The Beagle Smalltalk Virtual Machine

# About this document

The Beagle Smalltalk Virtual Machine is the engine that drives Smalltalk. For most users of Beagle Smalltalk, you don’t need to understand how the VM works in order to use Smalltalk. This document is provided for people who want to learn more about the inner workings of Smalltalk and who want to be able to make changes to it.

Fully understanding this document requires knowledge of the C programming language.

# Introduction

The Beagle Smalltalk virtual machine is a pure interpreter of Beagle Smalltalk bytecodes written in C. It’s a 64-bit application which uses 64 bit object pointers. The VM consists of several components.

* The interpreter (interpret.c)
* Memory allocation and garbage collection (memory.c)
* Image loading (image.c)
* Primitives (primitives.c, integer\_primitives.c, float\_primitives.c, file\_primitives.c, socket\_primitives.c)
* WebSockets (websockets.c, websockets.js.c, websockets.js.h)
* Miscellaneous utilities (utility.c)
* Remote debugging tool (remote.c)
* Main program (WinMain.c)

# Compiling the virtual machine

To compile the VM in a Windows environment, you will need Cygwin with gcc and make installed. In the folder containing the image, simply type ‘make’.

# Object structure

All object pointers are 64 bits and point to an address that’s 8-byte aligned. This means that the bottom three bits are always zero. We take advantage of that fact by using those three bits as a tag. Using tags, we can embed some objects directly in the pointer rather than requiring space to be allocated for them.

The tags supported by Beagle Smalltalk are:

2r000 - Object pointer  
 2r001 - SmallInteger  
 2r010 - Character  
 2r011 - Float

Tag 2r000 is just a pointer to the header of an object.

Tag 2r001 means that the rest of the 61 bits are a signed integer.

61 bit signed integer

001 signed integer

Tag 2r010 means that the rest of the 61 bts are a Unicode character.

61 bit Unicode character

010 signed integer

Tag 2r011 means that the remaining 61 bits are a double precision IEEE float with the exponent reduced in size by 3 bits.

8 bit exponent

011 signed integer

52 bit mantissa

1 bit sign

This means that the range of floats is 2+/-127 or 10+/-38 rather than 2+/-1023 or 10+/-307.

# Memory structure

Memory is organized into spaces. Each space has a header with the following structure.

spaceSize

lastFreeBlock

firstFreeBlock

spaceType

spaceNumber

spaceFlags

remembered  
set space  
number

The space types are:

EDEN\_SPACE 0

SURVIVOR\_SPACE\_1 1

SURVIVOR\_SPACE\_2 2

REMEMBERED\_SET 3

WELL\_KNOWN\_OBJECTS\_SPACE 4

OLD\_SPACE 5

STACK\_SPACE 6

Every space has a space number. This is an integer (currently between 0 and 255) indicating the index into the Spaces[] global variable that points to this space.

The spaceFlags field is a bit field indicating the properties of this memory space.

SPACE\_HAS\_TOP\_HEADERS 1

SPACE\_IS\_OBJECT\_SPACE 2

SPACE\_IS\_POINTER\_SPACE 4

SPACE\_IS\_SCAVENGED 8

SPACE\_IS\_STACK\_MANAGED 16

SPACE\_IS\_MARK\_SWEEP\_MANAGED 32

SPACE\_HAS\_SPACE\_OBJECT 64

SPACE\_IS\_CURRENT\_SPACE 128

This header represents a memory space that looks like this.

headers

bodies

free space

spaceSize (in bytes)

firstFreeBlock (index of oops)

lastFreeBlock (index of oops)

The exception to structure is for stack space. Stack bodies are at the start of the space and headers are at the end. This allows us to grow the stack by increasing the size of the latest stack frame by simply extending the body further into the free space.

Enhancement note:

It might actually be wise to convert all spaces to have headers in high memory and bodies in low memory. This approach has several advantages. It makes it possible to grow objects (if they are the last objects allocated in a space) by adding to the end of the body. It also means that the first body in the space could be the space description itself. The header would be the first header at the top of the space. The object would be a byte object where the bytes represent the rest of the space. In this way, spaces become first class objects. It would take many tricky modifications to the VM and the image to make this happen and the benefits are minor, so I haven’t tried.

# Objects

Objects are allocated with a header and a separate body. The format of the header is:

size

flags

flips

number of named inst vars

class

identity hash

body pointer

Here’s the dump of an object header (an array of size 6):

6ffffffb5da0: 58 00 00 00 00 00 00 00

Size – least significant byte first

The header is 0x28 bytes and the body is 0x30 bytes or 48 bytes

For a total of 0x58 bytes

6ffffffb5da8: 02 00 00 00 00 00 00 00

flags: 02 00 (indexed)  
 flips: 00 00

number of named inst vars 00 00 00 00

6ffffffb5db0: a8 4b af fe ff 6f 00 00

class pointer

6ffffffb5db8: 30 b2 60 41 00 00 00 00

identity hash

6ffffffb5dc0: 30 b9 fe ff ff 6f 00 00

body pointer

The flags are:

BYTES 1

INDEXED 2

RELOCATED 4

FREE 8

MARK 16

QUEUED\_FOR\_MARK 32

SPACE\_OBJECT 64

VM\_MIGRATION\_NEW 128

For objects that have no instance variables, the body pointer is 0. Otherwise, the body pointer points to the start of the body. After every body (except in Stack space), we reserve one additional slot to point back to the header. This back pointer allows us to easily implement a compacting global garbage collector.

header

body

# The Interpreter

The interpreter (in interpret.c) is responsible for executing the bytecodes in the compiled methods and blocks. It has functions to lookup methods in method dictionaries and to invoke them. It can perform some simple actions inlined (such as SmallInteger arithmetic) and it can invoke primitives.

There are hooks for implementing a Polymorphic Inline Cache (PIC) but this is currently not implemented.

Finally, interpret.c contains code to dump a Smalltalk walkback when the image crashes.

# Primitives

Primitives are operations written in C which are callable from Smalltalk. Beagle Smalltalk has a table of size 2048 containing function pointers to the C functions to implement each primitive. Within the primitive function, the following operations are available:

getReceiver() - fetch the object pointer to the receiver of the message

getLocal(x) - fetch local number x (arguments are locals starting at index 0)

Primitive functions always push two values onto the stack. If the primitive succeeds, it pushes 0 followed by the return object of the primitive. If the primitive fails, it pushes a non-zero fail code (usually 1) and an object.

If you're coding primitives, you need to be careful about object identity. Any operation which allocates objects could trigger a scavenging garbage collection which could move objects around in memory. If you have a pointer to an object in a C variable, that pointer could be invalid after a garbage collection.

To avoid this problem, you should push the object pointers onto the stack (which is updated during a garbage collect) and always fetch object pointers from the stack instead of from C variables. To make this simpler, we provide a series of macros for defining local stack variables.

DEFINE\_LOCALS; // Start a section that has local variables

DEFINE\_LOCAL(x); // Define a local variable we'll call x

SET\_LOCAL(x, getReceiver()); // Store an object into the local variable x

oop result = GET\_LOCAL(x); // Fetch the value of the local variable x

FREE\_LOCALS; // Get rid of all local variables

If you always use these macros, you won't have problems with C variables being invalidated during a garbage collection.

# Garbage Collection

Beagle Smalltalk implements two forms of garbage collection: a generation scavenger and a compacting mark and sweep.

## Scavenger

The generation scavenger uses four spaces called Eden, SurvivorSpace1, SurvivorSpace2, and the RememberedSet.

Eden

SurvivorSpace1

SurvivorSpace2

RememberedSet

Only one of the survivor spaces is active at any one time. The other survivor space is empty. The RememberedSet contains pointers to objects in any other space that contain pointers to objects in Eden or the active survivor space.

Objects are allocated in Eden by default. When Eden begins to get full, it triggers a scavenge to copy all active object in Eden and the active survivor space into the inactive survivor space and flips the inactive survivor space to now be active.

The algorithm goes something like this.

Foreach object pointer in the remembered set

Foreach pointer from that object into Eden or the active space

If the new space object is not already copied to the inactive space

Copy the object to the inactive space

Write a forwarding pointer into the old object

Endif

Update the old space object to point to the new object instead

End foreach

End foreach

Make the inactive survivor space the active one and vice versa

## Compacting Mark and Sweep

Mark and Sweep starts from known fixed "root" objects and traces through everything that's accessible from that object. Anything not traced is considered garbage. To eliminate the garbage, we compact the space by moving objects and object bodies around in the space to eliminate any holes.

# VM Debugging

A special primitive called primitiveHalt will halt the execution of Smalltalk bytecodes and wait for a web socket connection to a debugging web page. You can open the debugging web page with BeagleDebugUI.html. From there, you can perform debugging on the image while it's frozen. You can look around at the stack, memory spaces, and objects in the image.

A screenshot of a computer

AI-generated content may be incorrect.

The debugging VM has the following buttons.

Spaces - Show all memory spaces and their sizes with memory addresses

Stack - Show a low-level view of the stack including memory spaces

Type a number into the top text editor to select which stack frame to use

0 is the current frame. 1 is the next caller, etc.

Show - Type a hex address into the top text editor and hit Show to show an object

Dump - Type a hex address hit Dump to get a raw memory dump

Over/Into/Peek - This allows you to single step through bytecodes

Run - Close the debugging UI and continue normal Smalltalk operation