Process Frames

L. Caniglia, V. Murgia & G. Richarte

December 26, 2008

This document applies to dynamic (a.k.a. JIT) compilers that use the actual execution stack to allocate Smalltalk *processes. For that matter, method and block activations are consecutive data structures embedded in the stack. However efficient, this implementation obscures the behavior of* Smalltalk *processes and makes them hard to understand for the system programmer. Following a more natural approach we provide a reification of process frames that exposes their underlying structure in a way that is significantly easier to teach and learn. These objects are used to implement Process Continuations in* Digitalk Visual Smalltalk *without modifying the Virtual Machine.*

Introduction

Even though dynamic (*JIT*) compilers use the CPU execution stack to incarnate *Smalltalk* processes, most implementations provide the *Smalltalk* programmer with the pseudo-variable thisContext. By referring to thisContext the programmer can see the current process as a linked list of first-class objects that represent meaningful method and block activations. Unfortunately, this pseudo-variable is not available in all dialects, as is the case in *Digitalk* ones. This forces the programmer to instead interpret the native stack as a complex data structure. The drawback of this shortcut is lack of clarity in two important classes at least, Process and Debugger. As a side effect, enhancing these classes or even fixing long-standing glitches turn otherwise routinely work into real challenges.

In times where the complexity of software is growing to limits that were unthinkable one or two decades ago, the inability to improve the debugger is something that professional development teams can no longer afford. Similarly, the irruption of *Process Continuations* as regular artifacts of broadly accepted frameworks [cf. 1], or the imminent adoption of multi-core / multi-threaded hardware, makes apparent the need for developers to become proficient in low-level aspects of *Smalltalk* systems.

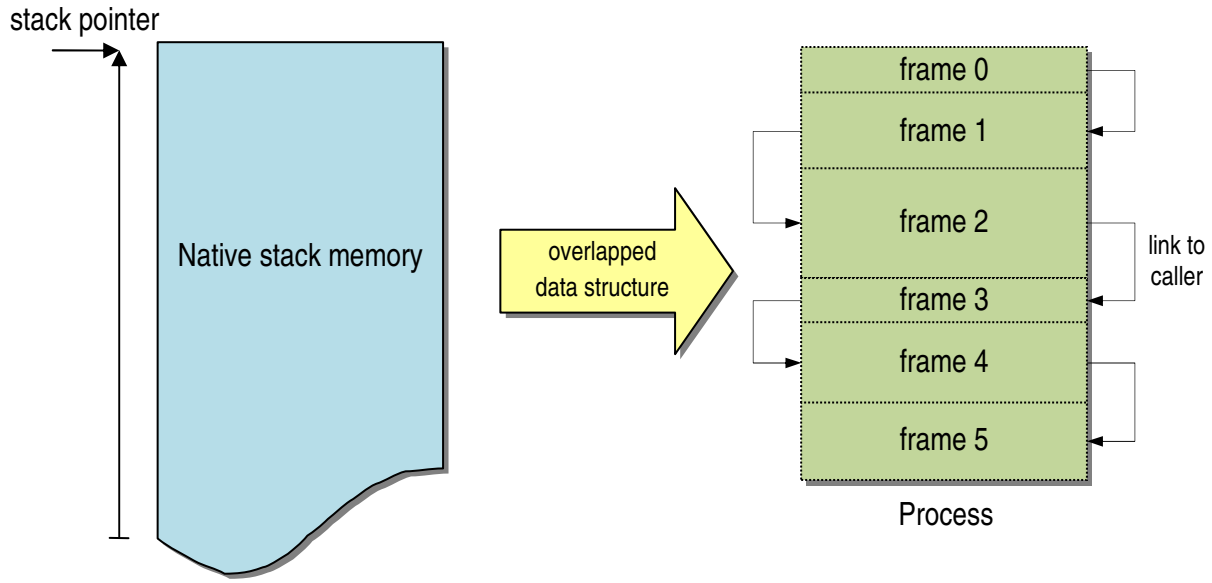
In this document we provide a detailed description of *Process Frames* in the form of a working reification of their features. Besides the documental value of our model, we show how to make use of it to implement *Process Continuations* entirely within the image. Even though our model has been implemented in *Visual Smalltalk*, we think that the concepts and techniques discussed here are applicable to a broader range of dialects.

The main reference of this document is [2]. This presentation assumes that the reader is familiar with the information discussed there.

The Execution Stack

The CPU execution stack embodies the current *Smalltalk* process, which can be accessed from the global CurrentProcess. In addition, primitive: 108 implements a service that, according to the official documentation, *answers a* Process *object containing the current stack contents*. This primitive is called from Process class >> #copyStack. Despite the copy prefix, the method gives access to the *current* process: the identity CurrentProcess == Process copyStack can be considered a system-wide invariant. More precisely, the primitive returns CurrentProcess after changing its contents field.

The class Process is a subclass of OrderedCollection that adds several instance variables described in subsequent sections. Entries (a.k.a. *slots*) of this collection are objects[1] consecutively allocated in the (native) stack. These slots are logically grouped in data structures known as (process) *frames*.



Except for some few special cases, process frames correspond to message sends, as suggested by the *Smalltalk* execution stack shown by the Debugger. Frames are activation contexts and most frames correspond to method or block activations.

For instance, frames hold the actual arguments of the activation, the receiver (self), and all involved temporaries. Frames associated with methods hold the corresponding CompiledMethod; those associated with blocks the associated BlockClosure instance.

As explained in [2], there are two kinds of temporaries. *Stack temporaries* are allocated as entries in the stack. *Environment temporaries* are local variables (method or block temporaries) referenced inside block closures. Given that a BlockClosure may survive the stack frame that

[1] Actually *object pointers* (a.k.a. oops). These are pointers to *Smalltalk* objects, with the exception of SmallIntegers which are codified inside their oop.

created it, these variables cannot be allocated in the stack. Therefore, they are stored in *Smalltalk* Arrays and are thus indirectly referenced by the nativized code.

Stack temporaries are not limited to those explicitly allocated by declaring them between pipes in the method (or block) header. There are also *implicit* stack temporaries created by the *Smalltalk* Compiler (actually the Encoder) to momentarily protect some objects from side effects of message sends (see [2] for more details on this).

Frames within the stack are simply linked: each frame has a slot that holds the number of bytes that separate it from its *calling* frame.[2] The calling frame is the preceding one in the stack. It corresponds to the method activation that sends the frame's message.

The link between frames and nativized code (i.e., machine code dynamically created by the *JIT* compiler) is given by the *return offset* slot. This is the number of bytes within the *native* calling method corresponding to the point where the current frame's method will return once it finishes. Note that the return offset is in correspondence with the value the native *program counter* will adopt when the current frame finishes. In other words, it is the offset of the instruction following the native *call* to the frame's nativized method.[3]

Method Frames

In a regular *Smalltalk* process most frames correspond to method activations. In this section we describe these frames and provide a conceptual model for them.

A naïve implementation of the class MethodFrame would have the following instance variables:

Instance variable	Description	Class
implicit	Implicitly allocated stack temporaries	Array
temporaries	Explicitly declared stack temporaries	Array
environment	Local variables accessed from within block closures (or nil)	Array
method	Method being activated	CompiledMethod
receiver	Message receiver (self)	Object
caller	Calling frame (or nil)	ProcessFrame
return	Offset (in bytes) of the return point within the nativized caller method	SmallInteger
arguments	Message arguments	Array
callee	Called frame (or nil)	ProcessFrame

The caller instance variable in our model corresponds to the Frame Pointer slot in [3]. Note moreover that, for the sake of convenience, the model includes a double link between frames so that every frame has direct access to both caller and callee.

[2] The native stack holds pointers from frame to frame. Primitive #copyStack converts these native pointers to distances from frame to frame. It also converts from native return addresses to return offsets.

[3] Note that when the CPU *calls* a routine it pushes the return *address* rather than its offset. Thus, the VM manages to present the VI information that is independent from actual memory locations.

The frame at the top of the stack will have nil in its callee field. Similarly, the last frame in the chain will have its caller set to nil. We will discuss these two special frames later in this document.

Example

Let's create a simple method:

```
Object >> #m
    $b between: $a and: $c
```

The mnemonic representation of which is:

```
Object>>m
    type: 0
    blocks: 0
    args: 0
    stk temps: 0
    literals:
        $a
        $c
        $b
        between:and:
(1)   1  <A3>    PushLiteral1
(2)   2  <A4>    PushLiteral2
(2)   3  <A0>    LoadLiteral3
(0)   4  <E2>    SendSelector1    #between:and:
(0)   5  <49>    ReturnSelf
```

As in [2] we have included the column on the left that shows the number of implicit temporaries allocated in the stack once the bytecode in the same line has been executed.

The message sent from this method corresponds to:

```
Character >> #between: min and: max
    ^(min asciiValue <= asciiInteger) and: [asciiInteger <= max asciiValue]
```

In turn, this method has the following mnemonic representation:

```
Character>>between:and:
    type: 0
    blocks: 0
    args: 2
    stk temps: 0
    literals:
    asciiValue
(0)  1  <50>  LoadArgument1
```

```
(0)  2  <E2>  SendSelector1    #asciiValue
(0)  3  <3B>  SendBinaryLessThanEqual    Inst,   1
(0)  5  <1C>  TestJumpFalse   15
(1)  8  <8C>  PushInstance1
(1)  9  <51>  LoadArgument2
(1) 10 <E2>  SendSelector1    #asciiValue
(1) 11 <09>  MoveRegToArg
(0) 12 <07>  PopR
(0) 13 <3B>  SendBinaryLessThanEqual    A-reg,1
(0) 15 <48>  Return
```

Assume we are in line 4 of the mnemonic representation of #m and in line 10 of #between:and:. In other words, assume we somehow manage to "take a photo" of the execution when #asciiValue is being sent for the second time.

Let's see what the method frame corresponding to this moment of the execution would be. The frame corresponding to the activation of Object >> #m is:

Instance variable	Contents	Description
implicit	#()	No implicit stack temporaries
temporaries	#()	No stack temporaries
environment	nil	No block closures in this method
method	Object >> #m	The method being executed
receiver	an Object	Message receiver
caller	don't care	Not relevant for this example
return	don't care	Not relevant for this example
arguments	#()	No arguments
callee	next frame	See below

Now let's see the callee frame as photographed at line 10. Since the first (and only) instance variable of the receiver $b is pushed on the stack in line 8, the frame object would look as follows:

Instance variable	Contents	Description
implicit	#(98)	Instance 1 of $b = 98
temporaries	#()	No stack temporaries
environment	nil	No block closures in this method[4]
method	Character >> #between:and:	The method being executed
receiver	$b	Message receiver
caller	prev frame	See above

[4] The argument of #and: does not generate a block closure because #and: is inlined (not *sent*.)

return	54	Computed as: (m's inlineTable at: 5)
arguments	#($a $c)	Message arguments
callee	don't care	Not relevant for this example

Note that the return offset can be computed as the entry at position 5 in the inlineTable of Object >> #m, where 5 is the bytecode index (a.k.a. *instruction counter*) immediately following the send bytecode at index 4.[5]

In a real execution these frames do not exist as *Smalltalk* objects. All that exists is the CPU execution stack and its registers. Our model represents the relevant data stored in the stack as first-class objects. The Debugger's approach is different: it deals with indexes and other conventions to directly access an amorphous array and make sense of the data structure embedded in the Process object.

Native Process Frames

Our model of method frames captures important bits of the *Smalltalk* execution architecture. In this section we will analyze the actual contents of the stack to prove that our model has all the information stored in the stack.

A careful study of the class Process reveals the underlying structure of the execution stack. Remember however that this structure is ultimately determined by the way the native code uses the stack by pushing and popping data. According to the *Smalltalk* calling convention, before sending a message the sender pushes the arguments on the stack and then loads register R with the receiver. At this point a native call to the nativized method is issued. Since native calls push the program counter of the instruction following them before jumping to the target address, we can see that above the arguments we will find the address to get back to the caller. The environment (if present) and stack temporaries are allocated after that and therefore they are located above the previous objects. Implicit temporaries (if any) are generated dynamically and should therefore be found at the top of the stack.

The common practice in assembler programming consists in using an auxiliary CPU register to reference any data stored in the stack. In the Intel 80386 processor family the stack pointer is held in ESP, while EBP is used to reference stack temporaries within every routine. Because of this, assembler routines usually begin with the following two instructions:

```
push ebp            ; save EBP on the stack
mov ebp, esp        ; copy ESP to EBP
```

At this point, stack temporaries can be accessed by indexing the stack with displacements relative to register EBP, using the constructor [ebp + *displacement*].[6] These two assembly

[5] As shown in [2] the inlineTable of a compiled method cm can be computed as CompilationInfo new compiledMethod: cm; inlineInfo, where CompilationInfo is the only subclass of CompilationResult (also reachable as the infoClass of the compiler.)

[6] Register EBP is restored back to its original value by adding the instruction pop ebp before the end of the routine.

instructions imply that immediately above the return address the stack will contain a pointer to the calling frame.

Summarizing all this information we should expect every frame to have a structure similar to the following:

Implicit variables
Stack temporaries
Environment array
Caller frame pointer
Return address
Arguments

Of course, we cannot assure that there is no other information in the stack. In fact, as we will see now, there is more. However, the purpose of this discussion is to help the reader make sense of the actual contents one finds when inspecting a Process. For now the reader should note the close correspondence between our summary list and the structure of the model we proposed for MethodFrame.

Copying the Stack

The simplest way to look at a real stack is to inspect Process copyStack. This primitive answers the CurrentProcess. Unfortunately, the inspector for instances of Process is the basic one, #basicInspect, and there is therefore no indication of the underlying structure of this sub-instance of OrderedCollection.

Nevertheless, a quick browse of its entries shows no indication of any memory addresses like the ones we predicted for the return address and the pointer to the calling frame. The reason is that absolute addresses are meaningless in a dynamic system where the machine code is created on the fly and all data (objects) is subject to unpredictable movements issued by the garbage collector.

Because of these reasons, the copyStack primitive replaces these absolute addresses with relative offsets before exposing them to the Virtual Image. In consequence, when inspecting a Process in *Smalltalk* what we see are offsets rather than memory addresses. This makes sense, because offsets do not depend on eventual relocations.[7]

Other than this (arithmetic) transformation, no additional "copy" operation is implemented by copyStack. Therefore, in order to have a frozen version of the CurrentProcess we need to explicitly copy it.

[7] The resume primitive transforms these offsets back to absolute addresses before giving the control back to the Process.

There is a point worth mentioning here: the default implementation of #shallowCopy inherited from IndexedCollection does not suffice because it only copies *indexed* entries but not the other named instance variables. Thus, we have to override #shallowCopy with:

```
Process >> #shallowCopy
    ^self objectShallowCopy postShallowCopy
```

where #objectShallowCopy is Object >> #shallowCopy and #postShallowCopy is:

```
Process >> #postShallowCopy
    contents := contents shallowCopy
```

Since Process >> #copy just sends #shallowCopy this is enough to have the #copy we were looking for.

Process Inspection

By inspecting (a copy of) Process copyStack we can take a look at the actual slots that make up every frame. Since frames are of different lengths, some guidance is required to identify them. The protocol of class Process has methods that might be useful, but the user will notice that the basic inspector is not of much help. Let's make use again of the following

Example

Let's go back to the case above. This time, however, we will introduce some code in order to capture the real process so we can compare our model of MethodFrame with real frames. Let's modify Character >> #asciiValue so that it inspects the current process. Since this method gets sent very often we must be careful not to crash the VM. Here is how:

```
Character >> #asciiValue
    (Flag isNil and: [self == $c]) ifTrue: [
        Flag := 0.
        Process copyStack copy inspect].
    ^asciiInteger
```

The global Flag ensures that the inspection does not happen every time the method is executed. Now we can safely evaluate:

```
Flag := nil. Object new m
```

and get an inspector open on the process.[8] Here is what we get:

Slots 1 & 2 are special and they hold information we will discuss later in this document:

Slot	Contents	Description
1	(Some) BlockClosure	Previous environment
2	134	Return offset

[8] Note that we have avoided using #halt to prevent side effects created by the Debugger.

Slots 3 to 6 make up the top frame:

Slot	Contents	Description
3	Character >> #asciiValue	The method being executed
4	$c	Message receiver
5	20	Caller frame offset
6	110	Return offset

Slots 7 to 13 make up the next frame:

Slot	Contents	Description
7	98	Implicit: Instance 1 of $b = 98
8	Character >> #between:and:	The method being executed
9	$b	Message receiver
10	24	Caller frame offset
11	54	Return offset
12	$c	Message argument 2
13	$a	Message argument 1

Slots 14 to 17 make up the next frame:

Slot	Contents	Description
14	Object >> #m	The method being executed
15	an Object	Message receiver
16	16	Caller frame offset
17	69	Return offset

The stack continues with more slots that correspond to frames previous to the evaluation of the expression Flag := nil . Object new m.

Since the caller frame offset is expressed in bytes, we have to divide it by 4 (in a 32-bit machine) to get the same offset expressed in slots. Thus, for instance, the offset 20 in slot 5 corresponds to 20 / 4 = 5 slots, and that's why the next frame offset is found at index 5 + 5 = 10. Note that if we send the message topFrame to the process we get 5, meaning that the first frame has its origin in slot 5. The caller frame offset is called the *frame pointer* (FP) in [3].

Block Frames

Block execution gives birth to *block* frames. These are similar to method frames, only differing in a few details.

As we have seen, when a method has block closures that use method temporaries, all these temporaries are stored in an Array called *environment*. A new method environment is generated every time the method is activated.

Method frames with blocks that do not access their temporaries still may have an environment, which in such cases is an empty Array. This happens with blocks that have explicit returns (^). The purpose of generating a new empty Array every time a method has a block with explicit return is to allow the block frame to find the frame where it has been (syntactically) defined. Since the explicit return must exit from the enclosing method, the corresponding frame must be found in the stack. The search proceeds by following the calling chain and comparing the environment of every frame with the environment stored in the block's fourth slot (see Object >> #methodEnvironment). If such a frame is not found, then the VM assumes that the method has already been exited and sends the Object >> #cantReturn message that generates the "familiar" error 'attempt to exit from the same method more than once'.

Our model for MethodFrames can be used to hold BlockFrames. In this case, however, the meaning of some instance variables changes accordingly:

Instance variable	Description	Class
implicit	Implicitly allocated stack temporaries within the block	Array
temporaries	Explicitly declared block stack temporaries	Array
environment	BlockClosure being activated	BlockClosure
method	The block's method (see BlockClosure >> #method)	CompiledMethod
receiver	Message receiver (self)	Object
caller	Calling frame (or nil)	ProcessFrame
return	Offset (in bytes) of the return point within the nativized caller	SmallInteger
arguments	Block arguments	Array
callee	Called frame (or nil)	ProcessFrame

Pseudo-code that finds the method frame would work like this:

BlockFrame >> #methodFrame

```
| frame env |
env:= environment methodEnvironment.
frame:= caller.
[
    frame isNil ifTrue: [self cantReturn].
    env == frame environment]
    whileFalse: [frame:= frame caller].
^frame
```

To see this error try the following code:

Object >> #m

```
| block |
block := [:arg |
    Process copyStack copy inspect.
     ^arg].
^block
```

If we now evaluate Object new m value: 1, we get the error and an inspector on the process that will show that the block environment is not in the caller chain.

The Top Frame

Instances of Process respond to the message topFrame with the index of their first frame pointer. There is nothing special about this first frame except that it has no callee. For our modeling purposes, however, this is a problem because we use the callee to determine the frame's instruction counter. Fortunately, instances of Process use the second of its entries to store the offset of the program counter within the method at the first frame, i.e., the method that was active at the time the process was photographed. Since this frame has not sent any message yet, there is no returning routine depending on it. However, we can use this value to compute the instruction counter (bytecode index) of the first frame using its method's inlineTable.

For the sake of uniformity, we will think of this entry as being called by the first frame. In doing this we can link it to the first frame using the caller and callee instance variables. Our model for this frame looks as follows:

Instance variable	Description	Class
caller	Calling frame (first frame in Process)	ProcessFrame
return	Program counter offset (in bytes) within the nativized caller (second entry in Process)	SmallInteger

Example

Let's consider the method:

```
Object >> #m
    Process copyStack.
    ^self
```

The mnemonic representation is:

```
Object>>m
    type: 0
    blocks: 0
    args: 0
    stk temps: 0
    literals:
        Process ==> Process
        copyStack
(0)   1  <5A>    LoadAssoc1
(0)   2  <E2>    SendSelector1    #copyStack
(0)   3  <0A>    LoadSelf
(0)   4  <48>    Return
```

By sending the #inlineTable message to this method we get #(34 39 44 46). Therefore, the third bytecode corresponds to the return offset 44. This means that the execution of Object new m at the point where copyStack is being sent would have the following top frame:

Instance variable	Contents	Description
caller	next frame	See below
return	44	As computed above

The calling frame corresponds to the activation of Object >> #m:

Instance variable	Contents	Description
implicit	#()	No implicit stack temporaries
temporaries	#()	No stack temporaries
environment	nil	No block closure in this method
method	Object >> #m	The method being executed
receiver	an Object	Message receiver
caller	don't care	Not relevant for this example
return	don't care	Not relevant for this example
arguments	#()	No message arguments
callee	top frame	See above

Launch Frame

Every process has a calling chain that ends up with an activation of SystemDictionary >> #launch.

SystemDictionary >> #launch

```
"Private - Launch the session."
ErrorCode := 0.
Launching ifFalse: [self recursiveMessage].
Launching := false.
SessionModel current startupSession
```

This frame is special in that it does not actually call its callee. Because of its special meaning, we can model this frame in a subclass of MethodFrame named LaunchFrame. Even though #launch does not have blocks or temporary variables, actual frames may have implicit temporaries and environments. A default LaunchFrame can be defined as follows:

Instance variable	Contents	Description
implicit	#()	No implicit temporaries
temporaries	#()	No stack temporaries
environment	nil	No block closure in this method
method	SystemDictionary >> #launch	This is always the case
receiver	Smalltalk	The global object

caller	nil	No caller, this is at the bottom of the stack
return	0	No caller, no return offset
arguments	#()	No message arguments
callee	unknown	Process-dependent

Instead of trying to deduce the actual extent of LaunchFrames (as we do with other types of frames), our model uses an additional instance variable named implicitCount. This variable is initialized to 0 when a LaunchFrame is created from scratch but is updated to its actual value when the frame is read from an instance of Process. In fact, that number can be calculated from the number of entries at the end of the stack that do not belong to any other frame.[9]

Interrupt Frames

When a process is interrupted (by any of the interrupts declared in the global InterruptSelectors), its normal flow is altered and the stack acquires a so-called *interrupt* frame that is different from regular ones. The content and shape of this frame depends on the source and type of the interruption. Therefore, as was the case with LaunchFrame, we have to keep some additional information to deduce the extent of the frame inside the stack. For this matter our model adds the instance variable framePointer. This variable is read when the frame is created from an instance of Process and holds the number of slots that separate it from its calling frame (see InterruptFrame >> #readCallerOffsetFrom: aProcess index: index in the next section).

Regardless the variety of interrupt frames, there is one particular kind that deserves attention. When the debugger executes *hop* and *skip* commands it asks the VM to generate a *debuggable* version of the compiled method by sending the message #asDebuggableMethod. As a consequence, the VM generates a #stepInterrupt that ends up being handled by the Debugger. Interrupt frames generated in these cases hold information about the message that was interrupted before it got sent. We can model this special type of frames by adding the following instance variables to our model:

Instance variable	Contents	Description
interruptSelf	an Object	The interruption receiver
nextSelector	a Symbol	Selector of the interrupted message
nextReceiver	an Object	Receiver of the interrupted message
nextArguments	an Array	Arguments of the interrupted message
framePointer	an Integer	Offset to the caller frame

The corresponding offsets in the Process frame are:

Instance variable	Offset
base	ArgumentOffset + self argumentCount - 1

[9] The method SystemDictionary >> #launch also appears in other frames of processes that have interrupts and callbacks. In those cases, however, these frames are not LaunchFrames.

interruptSelf	base + 1
nextSelector	base + 3
nextReceiver	base + 4
nextArguments	base + 6 to: base + 6 + nextSelector arity - 1
framePointer	0 (as usual)

Interruption frames not coming from the debugging mechanism have the same shape but different meaning since they hold different information.

The Process Frame Hierarchy

Process frames can be represented as instances of the following class hierarchy:[10]

ProcessFrame (return caller callee)
  SendFrame (implicit temporaries environment method receiver arguments)
    BlockFrame ()
    MethodFrame ()
      LaunchFrame (implicitCount)
    InterruptFrame (framePointer)
  TopFrame ()

The interesting thing about this hierarchy is that its classes can be instantiated from a Process and a frame index. In Appendix A we show how to create sub-instances of ProcessFrame.

Linking Frames

The model we have discussed so far corresponds to individual process frames. In order to represent processes rather than their frames we need a new class, namely ProcessStack. This class will have the responsibility to read every process frame into the appropriate sub-instance of ProcessFrame and link it to its caller and callee.

For this purpose we will ignore now the fact that Process defines several instance variables and will concentrate only in how to load and link frames. Therefore, we will start with a very simple implementation of ProcessStack that defines only two instance variables:

```
Object
    subclass: #ProcessStack
    instanceVariableNames: 'topFrame launchFrame'
    classVariableNames: ''
    poolDictionaries: ''
```

This way, when reading a ProcessStack from a Process, its first frame will be a TopFrame and its last one will be a LaunchFrame. Here is the code:

[10] *Italics* identify abstract classes.

ProcessStack >> #readFramesFrom: aProcess

```
| index class frame prev offset |
index := 1.
[
    class := ProcessFrame subclassFor: aProcess index: index.
    frame := class new.
    prev isNil ifTrue: [topFrame := frame] ifFalse: [prev caller: frame].
    frame fromProcess: aProcess index: index.
    offset := frame readCallerOffsetFrom: aProcess index: index.
    index := index + offset.
    prev := frame.
    offset > 0] whileTrue.
launchFrame := frame.
```

ProcessFrame class >> subclassFor: aProcess index: index

```
| method environment |
index = 1 ifTrue: [^TopFrame].
(aProcess at: index) = 0 ifTrue: [^LaunchFrame].
method := aProcess at: index + MethodOffset.
environment := method hasBlock ifTrue: [aProcess at: index + ContextOffset].
environment isBlockClosure ifTrue: [^BlockFrame].
(aProcess at: index + ReturnOffset) = 0 ifTrue: [^InterruptFrame].
^MethodFrame
```

Process-Named Variables

Frames are held in the indexed part of Process that is seen as an OrderedCollection. There are, moreover, several named instance variables that we must include in ProcessStack. What follows is a brief discussion of each of them.

name

Processes have names. The *User Interface Process*, which is treated specially by the system, receives the name 'User I/F'. The default name for a process is 'Background'.

priority

Every process has a priority represented by an integer between Processor idleTaskPriority (=1) and Processor topPriority (=7). The priority is an index to the corresponding queue held by the ProcessScheduler in its readyProcesses collection.

runable

This is an integer codifying several bit fields. Known values are:

value	meaning
0	not restartable, not resumable

1	restartable, not resumable
2	resumable — <primitive: 108> (#copyStack) sets runable to 2 in the CurrentProcess
4	?
5	resume with interrupts disabled
6	restart mode
16	resume with interrupt

isUserIF

This Boolean indicates whether the process is the *User Interface Process*.

debugger

If the process isBeingDebugged, this instance variable holds the running instance of Debugger; otherwise it is nil. When the debugger *hops* or *skips* the process is run and the VM interruption Process class >> #stepInterrupt is issued to give control back to the debugger.

interruptFrame

If this value is not nil then the process has been interrupted. In that case the value is the native stack pointer to the frame that was interrupted, divided by 2. The callee of the interrupted frame corresponds to what we have modeled as an InterruptFrame.[11] When the process has been interrupted the value of interruptFrame equals that of sendFrame (explained below).

exceptionEnvironment

If this value is not nil then the process is running inside some #on:do: block and this variable contains the ExceptionHandler.

terminationBlock

This is the block that gets evaluated when the process is terminated while it is active, i.e., equal to CurrentProcess (see Process >> #terminateWhileActive).

protectionBlock

If not nil, this block is the argument to an #ensure: message that must be evaluated because its receiver block will not return (i.e., the stack frame that evaluates it is about to be dropped). Before evaluating this block the frame's receiver (which had been set to the FrameMarker) must be replaced with the receiver of its calling frame (see the section about #ensure: later in this document and also Process >> #firstProtectionBlockWithin:removeMark:).

[11] To corroborate this information open a Debugger, inspect its debuggedProcess instance variable and verify that the interruptFrame ivar is not nil. In the inspector evaluate self stackPointerToProcessIndex: interruptFrame * 2. The value answered is the index within the process corresponding to the interrupted frame. The previous one (immediately above in the stack) is the interruption frame.

frameBias

This is an integer constant used by Process to compute indexes to frame pointers.

sendFrame

The debugger sets this variable with a value that depends on the command it is about to execute. Afterwards it resumes the debugged process. The VM uses the value set by the debugger to generate the interrupt that passes the control back to the debugger. Before that, the VM sets the variable to nil. The correspondence between the commands and the values of sendFrame is:

command	sendFrame
#hop:	=0
#skip:	=native stack address of current frame (this value is computed with: Process >> #processIndexToStackPointer:)
#jump	=16r20000000
#resume	=0 (debugger is set to nil)

More about ProcessStack

The complete definition of the class ProcessStack includes instance variables similar to the ones of Process.

```
Object
    subclass: #ProcessStack
    instanceVariableNames:
        'name priority runLevel ui debugger interruptFrame exceptionEnvironment terminationBlock
        protectionBlock frameBias sendFrame topFrame launchFrame'
    classVariableNames: ''
    poolDictionaries: ''
```

To mimic the initialization of instances of Process we define:

ProcessStack >> #initialize

```
    priority := CurrentProcess priority.
    ui := false.
    name := 'Background'.
    runLevel := 2.
    interruptFrame := 0.
    frameBias := 1.
    self initializeTerminationBlock; initializeLaunchFrame; initializeTopFrame.
    topFrame caller: launchFrame
```

ProcessStack >> #initializeTerminationBlock

```
    terminationBlock := [
        CurrentProcess isUserIF
            ifTrue: [
                CurrentProcess := Process new.
```

```
            CurrentProcess makeUserIF.
            SessionModel current isGui ifTrue: [SessionModel current runNotifier]]
        ifFalse: [Processor schedule]]
```

ProcessStack >> #initializeLaunchFrame

```
launchFrame := LaunchFrame new
```

ProcessStack >> #initializeTopFrame

```
topFrame := TopFrame new
```

ProcessFrame >> #initialize

```
return := 0
```

SendFrame >> #initialize

```
super initialize.
arguments := #().
temporaries := #().
```

LaunchFrame >> #initialize

```
super initialize.
method := Smalltalk methodFor: #launch.
receiver := Smalltalk.
arguments := #().
temporaries := #().
implicitCount := 0
```

TopFrame >> #initialize

```
super initialize.
return := 1
```

Creating a Process from a ProcessStack

The methods we have used to read Process frames into the appropriate sub-instances of ProcessFrame can be easily reversed to dump our model onto consecutive entries of a newly created Process. The only issue with this technique is that our model does not capture all the entries of the OrderedColletion. For instance, as have seen, the so called FramePointer is the offset in bytes to the caller frame. Our model does not save this value because offsets do not make sense in a linked list. Therefore, when the time comes to dump a ProcessFrame onto a Process our model must compute its FramePointer.

Another example is the entry at offset -3 within each frame. When the frame's method hasBlock (and the frame has a notNil environment) this entry contains the last notNil environment (if any) in the frame's caller chain. This slot corresponds to a native PUSH executed to save the value of the current environment in the stack. This means that the VM holds the current environment in a CPU register and saves it on the stack before loading the register with the new environment. This operation is only necessary if method hasBlock. We will call this value the previous environment or prevEnvironment.

In addition, the topmost notNil environment is preserved in the first Process entry (see Process >> #dropFrameWithoutProtection).

The case of InterruptFrames is also special because they always have prevEnvironment, which is found at offset 2.

Using all this information we can implement:

ProcessFrame >> #prevEnvironment

```
| prev env |
self hasPrevEnvironment ifFalse: [^nil].
prev := caller.
[(env := prev environment) isNil] whileTrue: [prev := prev caller].
^env
```

ProcessFrame >> #hasPrevEnvironment

```
^method hasBlock
```

TopFrame >> #hasPrevEnvironment

```
^true
```

InterruptFrame >> #hasPrevEnvironment

```
^true
```

The computation of the FramePointer is also easily derived from the information held by the frame and its caller.

SendFrame >> #framePointer

```
| end start |
end := self lastOffset.
start := caller firstOffset.
^end - start + 1
```

LaunchFrame >> #framePointer

```
^0
```

ProcessFrame >> #lastOffset

```
^ArgumentOffset + arguments size - 1
```

ProcessFrame >> #firstOffset

```
| right |
right := self tempOffset - self tempCount.
^right - self implicitTempCount + 1
```

Note that the methods above can be used to implement the following service:

ProcessFrame >> #processIndex

```
^callee processIndex + callee framePointer
```

TopFrame >> #processIndex

```
^1
```

The number of slots (size) of the Process can be determined as:

```
ProcessFrame >> slots
    ^self lastOffset - self firstOffset + 1
```

Useful Services

There are a number of useful services that are easier to implement in ProcessStack than in Process. Here are some examples:

Method for enumeration:

```
ProcessStack >> do: aBlock
    | frame |
    frame := topFrame.
    [frame notNil] whileTrue: [
        aBlock value: frame.
        frame := frame caller]
```

```
ProcessStack >> reverseDo: aBlock
    | frame |
    frame := launchFrame.
    [frame notNil] whileTrue: [
        aBlock value: frame.
        frame := frame callee]
```

```
ProcessStack >> detect: aBlock
    self do: [:frame | (aBlock value: frame) ifTrue: [^frame]].
    ^nil
```

Methods for converting:

```
ProcessStack >> asProcess[12]
    | process |
    process := Process new: self slots withAll: nil.
    self dumpOnProcess: process.
    ^process
```

Methods for flow control

```
ProcessFrame >> #send: selector to: anObject withArguments: anArray
    | frame |
    frame := MethodFrame new.
    frame
        receiver: anObject;
        method: (anObject methodFor: selector);
        returnOffset: callee returnOffset.
```

[12] The details of #dumpOnProcess: are not included but can be easily deduced from the information and code given in the section titled *Creating a Process from a ProcessStack*

```
    anArray withIndexDo: [:arg :i | frame argNumber: i put: arg].
    self topFrame returnOffset: frame initialInstructionCounter; caller: frame.
    frame caller: self.
    ^frame
```

ProcessFrame >> #send: selector to: anObject

```
    ^self send: selector to: anObject withArguments: #()
```

ProcessFrame >> #initialInstructionCounter[13]

```
    ^17
```

ProcessStack >> #resume

```
    Processor resume: self asProcess
```

Understanding BlockClosure >> #ensure:

The implementation of BlockClosure >> #ensure: is based on a specific VM manipulation of the execution stack.

Let's recall the implementation of this method:

BlockClosure >> #ensure: terminationBlock

```
    | result |
    result := self setUnwind: [:aContext :returnValue |
        terminationBlock value.
        aContext return: returnValue].
    terminationBlock value.
    ^result
```

BlockClosure >> #setUnwind: twoArgumentBlock

```
    ^self valueMarked
```

BlockClosure >> #valueMarked

```
    <primitive: 178>
    ^self primitiveFailed
```

The key of this implementation is inside primitive: 178. This service modifies the native stack by replacing the receiver slot in its calling frame with the global FrameMarker, which is the sole instance of ProtectedFrameMarker. The reader can verify this by implementing

ProtectedFrameMarker >> #isActivationVisibleForInstanceSelector: aSymbol

```
    ^true
```

and debugging a process that uses #ensure:.

If the block that receives #ensure: is curtailed or contains an explicit return, then before dropping the dismissed frames the native stack is searched for a frame with receiver == FrameMarker. If such a frame is found, then the receiver is replaced with the receiver of its caller and the argument block is evaluated. This is the twoArgumentBlock, apparently unused by #setUnwind:. This way the

---

[13] This *magic* constant can be found by trial and error, testing values from 1 to self inlineTable detect: [:pc | pc notNil].

evaluation of terminationBlock takes place. If the block receiving #ensure: had an explicit return, then the first argument of twoArgumentBlock is the receiving block and the second one, returnValue, its result. If, instead, the receiver raises an Error, the first argument is the terminationBlock defined in Process >> #safelyEvaluate: and the second one is a ProcessTermination signal.

A *Smalltalk* implementation of this mechanism can be found in the method Process >> #firstProtectionBlockWithin:removeMark: which is used by the Debugger when the user *restarts* the debugged process from some stack frame.

The Manfred Möbus Fix

A long standing bug in *Visual Smalltalk* was brilliantly fixed by Manfred Möbus, who published a patch in the VSE emailing list.

Manfred realized that when the Debugger replaces the method of the current frame with its *debuggable* version,[14] an inconsistency may be created in the callee chain. Such inconsistency happens when at least one of these frames references the regular method. The original version of Debugger >> #expandFrame: failed to search for such references and replace them with the debuggable version just compiled.

The equivalent of #expandFrame: in our model looks like:

```
SendFrame >> #beDebuggable
    | compact |
    self isDebuggable ifTrue: [^self].
    compact := method.
    method := compact asDebuggableMethod.
    callee callerExpanded: compact

ProcessFrame >> #callerExpanded: aCompiledMethod
    | expanded frame |
    expanded := caller method.
    return := self convert: aCompiledMethod to: expanded offset: return.
    frame:= self.
    ["Manfred's fix:"
        frame replaceCompiledMethod: aCompiledMethod with: expanded.
        frame := frame callee.
        frame notNil] whileTrue

ProcessFrame >> #convert: aCompiledMethod to: aDebugCompiledMethod offset: anInteger
    <primitive: 135>[15]
```

[14] By making a CompiledMethod *debuggable* the VI tells the *JIT* to nativize it differently, inserting additional (machine) code between nativized bytecodes that is able to generate interruptions depending on certain settings.

[15] This primitive acts as a mapping in that it transforms offsets within the nativization of the first argument into offsets within the nativization of the second, which is assumed to be the debuggable version of the first. The mapping happens because of the additional machine code inserted to the latter.

```
ProcessFrame >> #replaceCompiledMethod: aCompiledMethod with: anotherCompiledMethod
    "do nothing"

SendFrame >> #replaceCompiledMethod: aCompiledMethod with: anotherCompiledMethod
    super replaceCompiledMethod: aCompiledMethod with: anotherCompiledMethod.
    arguments
        select: [:arg | arg isBlockClosure and: [arg method == aCompiledMethod]]
        thenDo: [:block | block method: anotherCompiledMethod]

BlockFrame >> #replaceCompiledMethod: aCompiledMethod with: anotherCompiledMethod
    super replaceCompiledMethod: aCompiledMethod with: anotherCompiledMethod.
    environment method == aCompiledMethod
        ifTrue: [environment method: anotherCompiledMethod]
```

Within our model the fix consists in replacing references to the compact method in arguments and environments of frames in the callee chain with references to the debuggable one.[16]

The [firstTime] Trick

Some methods in Process and ProcessorScheduler have an obscure implementation that includes the block [firstTime], which looks quite useless. The pattern these methods use is on the lines of:

```
| firstTime oldProcess |
[firstTime]. "apparently useless block"
oldProcess := CurrentProcess.
CurrentProcess := Process new.
firstTime := true.
self copyStack.
firstTime
    ifTrue: [
        firstTime := false.
        self doThis]
    ifFalse: [self doThat]
```

Examples are ProcessorScheduler >> #fork:at:, ProcessorScheduler >> #suspendActive, Process >> #terminate, etc.

Note that #copyStack copies the current native stack onto (the newly created) CurrentProcesss. Therefore, when CurrentProcess resumes, it will start just after #copyStack.

The trick consists in that the inclusion of the block [firstTime] forces this variable to be saved in the environment. Therefore, the next time the process is resumed, firstTime will still have the very same value (finally) set by the original process. Since in our code snippet that value was false, the process will resume and follow the ifFalse: branch.

---

[16] Another fix implemented by Möbus consisted in expanding the caller of the protected frame, if any, that takes control when the user *hops* out of a block with explicit return (i.e., the user *hops* out of the receiver of an #ensure: message with ^). That fix, however, relies on Process >> #homeFrameOfClosure: which contains another issue because it compares the home it is searching for using #= rather than #==. Since contexts are Arrays, this method might answer with the wrong home frame and the crash would still happen. Therefore, the comparison selector must be fixed too.

Note moreover that the code in the ifTrue: branch schedules the newProcess and reestablishes the old one, giving the task-switch a chance to happen (e.g., by sending #yield).

doThis

```
Scheduler schedule: CurrentProcess.
newProcess := CurrentProcess.
CurrentProcess := oldProcess.
Processor yield.
^newProcess
```

The drawback of this trick is that newProcess inherits "garbage" from the old process. In fact, when that process is initialized by #copyStack it acquires the chain of calling frames present in the stack at that moment. That's why Process >> #evaluate: is forced to #dropSenderChain.[17]

It is interesting to note that this trick implements a continuation-like flow of control: the current process is copied and resumed later on at the same point where it had left, and with the same environment.[18]

From Blocks to Processes

In this section we simply use ProcessStack as a convenient intermediary to convert blocks into Processes.

BlockClosure >> #asProcess

```
^self asProcessStack asProcess
```

BlockClosure >> #asProcessStack

```
| process message |
process := ProcessStack new.
message := Message receiver: self selector: #value.
process launchFrame
    send: #evaluate:
    to: Processor
    withArguments: (Array with: message).
^process
```

BlockClosure >> #fork

```
^self asProcess fork
```

BlockClosure >> #forkAt: priority

```
| process |
process := self asProcess.
process priority: priority.
process fork
```

Process >> #fork

```
Processor forkProcess: self
```

---

[17] For a cleaner implementation of #fork see the section titled *From Blocks to Processes*.
[18] An implementation of *Continuations* is described below in this document.

ProcessScheduler >> #forkProcess: aProcess

```
(Process enableInterrupts: false) ifFalse: [
    Process enableInterrupts: true.
    ^self error: 'fork with interrupts disabled'].
self resume: aProcess
```

ProcessScheduler >> #evaluate: aMessage

```
Process enableInterrupts: false.
CurrentProcess basicEvaluate: aMessage.
self schedule
```

Process >> #basicEvaluate: aMessage[19]

```
self safelyEvaluate: aMessage.
self terminationBlock: [
    Notification signal: 'Process already terminated'.
    self terminateWithoutProtection ].
self runable: 0.
"Do not trace in the debugger past this point."
self class enableInterrupts: false.
self terminateWithoutProtection
```

Implementing Continuations

A *continuation* is an object that captures a process and implements a service that resumes that process at the point it had been held, making it return whatever value the client defines. Continuations are able to resume the process with any user-defined value, as many times as requested.

Our model for continuations is very simple:

```
Object
    subclass: #Continuation
    instanceVariableNames: 'process'
    classVariableNames: ''
    poolDictionaries: ''
```

Continuation class >> #currentDo: aBlock

```
| continuation |
continuation := self fromProcess: Process copyStack copy.
^aBlock value: continuation
```

Continuation class >> #fromProcess: aProcess

```
^self new process: aProcess
```

Continuation >> #process: aProcess

```
| process current sender |
process := aProcess asProcessStack.
current := process topFrame caller.
```

---

[19] This method is a fragment of Process >> #evaluate: and therefore can be used to factor the latter out.

```
    sender := current caller.
    sender send: #return: to: self
```

Continuation >> #return: anObject

```
    self ensureFullEpilogue.
    ^anObject
```

Continuation >> #ensureFullEpilogue

```
    "Nothing to do: any message send forces a full epilogue into its sender"
```

Continuation >> #value: anObject

```
    process topFrame caller argNumber: 1 put: anObject.
    process asProcess resume
```

A good source of examples comes from the *Squeak* unit tests [4]. There is a simple service that captures the CurrentProcess at specific points in the test and resumes it later to verify the behavior of the continuation. This is the service:

ContinuationTest >> #callcc: aBlock

```
    ^Continuation currentDo: aBlock
```

ContinuationTest >> #testSimpleCallCC

```
    | firstTime continuation |
    firstTime := self callcc: [:cc |
        continuation := cc.
        true].
    firstTime ifTrue: [continuation value: false].
    self deny: firstTime
```

In the test above, when #callcc: is sent, continuation is initialized and firstTime becomes true. The process captured by continuation is changed so that instead of evaluating the block, it sends the message #return: to itself. At that point no argument is specified. Then the continuation receives #value:. This message stores the argument, false in this case, into the argument slot of the #return: frame and resumes the process. In consequence, the process now assigns false to firstTime and continues the normal execution.[20]

It is interesting to note that continuations preserve environment temporaries. Here is an example:

ContinuationTest >> #testEnvironmentTemp

```
    | stack env firstTime continuation |
    stack := 1.
    firstTime := self callcc: [:cc |
        continuation := cc.
        env := 1.
        true].
    firstTime ifTrue: [
        stack := 2.
```

[20] These methods can be used as a guide to write an implementation directly using Process. The use of ProcessStack makes the main ideas easier to teach and learn, but this class is not essential for practical purposes.

```
        env := stack.
        continuation value: false].
    self assert: env = 2; assert: stack = 1
```

As we can see, when continuation is created stack == env == 1. Since firstTime == true, env and stack acquire the value 2. However, when continuation is resumed with value: false, firstTime is false and the ifTrue: branch does not take place. In particular, the re-assignation stack := 2 does not happen. Therefore, in the resumption, stack keeps its initial value 1, while env still has the value set in the previous process. This is similar to what regularly happens with environment variables, they preserve the value they acquired inside a block after the block execution finished. It is also similar to the [firstTime] trick we explained above.[21]

Appendix A – Creating Sub-instances of ProcessFrame

In what follows we show how to create sub-instances of ProcessFrame. We will assume that these classes use the pool dictionary StackOffsets.

Reading the return offset

```
ProcessFrame >> #readReturnOffsetFrom: aProcess index: index
    return := aProcess at: index + ReturnOffset

InterruptFrame >> #readReturnOffsetFrom: aProcess index: index
    return := aProcess at: index + ArgumentOffset + self argumentCount - 4

SendFrame >> #argumentCount
    ^method argumentCount

BlockFrame >> #argumentCount
    ^environment argumentCount
```

Reading the caller offset

```
ProcessFrame >> #readCallerOffsetFrom: aProcess index: index
    | bytes |
    bytes := aProcess at: index + 0.
    ^bytes // 4

InterruptFrame >> #readCallerOffsetFrom: aProcess index: index
    framePointer := super readCallerOffsetFrom: aProcess index: index.
    ^framePointer
```

[21] It is possible to implement another kind of continuations that *save* environment variables when the continuation is created and *reset* to these original values on every invocation of #value:. However, we don't see any application for such a feature.

When a process is being debugged the Debugger saves the real top frame index in its realFrame instance variable to hide the interruption mechanism used by the VM. Therefore, TopFrame has to override this method as follows:

```
TopFrame >> #readCallerOffsetFrom: aProcess index: index
    | real offset |
    real:= aProcess debugger ifNotNil: [:d | d realFrame].
    offset := real isNil ifTrue: [aProcess topFrame] ifFalse: [real].
    ^offset - 1
```

Reading the environment

```
SendFrame >> #readEnvironmentFrom: aProcess index: index
    method hasBlock ifFalse: [^self].
    environment := aProcess at: index + ContextOffset
```

Reading the method

```
SendFrame >> #readMethodFrom: aProcess index: index
    method := aProcess at: index + MethodOffset
```

Reading the receiver

```
SendFrame >> #readReceiverFrom: aProcess index: index
    receiver := aProcess at: index + ReceiverOffset
```

Reading the arguments

```
SendFrame >> #readArgumentsFrom: aProcess index: index
    | interval |
    interval := ArgumentOffset to: ArgumentOffset + self argumentCount - 1.
    arguments := interval collect: [:offset | aProcess at: index + offset]
```

Reading stack temporaries

```
SendFrame >> #readTemporariesFrom: aProcess index: index
    | right interval |
    right := self tempOffset.
    interval := right - self tempCount + 1 to: right.
    temporaries := interval collect: [:offset | aProcess at: index + offset]

SendFrame >> #tempOffset
    | offset |
    offset := TemporaryOffset.
    method hasBlock ifTrue: [offset := offset - 2]. "2 = home + context"
    ^offset - 1
```

SendFrame >> #tempCount

```
^method tempCount
```

BlockFrame >> #tempCount[22]

```
^environment tempCount
```

Reading implicit stack temporaries

SendFrame >> #readImplicitTemporariesFrom: aProcess index: index

```
| right interval |
right := self tempOffset - self tempCount.
interval := right - self implicitTempCount + 1 to: right.
implicit := interval collect: [:offset | aProcess at: index + offset]
```

SendFrame >> #implicitTempCount[23]

```
^self bytecodeReader stackDepthAt: self instructionCounter
```

LaunchFrame >> #implicitTempCount

```
^implicitCount
```

LaunchFrame >> #readImplicitTemporariesFrom: aProcess index: index

```
implicitCount := index - callee lastProcessIndex - 3 - self tempCount.
super readImplicitTemporariesFrom: aProcess index: index
```

ProcessFrame >> #lastProcessIndex[24]

```
^self processIndex + ArgumentOffset + self argumentCount - 1
```

Appendix B – Additional Methods

Methods to compute the number of block stack temporaries

BlockClosure >> #tempCount

```
| index reader |
index := self templateIndex ifNil: [^nil].
reader := ByteCodeReader on: self method.
^reader tempCountOfBlock: index
```

BlockClosure >> #templateIndex

```
| method n index |
method := self method.
n := self blockNumber + 1.
index := 0.
^method literals
    findFirst: [:literal |
        literal isBlockClosure
```

---

[22] See the implementation of BlockClosure >> #tempCount in Appendix B.
[23] See the details of this method in Appendix B.
[24] See the implementation of #processIndex below in this document.

```
            and: [literal method isCompiledMethod not]
            and: [(index := index + 1) = n]]
```

ByteCodeReader >> #tempCountOfBlock: blockIndex

```
    "See ByteCodeReader >> #loadBlockBC:on:"[25]
    | bytecodes stream c idx |
    bytecodes := self bcOfBlock: blockIndex.
    stream := bytecodes readStream.
    stream skip: 1.
    c := stream next.
    idx := self indexFromStream: stream.
    idx := stream next + (stream next * 256).
    [stream atEnd] whileFalse: [
        c := stream next.
        idx := self indexFromStream: stream.
        c = 0 ifFalse: [^idx]].
    ^0
```

ByteCodeReader >> #bcOfBlock: blockIndex

```
    | patterns bc |
    patterns := self class startBlockPatterns.
    self reset.
    [
        bc := self scanPatterns: patterns.
        bc notNil and: [(self blockIndexOf: bc) = blockIndex]] whileFalse.
    ^bc
```

ByteCodeReader >> #blockIndexOf: bytecode

```
    | idx |
    idx := bytecode third.
    idx >= 128
        ifTrue: [idx := idx - 128 + (128 * (bytecode fourth bitShift: 7))].
    ^idx
```

Methods to compute the number of implicit stack temporaries

SendFrame >> #implicitTempCount

```
    callee isNil ifTrue: [^0].
    ^self bytecodeReader stackDepthAt: self instructionCounter
```

LaunchFrame >> #implicitTempCount

```
    ^implicitCount
```

SendFrame >> #instructionCounter

```
    | table index offset first |
    table := self inlineTable.[26]
```

---

[25] This is the method that displays block bytecodes in their mnemonic form. Experience shows that a good way to understand the structure of bytecodes is to debug how the ByteCodeReader displays them in mnemonic form. The implementation of #tempCountOfBlock: closely mimics that of #loadBlockBC:on:.

```
    offset := callee returnOffset.[27]
    first := table findFirst: [:pc | pc notNil].
    index := table
        findFirst: [:pc | pc notNil and: [offset <= pc]]
        ifAbsent: [^0].
    index = first ifTrue: [^0].
    [
        index := index - 1.
        index = 0 ifTrue: [self error: 'Invalid bytecode index'].
        (table at: index) isNil] whileTrue.
    ^index
```

Methods to compute depth stack changes[28]

```
ByteCodeReader
    subclass: #StackTracer
    instanceVariableNames: 'mapping currentDepth blocks'
    classVariableNames: ''
    poolDictionaries:  'XCByteCodes'
```

StackTracer >> #nextByteCode

```
    | bc delta idx n interval |
    bc := super nextByteCode.
    bc isNil ifTrue: [^nil].
    delta := self depthChangeFor: bc.
    self currentDepth: self currentDepth + delta.
    mapping at: currentIndex put: self currentDepth.
    (self beginsBlockClosure: bc) ifTrue: [
        idx := self blockIndexOf: bc.
        n := self blockNumberAt: idx.
        interval := currentIndex to: (self blockEndIndexFor: bc).
        blocks at: n put: interval.
        currentDepth push: 0].
    (self endsBlockClosure: bc) ifTrue: [currentDepth pop].
    ^bc
```

StackTracer >> #depthChangeFor: bc

```
    | op |
    op := bc first.
    (op = LoadBlockContextN and: [bc second = 3]) ifTrue: [^1].
    (op between: PushArgument1 and: PushArgumentN) ifTrue: [^1].
    (op between: PushAssoc1 and: PushAssocN) ifTrue: [^1].
    (op between: PushContextTemporary1 and: PushContextTemporaryN) ifTrue: [^1].
```

---

[26] See [2] for further details on this method.
[27] This is the value of the instance variable return.
[28] The instance variable currentDepth is implemented as a Stack. This allows the current stack depth to be saved and restarted every time a block starts (push: 0) and recovered when a block exits (pop). See [2] for further details.

```
    op = IndirectEscape ifTrue: [^self depthChangeForIndirect: bc].
    op = PushFalse ifTrue: [^1].
    (op between: PushInstance1 and: PushInstanceN) ifTrue: [^1].
    op = PushSmallInteger0 ifTrue: [^1].
    op = PushSmallInteger1 ifTrue: [^1].
    op = PushSmallInteger2 ifTrue: [^1].
    op = PushSmallInteger ifTrue: [^1].
    (op between: PushLiteral1 and: PushLiteralN) ifTrue: [^1].
    op = PushNil ifTrue: [^1].
    op = PushR ifTrue: [^1].
    op = PushSelf ifTrue: [^1].
    (op between: PushTemporary1 and: PushTemporaryN) ifTrue: [^1].
    op = PushTrue ifTrue: [^1].
    op = DropTos1 ifTrue: [^-1].
    op = DropTos2 ifTrue: [^-2].
    op = DropTosN ifTrue: [^bc second negated].
    op = PopR ifTrue: [^-1].
    (op between: SendSelector1 and: SendSelectorN - 1)
        ifTrue: [^(self literal: op - SendSelector1 + 1) arity negated].
    op = SendSelectorN ifTrue: [^(self literal: bc second) arity negated].
    (op between: SendSpecial1 and: SendSpecial22)
        ifTrue: [^(self specialSelectors keyAtValue: op - SendSpecial1 + 1) arity negated].
    op = SendSuper1 ifTrue: [^(self literal: 1) arity negated].
    op = SendSuperN ifTrue: [^(self literal: bc second) arity negated].
    op = SendSuperSpecialN
        ifTrue: [^(self specialSelectors keyAtValue: bc second) arity negated].
    ^0
```

StackTracer >> #depthChangeForIndirect: bc[29]

```
    | stream |
    stream := '' writeStream.
    self indirectEscape: bc on: stream.
    ^(stream contents beginsWith: 'Push') ifTrue: [1] ifFalse: [0]
```

StackTracer >> #literal: index

```
    ^method at: method size - index + 1
```

StackTracer >> #currentDepth

```
    ^currentDepth top
```

StackTracer >> #currentDepth: anInteger

```
    currentDepth top: anInteger
```

ByteCodeReader >> #blockEndIndexFor: bytecode

```
    | stream c dummy r |
    stream := ReadStream on: bytecode.
```

[29] This implementation is admittedly inelegant.

```
    stream skip: 1.
    c := stream next.
    dummy := self indexFromStream: stream.
    r := stream next.
    r < 128 ifTrue: [^r - 1].
    ^stream next * 128 + (r - 128) - 1
```

ByteCodeReader >> #beginsBlockClosure: bytecode

```
    ^self class startBlockPatterns includes: bytecode first
```

ByteCodeReader >> #endsBlockClosure: bytecode

```
    ^(self class endBlockPatterns includes: bytecode first) and: [
        blocks
            anySatisfy: [:interval | interval notNil
                and: [interval last = currentIndex]]]
```

References

[1] SEASIDE – http://www.seaside.st.

[2] DIGITALK BYTECODES – L. Caniglia, V. Murgia & G. Richarte. December 13, 2008.

[3] A SMALLTALK VIRTUAL MACHINE ARCHITECTURE MODEL – Allen Wirfs-Brock, Paul Caudill. 1999.

[4] SQUEAK – http://www.squeak.org