LittleSmalltalk – The Byte Code

Author: Danny Reinhold
Version: 0.1
Status: Published
Date: April, 3nd, 2007

Summary:
This document explains the byte code that is used within LittleSmalltalk. For the typical Smalltalk developer this is probably not very interesting. But if you intend to write low level tools or to modify the virtual machine etc. you should know these things.

What are Byte Codes?

The LittleSmalltalk system consists of two main parts: The Smalltalk image that is loaded at program startup and the virtual machie (VM) that executes Smalltalk code and manages the object memory etc.
Interpreting the Smalltalk source code during execution would be very, very slow and would cause the VM to be really complex. So to speed up the execution the source code is first compiled into an intermediate format that is pretty simple (for a computer) to understand and to execute.
This indermediate code format is the so called Byte Code.

If you know other languages such as Java or frameworks like .NET this concept is not new to you. Those systems also work with byte code representations.
But all those forms of byte code tend to be completely incompatible. You cannot run Java Byte Code within the .NET framework directly and you cannot run .NET byte code within the LittleSmalltalk VM etc.

The abstraction level of LittleSmalltalk's byte code is roughly comparable to machine language for an abstract computer system.

How to see the byte code of a method?

LittleSmalltalk can display the byte code of a method in the Transcript window.
If you want to see the byte code of the method Class>>listMethods you can simply write:

```
Inspector new showByteCodes: (Class methods at: #listMethods)
```

The class `Inspector` allows you to look into methods a bit deeper.

You will see something like:

```
18
192
8
64
48
130
145
242
```

```
130
146
245
67
242
245
241
0
0
0
0
0
```

Not really informative you may think.
Right, so let's go a step further…

Disassembling byte codes

The Inspector cannot only show a list of numeric byte codes but also disassemble a sequence of byte codes.
So for example type:

```
Inspector new disassemble: (Class methods at: #listMethods)
```

Then you will see something like this:

```
Disassembly of method: #listMethods
18          1    2  PushInstance  instanceVariable: 2
192        12    0  PushBlock
8           0    8  Extended
64          4    0  PushLiteral
48          3    0  PushTemporary
130         8    2  MarkArguments
145         9    1  SendMessage
242        15    2  DoSpecial
130         8    2  MarkArguments
146         9    2  SendMessage
245        15    5  DoSpecial
67          4    3  PushLiteral
242        15    2  DoSpecial
245        15    5  DoSpecial
241        15    1  DoSpecial
0           0    0  Extended
0           0    0  Extended
0           0    0  Extended
0           0    0  Extended
```

This looks more interesting, doesn't it?
We will look deeper into this soon.

For know we should notice that a method is compiled into a byte code form. The byte code representation of a method is an array of 8 bit numbers, the byte codes.

The disassembly show us that each 8 bit byte code consists in fact of two 4 bit sub codes.
In the disassembly you see the 8 bit byte code in the first column, the higher 4 bits of the byte code in the second column, the value of the lower 4 bits of the byte code in the third column and finally a descriptive text (we take this as the disassembled form of the byte codes) in the fourth column.

You will always find: `byte code = high * 16 + low` where high and low are the values of the higher nibble (4 bits) or the lower nibble respectively.

How to read the disassembly?

Take the above disassembled version of Class>>listMethods and look into the source code of that method:

```
listMethods
  methods binaryDo:
    [ :nameStr :meth | Transcript print: nameStr ].
  ^ 'end of list'
```

The first action in this method is an access to the instance variable `methods`.
And the first line in the disassembly is:

```
18          1    2  PushInstance  instanceVariable: 2
```

The high code `1` causes the virtual machine to push an instance variable to the stack. The low code `2` specifies which instance variable to push, which is the third instance variable (counting starts at 0). Look into the class browser and verify that `method` indeed is the third instance variable of all classes.

So reading the assembly is essentially simple.

Please note that in some cases you only see the disassembly ‚Extended'. This only means that the disassembler is not really complete and cannot show some useful description of every possible byte code. This will be extended and improved in later LST versions.
In fact there is a special logic with those extended byte codes:
While usually the high code is the instruction code and the low code is an argument, the logic is different if the high code is 0.
Then the low code is the instruction code and the next byte contains the argument.
This way the system can work with arguments that are larger than 4 bit (in fact such an argument is an 8 bit value).
This is currently not correctly supported by the disassembler.

The execution model

The virtual machine basically emulates an abstract computer system. The most important parts of this system are the execution engine (the abstract CPU) and the object management system (basically the garbage collector).
The object memory is described in another document. So lets concentrate on the execution engine.

Execution always takes place within a `Process` object. This is not neccessarily a separated OS level process or thread, but an object of class `Process`. A process can create and execute other processes. The outer `Process` object is created by the VM itself.

When executing code this always happens within a specific `Context`. `Context` also is a Smalltalk class that describes which method with what arguments etc. is to be executed, what are the values of the temporary variables etc.
A context also provides a stack that is highly important during the evaluation of Smalltalk methods.

So in fact you don't execute methods directly. Instead you specify a context and tells that context to perform a specific method with specific arguments etc.

A highly important instance variable of class Context is `bytePointer`. The byte pointer in an execution context always points to the next byte code that will be executed.
Byte Codes are executed sequentially. So byte codes of a method are always represented as an array of byte codes that are executed one by one. Some byte codes will cause the interpreter to skip several byte codes or to jump to another location and for example continue the byte codes of another method. When starting the execution of a context, the bytePointer is set to the start of the method's byte code array.
In fact bytePointer is not a real memory pointer, instead it is an index within the byte code area of the context's method.

So we have all typical concepts of CPUs: A stack, a programm counter (the bytePointer) and registers (temporary variables).

The Byte Codes in detail

The following table lists all byte codes known by the LittleSmalltalk system.

High	Low	Name	Description
1	N	PushInstance	Push instance variable N to the stack
2	N	PushArgument	Push argument N to the stack
3	N	PushTemporary	Push temporary variable N to the stack
4	N	PushLiteral	Push literal N of the method to the stack
5	N	PushConstant	Push constant N to the stack. N must be one of 0,1,2,3,4,5,6,7,8,9,nil,true,false
6	N	AssignInstance	Copy the top of stack (TOS) to instance variable N. Leave TOS unchanged

7	N	AssignTemporary	Copy the TOS to temporary variable N. Leave TOS unchanged
8	N	MarkArguments	
9	N	SendMessage	
10	N	SendUnary	
11	N	SendBinary	
12	N	PushBlock	Push the block to the stack. N is the argument location. **Attention:** This opcode is followed by a byte that specifies the value of bytePointer after executing this byte code (this is a form of goto to skip the block's byte codes)
13	N	DoPrimitive	Call a primitive. N is the number of arguments to the primitive. The next byte after the byte code is the number of the primitive. The arguments are read from the stack.
15	N	DoSpecial	Perform special operation N as follows:
15	1	SelfReturn	
15	2	StackReturn	
15	3	BlockReturn	
15	4	Duplicate	
15	5	PopTop	
15	6	Branch	
15	7	BranchIfTrue	
15	8	BranchIfFalse	
15	9	SendToSuper	

Extended Byte Codes

As said earlier LittleSmalltalk knows extended byte codes. Use this mechanism if the low code takes more than 4 bits (but not more than 8 bits).
You can specify such an extended byte code by setting high code to 0 and low code to the opcode you whish to execute. Then LittleSmalltalk interpretes high code as the opcode and reads the low code from the next execution byte.
For example if you want to push instance variable 223 the code looks like:

```
1 123
```

While pushing instance variable 5 simply looks like:
`21` (= 1 * 16 + 5)