations is as follows:

- The method `Block>>catch:` evaluates the receiving block, but ensures that if an exception is thrown during the evaluation, the argument block is evaluated as an exception handler with the caught exception as an argument.
- The method `Object>>throw` throws the receiving object as an exception. As a result, the virtual machine unwinds stack frames until it reaches an activation of the method `Block>>catch:`. Before evaluating the handler block, the stack frame for the invocation of `Block>>catch:` is also unwound. The virtual machine passes the exception object to the handler block as an argument.
- The method `Block>>catchWithStackTrace:` works like `catch:`, but constructs a stack trace, which is provided to the exception handler block as an extra parameter. Constructing a stack trace object is a fairly costly operation in terms of time and memory space and by having both a choice between this method and the plain `catch:` method we are able to dispense with the construction of the stack trace in the cases where it is not needed. It is our experience that the decision of whether to construct a stack trace object is best placed at the point in the program where the exception is caught, rather than at the place where it is thrown.
- The method `Block>>catch:ifMatch:` allows the exception handler to provide a type. When the VM is unwinding stack frames in response to an invocation of `throw` it will skip `catch:ifMatch:` frames where the thrown object is not an instance of a subtype of the type specified in the `catch:ifMatch:` invocation. The alternative way to implement this (which was previously used) is for a compiler or programmer to generate code that tests the type of the thrown object in the exception handler block, and rethrows the exception if it is not of the correct type. The new method is both faster and better at making the decision of whether to generate a stack trace, since the VM is aware of which exception handler is the final destination of the exception throw.
- The method `Block>>catchWithStackTrace:ifMatch:` works, as might be expected, like a combination of `catchWithStackTrace:` and `catch:ifMatch:`. It allows a type to be specified for the thrown exception, and also provides a stack trace as a parameter to the exception handler.

**Idioms**

To illustrate how the `catch:` and `throw` methods are used, we present the well-known Java idiom for exception throwing and catching side-by-side with the equivalent idiom as implemented using the PalCom runtime environment support and expressed in Smalltalk:

```
try {                                   [
  ...                                     ...
  throw new IOException();                IOException new throw.
  ...                                     ...
} catch (Throwable e) {                 ] catch: [ :e |
  // handle any exception                 "handle any exception"
}                                       ].
```

With the `catch:ifMatch:` method it is possible to implement type-specific catching of exceptions as follows:

```
try {                                   [
  ...                                     ...
  throw new IOException();                IOException new throw.
  ...                                     ...
} catch (IOException e) {               ] catch: IOException ifMatch: [ :e |
  // handle IO exception                  "handle IO exception"
}                                       ].
```

For Java exceptions, which are instances of subclasses of `java.lang.Throwable`, the methods that provide the stack trace as an extra parameter to the catch block are not used (these are `catchWithStackTrace:` and `catchWithStackTrace:ifMatch:`). Instead, the stack trace is constructed by the constructor of the object, which calls the `fillInStackTrace()` method. This means that our runtime environment is compatible with the traditional Java idiom of overriding `fillInStackTrace()` with an empty method to prevent costly stack trace construction.

## 12.1 Interoperability with exceptions

The `pal-vm` supports throwing and catching of any object. This means that code written in any language that includes exceptions can throw and catch whatever objects are suitable. In the case of Smalltalk, it is traditional that `Symbols` (canonicalised strings) are thrown. In the case of Java, the language specification dictates that only subclasses of the `java.lang.Throwable` class can be thrown and caught.

Problems arise when code from different languages is mixed. Since Smalltalk can handle thrown objects of any class, and since Smalltalk does not have statically checked exceptions, there are no problems catching Java exceptions in Smalltalk code. However, there is no way to catch a Symbol in Java, and it was thought too onerous to insist that all Smalltalk code should throw only subclasses of `java.lang.Throwable`. This would preclude running any system without the `java.lang` component, something that is currently possible, and it would be a solution that was obviously not extensible to other languages that might have differing requirements.

Instead, the static initialiser of a Java exception class (executed at the moment where a Java component is loaded into a new process) can register classes or objects (including symbols) that should be translated into Java exception objects. These registrations are compiled into a table of correspondencies between thrown exceptions and classes.

When the exception is thrown, the VM code that unwinds the stack, looking for a match between the type of the exception and the type of the a clause must take the table of correspondencies into account.

As an example of this, the class `java.lang.ArrayIndexOutOfBoundsException` has a static initializer that registers the symbol `ArrayIndexOutOfBoundsException` in the exception translation table, together with an instance of itself as the target of the translation. This is done as following:

```
static {
  Exception me = new ArrayIndexOutOfBoundsException();
  me.nullStackTrace();
  ist.palcom.base.System.instance().addExceptionMapping(
    "ArrayIndexOutOfBoundsException", me);
}
```

The effect of this is to instantiate a prototype instance of `ArrayIndexOutOfBoundsException`. Like all Java exception objects, the instance contains a stack trace generated at the point where it was created (in this case the static initializer). This stack trace is deleted immediately, since it is not useful in this case. The instance is then registered with the VM as a translation of the Smalltalk symbol `ArrayIndexOutOfBoundsException`. This symbol is thrown by the Smalltalk `Array` class when an index argument is out of bounds.

When an index error occurs, this results in the throwing of the `ArrayIndexOutOfBoundsException` symbol. When the symbol is thrown, the VM unwinds the stack, looking for a `catch:` method invocation that will catch it. A Java `catch` clause is compiled to a `catch:` method invocation that contains a subclass of `java.lang.Throwable` (see above). If the class in the `catch:` method is a superclass of the Java class `java.lang.ArrayIndexOutOfBoundsException` then the stack unwinding of the VM will use the table of exception correspondencies to translate the symbol into a Java exception. This is done by making a shallow copy of the instance registered by the static initializer, and filling in the correct stack trace. These last steps, copying and adding a stack trace, are necessary in order to provide the Java exception object with a correct stack trace. Such stack traces are vital for debugging, since they indicate the exact point where the exception was thrown.

The net result of the above is that a Symbol is thrown by Smalltalk code, and if it is to be caught by a Java `catch` clause it is translated first to an instance of the correct Java class.

The exception translation mechanism has been described here in terms of Java and Smalltalk, but we anticipate that it will be usable almost unchanged for other languages.

# 13 Resource Awareness

In order to support Work Package 5 [25] the runtime environment has been augmented with some low level support for resource management and migration. In particular, the VM has primitives and system library code to make the following possible:

- Provide a count of the amount of memory that has been allocated on a system. By calling this method before and after an operation, a resource management framework can measure the memory usage of that operation. Note that this count includes memory that has been allocated and is now no longer in use (that is, the memory could be reused after a garbage collection).
- Provide a signal to a scheduler that is triggered when a certain amount of memory has been allocated. The signal is resettable by the scheduler. This allows a resource management framework to limit the memory allocated by a process when performing a certain operation.
- Provide the above counter and signals based on the number of bytecodes executed rather than the number of bytes allocated.

The memory allocation and bytecode execution data is currently collected for each coroutine. Fairly simple scheduler changes could make it possible to collect the data on a per-process basis. It is an open issue how memory allocated and bytecodes executed by the system on behalf of a process should be accounted for.

Delivering allocation data net of subsequently freeable memory is possible, but determining which memory is freeable can only be done by performing a full garbage collection. This is an unavoidably slow and costly operation, which would be likely to make it unusable in practice.

Delivering allocation data that is both net of freed memory and accounted for on a per-process basis would require separate per-process memory allocation heaps, and the resulting extra complexity and fragmentation issues are likely to make this approach infeasible.

When an allocation or bytecode execution trigger level is reached, the scheduler is triggered, by means of a thrown exception. The act of throwing the exception causes stack frames to be unwound and discarded. This means the scheduler does not currently have the option of letting the coroutine or process continue with normal execution after the trigger level has been reached. If it is found to be desirable, implementing the signalling as a specialised variant of the `suspend` primitive would give the scheduler more choices on the further course of action.

It is also desired to provide available bandwidth estimates for communication channels. This is likely to be best implemented in the communications subsystem with the aid of information from the driver software for the communication channels. As such it is not a runtime environment issue.

## 13.1 Resource API

The memory counter and bytecode counter can be active or inactive. They can be made inactive by the Smalltalk calls:

```
system nullMemoryCounter.
system nullBytecodeCounter.
```

or the Java calls:

```
ist.palcom.base.System.instance().nullMemoryCounter();
ist.palcom.base.System.instance().nullBytecodeCounter();
```

When they are active, the counters count *downwards*. If they reach zero, an exception is thrown. This is either a `#MemoryCounterReachedZero` symbol or a `#BytecodeCounterReachedZero` symbol. Using the techniques described in Section 11 they can be translated to Java exceptions.

To activate the counters, the following calls are used in Smalltalk:

```
system setMemoryCounter:  anInteger.
system setBytecodeCounter:  anInteger.
```

and in Java:

```
ist.palcom.base.System.instance().setMemoryCounter(x);
ist.palcom.base.System.instance().setBytecodeCounter(y);
```

If the intention is to measure, not limit the resource usage of a piece of code, then the counters should be set to very high values, to avoid them reaching zero. In that case they can be read again with the Smalltalk calls:

```
memCounter := system getMemoryCounter.
bytecodeCounter := system getBytecodeCounter.
```

or in Java:

```
x = ist.palcom.base.System.instance().getMemoryCounter();
y = ist.palcom.base.System.instance().getBytecodeCounter();
```

This represents a very simple low level API on which more advanced abstractions can be built.

# 14 Reflection Mechanisms

Reflection has been defined as the capability of a system to inspect and possibly modify itself[19]. The basis of reflection is access to reflective data. This reflective data can potentially be used to analyze, debug, modify and optimize a system.

One advantage of basing a system on a virtual machine is that the virtual machine software has access to reflective data in order to run. Thus, a VM naturally lends itself to a reflective system with inspectability and resilience.

## 14.1 Type annotation

Traditional VM-based object-oriented reflection as in Java and .NET is based on the existence of *full metainfo* for all methods, parameters etc. in the compiled deployment units.

The `pal-vm` does not carry full metainfo in the components. Though some metainfo is available on methods and fields, the types of fields and method parameter types are not included. In the case of components that were written in Smalltalk, this is natural, since the types are dynamic and therefore cannot be determined by the compiler or VM ahead of time. The actual objects that are used when the program is executed are of course all marked with their types (classes and interfaces). Thus, the lack of type annotations cannot lead to unpredictable behaviour: an exception will the thrown if the type assumptions of the programmer fail to hold at run time.

In the case of Pal-J code we also do not have type annotation on fields and methods. This is unusual for a virtual machine executing Java-like code, but since Java is for the most part statically typed, this has surprisingly few consequences: For the most part, the type annotations are used only by the compiler when generating code. At runtime, the type annotations are not needed. The main place where type annotations are needed is for arrays, which are partially dynamically typed in Java: Whenever an object is stored in an array, a virtual machine implementing Java must check whether the object is of a type that is allowed for the given array. If this is not the case, the VM throws an `ArrayStoreException`. In order to replicate this behaviour on `pal-vm`, arrays are marked at creation (using a special primitive) with the type of their contents. Arrays that allow any objects to be stored in them (including normally all arrays used in Smalltalk code) need not perform the check and consequently do not need to store a type.

The advantages of not storing type annotation are twofold. Firstly it saves a lot of space, which can be at a premium in palpable devices. Secondly it would make it much simpler to load and update code on a device: Since there are no type annotations on methods and classes, they cannot become inconsistent when related classes and methods are changed.

It is perhaps worth noting that type-overloaded duplicate method names in Java can be handled without needing runtime type annotation of methods. The overloading is used by the Java compiler to rename methods, which has the effect of removing duplicates. At runtime, only the renamed method names are present, and the type information as such is not needed.

## 14.2 A two-part VM

For smaller devices, the quest for reduced memory usage is in conflict with a the storage of large amounts of reflective metainfo in the system. However, having metainfo on the device is very attractive in terms of the support for the interoperability, visibility and inspection qualities that it can give a system. This is especially true when it is made accessible using simple, generic mechanisms like HMAPs (see below).

One solution to this conflict was pioneered by the OSVM system[6]. This was a ”reflective channel” solution, characterised by:

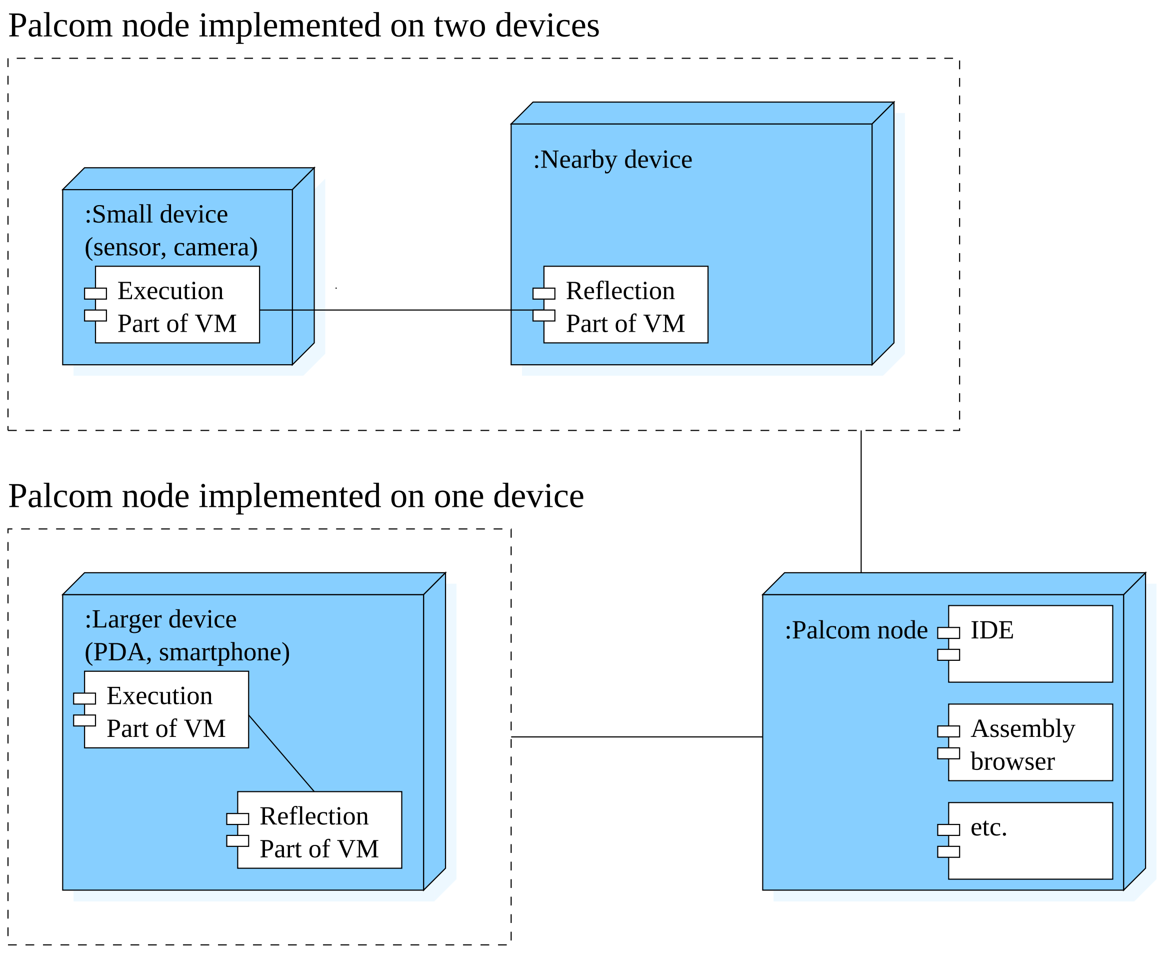


Figure 10: Possible design for a two-part VM

- Using a small reflective interface on the device. The bulk of the reflective data was kept on the development platform (or a reflection server device), a principle which could be called *remote reflection*.
- The reflective interface on the device had primitives for inspecting, updating, debugging and profiling the device.
- The device itself did not include much support for generating (updating) code, but rather allowed the development platform (or some other device) to upload new code through a secure transaction mechanism that would ensure that the device internal state was not compromised by the update. Access to the updating mechanisms from the outside was protected by an authorisation mechanism.

The disadvantage of this approach is the necessity of using a development environment or server to access the device. The development environment is a large, graphical program, unsuitable for small devices and the server is a heavy application designed to administer multiple devices and incapable of running on a smallish device. Thus the device nodes are not in themselves in possession of the reflective metainfo needed to let them be visible and inspectable.

Not all applications of the `pal-vm` will be on memory-constrained devices, but the challenges of heterogeneity and scalability moves us to try to run the VM on large as well as small devices.

For the PalCom project we are considering a variation on the OSVM approach, where some of the support for reflective operations is moved to a separate program that is closely coupled to the VM. In effect, the VM would be divided into two parts, the execution part and the reflective part. These two parts would be tightly coupled in terms of execution time (they are started and stopped together) and in terms of communication (they have extensive implicit and explicit knowledge about each others' internal state).

On medium sized PalCom devices (PDAs, stationary devices, in-vehicle devices with large amounts of memory, for example 16Mbytes) the reflective and execution parts could run together on the same device. On small devices like sensors or battery-powered sub-PDA devices where space is at a premium it might be possible to move the reflective part of the VM to some other hardware that was nearby. The communication between the two parts of the VM might be intermittent, so the execution part would have to be able to perform all normal operations without contact to the reflective part. However, if an external user or service required access to the reflective interface that would presuppose that some form of network connection existed. This network connection could also be used to contact the reflective part of the VM. Thus the separation of the VM would be transparent to the users.

In this scenario, illustrated in Figure 10, we would no longer be limited by the very smallest devices in terms of the reflective services we could provide. At the same time, devices of a certain size would be able to run totally independently, while providing the full gamut of reflective features.

## 14.3 HMAPs

A novel approach for obtaining introspection (visibility), already examined by the Java based `pre-vm` described in Deliverable 22[23], is to use HMAPs [20] to expose the internal state of components, either by adding a HMAP data structure on top of the virtual machine, or by adding HMAP support directly in the virtual machine, the latter approach allowing exposure of VM internal structures, such as methods, parameters etc.; see the ECOOP'05 Workshop Paper [34].

For the `pal-vm` we have a similar solution, with HMAPs implemented in the core libraries. However, expanding the scope of the metainfo available through the HMAPs, though desirable in terms of visibility and inspectability may conflict with the size constraints of small, palpable systems. Therefore it is possible that in a future redesign of the VM and HMAPs we might require that the subtrees of the HMAPs that contain reflective data about the VM only be accessed through the reflective part of the VM (as discussed above). By automatically forwarding HMAP requests from the execution part of the VM to the reflection part of the VM this change could be made transparent at those times where the two parts of the VM are in contact with each other. On larger devices like PDAs this would be all the time.

# 15 Persistence

The design of HMAPs (see Section 14.3 for references) mandates that HMAPs can only contain simple objects such as strings and integers. This, together with the simple, tree-based structure of HMAPs, means that it is very simple to serialise and deserialise HMAPs into a text-based format like XML. More complex objects can be mapped onto HMAPs in a variety of application-defined ways, where object structure is mapped to the HMAP's hierarchical structure, and the individual pieces of object state are converted to strings.

On nodes that have access to persistent storage (for example files on hard disk or flash memory) we can thus use HMAP serialisation and deserialisation to present the programmer with a generic storage facility that is well matched to the other uses we have for HMAPs: transparency and reflection. The use of a standard structure allows generic tools to inspect, present and modify the data without having to be aware of the exact applications and versions that are writing and reading data.

A style of programming where data that is intended to persist over a reasonable time period is stored in the HMAPs can facilitate remote inspection, transparency and simplify file-based persistent storage. If this model is followed, it implies that non-HMAP objects are used mainly for intermediate results and short-lived data.

## 15.1 Migration

If persistent data is always stored in the HMAPs, we can support a simple but powerful form of process migration. Since services are implemented in processes, this implies that services could be migrated too. Migration could be achieved by pausing a process, serialising the HMAPs, deserialising the HMAPs on another node, then restarting the process on the other node. In order to restart the process, the same components would be needed as had been used on the first node. Since the component files are device independent, there should be no difficulties with compatibility on the new device.

Such a migration model, which is not yet implemented, would need to be integrated with the resource management subsystem described in Deliverable 42[30] and with the service management framework described in Deliverable 39, Section 5[27]. The resource management subsystem is needed to determine the policy and make the decision on whether and where to migrate applications. The service management framework needs to define what it means to 'pause' an application, and probably needs to include mechanisms to simplify adding and inspecting the information in the HMAPs.

If services are to be migrated, this also places requirements on the network protocols, which would need to be able to handle communication with the service in its new location. Network protocols are documented in Deliverable 41[29].

## 15.2 Comparison with Object-based Migration and Persistence

Processes on the `pal-vm` virtual machine are defined in such a way that their object graphs are disjoint. This means that no process contains a reference to an object in another process. All processes may hold references to immutable system objects, but such immutable system objects may not contain references to process data. Since they are immutable it is also not possible for a process to add a reference to its own data to such an object or to use such an object to gain access to the data in another process. HMAPs allow a safe way for process to exchange data without their object graphs becoming entangled. This is described in more detail in Section 8.

This process separation constraint mirrors that of protected multitasking operating systems. It is important in order to prevent program instability in one process from spreading to another process. However, it also means that it would be theoretically possible to serialise all objects in a process, providing a complete persistent or migratable data set for a running process. Such a facility, storing all transitively reachable objects, would be somewhat more heavyweight in terms of implementation effort and space requirements.

A transitive object storage facility has the advantage of being simpler for the service programmer to use than HMAP-based persistence facilities: The programmer would not have to separate his data into persistent and transient portions, and would not have to deal with the less direct access to data that HMAPs offer when compared with simple objects with fields and inter-object references.

On the other hand, the automatic nature of an object-based persistence facility can be the source of disadvantages. Experience with a similar system in BETA[17] shows that it is easy to accidentally save or migrate too much data, as the storage system blindly follows inter-object references into data that need not be stored. In addition, loading such persistent data into a newer version of an application, where objects may have changed size, or program changes may have altered the interpretation of data in objects may produce unexpected results.

The process separation principle has as its prime motivation the desire to ensure process stability. However, it is interesting to note that the principle also opens the way to an alternative implementation of persistence, based on the object graph of a process. Nevertheless, it is unlikely that we will have time to implement a persistence system based on this.

# 16 VM Performance and Size

The challenges of PalCom include invisibility and construction/deconstruction. To meet this challenge, we aim to build our systems out of small building blocks that can be hidden in the environment (but visible in some form, when needed), and that can be composed together in flexible ways in order to build larger systems. In order for this to be possible the systems must be small, both in size, cost and power requirements. A simple way to save power is to run a system at a lower speed, so it is also important that the software run as fast as possible on a given hardware with a given CPU speed.

In order to facilitate the development of prototype applications for PalCom it is also important to have a reference implementation that exhibits these qualities of speed and economy of space – see, e.g., Section 19.

## 16.1 VM size

As mentioned in deliverable 22[23], the goal for the `pal-vm` is to be able to run on devices with as little as 256kbytes of memory. There are a number of designs decisions, that will help keep the size of the running VM low. These were detailed in deliverable 22.

As was also mentioned in deliverable 22, the Java based `pre-vm` was not made for efficiency and compactness, and did not in any way meet these size expectations. The currently used and developed `pal-vm` is the version that is attempting to meet these requirements. The current implementation, when compiled for the ARM processor, takes 124360bytes, or around 122kbytes (as determined by the `size` program on the `.o` files that make up the compiled VM). In order to run on embedded devices with Linux the VM must be linked to the system libraries. When this is done, the current size is 303944bytes or 297kbytes (this figure is for the VM version that is compiled to run on the ARM-Linux based Axis cameras). It is to be expected that this figure can be reduced by reconfiguring the linker, and it would be considerable lower for a VM that was designed to run directly on the hardware without an operating system. Nothing in the VM design precludes creating such a version of the VM, but it is currently unclear whether any of the PalCom prototypes require such an effort.

The runtime memory consumption of the VM is still too high. One reason for this is that the garbage collector does not yet implement older generations of the object heap, which means that the two first generation heaps, or *semispaces*, must be large enough to hold all live objects. At the last measurement, 2006-12-07, the heap needed to be 452kbytes large to run the simple BenchPress benchmarks (apart from the `Storage` benchmark, which allocates a large amount of memory as part of its operation). This figure was reduced from 793kbytes in the summer of 2006 by a series of efficiency improvements and program changes. It is to be expected that, if necessary, the figure could be reduced further with the help of changes in the structure of basic objects in the VM and a multi-generation garbage collector.

The core libraries, including the communication libraries are still under development. Since a typical PalCom node will need these libraries, the size of the libraries, their memory use, and the size of any utility libraries they use will be an important factor in determining the effective minimum memory footprint of a device based on the PalCom architecture. Precise measurements of the space requirements cannot yet be made, since the specification and implementation of the communication protocols is ongoing. See Deliverable 41 for details[29].

## 16.2 VM speed

The initial proof-of-concept VM, `pre-vm`, written in Java, was abandoned. The PalCom VM specification was then reimplemented in the `pal-vm`, written in C++. One of the main reasons for this move was in order to improve the execution speed. Our measurements on the small BenchPress benchmark suite indicated that the speed improvement on the first versions of the `pal-vm` was of the order of a factor of 3 in throughput, and a

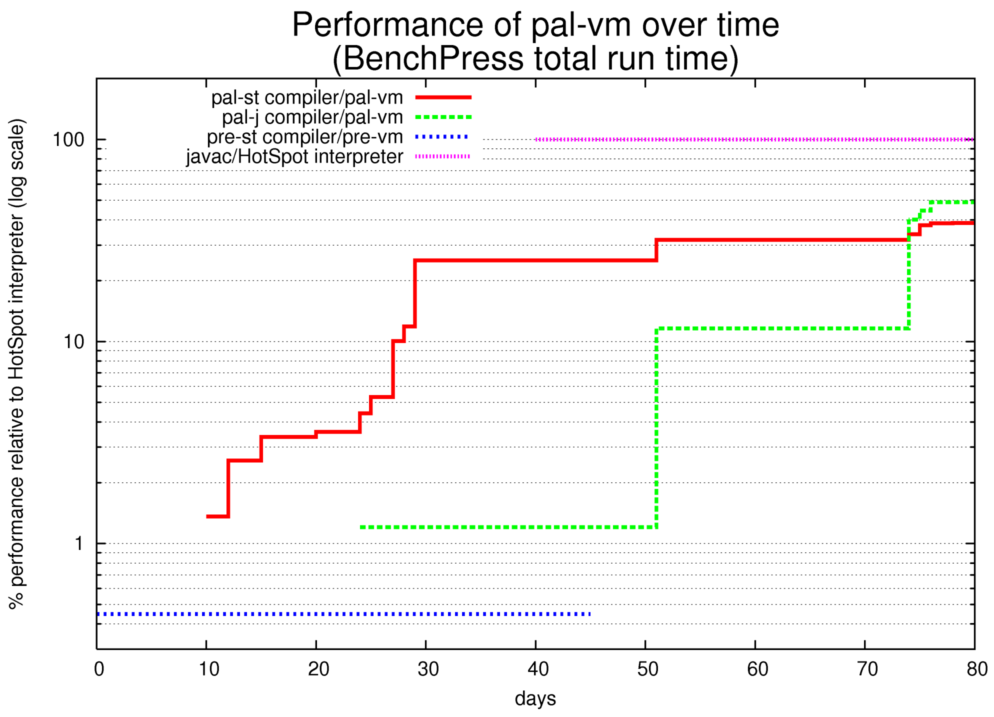


Figure 11: Performance of VM over time

relatively simple change to the way primitive operations were implemented brought this up to a factor of more than 5 relative to the `pre-vm`.

However, the performance of the `pal-vm` was still not satisfactory for those prototypes that were based on ARM processors. The groups working on these prototypes, which include all PDA-based devices, all UNC20-based devices (for example the tiles) and the Axis Cameras reported that performance was not acceptable, see Section 19. Therefore we had a period of profiling, analysing and measuring the VM speed, using our BenchPress benchmarks.

For comparison purposes, we translated the BenchPress benchmarks into Java (from Smalltalk), taking care to use 'natural' constructs, in order that the benchmark code should reflect typical use of the Java language. This enabled us to run the benchmarks using the industry standard `javac` compiler and the `HotSpot` VM[1]. These are standard Java tools, released by Sun Microsystems.

The HotSpot VM is is not intended for embedded use, but runs on desktop machines. Our tests on the `pal-vm` were therefore also performed on a desktop machine. However, we found that the performance improvements were generally seen both on desktop and embedded targets. Since `pal-vm` is interpreter-based for compactness and simplicity, we compared ourselves with the interpreter built into `HotSpot`. This interpreter is a highly optimised commercial product that has been in very widespread use and continual development for over a decade.

The VM was built in a version that could generate profiling information at the level of C++ source code lines. In addition, a version of the VM was augmented with tools for profiling byte codes executed on a Java or Smalltalk source code level. Performance was initially analysed mainly for the Smalltalk version of the benchmark, compiled with the PalCom Smalltalk compiler `pal-st`. Later performance work also used the Java version of the benchmarks, compiled with the PalCom `pal-j` compiler.

By analysing which parts of BenchPress had the most problematic performance, and by using profiling information at several levels we were able to initiate an iterative process, concentrating on the areas that appeared the most promising for a performance boost. See Figure 11 for the performance improvements we made. After each improvement to the VM the performance was tested on code compiled with `pal-st`. Performance was not tested as frequently on code compiled with `pal-j`. This is the reason that there are not so many steps in the graph for `pal-j`. No development occurred on the Java-implemented `pre-vm` or the version of `pal-st` used for `pre-vm`. This is why the `pre-vm` performance does not change over time. No major changes occurred in Sun's `javac` or `HotSpot` software in the time period in question, therefore the performance for these tools is also constant.

The current performance of the PalCom tools is around 50% of that of the Sun tools. As the main focus of the PalCom project is not performance, this is good enough that we do not expect speed to impede the progress of development of prototypes that use our tools.

# 17 Languages and Compilers

The virtual machine implemented by `pal-vm` is currently targeted by two compilers and an assembler (described in Appendix C). The first, `pal-st` compiles a dialect of Smalltalk we shall call PalST and generates component (`.prc`) files. The second, `pal-j` compiles a language we shall call PalJ, which is very similar to Java. It also produces `.prc` files.

In general we strive to be as compatible as possible to Java and Smalltalk, at the source code level. This enables us to use existing code in the PalCom project, and also enables the dual execution platform model, described in Section 20.

The differences between PalJ and Java and the differences between standard Smalltalk, and PalST are motivated by the following factors:

- Size: The requirements of palpability place constraints on the size of devices, which translate into memory space constraints for the VM. The constraints cannot be met without some incompatibilities and missing features relative to the standard languages. See also Section 16.1.
- Simplicity: Keeping the system simple helps keep it understandable, which helps make it inspectable and constructable/deconstructable. It also cuts down on implementation effort.
- Speed: Features that have proved costly to implement in terms of execution speed have been omitted. See also Section 16.2.
- Multiprocessor: Language features designed to improve performance on multiprocessor devices are not relevant to palpable devices, which are invariably single-processor, and are likely to continue to be so, given the constraints of cost, power consumption and heat production that are typical of palpable devices.
- Interoperability: Some changes have been made to the languages in order to accommodate running code from more than one language in the same virtual machine. See Section 11 for more details.

## 17.1 Smalltalk/PalST differences

Standard Smalltalk[16] has two different classes for characters and integers. In PalST and `pal-vm` there is only one class, called Integer. This makes the VM *simpler*. Experience suggests that this is not a problem in practice.

Standard Smalltalk does not define a syntax for class or method definitions. Method bodies are typed directly into a running Smalltalk virtual machine, which conventionally includes a compiler. (A snapshot recording of the internal state of the Smalltalk VM can be saved on disk and restarted at a later time.) Therefore there is no standard Smalltalk syntax with which to be compatible. PalST defines its own format.

The syntax of the body of a Smalltalk method is very simple. There are no big differences between the syntax of PalST and standard Smalltalk. Since characters are integers in `pal-vm` the dollar-based character constant syntax creates an integer.

For *simplicity* there are no floating point numbers in `pal-vm` and so the syntax for immediate floating point numbers is not supported in PalST. Likewise, there is no support for arbitrarily large integers. Integers in the `pal-vm` are 32 bits and arithmetic operations are on signed two's complement 32 bit numbers with no exceptions for overflows. This aids *interoperability* as it conforms to standard Java behaviour for integers, and it is also *simpler* than supporting arbitrary precision arithmetic.

The `pal-vm` does not include a compiler and so code must be compiled ahead of time. This is unusual for a Smalltalk implementation. Instead, there is support for reflective operations with the aid of external programs. In particular, the VM has support for loading new code at run time from component (`.prc`) files. This keeps the VM *small* and *simple*. Likewise, there is no support for adding a method to a class and other changes to classes. This would conflict with the aim of making classes read-only, which is important to keep the VM *small* when several processes share access to the same classes. See Section 8 for more details.

Although a method is a first class object (in keeping with Smalltalk tradition), the `pal-vm` does not in general make it possible to obtain the name of a running method or the methods in a class in the form of a printable string. This would preclude the compression or integer-encoding of method names, which (though it is not currently done) would help keep the memory *size* of the running program down. Alternative methods to get method names (for example for debugging) are detailed in Section 18.

Standard Smalltalk has two different kinds of static variables. *Class variables* are visible to subclasses. The superclass and subclasses share access to the same static variables. On the other hand *class instance variables* are inherited by subclasses. Each subclass gets its own copy of the class instance variables from the superclass, and these class instance variables are independent.

In `pal-vm` the variables of superclasses are not visible or inherited by subclasses. Access to the variables of superclasses is available only through getter and setter methods. This applies both to instance variables and class (static) variables. Because of this, the distinction between class variables and class instance variables disappears. The restriction on access to the variables of superclasses is present in order to decouple the representation of a class and its subclasses. This makes it simpler to introduce incremental update and versioning features to the VM at a later date, since program changes are likely to be isolated to a smaller number of methods or classes.

Each process has its own set of class variables in the `pal-vm`. Standard Smalltalk does not have separate processes, so the question of whether they are shared between processes does not arise.

### 17.2 Java/PalJ differences

Instances of the 'primitive types' (`int`, `boolean`, etc.) in Java are not real objects, having no methods and no inheritance hierarchy. For situations where the programmer needs a primitive instance that is a real object, there is a set of real classes that mirror the primitive types. These are `java.lang.Integer`, `java.lang.Boolean`, etc.

In `pal-vm` everything is an object. Though integers/characters are efficiently implemented this is transparent in the programming model, where they can be used just like any other objects. This means that the dichotomy between `int` and `java.lang.Integer` is not necessary. Therefore in PalJ both the `int` type and the `java.lang.Integer` class are mapped to the basic `Integer` class from the base library. This is both *simple* and helpful for interoperability.

In practice this does not lead to much incompatibility with Java, since the difference cannot be detected by a Java programmer. The static Java type system, enforced by the `pal-j` compiler, ensures that an `int` is never encountered where a `java.lang.Integer` was expected and vice versa. It is only when writing code that interoperates with code written in other languages (ie. Smalltalk) that the difference can be detected with a suitably constructed example.

Some packages and classes that are required to be present in a standard Java environment are not present in the PalCom system. See Appendix G for more details. This is to keep the system *small* and *simple*. One of the missing packages is the `java.lang.reflect` package. This is because we handle reflection differently in the PalCom system to the way it is handled in Java. See Section 14 for more details.

In standard Java, a lock can be taken on any object. In the PalCom system we handle concurrency and threads with the `PalcomThreads` library, and so this operation is not available in this form. The omission of this operation also has the effect of making the VM implementation *smaller* and *simpler*.

Similarly, in standard Java a hash code can be produced for any object using the static method on the `System` class, `identityHashCode(Object)`. A call to this method is also the default implementation for the `hashCode()` method on `java.lang.Object`. Once a hash code has been produced using this method, it is required never to change for that object. This makes it difficult to keep the per-object space overhead low.
The initial Java implementations had a non-moving garbage collector, which meant the address of the object could not change, and the address could be used to generate the hash code with no overhead. However, it is generally agreed that an exclusively non-moving garbage collector is less efficient than a moving garbage collector and has unresolvable memory fragmentation issues, so that course is no longer feasible when implementing a Java VM. There are complex solutions that involve only using space for the hash code of those objects where the hash code has been demanded by the program, but the complexity of such solutions works against our aim to keep the `pal-vm` *simple*.
In the `pal-vm` there is no `identityHashCode()` method. Only those classes that explicitly override the `hashCode()` method have a fixed immutable hash code. This includes strings and integers, since they can generate a fixed hash code from their (immutable) contents. But the base `Object` class does not have this operation, since that would require space in every object to store a hash code. It is possible to add a `hashCode()` operation to any class that needs it. This may be done by reserving space in the object for a hash code, but experience with Java shows that most objects in a system do not ever have their `hashCode()` method called. If the `hashCode()` method is called on an object whose class has not overridden the `hashCode()` method then a `#HashCodeMethodNotOverridden` exception will be thrown.

Thus, due to hash code and locking differences, the `pal-vm` has a per-object overhead of 4 bytes, whereas most Java VMs have a per-object overhead of 8 bytes. Since this overhead is encountered in every single object of a system we think this is a significant contribution to keeping the PalCom system *small*.

The null reference is implemented as a real object in `pal-vm`. This object is the singleton instance of the `Nil` class, usually called `nil` in Smalltalk. Since `Nil` inherits from `Object`, the base class at the top of the inheritance hierarchy, this means that all the methods from `Object` are available for null references. Thus, where a Java program would throw a NullPointerException when the `toString()` method is called on a null reference, a PalJ program merely returns the string `"nil"`. For methods not present in the `Nil` class, the `nil` singleton will throw an exception that is converted into a `java.lang.NullPointerException` as expected. Details of how this is done are in Section 12.

# 18 Debugging

The `pal-vm` needs to have facilities to enable the debugging of applications running on it. The features that support transparency can also be very useful for debugging purposes. For example, being able to remotely inspect HMAPs gives a programmer insight into the state of a program. The VM internal data that is made visible in the HMAPs can also be used to analyse a program that is not behaving as expected.

In addition to this, the `pal-vm` also has the ability to generate stack dumps, showing the sequence of method calls on the stack, with the types and contents of parameters, message receivers and local variables. The source code line number of the position in the code corresponding to each method invocation is also shown.

The data needed to generate readable stack dumps, including variable names, bytecode-to-line number mapping information is generated by both the `pal-j` and `pal-st` compilers. This is described in detail in Appendix F.

In addition to the reflection support and the stack dump support already implemented, the VM also needs more conventional debugging facilities. These include the ability to do one or more of the following:

- Inspect objects and stacks.
- Pause threads, coroutines and entire processes on demand
- Pause or log the program as it encounters predetermined code or data breakpoints
- Pause or log events such as exceptions being thrown
- Step through the program one line or one byte code at a time.
- Step over method calls, pausing the thread on return
- Evaluate expressions written in source form in the debugger.

Adding these capabilities to the VM is likely to increase the size of the VM considerably. In order to keep the size down there are several options:

1. Write the necessary code in Smalltalk or Java and only load the code if it is needed. This precludes debugging on very small devices, where the debugging code would not fit.
2. Make two versions of the VM, one with and one without debugging support. This also has the disadvantage that debugging is not possible on very small devices. In this case the debugging code would have to be written in C++.
3. Divide the VM in two parts, as shown in Section 14. Most of the debugging code could probably be placed in the reflective part where space is not so constrained. The reflective part can be written in Java.

It is our experience that writing large amounts of complex code in C++ is slower and more time consuming than writing in Smalltalk or Java. Therefore we are likely to choose a combination of the first and third options.

# 19 Example Usages of `Pal-vm`

The `pal-vm` is used within a number of Application Prototypes within the project: In the previous WP3 Deliverable 22 [23], it was described, how the VM (then known as `pre-vm-c`) was used in the WP8 Major Incidents prototype called *BlueBio*.

Since then, the VM has been taken into use in one more prototype, and a couple more are planned for the near future.

## 19.1 Use of `Pal-vm` In Active Surfaces

The WP11 Care Community prototype called *Active Surfaces* (or often just *Tiles*) consists of a number of waterproof tiles able to communicate with each other on four sides using infrared (IR) communication. Each tile can be fitted with any of a number of different surfaces (see Figure 12), with e.g. picture fragments or letters on them.

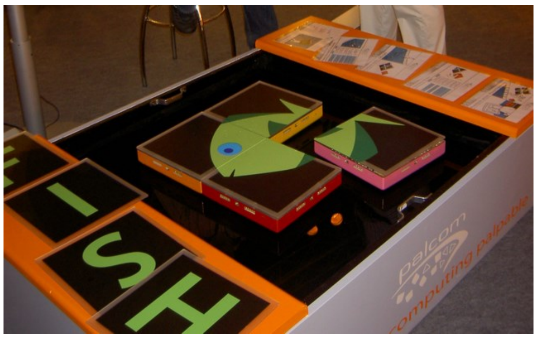


Figure 12: Active Surfaces Demonstrated at IST-Event 2006

The goal is to have mentally and/or physically handicapped children organise the tiles in games, like puzzle or scrabble, and thus provide them with an enjoyable rehabilitation process, where they learn while having fun.

An array of LEDs along each of the four sides of the tile provides the children with simple feedback, to indicate success as well as progress in the game being played.

### 19.1.1 Implementation

At the second PalCom review in 2006, a very first use of the VM within a Tile was shown, see left picture in Figure 13.

At the heart of each tile is an embedded Linux device, UNC20 [36], that runs `pal-vm`, as depicted in Figure 14. This device communicates with the PalCom-designed hardware that handles the actual communication and toggling of feedback LEDs. The right picture in Figure 13 shows the latest version of the Tile (without the lid). This is the version used at the IST-Event 2006.

At the top of the PalCom “stack” (see Section 20) is the actual game logic, which is currently just a proof-of-concept implementation of a puzzle game written in the PalJ language.

The game logic makes use of the PalCom communication layers (see WP4 Deliverables [24] [29]), that abstracts away the IR communication details, which in turn is implemented as a collection of `pal-st` classes in the `ist.palcom.base.networking` component.

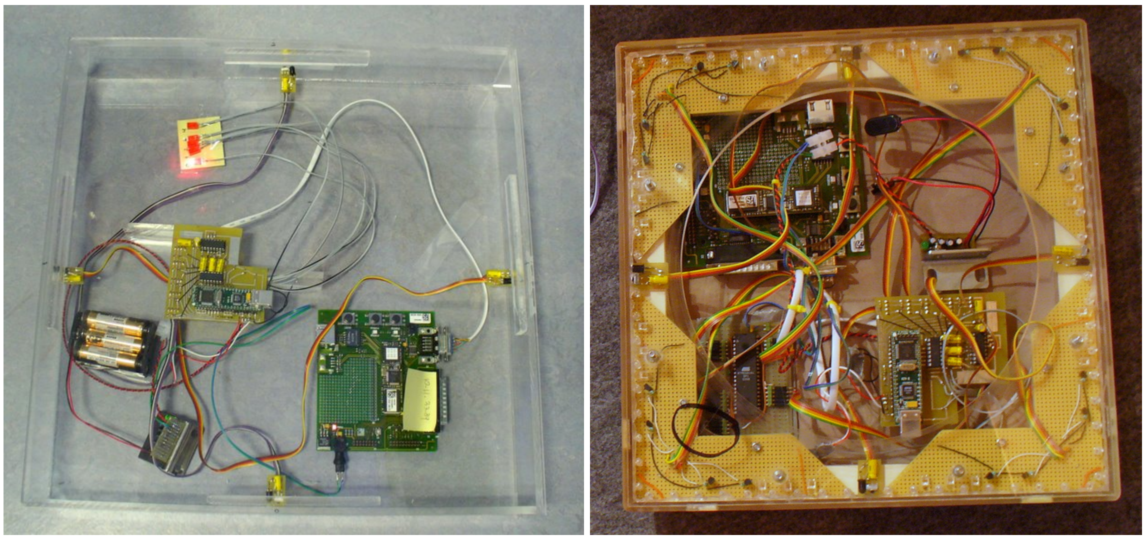


Figure 13: Tiles with VM at PalCom Review 2 and IST Event 2006

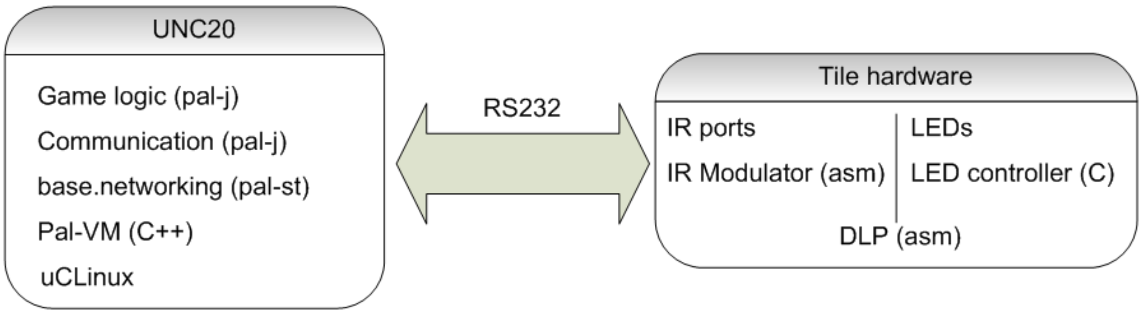


Figure 14: Tile Overall Structure

The current implementation of the Tiles include four physical Tiles. A fifth so-called "Assembler Tile" is being planned. This is intended to be the one, that the physicians use when configuring a new "game", and possibly also when inspecting the current game, e.g. using the WP6 assembly browser. The diagrams in Figure 15 show a UML-like view of a possible Tile architecture with Assembler Tile. The entities named `:HappyLocal` and `:Assembler` are expressed in a non-standard assembly notation.

For more information, consult the previous WP11 Deliverable [26], and upcoming WP7-12 Deliverable [31].

### 19.1.2 Influence on the PalCom Open Architecture

The prototype has influenced on the PalCom open architecture in a number of ways.

Firstly, the ongoing development of the tiles has lead to testing of and, indirectly, various improvements of the architecture reference implementation; in particular the `pal-vm` as well as the `pal-st` and `pal-j` compilers.

The tile hardware has only very limited resources available, as far as processing power, memory and communication bandwidth is concerned. Due to this, performance improvements have been implemented on the `pal-vm`, and the compilers, and considerable improvements have been achieved in this regard, c.f., Section 16.

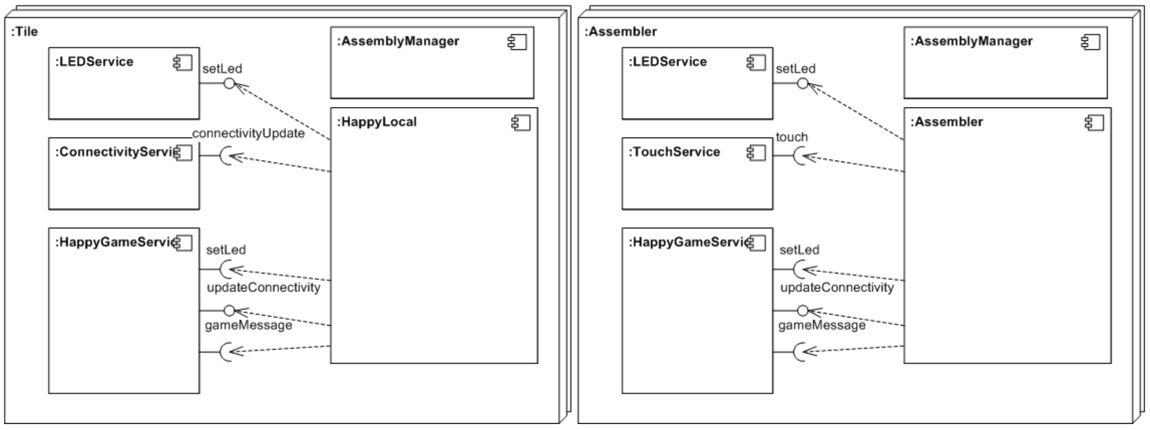


Figure 15: Tile and Assembler UML

Currently work is also being carried out to minimise the amount of traffic needed in order to support the bare minimal requirements of the PalCom device and service discovery protocol, with various approaches being discussed; among others a lightweight binary format as opposed to the current XML-based format, see deliverable 41 [29].

#### 19.1.3 Placement of the Code

All available source code specific to the tiles prototype can be found in the following location of the internal `palcom-i` CVS repository:

- `palcom-i/developer/services/communication/active-surfaces`

The contributions to the communication modules and the base component can be found in the (to-be) open source `palcom` repository at these paths, respectively:

- `palcom/developer/core/communication`
  The `ist.palcom.mal` component.
- `palcom/developer/runtime/pal-base/src`
  The `ist.palcom.base.networking` component.

## 19.2 Planned Use of the VM on AXIS Cameras

A couple of AXIS network cameras [8] are used in the application prototypes. For instance in the WP7 On-Site prototype, see Figure 16, where one such camera is in use at the PalCom exhibition at IST-Event 2006.

This (or similar) camera is planned to be further used in the WP7 prototypes, as well as in the WP8 Overview prototype. As described in Section 21.2, the VM has been ported to run on the AXIS platform, and we thus expect the VM used in these prototypes as they mature.

## 19.3 Planned Use of the VM for FROBS

The primitive mechanisms for resource accounting described in Section 13 have been specifically designed for the FROBS system [14] experimented with in WP5. It is expected that the FROBS system will be (partially) ported to run on `pal-vm`. This should not be a major task, since FROBS is written in Java, and will probably be fairly easy to get to run using `pal-j`. For more information, refer to the WP5 deliverable [30].

Figure 16: AXIS Camera at PalCom IST 2006 Exhibition

# 20 `Pal-vm` Programming Model

The general programming model for PalCom is described in the WP2 deliverable 39 [27], Chapter 7. That chapter also contain specific examples based on the `pal-vm` implementation of the PalCom Runtime Environment.

This chapter will add some more details to the "programmer's view" of the `pal-vm` implementation.

## 20.1 Dual Execution Engine

Chapter 5 of the WP2 deliverable 39 [27] contains a figure showing the layered architecture of a given PalCom Node. The first layer above the (optional) Operating System is named "Runtime Engine" in that figure. In the concrete implementation of this "PalCom Stack" in the (to-be) open source CVS tree (see Section A), a dual execution model is implemented, as detailed in Figure 17.

The essence is, that on a Desktop machine, one can choose to execute PalCom code on either the Java Virtual Machine (JVM) or the PalCom Virtual Machine (`pal-vm`). The Core Libraries, Middleware Management, and Utilities are all written in Java code, compatible with `pal-j` (cf. Section 17), which means that they can be compiled with either the standard Java Compiler `javac` and executed on JVM, or with `pal-j` and executed on the `pal-vm`. See Section 20.3 below for details.

## 20.2 `Pal-vm` Out-of-the-box or Developer Use

As can be seen from the code base described in Appendix A, there are two different ways to work with the `pal-vm`: The developer can either use the "out-of-the-box" precompiled binaries and libraries in `palcom/bin`

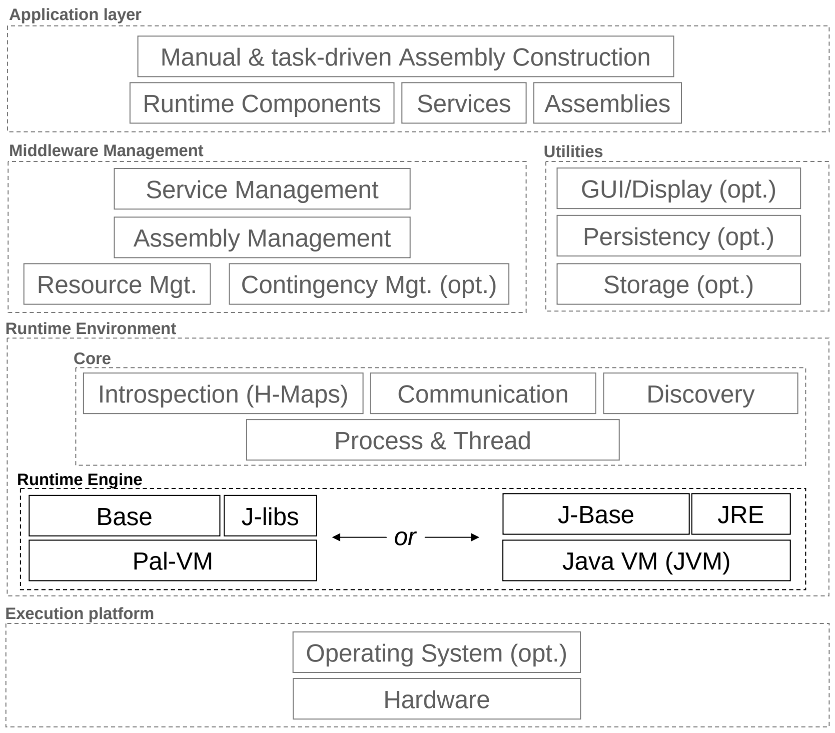


Figure 17: Runtime Layers on one PalCom Node

and `palcom/lib`, i.e., `pal-vm`, `pal-st`, and `pal-j`. Working this way requires no other preparation than checking out the code base, and setting the PATH to include `palcom/bin`.

Alternatively, the developer may work with the developer tree included in the PalCom CVS. This means that they are using the `pal-dev-*` scripts included in `palcom/bin/developer` (e.g., `pal-dev-vm`, `pal-dev-st`, `pal-dev-j`), which use the latest developer version of the tools and libraries in the CVS tree. Doing so requires the developer to have `palcom/bin/developer` in their PATH, and have executed the script `pal-dev-build`.

Unless otherwise noted, the programming cycle for `pal-vm` described below assumes "out-of-the-box" use. For "developer" use, simply use the `pal-dev-*` scripts instead.

## 20.3 Programming Cycle

For a Java programmer, a programming cycle would typically start with initial programming on a standard Java Desktop installation, perhaps using the open source Eclipse development environment [12]. The PalCom CVS contains a number of directories containing Eclipse projects in the form of `.project` files and supporting stuff, e.g. ANT scripts with CLASSPATHs set up appropriately for JVM execution. The programmer can then run their program from within Eclipse, and thus execute on the JVM.

The next step for the Java programmer would then be to recompile the source code using the `pal-j` compiler (on the command line), and then execute the resulting component(s) on the `pal-vm`, cf. below. A thorough tutorial on how to program with `pal-j` is available in the PalCom CVS at `palcom/doc/tutorials/pal-j`.

A different programming approach can be taken by programming in Smalltalk, and compiling it with the `pal-st` compiler. Using this approach, the developer will typically skip the use of, e.g., Eclipse completely. They will still be able to utilise the Core, Infrastructure, etc. libraries (written in Java), through the use of the language interoperability mechanisms described in Section 11. But often, the program will directly use the (Smalltalk based) methods found in Base. This will provide the fastest and smallest code, compared to doing it in Java, but the resulting program will, of course, only be able to run this on the `pal-vm`, and not the JVM.

When ready, the program can then be executed on the `pal-vm`, typically first on the Desktop machine. Concretely the outcome of the `pal-j` or `pal-st` compilation(s) will be one or more PalCom Component Files (with extension `.prc`), c.f., Section 6. Assuming these are placed in the current directory, the execution using `pal-vm` is done from the command-line using the command

```
        pal-vm -cp . mycomponent.prc
```

The `-cp .` argument instructs the VM to add the current directory to the *component path* it uses to find components. The `pal-vm` in this command line is one of the scripts explained in Appendix A and it automatically adds `palcom/lib` to the component path. This is the location of the precompiled components for the standard "PalCom Stack" libraries, further explained in Appendix A. If the developer is using the "developer" approach, the `pal-dev-build` script generates components into the default component path `palcom/developer/dev-lib`, which is automatically added to the component path by the `pal-dev-vm` script.

Once the code is running on `pal-vm` on a Desktop machine, the developer may want to copy the components (including necessary system components from `palcom/lib` or `palcom/developer/dev-lib`) to the embedded device. Furthermore, the developer will need to build and copy the binary `pal-vm` for that platform. This is done by

1. Invoking the script `pal-dev-build-vm` with a cross build option for the target platform. E.g., to build the VM for UNC20, the command is `pal-dev-build-vm -t unc20`. The available options are displayed by executing `pal-dev-build-vm -h`.
2. Copying the binary VM to the device. The `pal-dev-build` script reports where the binary to copy is placed.

#### 20.3.1 JVM or `Pal-vm`?

Depending on the ultimate goals of the project, the coding cycle described above may be more or less appropriate.

Using the PalCom stack on the JVM provides (among others) the following advantages:

1. Full-featured programming environments like Eclipse, with powerful source level debugging
2. The application may make use all the powerful standard libraries, including advanced graphics

Using the PalCom stack on `pal-vm`, on the other hand, at the present time provides the following advantages:

1. Small devices, not just Desktop machines.
2. Support for resource monitoring
3. Dynamic languages (currently Smalltalk) as well as strongly typed (currently Java)
4. Language interoperability
5. Isolated processes on one VM

6. Efficient lightweight scheduling in the form of Coroutines

and later

1. VM state inspection (HMAPs)
2. Debugging support, possibly remote

If the application is only to run on the Desktop platform and does not have the need for the `pal-vm` specific primitives, or needs, e.g., advanced graphics, then the developer may want to (or need to) only code for the JVM. We recommend that development is done in Eclipse.

If, on the other hand, the application need the special things provided by `pal-vm`, it is, of course, optional whether the developer wants to code it in Java in Eclipse first. One benefit from doing this, though, is that the developer currently gets much better debugging support when running on the JVM, than when running on `pal-vm`. This may, however, change, as described in Section 18.

## 20.4 Language Restrictions

Even though Figure 17 indicates that if one programs in Java, one can freely choose between running on the JVM and `pal-vm`, this is not unrestrictedly the case. Firstly, as described in Section 17.2, the entire Java language is not supported by the `pal-j` compiler, so if one uses, e.g., floating point numbers in the code running on JVM, this will not function with `pal-j`/`pal-vm`. Furthermore, the full set of Standard Java libraries are not supported on `pal-vm` – see the following section – whereas some specific PalCom primitives are only available on `pal-vm`, e.g. the resource monitoring primitives described in Section 13.

## 20.5 Base Libraries

In Figure 17 the libraries depicted right above `pal-vm` and JVM, respectively, need a little extra explanation: As noted, all libraries above the Runtime Engine layer are currently written in Java, allowing for reuse on the two execution paths through the layers. As also explained in Appendix A, the Base library used by `pal-vm` (in Appendix A referred to as `pal-base`), contains the basic classes like `Object`, `Integer`, `Boolean`, etc., but also slightly more complex classes like `CheckedArray`, `Process` and the `System` class. For details on the Base classes, refer to Appendix G.

When programming for the JVM, corresponding Java classes are available in the J-Base libraries (`pal-jbase`). Analogue to this, a small subset of the standard Java libraries (called JRE in the figure) is available when running on the `pal-vm` in form of the J-Libs library (in a mixture of Smalltalk and Pal-J code). When coding in Eclipse, the CLASSPATHs of the projects are actually set up in such a way, that Eclipse checks application code against the J-Libs libraries there. This prevents use of JRE classes not supported on `pal-vm`.

## 20.6 Components and Interfaces

The `pal-vm` implementation of the PalCom component model is described in Section 6; here it is emphasised what this means from a programmer's perspective:

As already explained above, the `pal-vm`, loads and executes PalCom components in the form of PRC files, e.g. `mycomponent.prc`. The binary format of these PRC files is described in Appendix E, but this is mostly relevant for compiler constructors. However, Section 6 and Appendix D describes the Palcom Component Description files, which are textual files describing the contents of a given PRC file. These so-called PCS files are typically named, e.g., `mycomponent.pcs` or often just `Component`. A PCS file can be thought of as a "compiler script" that specifies, among other things, which classes should be included in a component, and which other components they depend on. One PCS file can only function as a "compiler script" for source files in one language. If components written in another language are required, they have to be compiled into separate PRC files and added as required components in the given PCS file.

The PCS files thus list the provided and required interfaces of the component. Currently only the *class names* are listed; it is being investigated whether explicit signatures for the various methods in the constituent classes should be (optionally) declared in the PCS files too, but this is not yet possible.

See also Section 11 for a discussion of interface description annotations, which is the current way to decorate Smalltalk code with signature type information.

## 20.7 Services

As defined in the Open Architecture Deliverable 39 [27] one characteristic of a Service is that it is *self-contained*. `Pal-vm` does not have a Service concept (this is provided by the Core libraries on top of the VM). `Pal-vm` instead provides *processes* which can be used to implement services: As described in Section 8, processes have the characteristics of being independent from each other by not allowing object-references to one process in another process.

This characteristic of processes is enforced by the structure of the `System` classes and VM primitives, and currently there is no need for the compilers or VM to do extra checking to ensure this.

The Base libraries also provide the `System>>startProcess` and `System>>startProcessByPath: path` methods, that can be used to launch services dynamically, if written using processes.

Cooperation of services residing either in different processes, in different VMs on a single node, or on different nodes will be using the communication protocols defined in Deliverable 41 [29].

It is of course also possible to program more than one service in a single process, if the "self-contained" characteristic is not interpreted as mandating complete separation. For services that cooperate significantly with other services this may be the easiest approach. The services that are together in one process will, however, not be safe from each other in the sense that:

- a fault in one service could crash another service running in the same process.
- for migration of services (cf. Section 15.1) between nodes it is likely that all the services in one process will have to be migrated together as a unit

# 21 Platforms and Dependencies

Currently `pal-vm` can be used on the following platforms:

1. Intel based Linux (Desktop)
2. ARM-9 based Linux (BlueGiga Router [9])
3. ARM-7 based Linux (UNC20 Board [36], AXIS Network Camera [8])
4. Intel based Windows XP with Cygwin [10] (Desktop)
5. PowerPC based Mac OS X 10.4 (Desktop)
6. Intel based Mac OS X 10.4 (Desktop)

Planned next platforms include

1. ARM based Windows Mobile 2003 (PDA)

## 21.1 Operating System Dependency?

From the start of the project, it was argued, that for very small devices it should be considered to run on machines without a traditional operating system (OS), i.e. on the "bare metal", see, e.g., WP3 Deliverable 22 [23, section 4.2], WP2 Deliverable 39 [27, section 5, architecture layers figure]. This was also inspired by the OSVM platform [2], which supported running without a traditional operating system.

As noted in Section 16.1, there is nothing in the general design of `pal-vm`, the precludes such a realisation. In the current implementation of `pal-vm`, however, the existence of an operating system is assumed a few places: In Section 9 it is mentioned that current implementation of preemptive scheduling uses an OS-level (or CPU-level) interrupt-handler. This functionality would be provided by a periodic interrupt in the absence of an operating system. The native futures described in Section 10.3 again rely on the existence of an OS thread to do the waiting for the result.

However, despite these dependencies on traditional OS behaviour, it would be straight-forward (but time-consuming) to run without a traditional OS. These dependencies are few in number and isolated to a few places in the implementation, and a non-OS based implementation would typically supply similar functionality, either provided directly by the System libraries, or by including a home-made "micro-operating-system" as part of the VM. This was indeed the case for the OSVM platform.

## 21.2 Platform Dependencies

During year three of the project, the platform dependencies in the VM code have been much more isolated than before. The source code is all included in the PalCom (to-be) open source tree in `palcom/developer/runtime/pal-vm/src/VM`. The platform specific files are:

**GNUMakefile**
Contains most of the platform specific settings for all platforms. As mentioned in Appendix A a major restructuring of the code base has been completed during the third year of the project. One of the major restructurings done within the VM sources is, that we used to use Makefiles automatically generated by Eclipse for each platform. Although convenient with the user interface for changing platform settings, it turned out to be too time-consuming to maintain settings for each platform individually. Especially since it is not easy to change for other platforms, than the one you are developing on.

To overcome these difficulties, a single GNU Makefile was produced, that basically contains three sections: A section with platform specific settings of compile time options, includes etc., a section with a common listing of the constituent source files, and a section with make targets for the user to invoke.

This simple structure has made it much easier to add new platforms, or to change settings for all or specific platforms. For instance the support for the Axis camera mentioned above, was added in a few hours.

**Platform.hpp**
Definition of interface to platform specific operations used by the VM. This includes some timing functionality as well as basic file I/O.

**UnixPlatform.cpp**
UNIX implementation of platform specific functions. Used on Linux and Mac OS X platforms.

**CygwinPlatform.cpp**
Cygnus Windows implementation of platform specific functions. Since Cygwin is an attempt to mimic UNIX on Windows, the differences from UnixPlatform.cpp are very small, and mostly concern various constants and path notation.

**NativeSymLookup.cpp**
This is used for looking up symbols after loading third party libraries. This is needed when a PalCom component is loaded, that references external third party code using the Native interface, see Section 10. This file should be split in parts corresponding to each platform, and included in the above mentioned platform specific source files.

**Threads.cpp**
The implementation of threads currently assumes POSIX Threads to be available. This may not always be the case, and as such this may need to be split out into the platform specific files.

Besides the above mentioned files, a few files use non-ANSI C++ code. Specifically the GCC [15] extensions for *computed goto*, *address-of labels*, and *asm bindings of variables to hardware registers* are used. All of these usages are, however, only for optimisation purposes, and ANSI C++ compliant code is always included as fallback these places in the code. So even for a non-GCC compiler, this code should work as-is.

To summarise, the platform specific parts of the VM source code are quite isolated, and it should be straightforward to port the code to other platforms, as the need arises. As mentioned above, the next platform likely to be considered is a Windows Mobile based PDA. The major challenge expected are is Windows specific (non-POSIX) threads.

# 22 Open Issues

The following are a number of open issues to be dealt with in the future work in WP3:

- Two-part VM. As described in Section 14.2 we are considering a change to the VM to divide it into two tightly connected parts; the executing part and the reflective part.
- HMAPs. As mentioned in Section 14.3 it is still an open issue how much reflective information to make available through HMAPs and how HMAPs can be implemented in a two-part VM.
- Resource support. Some mechanisms have been proposed and implemented for low level resource management. See Section 13 for details. It is not yet clear whether these mechanisms are sufficient or correctly designed for allowing resource monitoring and manipulation, as needed by WP5 [25].
- Reflection. As mentioned in Section 14, some work remains in order to make the best design decisions for the reflective mechanisms.
- Dynamic upgrade, versioning, migration. The adequacy of the simple HMAP-based model for persistence (Section 15) has not yet been fully determined. This persistence model suggests a simple migration capability that may or may not prove to be useful in practice. The process model (Section 8) has been designed to ensure that processes are independent of each other, which leaves the possibility open of creating a migration mechanism that encompasses more of the state of a running process.
- Code verification. Though security issues are outside the formal scope of the project, a practical system may need to address issues of verifying code safety and enforcing a security policy in order to ensure system stability in the presence of code from multiple sources. This includes formulating a policy on which code may contain primitives.
- The benchmarks currently used to assess performance (Section 16) do not include the use of exceptions. If it proves that exception performance is inadequate, then the exception mechanism described in Section 12 may need to be revised.
- Optimisations. As mentioned in Section 16 a number of optimisations remain to be done in `pal-vm`. E.g. a multi-generation garbage collector must probably be implemented. The two-part VM design described in Section 14.2 would allow a just-in-time (JIT) compiler to be added to the VM. This JIT could generate either machine code, or it could generate more efficient byte codes, based on feedback from program execution.
- BETA. There exists a compiler for the BETA language that generates Palcom code, but it is not currently up to date, and does not generate code compatible with the `pal-vm`. The interoperability framework (Section 11) has been designed with BETA in mind, but BETA support has not been maintained the last 12 months and needs an update to work with the current mechanisms. Adding support for a third language is considered important, since this will demonstrate the generality of the mechanisms.
- Separation of code for Basename Interoperability: As mentioned in Section 11.4, currently the generation of base names from method names is hard-coded into the VM. The design allows for this functionality to be supplied as an “add-on” library, so that additional languages in principle can be supported by adding such a library. This has yet to be realised for the currently supported languages.
- Debugger. The design of a debugger and details of the mechanisms needed are not finalised. See Section 18.
- Native calls. An application that needs to do a lot of different native calls is awkward to program at the moment, since arguments must be packed into an array prior to every call. It may be possible to improve this process with better library and/or VM support.
- Scheduling. As mentioned in Section 9, the mechanisms for preemptive scheduling still have a number of open issues.

- Synchronisation between Threads. Although the mechanisms for preemptive scheduling of processes and Threads are in principle the same, cf. Section 9, the current implementation only allows for preemptive scheduling of Processes. This is mainly due to the lack of VM-level synchronisation mechanisms between Threads (this is not an issue between processes, since they cannot refer shared data). Thus if preemptive scheduling between Threads is requested, synchronisation will have to be designed and implemented.

- PCS interface declarations. As mentioned in Section 20.6, it is being considered to augment the Component Specification File descriptions to allow for interface declarations as an alternative to the IDA annotations currently used for Smalltalk code.

- Ports. The VM must be ported to a larger variety of devices, including PDAs.

- Non-OS based implementation. As summarised in Section 21.1 the current implementations all utilise the underlying Operating System, whereas the general design of the VM does not dictate the presence of an OS. It is still, though, unclear how much effort will be needed, should a non-OS implementation be requested on a given platform. Experiences from OSVM can be used to address this.

- Investigate possibility for offering palpable qualities on top of JVM, cf. Toolbox Contribution 6 listed in the Introduction, Section 4. Even though the current implementation provides a "dual execution engine" approach – see Section 20.1 – a number of specific mechanisms addressing palpable qualities, that are currently supported by (or planned for) `pal-vm` – e.g. support for resource monitoring, dynamic languages, language interoperability, isolated processes, VM state inspection, efficient coroutines – are not directly offered by JVM. It remains to be investigated to what extend (some of) these can be supported on top of the JVM using specially crafted tools and libraries.

# 23 Summary and Conclusion

The preceeding sections together with the appendices of this report show that although a number of open issues remain, substantial progress has been made in the design and reference implementation of the Palcom Runtime Environment:

The objectives were detailed in the introduction, below it is outlined how Task 1 and Task 2 listed on page 10 have been addressed. As described in Task 3, also on page 10, Tasks 1 and 2 together form the objectives of this deliverable.

## 23.1 Task 1: Support for resource and contingency management and code base for open-source

As was mentioned on page 10, this task involves toolbox contributions 1-4, also listed in the introduction. Progress has been made in all of these:

1. *Further specification of palpable runtime environment.*
   The further definition of the base class libraries is explained in Appendices A and G and a clarification of what parts of the VM are platform dependent was given in Section 21.
   With respect to *Improved Virtual Machine Reference implementation & supporting libraries* the improvements of the process handling was described in Section 8, support for preemptive scheduling was explained in Section 9, and exception handling was detailed in Section 12. No work has been done on improving garbage collection, since as explained in Section 16, other performance issues turned out to be more important. Improved garbage collection was mentioned as a still open issue in the preceeding section. As for the other performance issues, large improvements with respect to reduced memory consumption and optimisations have been obtained, as explained in Section 16. Finally access to legacy code has been streamlined by the introduction of the native mechanisms described in Section 10.
2. *Support for Resource - and contingency management*
   Primitive mechanisms for the new resource monitoring were described in Section 13, and the VM exception handling mechanisms – likely to be used by contingency management – was described in Section 12.
3. *Continued support for networking*
   The VM no longer provides specific primitives for networking; instead the native mechanisms, described in Section 10, for calling external libraries are now used in the core communication libraries and `ist.palcom.base.networking`.
4. *Support for introspection and reflection*
   A possible design for supporting reflection on resource constrained devices was presented in Section 14. The HMAP mechanisms mentioned in Sections 14 and 15 are already supported to some extent in the implementation of the VM.

Task 1 also suggests that toolbox contributions 5 and 6 are addressed, if time allows. This has been partly done:

5. *Support for program construction, analysis and supervision*
   Most of the mechanisms described in the preceeding sections have required parallel work on the VM and the compilers. Thus most of the improvements described are also dependent on improved Smalltalk and PalJ compilers. Specifically the language interoperability mechanisms described in Section 11 have required substantial changes in the compilers too.
   At the same time the documentation of how to work with the construction tools has been significantly improved, e.g. by the tutorials in `palcom/doc/tutorials`, mentioned in Appendix A, and further explained in Deliverable 36 [28]. But also by the programming model explanation in this deliverable (Section 20), and the codebase overview given in Appendix A. As mentioned in Section 18 the design and implementation of support for program analysis and supervision is still an open issue.

6. *Investigate possibility for offering palpable qualities on top of JVM*
   As mentioned in Section 20, there are some differences in possibilities when the JVM or the `pal-vm` runtime execution is chosen. As summarized in the list of open issues above, it remains an open issue to investigate to what extend (some of) the `pal-vm` specific mechanisms can be supported on top of the JVM.

## 23.2 Task 2: Further specification of runtime and improved code base for application prototypes

The first objective of this task was to deal with issues arising with the internal use of the open source code base, and maturing this for the IST Event 2006 exhibition. This has been an ongoing task for the entire year 3, and as explained in Appendix A, a coherent and understandable code base has now been established. This involved a lot of cleaning up and restructuring, as described in Appendix A.3. At the IST Event 2006, the PalCom Open Source Dissemination Kit was not announced – for the reasons explained in WP14 Deliverable 36 [28] – but the code base was still mature enough, that it was used in a number of the demonstrations shown, cf. Section 19.

The second objective was to provide WP2 with updates on the Runtime Environment specification. This has taken place by mutual discussion throughout the year, e.g. informing WP2 of the dual execution engine approach explained in Section 20 about the programming model. This deliverable – which has been produced in parallel with WP2 Deliverable 39 [27] – also details a number of designs from the WP2 Deliverable, specifically sections 6–18, 20, 21, and all the appendices.

To repeat the beginning of this section, substantial progress has thus been demonstrated in the design and reference implementation of the Palcom Runtime Environment.

A number of open issues still remain. As many as possible of these will be addressed in the last year of the project.

# A Code Base

The code base for the VM and supporting tools are all in the PalCom CVS Root simply called `palcom`, i.e., the CVS tree under preparation for open source. The open source strategy is further described in WP14 Deliverable 36 [28]. Appendix B of Deliverable 36 contains detailed setup instructions for the open source CVS tree. Below only the essentials of this are repeated. Furthermore we explain some of the changes to the code base, that have been necessary for this new CVS Root.

## A.1 PalCom CVS Root

The PalCom CVS root is currently placed at

- `cvs.daimi.au.dk:/users/palcom/palcom`

It can be accessed through either `ssh` or `:pserver` – both methods require that you are registered as a user, please contact *palcom2-admin* **at** *ist-palcom.org* if you need access.

It is also possible to access the CVS through a web interface at the address

- `http://www.ist-palcom.org/cvs`

(click on "Palcom Opensource CVS"). Again you will need credentials to log on, please contact *palcom2-admin* **at** *ist-palcom.org* .

With the exception of the `pal-beta` compiler, `pal-vm` and all supporting tools and base libraries are part of this CVS Root. This includes

- `palcom/developer/runtime/pal-vm/`
  The `src/VM` subdirectory contains the `pal-vm` (C++) source code. The platform dependent files were listed in Section 21.
- `palcom/developer/runtime/pal-vm/benchmarks`
  Smalltalk programs used to measure the performance of the VM and compilers (see Section 16 for details).
- `palcom/developer/runtime/pal-vm/tst`
  Smalltalk programs used as a unit test for testing the functionality of the VM and the `pal-st` compiler.
- `palcom/developer/runtime/j-libs/`
  Basic libraries - written in a mixture of Smalltalk and Java source code - to be used on top of the `pal-vm`. Contains a small subset of the standard Java class libraries. Mainly used to allow for execution of Java programs and libraries on `pal-vm`, but can also be used from Smalltalk. See Figure 17 on page 52.
- `palcom/developer/runtime/jbenchmarks/`
  Java programs equivalent to the Smalltalk programs in `palcom/developer/runtime/pal-vm/benchmarks` used to compare the performance of programs written in Java source code, with equivalent programs written in Smalltalk (see Section 16 for details).
- `palcom/developer/runtime/jtst/`
  Java programs used as a unit test for testing the functionality of the VM and the `pal-j` compiler.

- `palcom/developer/runtime/pal-base/`
  Minimal base libraries used by programs running on `pal-vm`, see Figure 17 on page 52. Contains base classes like `Object`, `Integer`, `Boolean`, etc., but also slightly more complex classes like `CheckedArray`, `Process` and the `System` class.
  Furthermore the subdirectories `palcom/developer/runtime/pal-base/networking/` and `palcom/developer/runtime/pal-base/storage/` contains classes supporting networking and (very simple) disk-based storage.
  See Appendix G for a listing of the base classes.
- `palcom/developer/runtime/pal-jbase/`
  Java library corresponding to `pal-base` to be used when running a (Java) PalCom program on the JVM platform. See Figure 17 on page 52.
- `palcom/developer/tools/bytecode/pal-asm/`
  The (Java) source code of the bytecode assembler.
- `palcom/developer/tools/bytecode/pal-dis/`
  The (Java) source code of the bytecode disassembler.
- `palcom/developer/tools/bytecode/pal-bytecodes/`
  Common bytecode (Java) libraries used by the bytecode assembler, disassembler, and the Smalltalk compiler.
- `palcom/developer/tools/compilers/pal-j/`
  The (Java) source code for the PalCom Java subset (PalJ) compiler. It contains two main subdirectories: One for a generic Java 1.4 frontend, and one for a backend producing `pal-vm` bytecodes. Both of these are written using the JastAdd [37] Java based attribute grammar system. The Java frontend is a copy from another project, whereas the backend is developed in PalCom. For more about the `pal-j` compiler, and the Java subset, it supports, refer to Section 17.
- `palcom/developer/tools/compilers/pal-st/`
  The (Java) source code of the PalCom Smalltalk compiler. The frontend of this compiler dates back to an older project called SOM [4], and its successor called POMP [3][33], whereas the PalCom bytecode backend is developed in PalCom. For more about the `pal-st` compiler, and the Smalltalk variant, it implements, refer to Section 17.

The `pal-beta` compiler is not under the PalCom CVS, but is included in the source for the BETA system. A binary distribution is available under the PalCom BSCW, though, at the address

- `http://bscw.ist-palcom.org/Workpackages/WP03/PAL-VM/PRE-BETA`

Notice, that the `pal-beta` compiler is currently not compatible with the latest `pal-vm`.

## A.2 Scripts

To ease the execution of the various tools in the code base, as well as the building of the code base (for developers), a number of command line scripts are available.
The use of these is described in `palcom/doc/setup/Command-line-tools.txt` but the essential part is:

- The scripts in `palcom/bin/` use precompiled, binary distributed tools in `palcom/bin/` and `palcom/lib/` and as such are ready to use “out of the box”, when you have checked out the repository, see Section 20.2.
  The developer scripts include

`pal-vm`
: Executes the binary distributed VM. Script command line options allows selecting between two builds of the VM (Release, Debug).

`pal-asm`
: Executes the binary distributed bytecode assembler

`pal-dis`
: Executes the binary distributed bytecode disassembler

`pal-st`
: Executes the binary distributed Smalltalk compiler

`pal-j`
: Executes the binary distributed Palcom Java compiler

- The scripts in `palcom/bin/developer` use the latest build of the tools in `palcom/developer`, and as such requires `pal-dev-build` to be run as the first thing after checkout. All developer scripts have names that start with `pal-dev-`, and it is thus possible to have both the developer-, and non-developer scripts in ones PATH. The developer scripts include

`pal-dev-build`
: Builds the basic code base including compilers, VM, base libraries, and core libraries. Use a number of other build scripts (not listed here) to build the various parts of the code base.

`pal-dev-clean`
: Cleans the developer tree of all generated files

`pal-dev-test`
: Executes various tests to ensure a consistent code base. This includes the `tst` and `jtst` unit tests, a number of test programs for the core libraries, all the tutorials in `palcom/doc/tutorials`, and (optionally) the benchmarks programs listed above.

`pal-dev-export`
: Makes an "export" of the current binaries in the developer tree into `palcom/bin` and `palcom/lib`.

`pal-dev-vm`
: Executes the developer version of the VM. Script command line options allows selecting between four builds of the VM (Release, Debug, Checked, Profiling).

`pal-dev-asm`
: Executes the developer version of the bytecode assembler

`pal-dev-dis`
: Executes the developer version of the bytecode disassembler

`pal-dev-st`
: Executes the developer version of the Smalltalk compiler

`pal-dev-j`
: Executes the developer version of the Palcom Java compiler

As said, the scripts are intended to make life easier for the programmers and users. They act as wrappers for the binaries, selecting the appropriate binaries for the current platform (in the C++ written VM case), setting up appropriate CLASSPATHs for Java programs (e.g. the compilers) and appropriate PalCom component paths, abstract away differences between platforms, like path notations on UNIX versus Windows, and setting default command line options. As described in Deliverable 36 [28, Appendix B], and further in the next section, the developer code base may be checked out in two different "topologies": A flat structure if using Eclipse project sets, or a tree structure if using a full CVS checkout. This further complication is also tested for, and handled by the developer command line scripts.

## A.3 Restructurings and Toolbox Contributions

Comparing the above overview of the current code base with the overview given in the previous WP3 Deliverable [23, Appendix G], it is obvious that a lot of restructuring of the code has taken place over the last year in preparation for going open-source.

As described in WP14 Deliverable 36 [28], this restructuring has involved a number of work packages, WP3 not the least.

The motivation for the new structure has been

1. The new structure should be easily comprehensible to external open source users and contributors.
2. It should be as easy as possible to get the system checked out or downloaded, and up and running
3. Development should work for both Eclipse users, and "command-line" users, not used to working in a graphical development environment.

As described in [28, Appendix B], this has led to a structure, where you can check out the entire (to be) open source development tree *and* the ready-to-use "out-of-the-box" binaries using just one CVS command. This work involved collecting all the previously separate CVS repositories into one. The overall structure of this new tree is listed in Figure 18.

```
palcom
   doc/
   lib/
   bin/
     developer/
   developer/
     runtime/
     core/
     tools/
     services/
     applications/
```

Figure 18: Overall structure of PalCom CVS Tree

As can be seen, the WP3 contributions listed in the preceding section spans a subtree of this.

With the aim of supporting Eclipse users better, an alternative check-out technique has also been provided for: In Eclipse, you can specify so-called ProjectSet files, which are XML files, that specify how to check out a specific sub-tree of a CVS Root. Since the PalCom developer tree contains a number of Eclipse projects (e.g. the compilers and the VM are separate Eclipse projects), a number of such Eclipse ProjectSet Files are also supplied. Using these, you get a flat directory structure in your Eclipse workspace, unlike the nested structure shown in Figure 18. As mentioned in the above section, this dual structure support complicates the command line scripts somewhat.

As described in Section 21.2, another major restructuring that has taken place is the creation of one unified GNUMakefile covering all build variants of the VM for all platforms, instead of the previous automatically generated platform specific Makefiles provided by the Eclipse CDT extension [11]. We are still programming the VM from within Eclipse, using the CDT extension, but CDT now uses our manually crafted GNUMakefile instead of building its own.

The restructuring has also involved identifying "Toolbox Contributions" from each work package, cf. Figure 19, which also comes from [28].

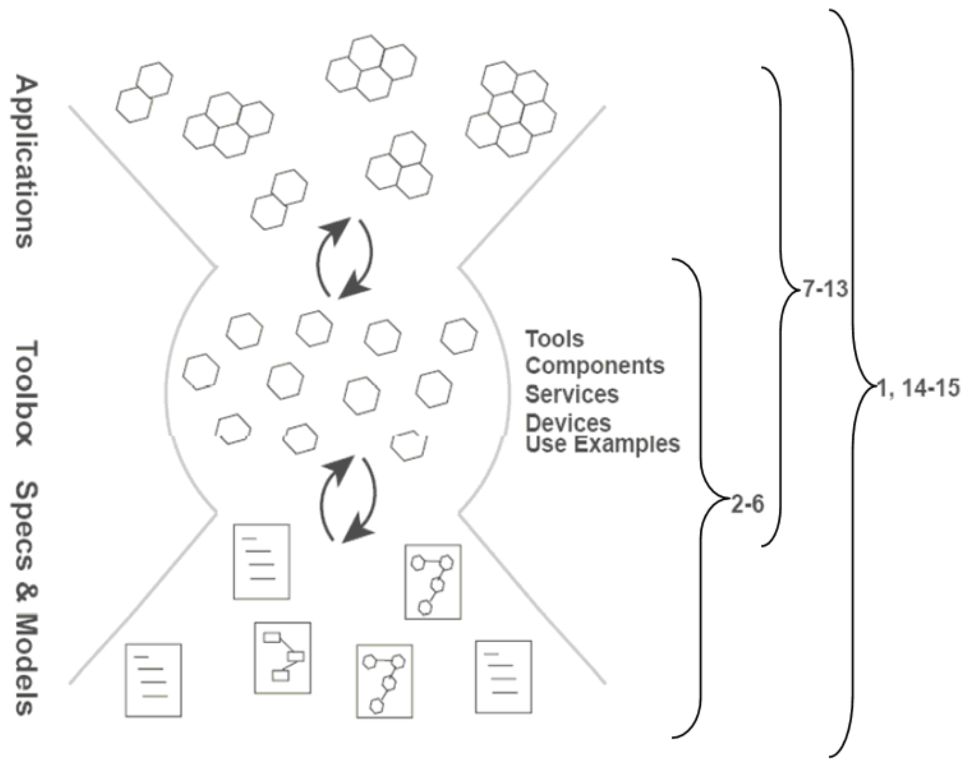


Figure 19: Abstract presentation of the PalCom "Toolbox"

The contributions to this Toolbox from WP3 are also listed in the "Toolbox Contributions" table on page 9 of the Introduction of this WP3 Deliverable. As it is illustrated in Figure 19, the WP3 contributions (together with other WP2-6 Contributions) lie in the bottom half of the middle part of this figure. This is further detailed in Figure 17 in Section 20.

# B Bytecode Reference

This appendix describes the binary bytecodes defined for the `pal-vm`. Discussion of the rationale behind the bytecode set can be found in Section 7.

In addition to the bytecodes, each method holds an array of values used by the bytecodes. These are called the literals. The literal array is implicit in the assembler syntax (Appendix C), but explicit in the binary component format (Appendix E).

**Notation.** The assembly syntax summarised for each bytecode is described in detail in Appendix C. All indexes into the literal array are marked with an asterisk (*) below. Each of the numbers in square brackets "[]" are bytes (8-bit values).

## B.1 halt

| Binary Format | Assembler Format |
|---|---|
| [0] | `halt` |

**Description**
Stop the execution of bytecodes.

**Stack Transformation**
... $\rightarrow$ ...

**Exceptions**
None.

## B.2 push local

| Binary Format | Assembler Format |
|---|---|
| [2] [local index] [depth] | `push.local <symbol> <depth:number>` |

**Description**
This byte code reads a value from a local variable slot of the current frame. The value is pushed onto the stack. The *depth* parameter determines how many levels of lexical nesting the local variable can be found at. If the *depth* is zero then this byte code accesses local variables in the current stack frame. If it is 1, then the byte code accesses local variables in the method or block in which the current block is lexically nested, etc. In order to implement this byte code, a block object must contain some form of reference to the stack frame in which it is defined.

**Stack Transformation**
... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**
None.

## B.3 push argument

| Binary Format | Assembler Format |
|---|---|
| [3] [argument index] [depth] | `push.argument <argument_index:number> <depth:number>` |

**Description**

Push the argument found at the given *argument index* on the stack of the current frame. The *depth* parameter determines how many levels of lexical nesting the argument can be found at. If the *depth* is zero then this byte code accesses arguments in the current stack frame. If it is 1, then the byte code accesses arguments to the method or block in which the current block is lexically nested, etc. In order to implement this byte code, a block object must contain some form of reference to the stack frame in which it is defined.

**Stack Transformation**

... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**

None.

## B.4 push field

| Binary Format | Assembler Format |
|---|---|
| [4] [*field name index] | `push.field <symbol>` |

**Description**

Get the field name from the current method using the *field name index* to index into the literal table, and push the value of that field in the *self* object on the stack. This instruction implicitly uses the self reference. The self reference is implicitly copied to blocks, which makes it accessible in block methods.

At lexical depth 0 (outside block methods) the self reference is identical with the 0th argument to the current bytecode method. At other lexical depths, the 0th argument to the current bytecode method is a block object. The virtual machine uses the class definition for the self object to map the symbolic name to an offset into the object layout.

**Stack Transformation**

... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**

None.

## B.5 push block

| Binary Format | Assembler Format |
|---|---|
| [5] [*block method index] | `push.block <symbol>` |

**Description**

Push a new block with a static link pointing to the current stack frame on the stack. The code for the block is given by the block method found in the literal table using the *block method index.* The block can no longer be used after the current stack frame has been destroyed by returning from it or unwinding the stack past it.

**Stack Transformation**

... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**

None.

## B.6 push constant

| Binary Format | Assembler Format |
|---|---|
| [6] [*constant index] | `push.constant <argument>` |

**Description**

Push a constant to the stack from the literal table of the current method using *constant index* to index into the table. Arguments can be integers, symbols, strings, immutable arrays, and immutable hash maps. The assembler syntax for these arguments is documented in EBNF in Appendix C.

**Stack Transformation**

... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**

None.

## B.7 push global

| Binary Format | Assembler Format |
|---|---|
| [7] [*global name index] | `push.global <symbol>` |

**Description**

Get the global name from the literal table of the current method using *global name index* and lookup the name in the global dictionary; push the value of the global to the stack. If the name contains a colon (':') then the text before the colon is the name of a component in which the lookup should take place. Otherwise the lookup takes place in the current component (the component of the class in which the current method is defined), and if the name is not found in the current component then the components on which the current component depends are searched. If the component name is explicitly specified then it should match the name of the current component or one of the components on which the current component depends. By giving the name of a class as the parameter to this bytecode it can be used to look up classes in components.

**Stack Transformation**

... $\rightarrow$ ... $\boxed{\text{value}}$

**Exceptions**

Sends the message `unknownGlobal` to the *self* object if the global name is not found.

## B.8 pop

| Binary Format | Assembler Format |
|---|---|
| [8] | `pop` |

**Description**

Pop and discard the top of the stack.

**Stack Transformation**

... $\boxed{\text{value}}$ $\rightarrow$ ...

**Exceptions**

None.

## B.9 pop local

| Binary Format | Assembler Format |
|---|---|
| [9] [local index] [depth] | `pop.local <local_index:number> <depth:number>` |

**Description**

The inverse of `push.local`. Pops the top of the stack, writing the popped value into the local found at the given *index*. The *depth* parameter determines how many levels of lexical nesting the local variable can be found at. If the *depth* is zero then this byte code accesses local variables in the current stack frame. If it is 1, then the byte code accesses local variables in the method or block in which the current block is lexically nested, etc. In order to implement this byte code, a block object must contain some form of reference to the stack frame in which it is defined.

**Stack Transformation**

... $\boxed{\text{value}}$ → ...

**Exceptions**

If *depth is non-zero and the object to be popped is a block object, then the* `#BlockWrittenToContext` *exception is thrown.*

## B.10 pop argument

| Binary Format | Assembler Format |
|---|---|
| [10] [argument index] [depth] | `pop.argument <argument_index:number> <depth:number>` |

**Description**

The inverse of `push.argument`. Pop the top element from the stack and store the element at the given *argument index* in the frame with the given depth.. The index may not be zero, since it is not permitted to change the receiver during execution of a method. The *depth* parameter determines how many levels of lexical nesting the argument can be found at. If the *depth* is zero then this byte code accesses arguments in the current stack frame. If it is 1, then the byte code accesses arguments to the method or block in which the current block is lexically nested, etc. In order to implement this byte code, a block object must contain some form of reference to the stack frame in which it is defined.

**Stack Transformation**

... $\boxed{\text{value}}$ → ...

**Exceptions**

If *depth is non-zero and the object to be popped is a block object, then the* `#BlockWrittenToContext` *exception is thrown.*

## B.11 pop field

| Binary Format | Assembler Format |
|---|---|
| [11] [*field name index] | `pop.field <symbol>` |

**Description**

The inverse of `push.field`. Get the field name from the current method using the *field name index* to index into the literal table, pop the top element from the stack, and store that element in the field of the *self* object. This instruction implicitly uses the self reference. The self reference is implicitly copied to blocks, which makes it accessible in block methods. At lexical depth 0 (outside block methods) the self reference is identical with the 0th argument to the current bytecode method. At other lexical depths, the 0th argument to the current bytecode method is a block object. The virtual machine uses the class definition for the self object to map the symbolic name to an offset into the object layout.

**Stack Transformation**

... $\boxed{\text{value}}$ → ...

**Exceptions**

If the object to be popped is a block object, then the `#BlockStoredInObject` exception is thrown.

## B.12 send

| Binary Format | Assembler Format |
|---|---|
| [12] [*message selector index] | `send <symbol>` |

**Description**

Get the message selector from the current method using *message selector index* to index into the literal table, and send it to an object on the stack (the receiver object). The receiver object is determined by counting the number of arguments in the message selector and using this number to index the stack. The receiving object is below any arguments on the stack (note: this ordering of arguments results from using right-to-left evaluation order). If there are no arguments, the receiving object is simply at the top of the stack. [2] A new stack frame is created based on the receiver method, with a dynamic link pointing back to the current stack frame, and evaluation continues in the new stack frame.

**Stack Transformation**

... $\boxed{\text{arg}_0}\boxed{\text{arg}_1}\cdots\boxed{\text{arg}_n}$ → ... $\boxed{\text{result}}$

(Including the return from the callee)

**Exceptions**

If the message selector is not a defined method of the receiver object, then the VM ensures that the message `doesNotUnderstand:arguments:` is sent to the receiver object instead (see Appendix G for details).

## B.13 super send

| Binary Format | Assembler Format |
|---|---|
| [13] [*message selector index] | `send.super <symbol>` |

**Description**

Equivalent to `send`, except that method lookup starts in the super class of the class that holds the current method. The method to be called by a given occurrence of this byte code can be determined statically, since it depends only on the selector (method name) and the class in which the current method is defined. Thus, the method called is independent of the class of the self reference.

**Stack Transformation**

... $\boxed{\text{arg}_0}\boxed{\text{arg}_1}\cdots\boxed{\text{arg}_n}$ → ... $\boxed{\text{result}}$

**Exceptions**

See `send`.

[2]Currently, the number of arguments is determined by the number of times the character “:” occurs in the selector.

## B.14 return local

| Binary Format | Assembler Format |
|---|---|
| [14] | `return.local` |

**Description**

Pop the top of stack (the return value) and return to the previous stack frame following the dynamic link, then pop the arguments and push the return value.

**Stack Transformation**

Local stack frame: $\ldots\boxed{\text{returnvalue}} \rightarrow$ POP FRAME
Previous stack frame (by following the dynamic link): see `send` bytecode.

**Exceptions**

If the object to be popped is a block object, then the `#BlockReturned` exception is thrown.

## B.15 return non-local

| Binary Format | Assembler Format |
|---|---|
| [15] | `return.non_local` |

**Description**

Pop the top of stack (the return value) and follow the chain of block stack frame references (starting at the stack frame referred by the current block) to the end (the lexically enclosing method invocation). Return from this stack frame like return local: follow the dynamic link, pop the arguments and push the return value.

**Stack Transformation**

Local stack frame: $\ldots\boxed{\text{returnvalue}} \rightarrow$ POP FRAME
Previous stack frame (by following the chain of block stack frame references to find the stack frame from which we should return): see `send` bytecode.

**Exceptions**

If the object to be popped is a block object, then the `#BlockReturned` exception is thrown.

## B.16 branch

| Binary Format | Assembler Format |
|---|---|
| [16] [bci-relative offset low byte] [bci-relative offset high byte] | `branch <label>` |

**Description**

The relative offset is constructed as a signed 16 bit value from the two parameter bytes that follow the branch bytecode. Add the relative offset to the current bytecode index, and continue execution from the computed bytecode index. Branch with offset zero branches to the branch bytecode itself.

**Stack Transformation**

... $\rightarrow$ ...

**Exceptions**

None.

## B.17 branch.identical

| Binary Format | Assembler Format |
|---|---|
| [17] [bci-relative offset low byte] [bci-relative offset high byte] | `branch.identical <label>` |

**Description**

The relative offset is constructed as a signed 16 bit value from the two parameter bytes that follow the branch.identical bytecode. Pop the two topmost elements of the stack, and compare them. If they point to the same object (are identical), add the relative offset to the current bytecode index and continue execution from the computed bytecode index. Otherwise, continue execution from the bytecode index following this instruction. A taken branch with offset zero branches to the `branch.identical` bytecode itself.

**Stack Transformation**

... $\boxed{\text{value1}}\boxed{\text{value2}} \rightarrow$ ...

**Exceptions**

None.

## B.18 branch.if.true

| Binary Format | Assembler Format |
|---|---|
| [20] [bci-relative offset low byte] [bci-relative offset high byte] | `branch.identical <label>` |

**Description**

The relative offset is constructed as a signed 16 bit value from the two parameter bytes that follow the branch.if.true bytecode. Pop the topmost element of the stack. If it is true, the system-wide global Boolean value, then add the relative offset to the current bytecode index and continue execution from the computed bytecode index. Otherwise, if the topmost element is false, the system-wide global Boolean value, then continue execution from the bytecode index following this instruction. A taken branch with offset zero branches to the `branch.if.true` bytecode itself.

**Stack Transformation**

... $\boxed{\text{value}} \rightarrow$ ...

**Exceptions**

The symbol #IfSentToNonBoolean is thrown if the topmost element of the stack is not either true or false.

## B.19 branch.if.false

| Binary Format | Assembler Format |
|---|---|
| [21] [bci-relative offset low byte] [bci-relative offset high byte] | `branch.identical <label>` |

**Description**

The relative offset is constructed as a signed 16 bit value from the two parameter bytes that follow the branch.if.false bytecode. Pop the topmost element of the stack. If it is false, the system-wide global Boolean value, then add the relative offset to the current bytecode index and continue execution from the computed bytecode index. Otherwise, if the topmost element is true, the system-wide global Boolean value, then continue execution from the bytecode index following this instruction. A taken branch with offset zero branches to the `branch.if.true` bytecode itself.

**Stack Transformation**

... $\boxed{\text{value}} \rightarrow$ ...

**Exceptions**

The symbol #IfSentToNonBoolean is thrown if the topmost element of the stack is not either true or false.

## B.20 swap

| Binary Format | Assembler Format |
|---|---|
| [18] | `swap` |

**Description**

Swap the order of the two topmost elements of the stack

**Stack Transformation**

... $\boxed{\text{value1}}\boxed{\text{value2}}$ → ... $\boxed{\text{value2}}\boxed{\text{value1}}$

**Exceptions**

None.

## B.21 dup

| Binary Format | Assembler Format |
|---|---|
| [1] | `dup` |

**Description**

Duplicate the top value on the stack

**Stack Transformation**

... $\boxed{\text{value1}}$ → ... $\boxed{\text{value1}}\boxed{\text{value1}}$

**Exceptions**

None.

## B.22 call

| Binary Format | | Assembler Format |
|---|---|---|
| [19] | [primitive name index] | `call <name>` |

**Description**

Calls a primitive. Gets the primitive name from the current method using the *primitive name index* to index into the literal table. The effect on the stack is the same for a primitive call and a non-primitive message send. The primitive may throw an exception instead of returning conventionally.

**Stack Transformation**

... $\boxed{\text{arg}_0}\boxed{\text{arg}_1}\cdots\boxed{\text{arg}_n}$ → ... $\boxed{\text{result}}$
(Including the return from the callee)

**Exceptions**

Depends on primitive called

# C Textual Assembler Reference

This appendix describes the `pal-vm` textual assembler, and the syntax for its input files.

## C.1 File name convention

`Pal-vm` textual assembler files must have names that end with the `.pra` extension.

## C.2 Assembly Language Syntax

The assembly language syntax is as follows, using EBNF [5].

```
<component>          ::= ( <metainfo> | <comment> | <class> )*

<metainfo>           ::= .metainfo <symbol> <symbol> <newline>
<comment>            ::= ; <not_newline>* <newline>

<class>              ::= <classname> <supername> <instance_or_class>*
<classname>          ::= .class <name> <newline>
<supername>          ::= .super <name> <newline>
<instance_or_class> ::= <instance_side> | <class_side> | <instance_method>
                       | <class_method> | <instance_field> | <class_field>
<instance_side>      ::= .instance_side <newline> { <newline> <member>* }
                         <newline>
<class_side>         ::= .class_side <newline> { <newline> <member>* }
                         <newline>

<member>             ::= <local_method> | <local_field>
<local_method>       ::= .method <symbol> <newline> <method_body> <newline>
<instance_method>    ::= .instance_method <symbol> <newline> <method_body>
                         <newline>
<class_method>       ::= .class_method <symbol> <newline> <method_body>
                         <newline>
<local_field>        ::= .field <symbol> <newline>
<instance_field>     ::= .instance_field <symbol> <newline>
<class_field>        ::= .class_field <symbol> <newline>

<method_body>        ::= { <newline> <metainfo>* <method_locals>
                           <method_stacksize> <bytecode>* }
<method_locals>      ::= .locals <number> <newline>
<method_stacksize>   ::= .maxstacksize <number> <newline>
<bytecode>           ::= <label>? <code> <argument>* <newline>
<code>               ::= halt | dup | push.local | push.argument | push.field
                       | push.block | push.constant | push.global | pop
                       | pop.local | pop.argument | pop.field | send
                       | super.send | return.local | return.non.local
                       | branch | branch.identical | swap | call
                       | branch.if.true | branch.if.false
<argument>           ::= <integer> | <symbol> | <string> | <array>
                       | <hashmap>
<integer>            ::= <number> | -<number>
```

```
<symbol> ::= <quoted_symbol> | <simple_symbol>
<quoted_symbol> ::= '#' <string>
<simple_symbol> ::= '#' <word>
<word> ::= <alphanum>+
<words> ::= <word> (' ' <word>)*
<alphanum> ::= <letter> | <digit> | '_'
<string> ::= '"' (""|<anychar>)* '"'
<label> ::= <string> ':'
<not_newline> ::= <anychar>*
<anychar> ::= <letter> | <digit> | ' ' | ...
<number> ::= <digit>+
<array> ::= ( <argument>* )
<hashmap> ::= { ( ( <symbol> | <integer> ) := <argument> )* }
```

The semantics attached to the metainfo parameters are described in Appendices E.3 and F.

The exact types and numbers of arguments that are allowed for each bytecode are documented in Appendix B.

# D Component Specification File Format

This appendix describes the syntax of the component specification file.

## D.1 File name convention

Component specification files should have names that end with the `.pcs` extension (for palcom component specification). This convention is not enforced by the compilers. The name `Component` without extension is often used in Java applications for historical reasons.

## D.2 Grammar for Component Specification File

```
<component>        := "component" <name> <component-body>
<name>             := <identifier> ( "." <identifier> ) *
<component-body>   := "{" <feature> "}"
<feature>          := <class> | <mainclass> | <requires>
<class>            := "class" <name>
<mainclass>        := "mainclass" <name>
<requires>         := "requires" <string> "as" <name>
```

# E Binary Component Layout

This appendix describes the binary format of `pal-vm` components.

## E.1 File name convention

`Pal-vm` component files must have names that end with the `.prc` extension.

## E.2 Grammar for binary components

The grammar of the binary component format is as follows, using EBNF [5].

```
<component>          := <component marker> <version> <metainfo>
                          <number of classes> <class>*
<component marker>   := <fixed length int = 0xCABAFDAF>
<version>            := <fixed length int>
<metainfo>           := <number of metainfo> <metainfo item>*
<number of metainfo>:= <int>
<metainfo item>      := <string> <string>

<number of classes> := <int>

<class>              := <class name> <super class name>
                          <metainfo> <instance side> <class side>
<class name>         := <string>
<super class name>   := <string>
<instance side>      := <metainfo><fields and invokables>
<class side>         := <metainfo><fields and invokables>

<fields and invokables> := <number of fields> <field name>*
                              <number of invokables> <method>*
<number of fields>      := <int>
<field name>            := <string>
<number of invokables>  := <int>
<method>                := <method signature> <metainfo>
                              <number of locals> <maximum stack size>
                              <number of literals> <literal>*
                              <number of bytecodes> <bytecode>*
<method signature>      := <string>
<number of locals>      := <int>
<maximum stack size>    := <int>
<number of literals>    := <int>
<number of bytecodes>   := <int>
<bytecode>              := <byte>

<literal>                := <simple literal> | <literal string> |
                               <literal method> | <literal array> |
                               <literal hash map>
```

```
<simple literal>          := <literal symbol> | <literal integer>

<literal method>          := <literal method marker> <method>
<literal method marker>   := <byte = 1>

<literal string>          := <literal string marker> <string>
<literal string marker>   := <byte = 3>

<literal symbol>          := <literal symbol marker> <string>
<literal symbol marker>   := <byte = 4>

<literal integer>         := <literal integer marker> <int>
<literal integer marker>  := <byte = 5>

<literal array>           := <literal array marker> <array>
<literal array marker>    := <byte = 7>

<literal hash map>        := <literal hash map marker> <hash map>
<literal hash map marker>:= <byte = 8>

<fixed length int>        := <bits 0-7> <bits 8-15> <bits 16-23> <bits 24-31>
<bits 0-7>   := <byte>
<bits 8-15>  := <byte>
<bits 16-23> := <byte>
<bits 24-31> := <byte>

<int>                     := <one byte int> | <two byte int> | <three byte int> |
                             <four byte int> | <five byte int>
<one byte int>            := <bits 0-6>
<two byte int>            := <bits 7-13> <bits 0-6>
<three byte int>          := <bits 14-20> <bits 7-13> <bits 0-6>
<four byte int>           := <bits 21-27> <bits 14-20> <bits 7-13> <bits 0-6>
<five byte int>           := <bits 28-31> <bits 21-27> <bits 14-20> <bits 7-13>
                             <bits 0-6>

<fixed length int>        := <bits 0-7> <bits 8-15> <bits 16-23> <bits 24-31>

<bits 0-7>   := <byte>
<bits 8-15>  := <byte>
<bits 16-23> := <byte>
<bits 24-31> := <byte>

<bits 0-6>   := <byte <= 127>
<bits 7-13>  := <byte >= 128>
<bits 14-20> := <byte >= 128>
<bits 21-27> := <byte >= 128>
<bits 28-31> := <byte >= 128>

<char>       := <one byte int> | <two byte int> | <three byte int>

<string>                := <number of characters> <char>*
<number of characters> := <int>
```

```
<array>                 := <number of elements> <literal>*
<number of elements>    := <int>

<hash map>              := <number of entries> (<simple literal> <literal>)*
<number of entries>     := <int>
```

## E.3 Metainformation

The following metainformation keys are recognised by the virtual machine.

**requires** : Indicates a requirement on another component. The value of this attribute is the name of a component file. The value may optionally contain a name that the required component will be bound to in the name space of the component. Multiple attributes are allowed per component.

**process.main_class** : Indicates the main class used to start the process that is instantiated from this component. The value of this attribute is the name of a class. Only the first occurrence of this attribute is used when loading a component.

**component.class** : Indicates the class that the runtime component should be an instance of. This class should be a subclass of the `Component` class. When a process is started that uses a component, a singleton instance of this component class is instantiated.

**component.initializer** : Indicates the instance method on the component class that initialises the singleton instance of the component class. This initialiser method is executed before the process main_class is started.

# F Reflection Data

The component format has provision for embedding reflection data. This data may be used by the runtime system, or by the development tools or a reflection server. Reflection data that is unused is discarded by the runtime system when the component is loaded.

Those items of reflection data that are stored as metainformation are represented as a collection of key-value pairs, where the keys and values are in the form of strings of characters. At runtime the keys and values are symbols. This applies both to the binary and textual formats component file formats.

## F.1 Field names

The names of fields in objects are stored in the `<instance side>` and `<class side>` in the binary component. See Appendix E. The names can be defined in the textual (assembler) format using the `.field`, `.instance_field` and `.class_field` directives to the assembler. See Appendix C.

## F.2 Local variable names

Local variable names are written into a component as metainfo on their method. See Appendix E and Appendix C for the details of how metainfo is stored. Metainfo is in the form of key-value pairs of symbols.

In the case of arguments, the keys are in the form ”`var.a1`”, ”`var.a2`”, ”`var.a3`”, etc. The zeroth argument is not explicitly named (it is normally called self or this).

In the case of local variables, the keys are in the form ”`var.l0`”, ”`var.l1`”, etc.

## F.3 Source code files

The source code files used to generate a component are written into a component as metainfo on the component. See Appendix E and Appendix C for the details of how metainfo is stored. Metainfo is in the form of key-value pairs of symbols.

For the source code files they keys are in the form ”`sourceFile.0`”, ”`sourceFile.1`”, etc. The numbers in the keys are used elsewhere in the component to refer to the files. The value in the key-value metainfo pairs is the full path to the source file, as seen by the compiler at compile time.

## F.4 Source code line numbers

Methods have a piece of metainfo with the key ”`ln`” that describes the source code line from which they are generated. This information can be useful to profilers, stack dump generators and debuggers. The data is in the value part of the key-value metainfo pair in textual form. The syntax is as follows, using EBNF [5].

```
<line info>          ::= <line spec> ( ; <line spec> )*
<line spec>          ::= ( <line> = ) ? <bytecode range>
                         ( , <bytecode range> )*
<line>               ::= ( <source file number> : )? <line number>
<source file number> ::= <int>
<line number>        ::= <int>
<bytecode range>     ::= ( <bytecode index> - )? <bytecode index>
<bytecode index>     ::= <int>
<int>                ::= ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
```

The line number information is in the form of semicolon-separated line specifications.

The line specifications are in the form of a line designation and an equals sign, followed by one or more comma separated bytecode ranges. If the line designation and the equals sign are missing, then the line designation is taken to be the line in the same file immediately following that of the previous line designation.

The line designation is in the form of a source file number, a colon and a line number. If the source file number and the colon are missing, then the source file number is taken to be the same as in the previous line specification.

The bytecode ranges are in the form of a bytecode index, a dash, and another bytecode index. They are inclusive and relative to the start of the method. They represent the positions in the byte stream (including bytecode parameters) that correspond to the given source code lines. If the first bytecode index and the dash are missing then the start of the bytecode range is taken to be one more than the end of the previous bytecode range.

As an example, consider a method in which bytecodes 0-4 are from source file 42, line 13. Bytecodes 5-10 are from source file 42, line 14. Bytecodes 11-12 are from source file 42, line 15. The ”`ln`” metainfo entry for that method could be coded as follows:

```
42:13=0-4;42:14=5-10;42:15=11-12
```

Equivalently, the source file number can be omitted from all line specifications but the first, since it is unchanged:

```
42:13=0-4;14=5-10;15=11-12
```

Equivalently, the line can be entirely omitted from all line specifications but the first, since each line is one greater than the previous one:

```
42:13=0-4;5-10;11-12
```

Equivalently, start of each bytecode range can be omitted from all line specifications but the first, since each bytecode range starts immediately after the previous one:

```
42:13=0-4;10;12
```

This last form is preferred where possible for space efficiency reasons.

# G System Classes

The `pal-vm` runtime environment defines a base class library (BCL) with a number of general purpose classes that can be used from any language running on the `pal-vm` platform. These classes define a number of standard methods available to all `pal-vm` programs.

A number of *primitive* functions are also defined. These functions are called using a special *call* bytecode. See Section 7.3 for the rationale of what is implemented as a bytecode and what is implemented as a primitive. Examples of primitives are mathematical functions and functions for basic allocation of objects.

Below we include an overview of the `ist.palcom.base` library, presented in a Javadoc [13] alike manner. To reduce the size of this document, we only include the class descriptions, and not the individual method descriptions. Furthermore, only the classes contained in `ist.palcom.base` are described, neither the two other components in the Base libraries `ist.palcom.base.networking` and `ist.palcom.base.storage` nor the `j-libs`, `j-base` and `core` libraries. For a full documentation, consult PalCom Working Note 110 - PalCom Javadoc [35].

## G.1 CLASS **Array**

Array is a system class that represents an mutable sequence of objects. The index of the first element is 0. The size of an array is fixed at creation time.

DECLARATION:

public class Array **extends** ist.palcom.base.ReadableArray

## G.2 CLASS **ArrayList**

ArrayList is an array based list that supports most of the interface of Java ArrayList. The storage is automatically extended to make room for new elements. Adding elements to the front or back takes time amortized O(1).

DECLARATION:

public class ArrayList **extends** ist.palcom.base.Object

## G.3 CLASS **Block**

Block is a system class that is used to implement control structures in Smalltalk. A Block is created automatically by the Smalltalk compiler when the Smalltalk block-syntax is used. Blocks are also used to implement exception handlers.

A block is evaluated by calling the methods value, value:, value:with:, etc.

A block may not be used after the method in which it is defined has returned. In order to enforce this rule, there are certain restrictions on what may be done with a block. In particular, it is not permitted to store a block in an object, to return a block from a method or to throw a block as an exception.

DECLARATION:

public class Block **extends** ist.palcom.base.Object

## G.4 CLASS **Block1**

Block1 is a Block that takes one argument.

DECLARATION:

```
public class Block1 extends ist.palcom.base.Block
```

## G.5 CLASS **Block2**

Block2 is a Block that takes 2 arguments.

DECLARATION:

```
public class Block2 extends ist.palcom.base.Block
```

## G.6 CLASS **Block3**

Block3 is a Block that takes 3 arguments.

DECLARATION:

```
public class Block3 extends ist.palcom.base.Block
```

## G.7 CLASS **BlockMirror**

There are restrictions on the operations that can be performed on Blocks. This is in order to ensure that blocks are never used after their context has returned. Due to these restrictions, blocks cannot be placed in StackFrame objects. Instead of a block, a BlockMirror is put in the StackFrame objects. The BlockMirror contains a Block, but the Block is not accessible and so unsafe operations are preventer.

DECLARATION:

```
public class BlockMirror extends ist.palcom.base.Object
```

## G.8 CLASS **Boolean**

Boolean is a system class representing a true/false value.

In Smalltalk applications, control structures are implemented using methods on the boolean class. In Java/Beta applications, control structures are implemented directly by the compiler.

DECLARATION:

```
public class Boolean extends ist.palcom.base.Object
```

### G.9 CLASS **ByteArray**

ByteArray is a system class representing arrays of byte values. The bytes are represented efficiently by the VM. The byte values are manipulated as small Integer values when reading and writing values.

The index of the first element is 0. The size of a ByteArray is fixed at creation time.

DECLARATION:

public class ByteArray **extends** ist.palcom.base.Object

### G.10 CLASS **ByteArrayBuffer**

ByteArrayBuffer is a ByteBuffer that uses a ByteArray as storage.

DECLARATION:

public class ByteArrayBuffer **extends** ist.palcom.base.ByteBuffer

### G.11 CLASS **ByteBuffer**

ByteBuffer supplies a stream-like interface to manipulating byte-data. This class is an abstract superclass that has two implementations: ByteArrayBuffer and MemoryByteBuffer, that uses a ByteArray and Memory object as storage respectively.

DECLARATION:

public class ByteBuffer **extends** ist.palcom.base.Object

### G.12 CLASS **Channel**

Channel is an abstract superclass for socket-like objects. Sub-classes to the Channel class can be used as parameter to the system select function, that waits for data on one of a set of channels.

DECLARATION:

public class Channel **extends** ist.palcom.base.Object

### G.13 CLASS **CheckedArray**

This class can be used to implement Java-style type checked arrays. It is subclassed by the 'Object createCheckedArrayClass' primitive. The primitive generates a new class if one does not already exist. The new class has an instance method called type, which returns the class or symbol-representing-interface which elements in this array must conform to.

DECLARATION:

public class CheckedArray **extends** ist.palcom.base.Array

## G.14 CLASS **Class**

Class is a system class whose instances represents classes. Any object has a class including class-objects.

A class cannot be constructed directly - they are constructed automatically, when loading a component.

DECLARATION:

```
public class Class extends ist.palcom.base.Object
```

## G.15 CLASS **Coroutine**

Instances of Coroutine represents independent execution sequences. A coroutine has its own stack.

DECLARATION:

```
public class Coroutine extends ist.palcom.base.Object
```

## G.16 CLASS **False**

A singular instance, 'false', of the False class represents the canonical false value.

DECLARATION:

```
public class False extends ist.palcom.base.Boolean
```

## G.17 CLASS **HashEntry**

HashEntry is a system class. It is the basic building block of HashMaps.

DECLARATION:

```
public class HashEntry extends ist.palcom.base.ReadableHashEntry
```

## G.18 CLASS **HashIterator**

HashIterator is a system class. It is obtained from the HashMap's or ReadableHashMap's iterator method. It is never instatiated directly.

DECLARATION:

```
public class HashIterator extends ist.palcom.base.Object
```

## G.19 CLASS **HashKeyIterator**

Iterator over the keys in a HashMap or ReadableHashMap.

DECLARATION:

```
public class HashKeyIterator extends ist.palcom.base.Object
```

## G.20 CLASS **HashKeySet**

A set view of the keys in a HashMap or ReadableHashMap. An instance of this class is returned from the HashMap's keySet method, which gives access to the the keys of a HashMap as a set.

DECLARATION:

```
public class HashKeySet extends ist.palcom.base.Object
```

## G.21 CLASS **HashMap**

HashMap is a system class that maps keys to values. The objects serving as keys are restricted to Symbol and Integer. Unlike the superclass, ReadableHashMap, instances of HashMap are mutable.

DECLARATION:

```
public class HashMap extends ist.palcom.base.ReadableHashMap
```

## G.22 CLASS **HashValueIterator**

Iterator over the values in a HashMap or a ReadableHashMap. Returned by the HashMap's iterator method. Is never instantiated directly by the user of a HashMap.

DECLARATION:

```
public class HashValueIterator extends ist.palcom.base.Object
```

## G.23 CLASS **HashValues**

A collection view of the values in a HashMap or ReadableHashMap. An instance of this class is returned from the HashMaps values method, which gives access to the HashMaps values as a Collection.

DECLARATION:

```
public class HashValues extends ist.palcom.base.Object
```

## G.24 CLASS **Integer**

Integer is a system class that represents small integer values. Integer values are efficiently represented as immediate values instead of full objects. The immediate values are tagged and therefore less than 32 bits are available for the values. Instances of the Integer32 class are used to represent full 32 bit integers.

DECLARATION:

public class Integer **extends** ist.palcom.base.Object

## G.25 CLASS **Integer32**

Integer32 is a system class whose instances represents 32 bit integer values. This class is only used for integers outside the bit range supported by the Integer class. This class is never instantiated directly by the programmer. The VM automatically creates Integer32 instances when needed.

DECLARATION:

public class Integer32 **extends** ist.palcom.base.Object

## G.26 CLASS **Link**

Link is used by the implementation of the LinkedList class. Instances of this class represent a position in a LinkedList and contain a reference to the successor and predecessor links as well as a reference to the List element at this position in the List.

DECLARATION:

public class Link **extends** ist.palcom.base.Object

## G.27 CLASS **LinkedList**

LinkedList is a double linked list that supports the interface of both Java LinkedList and typical LinkedList datastructures in the Smalltalk world. A LinkedList shares interface with the ArrayList class.

DECLARATION:

public class LinkedList **extends** ist.palcom.base.Object

## G.28 CLASS **Memory**

Memory is a system class whose instances represents externally allocated memory using the native malloc and free functions. The programmer must ensure that every malloc'ed Memory object is also free'ed exactly once. If a Memory object is not free'ed, there is a memory leak, if it is free'ed more than once, there may be heap corruption that can lead to an access-violation or similar.

Accesses to the underlying memory are bounds checked.

DECLARATION:

public class Memory **extends** ist.palcom.base.Object

## G.29 CLASS **MemoryByteBuffer**

MemoryByteBuffer is a ByteBuffer implementation that uses a Memory object as storage. Supports manipulation of externally allocated memory using the ByteBuffer interface.

DECLARATION:

```
public class MemoryByteBuffer extends ist.palcom.base.ByteBuffer
```

## G.30 CLASS **Method**

Method is a system class describing the methods of classes. Instances of this class are created by the component loader, and can not be created directly by the programmer.

DECLARATION:

```
public class Method extends ist.palcom.base.Object
```

## G.31 CLASS **NativeFuture**

An instance of a NativeFuture represents the return value of an external call which is taking place in a different OS thread. NativeFutures are used to implement the nativeLongRunningXxx calls in the System class. A NativeFuture is also returned from the nativeFutureCall:with: method on the System class.

If NativeFutures are used directly rather than through the nativeLongRunningXxx calls in the System class, then it should be noted that a NativeFuture has a backing resource, which is released the first time the value is extracted. Thus, it is important to eventually call one of the xxxValue methods.

DECLARATION:

```
public class NativeFuture extends ist.palcom.base.Object
```

## G.32 CLASS **Nil**

Nil is a system class that has a singular instance, 'nil', that represents 'nothing' or 'null'. The singular instance is used to implement null references in PalJ.

DECLARATION:

```
public class Nil extends ist.palcom.base.Object
```

## G.33 CLASS **Object**

Object is the system class that is a superclass of all other classes. This class defines the basic behaviour of all objects in the system.

DECLARATION:

```
public class Object extends java.lang.Object
```

## G.34 CLASS **PersistentComponent**

PersistentComponent represent a loaded component and contains classes and metainformation.

DECLARATION:

```
public class PersistentComponent extends ist.palcom.base.Object
```

## G.35 CLASS **Process**

The process represents a running process. The process contains a collection of all components used to implement the process. It also contains a HashMap that is used to implement per-process class variables (known as static variables in Java).

DECLARATION:

```
public class Process extends ist.palcom.base.Object
```

## G.36 CLASS **ReadableArray**

ReadableArray is a system class that represents a read-only sequence of objects. The index of the first value is 0. It is the superclass of the mutable class, Array, and of all Java arrays.

DECLARATION:

```
public class ReadableArray extends ist.palcom.base.Object
```

## G.37 CLASS **ReadableHashEntry**

ReadableHashEntry is a system class. It is the basic building block of ReadableHashMaps. ReadableHashEntry is immutable, but it has a subclass, HashEntry, which can be modified.

DECLARATION:

```
public class ReadableHashEntry extends ist.palcom.base.Object
```

## G.38 CLASS **ReadableHashMap**

ReadableHashMap is a system class. The objects serving as keys are restricted to Symbol and Integer. This map is not writable (see HashMap).

DECLARATION:

```
public class ReadableHashMap extends ist.palcom.base.Object
```

## G.39 CLASS **RoundRobin**

RoundRobin is a simple scheduler used for scheduling processes.

DECLARATION:

```
public class RoundRobin extends ist.palcom.base.Object
```

## G.40 CLASS **Runner**

Runner is a thread that executes the run method on a process.

DECLARATION:

```
public class Runner extends ist.palcom.base.Thread
```

## G.41 CLASS **StackFrame**

StackFrame is a system class that represents stack frames on the execution stack of a coroutine. This class is used to manipulate stack-traces in exception-handlers.

DECLARATION:

```
public class StackFrame extends ist.palcom.base.Object
```

## G.42 CLASS **String**

The String class represents character string. All literals in a program is represented as instances of this class. A String is immutable and cannot be modified.

DECLARATION:

```
public class String extends ist.palcom.base.Object
```

## G.43 CLASS **StringBuilder**

StringBuilder is an efficient way to create String objects. A StringBuilder contains a buffer to which Strings and other objects can be appended. The asString method returns a concatenation of the result of calling the asStrng method on all objects in the buffer.

DECLARATION:

```
public class StringBuilder extends ist.palcom.base.Object
```

## G.44 CLASS **Symbol**

Symbol is a system class that represents unique symbol values in the program. A Symbol is a canonical character String. A String can be converted to a Symbol using the asSymbol method, and a Symbol can be converted to a String using the asString method. In the smalltalk language a special syntax exists for creating a Symbol literal.

DECLARATION:

```
public class Symbol extends ist.palcom.base.Object
```

## G.45 CLASS **System**

The System class contains useful global functionality. The VM contains one singular instance of the System class that is shared between all processes. It cannot be instantiated directly.

DECLARATION:

```
public class System extends ist.palcom.base.Object
```

## G.46 CLASS **Thread**

Thread represents an independent execution sequence that is scheduled together with other Threads in a system of Threads within a Process.

DECLARATION:

```
public class Thread extends ist.palcom.base.Coroutine
```

## G.47 CLASS **True**

A singular instance, 'true', of the True class represents the canonical true value.

DECLARATION:

```
public class True extends ist.palcom.base.Boolean
```

## G.48 CLASS **Vector**

The Vector class an array based list. Vector is an alternative implementation of ArrayList. This class will be removed in future versions of the base library (to be replaced by the ArrayList implementation).

DECLARATION:

```
public class Vector extends ist.palcom.base.Object
```

## References

[1] The java hotspot performance enging architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[2] Osvm object-oriented software platform. http://www.esmertec.com/solutions/M2M/OSVM/.

[3] The pomp project. http://wiki.daimi.au.dk/som/the_pomp_project.wiki.

[4] The som project. http://wiki.daimi.au.dk/som/_home.wiki.

[5] ISO/IEC 14977:1996. Information technology – Syntactic metalanguage – Extended BNF. Technical report, ISO Standards, 1996.

[6] J.R. Andersen, L. Bak, S. Garup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation and evaluation of the resilient smalltalk embeded platform. http://www.esug.org/data/ESUG2004/ESUG2004-RT-Resilient.pdf.

[7] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition, 2000.

[8] Axis communications, axis 207 network camera. http://www.axis.com/products/cam_207/index.htm.

[9] Bluegiga technologies. http://www.bluegiga.com.

[10] Cygwin: Gnu + cygnus + windows. http://www.cygwin.com.

[11] Eclipse c++ development toolkit. http://download.eclipse.org/tools/cdt/releases/new.

[12] The eclipse project. http://eclipse.org.

[13] Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1–2 June 1995*, pages 151–173. Springer, 1995.

[14] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper Honig Spring. Pervasive computing with frugal objects. In *Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07)*, page To appear., Niagara Falls, Canada, May 2007.

[15] The gnu compiler collection. http://gcc.gnu.org.

[16] Adele Goldberg and David Robson. *SmallTalk-80: The language and its Implementation*. Addison Wesley, 1983.

[17] Stephan Korsholm. Transparent, scalable, efficient oo-persistence. In *Proceedings of the Workshop on Object-Oriented Technology*, page 212, London, UK, 1999. Springer-Verlag. http://as15.iguw.tuwien.ac.at/desarte/OOPersistence.pdf.

[18] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Language*. ACM Press/Addison Wesley, 1993.

[19] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[20] Peter Ørbæk. Programming with Hierarchical Maps. Technical Report DAIMI PB 575, Institute of Computer Science, University of Aarhus, 2005. http://www.daimi.au.dk/publications/PB/575/PB-575.pdf.

[21] PalCom. PalCom External Report 20: Deliverable 15 (5.1): PalCom Component Model. Technical report, PalCom Project IST-002057, March 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-15-[5.1]-PalCom-Component-Model.pdf.

[22] PalCom. PalCom External Report 27: Deliverable 21 (2.4.1): Specification of Programming Models and Language Support for Palpable Computing. Technical report, PalCom Project IST-002057, August 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-21-[2.4.1]-programming-models-and-language-support.pdf.

[23] PalCom. PalCom External Report 28: Deliverable 22 (2.3.1): Specification of Virtual Machine and Reference Implementation on selected Embedded Device(s). Description of use of VM in Application Prototypes. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-22-[2.3.1]-virtual-machine.pdf.

[24] PalCom. PalCom External Report 29: Deliverable 23 (2.4.2): Specification of Component & Communication model. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-23-[2.4.2]-component-communication-model.pdf.

[25] PalCom. PalCom External Report 32: Deliverable 24 (2.5.1): Design Issues for Resource Awareness and Management. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-24-[2.5.1]-resource-awareness-and-management.pdf.

[26] PalCom. PalCom External Report 38: Deliverable 30 (2.11.1): Care Community Application Prototypes. Technical report, PalCom Project IST-002057, November 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-30-[2.11.1]-care-community-application-prototypes.pdf.

[27] PalCom. PalCom External Report 50: Deliverable 39 (2.2.2): Open architecture. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-39-[2.2.2]-open-architecture.pdf.

[28] PalCom. PalCom External Report 51: Deliverable 36 (2.14.1): Dissemination. Technical report, PalCom Project IST-002057, October 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-36-[2.14.1]-dissemination.pdf.

[29] PalCom. PalCom External Report 55: Deliverable 41 (2.4.3): Components & communication. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-41-[2.4.3]-components-communication.pdf.

[30] PalCom. PalCom External Report 56: Deliverable 42 (2.5.3): Resource & contingency management. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-42-[2.5.3]-resource-contingency-management.pdf.

[31] PalCom. PalCom External Report 58: Deliverable 44 (2.7.2): WP 7-12 Prototypes status after Year 3. Technical report, PalCom Project IST-002057, January 2007. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-44-[2.7.2]-prototype-status-after-year3.pdf.

[32] PalCom. Palpable Computing: A new perspective on Ambient Computing. Annex I – Description of Work, update Jan. 2007. Technical report, PalCom Project IST-002057, 2007. http://www.ist-palcom.org/publications/review3/contract/PalComDoW6.pdf.

[33] Kari Schougaard and Ulrik P. Schultz. POMP – Pervasive Object Model Project. 9th Workshop on Mobile Object Systems (Resource Aware Computing), ECOOP 2003, http://www.daimi.au.dk/~kari/publications/mos03.pdf, 2003.

[34] Ulrik Pagh Schultz, Erik Corry, and Kasper V. Lund. Virtual machines for ambient computing: A palpable computing perspective. Technical report, July 2005. http://www.ist-palcom.org/publications/files/OT4AMI-prevm.pdf.

[35] David Svensson. PalCom Working Note #110: PalCom Javadoc. Technical report, Institute of Computer Science, University of Aarhus, 2006.

[36] http://www.unc20.net.

[37] Lund University. Jastadd open source java-based compiler compiler system. http://jastadd.cs.lth.se/.