1984

Smalltalk-80 virtual machine

Ashraful Huq

Rochester Institute of Technology
School of Computer Science and Technology

SMALLTALK-80
VIRTUAL MACHINE

by

Ashraful Huq

Thesis submitted to the Faculty of the
School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED :

Guy Johnson

Professor Guy Johnson

Lawrence Coon

Professor Lawrence Coon

Warren R. Carithers

Professor Warren R. Carithers

September 1984
Rochester, New York

ACKNOWLEDGEMENTS

I dedicate this thesis to my parents. The writing and programming of this thesis has been an arduous task. My mother and my late father, gave me patient understanding and love without which this work could not have been completed. Both my brothers have spoken words of encouragement to me all along.

I wish to thank Professor Guy Johnson for the encouragement and guidance he has given me. I am very grateful to him for the endless hours he spent directing my effort on this project. I am indebted to Professor Lawrence Coon for giving me the main idea for the thesis. I also wish to thank Professor Warren R. Carithers for his cooperation and for time spent on my committee. I am grateful to both of them for their diligent patience and support.

I would also like to thank Professor John McCarthy for his invaluable help and encouragement to come to the United States for my degree. I also owe Dr. R. Giles special thanks for his support and help.

Lastly, my sincere thanks to all my friends and fellow students for their support and friendship.

1.0 INTRODUCTION AND BACKGROUND

Smalltalk-80 is an interactive, object oriented programming language developed by the XEROX Palo Alto Research Center, California to demonstrate the feasibility of a usable object-oriented programming environment. The challenge of the system (Goldberg, Adele, Robson, etc [1, 5, 6]) is to provide acceptable performance without compromising the simplicity of the object-oriented programming environment. The goals of the system are to provide:

1. A powerful and flexible user friendly object-oriented programming language and a graphics-based, interactive program development environment. The system should be powerful enough for expert users, but also easy to use by novice users.

2. A powerful information system, one in which the user can store, access and manipulate information so that the system can grow as the user's ideas grow, and provide increased flexibility in terms of user programmability.

3. A language of description which serves as an interface between the models in the human mind and those in computing hardware.

4. A modular, flexible and consistent system which can cater to a wide variety of information needs.

5. A system whose performance is to be comparable with existing interactive information systems.

Smalltalk-80 is an outgrowth of several experimental object-oriented systems, Smalltalk-72, Smalltalk-76 and Smalltalk-78 [5, 6, 7]. They were intended as a vehicle for research in object-oriented programming environment; eventually Smalltalk-80 evolved into a fully operational programming language product.

1.1 Overview Of The Smalltalk-80 System

A complete Smalltalk-80 system is made up of the following components; a) automatic storage management; b) file system; c) display handling; d) text and picture editing; e) keyboard and pointing device input; f) debuggers; g) performance profiler; h) processor scheduling and i) compilation and decompilation. In Smalltalk:

1. Every entity, no matter what its name or function, is an object. For example, objects represent; numbers, character strings, queues, programs, compilers, computational processes, text editors, etc.

2. Every object is an instance of a class, and behaves in a manner prescribed by that class.

3. Each object may maintain its own local state, or memory.

4. Objects communicate with each other by sending messages.

Thus, defining a class consists of enumerating the valid messages which it might receive, and defining the appropriate response associated with each message.

In Smalltalk the user interface component and the language interface component are object-oriented, as opposed to the procedure oriented slant of most older user interfaces and languages. Object-oriented programming replaces conventional operator and operand concepts with messages and objects. Object are private data, and the operations supported on that data; a message is a request for an object to perform one of its operations. For example the expression 1+9 can be viewed as the object 1 being sent the message +9. Recognizing the message '+', it then asks for the next value, which is 9; the addition is performed, and the result is returned.

In a conventional programming language when procedure and function calls are made, such as executing a specific piece of code appropriate to some data type, the decision as to which procedure and/or function is to be called, is made ahead of time (static binding), by contrast, in the

object-oriented model when a message is sent to perform some operation, the message is responsible only for deciding what should be done; the objects decide how to do it. Here the semantics are found in the class definition; the interpretation of a message (an operator) is intrinsic or internal to that class, and not external or lodged in a set of external routines. For example adding complex numbers to a compiler requires changing all the routines for addition, subtraction etc. Adding it to Smalltalk merely requires defining appropriate responses to the desired set of messages and defining complex number objects.

1.2 Smalltalk-80 System Implementation

The Smalltalk-80 system is divided into two layers:

1. Virtual Image - This image consists of all the objects in the system.

2. Virtual Machine - This consists of the hardware and software which give dynamics to the objects in the Virtual Image.

In Smalltalk, written programs are translated into sequences of eight-bit instructions (called bytecodes) by the Smalltalk-80 compiler. These bytecodes are used by the interpreter to manipulate objects in the Virtual Image. Below the interpreter is the object memory that holds the

objects which make up the Virtual Image.

The Virtual Image is provided by XEROX, and the task of any system implementor is to create a Virtual Machine. The Virtual Image then can be loaded into the Virtual Machine and the Smalltalk-80 system becomes operational. The external behaviour of the Virtual Machine must conform to the Smalltalk-80 specification, even though the underlying implementation of the Virtual Machine may be different than that presented by the Smalltalk-80 implementors [1]. The current size of the Virtual Image requires at least half megabyte of memory.

As mentioned earlier, the Smalltalk-80 system is divided into two distinct parts, in presenting the details of the Smalltalk-80 implementation, it is assumed that the reader is generally familiar with the Smalltalk-80 system, particularly Objects, Instance Variables, Classes etc. It is beyond the scope of this thesis to present a complete overview of the Smalltalk-80 system; rather, the emphasis is on implementation. Readers unfamiliar with the Smalltalk-80 system should refer to materials presented in the bibliography [1, 6, 7].

1.3 Thesis Description

The Smalltalk-80 System is divided into two parts: a) Smalltalk Virtual Image; and b) Smalltalk-80 Virtual Machine.

This thesis is going to be based on a partial implementation of the Smalltalk-80 Virtual Machine. Specifically:

1. Implement an interpreter to simulate the Virtual Machine
2. Implement all necessary memory management routines used by the interpreter.

The following areas will not be addressed/implemented:

1. Graphics support
2. File System
3. Multiple Processes

The thesis is divided into 7 chapters. Chapter 1 gives a brief overview of the Smalltalk-80 system. Chapter 2 describes the implementation of the system on a time-sharing system. Chapter 3 describes the interpreter; data stuctures, objects, bytecodes, messages are explained. Chapter 4 includes a complete Smalltalk-80 method execution. It lists in detail how source statements gets executed by the interpreter. Chapter 5 describes the object memory required by the system and its implementation. The

structure of the object memory is described, and memory management routines explained. Chapter 6 details the conclusions reached. Chapter 7 explains what needs to be done to make the Smalltalk-80 system fully functional.

2.0 IMPLEMENTATION OF THE SMALLTALK-80 VIRTUAL MACHINE

The Smalltalk-80 Virtual Machine is implemented on top of the 4.2 BSD UNIX^ Operating System. The system is implemented as prescribed by the book "SMALLTALK-80 : The language and its implementation" [1].

There are two distinct part of the Virtual Machine:

1. Smalltalk-80 Interpreter

2. Smalltalk-80 Object Memory Manager

The Smalltalk-80 Virtual Machine was developed using the VAX 11/780 UNIX system. The entire Virtual Machine is written in C. VAX 11/780 is a 32 bit word machine, and the standard Smalltalk-80 implementation uses 16 bit word. So to be consistent with the Smalltalk-80 implementation [1], in all our C programs, a word is treated as 16 bits long.

Even though the System is being developed on a VAX 11/780, the system can be ported to a variety of hardware which has support for C and UNIX. The only modification required in the programs will be to change the macro definitions used to define integer.

The Virtual Machine implementation described in the book is written in Smalltalk-80 [1]. This source code was translated into C code and modified to produce the same

---

^UNIX is a trademark of AT&T Bell Laboratories.

behaviour as prescribed by the Smalltalk source code.

The choice of C as an implementation language was made on the basis of the strength of C as a system programming language. UNIX software tools proved quite useful for program development work.

3.0 THE INTERPRETER IMPLEMENTATION

The Smalltalk-80 Virtual Machine and corresponding bytecode set are stack-oriented. Object pointers are pushed onto and popped from a stack, and when a message is sent, the top few elements of the stack are used as receiver and arguments of the method (methods correspond to programs, subroutines, or procedures in procedure oriented languages). These are replaced by the object returned as the value of that method. Some examples are given later on.

The State of the interpreter consists of:

1. The Compiled Method whose bytecodes are being executed. Compiled Method is an Object which contains the bytecodes to be executed. The data structure of the Compiled Method is described later.

2. The location of the next bytecode to be executed in that Compiled Method. This is the interpreter's instruction pointer.

3. The receiver and arguments of the message that invoked the Compiled Method.

4. Any temporary variables needed by the Compiled Method.

5. A stack

The interpreter uses the above five pieces of information and repeatedly performs a three step cycle [1]:

1. Fetch the bytecode from the Compiled Method indicated by the instruction pointer.

2. Increment the instruction pointer.

3. Perform the function specified by the bytecode.

Figure 1 thru 4 show some examples: The bytecodes are described, the action performed by the bytecodes and the contents of the stack are explained. The stack mentioned in some of the bytecodes is used for several purposes. It holds the receiver, arguments, and results of the two messages that are sent. The stack is also used as the source of the result to be returned by a message. The stack is maintained by the interpreter.

Bytecode	Action	Stack Contents After Execution (Top of stack to Right)
32	Push 3	(3)
33	Push 4	(3 4)
34	Push 5	(3 4 5)
176	Send +	(3 9)
184	Send *	(27)

Figure 1: Bytecodes for the expression 3 * (4 + 5)

Bytecode	Action	Stack Contents After Execution (Top of stack to Right)
32	Push 3	(3)
105	store into a	(3)
135	Pop	( )
33	Push 4	(4)
106	store into b	(4)

Figure 2: Bytecodes for the expression a <- 3. b <- 4.

Bytecode	Action	Stack Contents After Execution (Top of stack to Right)
32	Push 3	(3)
105	Store into a	(3)
135	Pop	(3)
128	Push a	(3)
124	Return top	( )

Figure 3: Bytecodes for the expression a <- 3. |a

Bytecode	Action	Stack Contents After Execution (Top of stack to Right)
32	Push 3	(3)
33	Push 4	(3 4)
176	Send +	(3 9)
105	Store into a	(7)

Figure 4: Bytecodes for the expression a <- 3 + 4

3.1 Bytecodes

The interpreter understands 256 bytecode instructions that fall into five categories: pushes, stores, sends, returns and jumps. Appendix A describes the exact meaning of each bytecode[1]. Since more than 256 instructions for the interpreter are needed (bytecodes are 8 bits long), some of the bytecodes take extensions. An extension is one or two bytes following the bytecode that further specify the instruction. An extension is not an instruction on its own; it is only part of an instruction.

The bytecodes in Appendix A are listed in ranges that have similar function. Each range of bytecode is listed with a bit pattern and a comment about the function of the bytecodes. The bit pattern shows the binary representation of the bytecodes in the range.

Push Bytecodes : A push bytecode indicates the source of an object to be pushed on the top of the interpreter's stack. The sources include: a) the receiver of the message that invoked the Compiled Method; b) the instance variables of the receiver; c) the temporary frame (the arguments of the message and the temporary variables); d) the literal frame of the Compiled Method; e) the top of the stack (i.e., this bytecode duplicates the top of the stack).

Store Bytecodes : The bytecodes compiled from an assignment

expression end with a store bytecode. The bytecodes before the store bytecode compute the new value of a variable and leave it on top of the stack. A store bytecode indicates the variable whose value should be changed.

Send Bytecodes : The send bytecode causes a message to be sent. The object pointers for the receiver and the arguments of the message are found on the active contexts stack. The send bytecode determines the selector of the message and how many arguments to take from the stack. A set of 32 send bytecodes refer directly to the special selectors (described later; Figure 8), the other send bytecodes refer to their selectors in the literal frame.

Return Bytecodes : There are six bytecodes that return control, and a value from a context. The value is usually found on the top of the stack. Four of these are special bytecodes. These are self (message receiver), true, false, nil.

Jump Bytecodes : The jump bytecodes change the active context's instruction pointer by a specified amount. Unconditional jumps change the instruction pointer whenever they are encountered. Conditional jumps only change the instruction pointer if the object pointer on the top of the stack is a specified boolean object, either true or false. The jump bytecodes are used to implement efficient control structures.

3.2 Objects Used By The Interpreter

When the Interpreter is running it needs to have access to certain objects. These objects are :

1. Compiled Method.
2. Contexts.
3. Classes.

3.2.1 Compiled Method

When the interpreter is running it fetches bytecodes from and object called the Compiled Method (Figure 6).

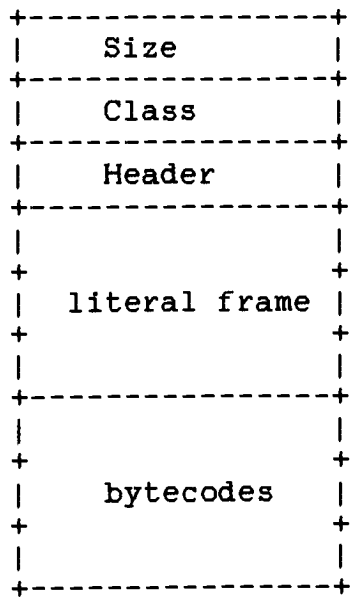


Figure 6

In the Smalltalk-80 Virtual Image when a programmer writes source statements, these are translated by the Smalltalk-80 compiler into bytecodes and stores them in an object called Compiled Method (figure 6). For example the bytecodes for the source statement:

```
extent:newExtent
    corner <- origin + newExtent
```

would be 0, 16, 176, 97, 120. The bytecodes are stored as 8 bit values, with two bytecodes to a word. In addition a Compiled Method also contains some object pointers. The first of these object pointers is called the method header and the rest of the object pointers make up the method's literal frame.

Header: The header field of the Compiled Method is encoded as an object pointer. It contains a small integer with the low order bit set to 1 to indicate it is not an object pointer. The header includes four fields as show in figure 7.

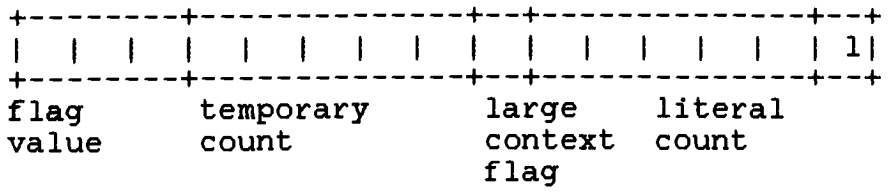


Figure 7

Flag value - This field is 3 bits long and is used to indicate the number of arguments a Compiled Method takes and whether or not it has an associated primitive routine.

Temporary Count - This field is 5 bits long. It indicates to the Interpreter the number of temporary variables including the number of arguments used by the Compiled Method.

Large Context Flag - This field is single bit. It indicates which of the two sizes of stack are needed. Smaller Compiled Method have room for 12 and larger have room for 32.

Literal Count - This field is 5 bits long and holds the size of the literal frame (Figure 6).

Special Primitive Methods: Smalltalk methods that only return the receiver of the message (self) produce Compiled Methods that have no literals or bytecodes, only a header with a flag value of 5. Similarly Smalltalk methods that only return the value of one of the receiver's instance variables produce Compiled Methods that contain only headers with a flag value of 6. All other methods produce Compiled Methods with bytecodes. When the flag value is 6, the index of the instance variable to return is found in the header in the bit field ordinarily used to indicate the number of temporary variables used by the Compiled Methods.

Method Header Extensions: If the flag value is 7, the next to last literal is a header extension. The header extension includes two bit fields that encode the argument count and primitive index of the Compiled Method. Figure 7.1 shows an example.

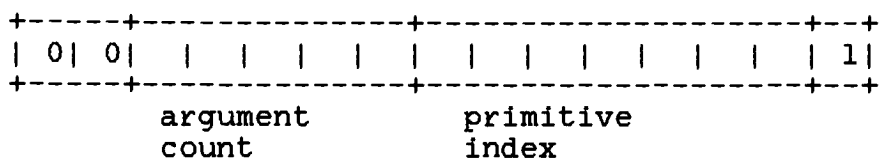


Figure 7.1

Literal Frame: Here object pointers are stored for objects that the Interpreter cannot refer directly. These include the selectors of messages that the method sends, plus shared variables and constants to which the method refers. Figure 8 shows the objects and the message selectors that can be directly referred by bytecodes. All other objects are entered in the literal frame.

Objects:

- the receiver and arguments of the invoking message

- the values of the receiver's instance variables

- the values of any temporary variables required by the method

- seven special constants (true, false, nil, -1, 0, 1, 2)

- 32 special message selectors.

The 32 selectors are:

+	-	<	>
<=	>=	=	~=
*	/	\	@
bitShift	\\	bitAnd	bitOr
at:	at:put:	size	next
nextPut:	atEnd	==	class
blockCopy	value	value:	do:
new	new:	x	Y

Figure 8

For example in the Compiled Method for the method "Rectangle intersects" (Figure 9)[1].

```
intersects:aRectangle
|(origin max:aRectangle origin) < (corner min:aRectangle
corner)
```

The four message selectors, "max:", "origin:", "min:", and "corner" are not in the set that can be directly referenced

by bytecodes (Figure 8). These selector are included in the literal frame and the send bytecode access these selectors by their position in the literal frame.

Rectangle intersects:

```
0       push the value of the receiver's first instance
        variable (origin) onto the stack
16      push the argument (aRectangle)
209     send a unary message with the selector in the
        second literal frame location (origin)
224     send a single argument message with the selector
        in the first literal frame location (max:)
1       push the value of the receiver's second
        instance variable (corner) onto the stack
16      push the argument (aRectangle) onto the stack
211     send a unary message with the selector in the
        fourth literal frame location (corner)
226     send a single argument message with the selector
        in the third literal frame location (min:
178     send a binary message with the selector <
124     return the object on top of the stack as the
        value of the message (intersects:)

Literal Frame
    max:
    origin
    min:
    corner
```

Figure 9

3.2.2 Contexts

The interpreter uses contexts to represent the state of execution of Compiled Methods and Blocks.

Push, store, and jump bytecodes require only small changes to the state of the interpreter. Objects may be moved to or from the stack, and the instruction pointer is always changed; but most of the state remains the same. Send and return bytecodes may require much larger changes to the interpreter's state.

When a message is sent the interpreter's state may have to change in order to execute a different Compiled Method. Before switching, the interpreter's old state is saved in objects called context. This is known as "context switching". A context can be a Method Context or Block Context. A Method Context represents the execution of a Compiled Method in response to a message. A Block Context represents a block encountered in a Compiled Method. A Block Context refers to the Method Context whose Compiled Method contained the block it represents. This is called the Block Context's home Blocks are objects used in many of the control structures in the Smalltalk-80 System. A block represents a deferred sequence of actions. A block expression consists of a sequence of expressions separated by periods and delimited by square brackets. For example: [index <- index + 1].

Figure 9.1 shows a Method Context and its Compiled Method and figure 9.2 shows a Block Context and its Compiled Method plus its home.

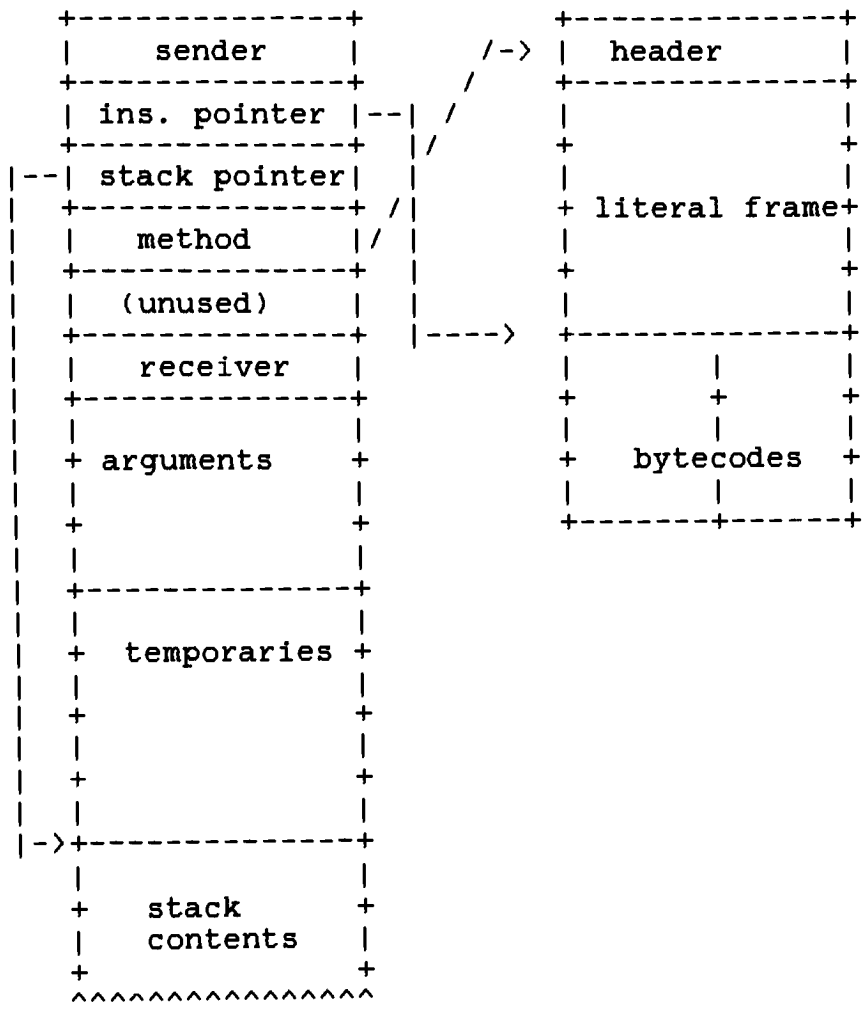


Figure 9.1

Both kinds of contexts have six fields corresponding to six named instance variables. These fixed fields are followed by some indexable fields. The indexable fields are used to store the temporary frame which includes arguments and temporary variables. These are followed by the contents of the evaluation stack.

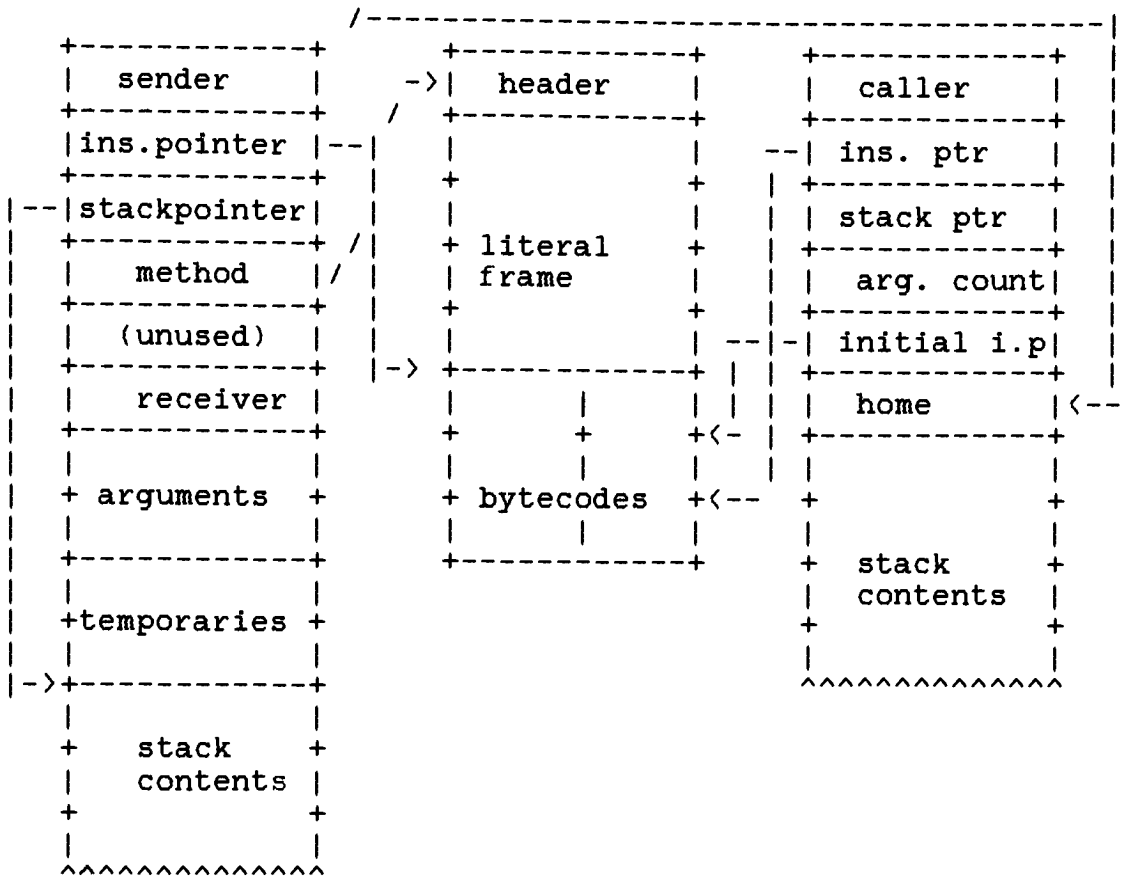


Figure 9.2

The context that represents the Compiled Method or block currently being executed is called the active context. When the active context changes the stack pointer and the instruction pointer are saved in the active context before a new context is made current and executed.

3.2.3 Classes

The interpreter finds the Compiled Method which is to be to executed in response to a message by searching a message dictionary. The dictionary is found in the class of the message receiver or one of the superclasses of that class. A class is an object that describes the implementation of a set of similar objects, and a superclass is the parent of the current class [1].

When a message is sent, the receiver and arguments are identified, and the interpreter searches for the appropriate method (Figure 10).

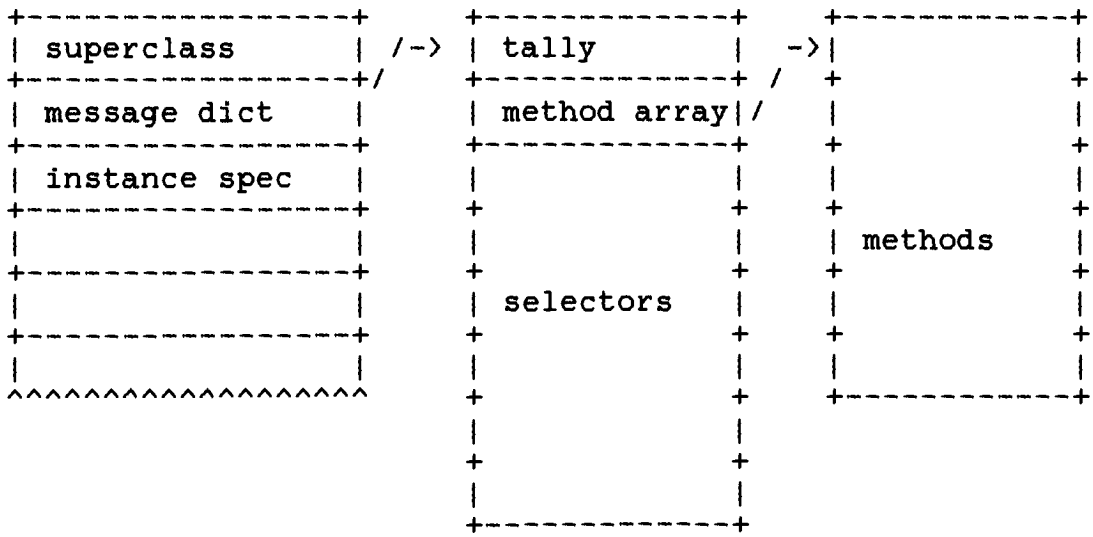


Figure 10

The technique used in Smalltalk-80 is to include in each class-describing object a dictionary, called the method dictionary, that associates selectors with methods. Pointers to the selectors that will be sent by any method are kept in the method (along with global variable pointers and bytecodes). The bytecodes that tell the interpreter to send a message encode a field offset in the literal frame where the selector is found, plus the number of arguments that the method needs. By convention, the top elements of the stack are the arguments, and the elements below the last argument is the receiver. For example, the send bytecode for the expression

3 * 4

will stand for "send the selector in field A of the method (which will be *), and it takes one argument". The interpreter will ask the Memory Manager for the A field of the method, will get the top of the stack (4) as the argument, and will get the next element down (3) as the receiver. It will locate the receiver's class, its method dictionary, search it for an association of the * selector with some method and, when found, execute the method.

If no such association is found, the searching does not end. The receiver's class may be a subclass of another class, called its superclass. If this is the case, the method for * may be defined in the superclass, so the interpreter must

check there. This means that each class must have a field that refers to its superclass. The interpreter searches the method dictionary of the superclass, an so on, until either an appropriate method is found or it runs out of superclasses, in which case an error occurs.

3.2.4 Primitive Subroutines

Since method lookup always involves searching the method dictionary, it is desirable to implement certain functions in machine code without any dictionary lookup to reduce overhead. These are:

1. Input/Output: connecting the Smalltalk-80 system to actual hardware.

2. Arithmetic: basic arithmetic for integers.

3. Subscripting indexable objects: fetching and storing indexable instance variables (Arrays).

4. Screen Graphics: drawing and moving areas of the screen bitmap quickly.

5. Object allocation: Creation of new objects.

In Smalltalk-80 these set of subroutines are called primitive subroutines. These routines are represented in Smalltalk-80 Virtual Image as method with a special flag

that says to run the corresponding subroutine rather than the Smalltalk-80 bytecodes. A few of these primitive methods are executed so often that even the cost of looking them up in their classes method dictionary would be excessive. These methods are represented as special version of the Send Message type of bytecodes. For example, the messages (+, -, *, /) are represented this way. When the bytecode is executed and the top two elements of the stack are small integers, the primitive method is called as a subroutine. When these bytecodes are executed and the top two elements are not small integers, the (+, -, *, /) messages are sent normally. Figure 11 shows an example for the + primitive method.

```
+addend
<primitive:1>
|super + addend


Small integer + associated with primitive #1

112         Push the receiver (self) onto stack
 16         Push the contents of the first temporary frame
            location (the argument addend) onto stack.
133,32      send to super a single argument message with the
            selector in the first literal frame location (+)
124         Return the object on top of the stack as the
            value of the message (+)

Literal Frame
  #+
```

Figure 11

When the interpreter is executing the bytecode from the Compiled Method to send a message and finds one of these flags set, it calls the subroutine directly and uses the value returned from it as the value of the method. If the flag is not set a new Method Context is created and made active.

Even when a primitive method is indicated in the Compiled Method, it is possible for the interpreter not to respond successfully. An example could be that the arguments of the + may not be small integers. If the interpreter cannot execute the primitive, the primitive fails and the a message is sent to its superclass for a response.

4.0 METHOD EXECUTION EXAMPLE

Below an example implementation is described. A complete trace is included of the method execution, context switching and responses to primitive methods and the results returned by each message.

The implementation describes three parts (Figure 12).

1. a class name
2. a declaration of the variables available to the instances
3. the methods used by the instances to respond to messages.

The example given below was defined in the "Smalltalk-80 : **The Language and its Implementation**" book [1]. Some slight modification has been made to the example. There are six methods associated with Financial History Class and they describe some transaction recording, and some queries.

Financial History is the new class, whose superclass is Object. cashOnHand, incomes, expenditures, moneySpent, moneyReceived are all instance variables of Financial History and are shared by all methods in the Financial History Class.

The six methods "someDemo", "initialBalance:", "receive:from:", "spend:for:", "totalReceivedFrom:", and "totalSpentFor:" describes how each object will perform one of its operation.

```
class                       FinancialHistory
superclass                  Object
instance variable names     cashOnHand
                            incomes
                            expenditures
                            moneySpent
                            moneyReceived

instance methods

someDemo
   self initialBalance:16

   self receive: 5 from: 1
   self receive: 6 from: 10

   self spend: 3 for: 10
   self spend: 4 for: 1

   moneyReceived <- self totalReceivedFrom: 10
   moneySpent    <- self totalSpentFor: 1


initialBalance:amount
   cashOnHand   <- amount
   incomes      <- Array new:10
   expenditures <- Array new: 10


receive: amount from: source
   incomes at: source put: amount
   cashOnHand <- cashOnHand + amount


spend: amount for: reason
   expenditures at: reason put:amount
   cashOnHand <- cashOnHand - amount


totalReceivedFrom: source
   Incomes at: source


totalSpentFor: reason
   Expenditures at: reason
```

Figure 12

Method "initialBalance:" initializes the Financial History class. It sets the cashOnHand variable to the initial amount(16 dollars). Then two messages are sent to the Array class, to create two instances of indexed variables of dimension 10 each. The method's (initialBalance:) data structure is shown in figure 13.

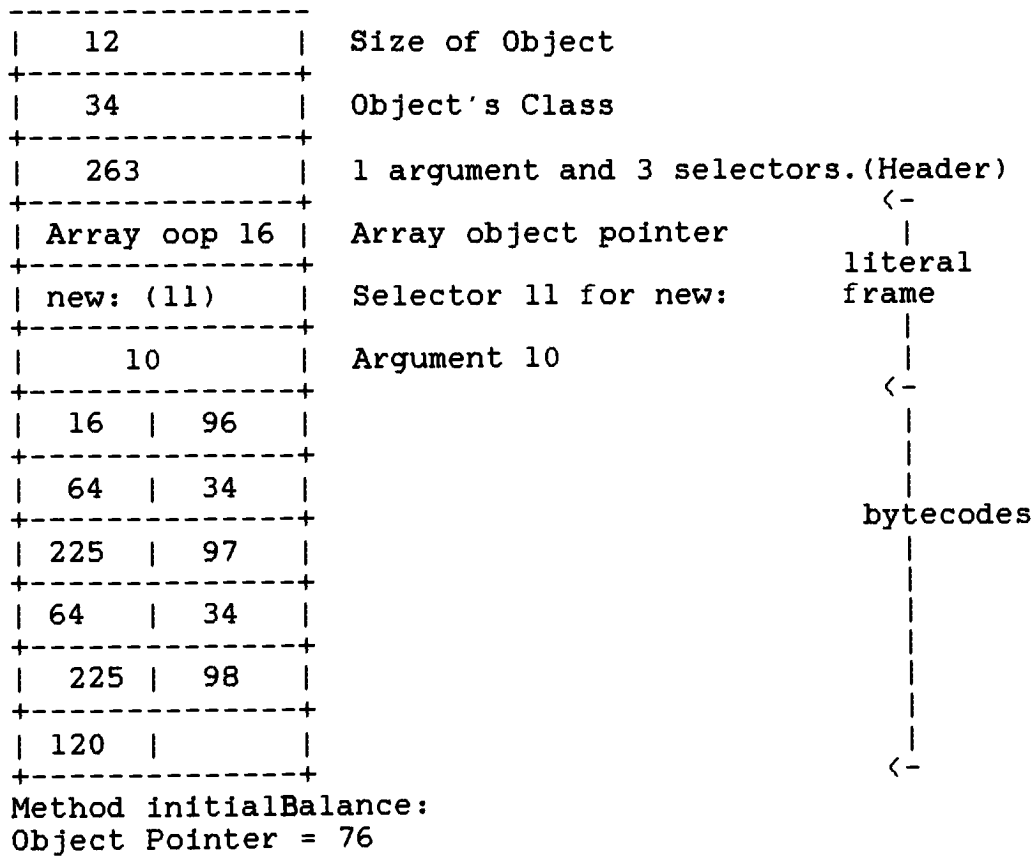


Figure 13

The 263 which is in the header field specifies that there are 1 argument and 3 selectors. The encoded header is shown

in Figure 13.1.

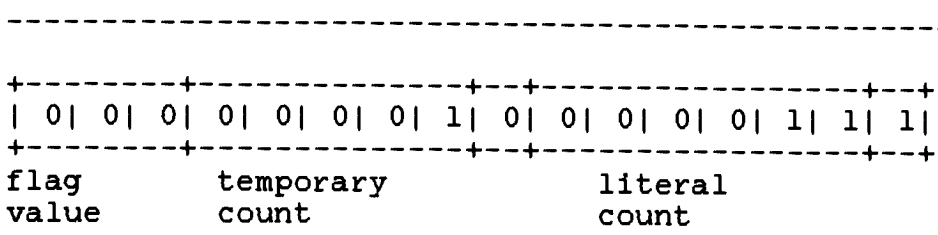


Figure 13.1

Method "receive: amount from: source" describes the method which handles income. The Compiled Method for "receive:from:" is shown in figure 14:

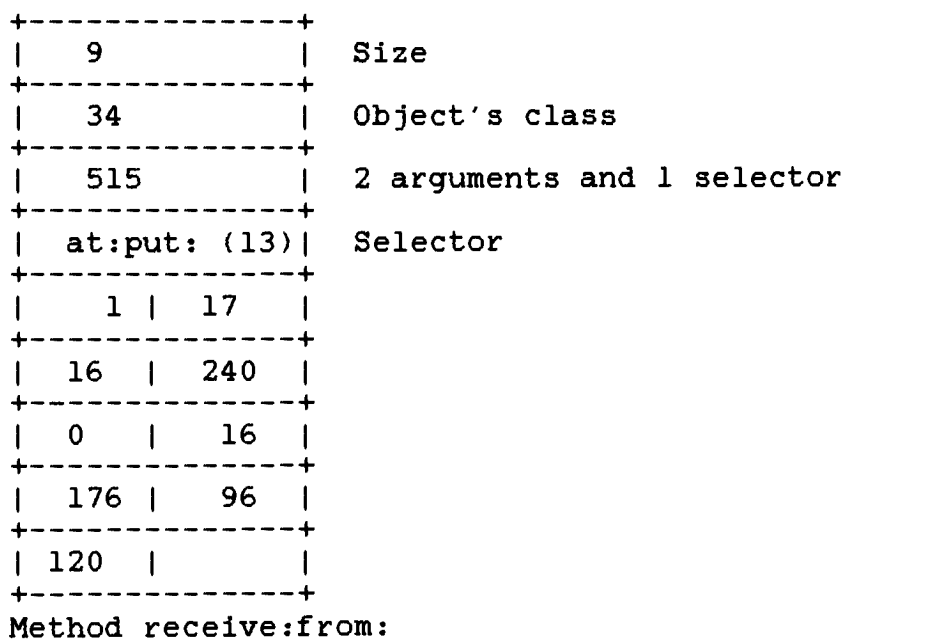


Figure 14

When this method is invoked with the amount and source arguments, the amount is put into the incomes array in the field specified by the source index. The amount is also added to cashOnHand instance variable.

Method "spend: amount for: reason" describes how the expenditures are to be handled. When this method is invoked with the amount and reason arguments, the amount is put into the expenditures array in the field specified by the reason index. The amount is deducted from cashOnHand. The Compiled Method for "spend:for:" is shown in figure 15:

```
+--------------+
|   9          |  Size
+--------------+
|   34         |  Object's class
+--------------+
|   515        |  2 arguments and 1 selector
+--------------+
|  at:put: (13)|  Selector
+--------------+
|    2 |  17   |
+--------------+
|  16  |  240  |
+--------------+
|  0   |   16  |
+--------------+
|  177 |   96  |
+--------------+
| 120  |       |
+--------------+
Method spend:for:
Object Pointer = 80
```

Figure 15

Method "totalReceivedFrom: source" queries the incomes object for a particular item (Figure 16). The source argument specifies the index into the incomes indexed fields. The income amount from the specified source is returned. Here at: method is executed primitively by the interpreter, however the selector associated with at: is still looked by in the dictionary. The method associated with the selector is then executed by the interpreter. No context switch occurs.

```
+--------------+
|       6      |   Size
+--------------+
|     34       |   Class
+--------------+
|    259       |   1 argument and 1 Selector
+--------------+
|     at: (12) |   Selector
+--------------+
|   1   |  16  |
+--------------+
|  224  | 124  |
+--------------+

Method totalReceivedFrom
Object Pointer = 82
```

Figure 16

Method "totalSpentFor: reason" queries the expenditure object for a particular item (Figure 17). The reason argument specifies the index into the expenditures indexed fields. The expenditures amount for the specified reason is returned (Figure 18).

```
+---------------+
|       6       |   Size
+---------------+
|      34       |   Class
+---------------+
|     259       |   1 argument and 1 Selector
+---------------+
|     at: (12)  |   Selector
+---------------+
|   2   |  16   |
+---------------+
|  224  | 124   |
+---------------+
Method totalSpentFor:
Object Pointer = 84
```

Figure 17

Figure 18 (continued next page)

```
+---------------+
|      33       |  Size
+---------------+
|      34       |  Object's class
+---------------+
|     544       |
+---------------+
|       16      |
+---------------+
|     (14)      |  Selector for initialBalance:
+---------------+
|       5       |
+---------------+
|       1       |
+---------------+
|       6       |
+---------------+
|       2       |
+---------------+
|     (15)      |  Selector for receive:from:
+---------------+
|       3       |
+---------------+
|      10       |
+---------------+
|      4        |
+---------------+
|      1        |
```

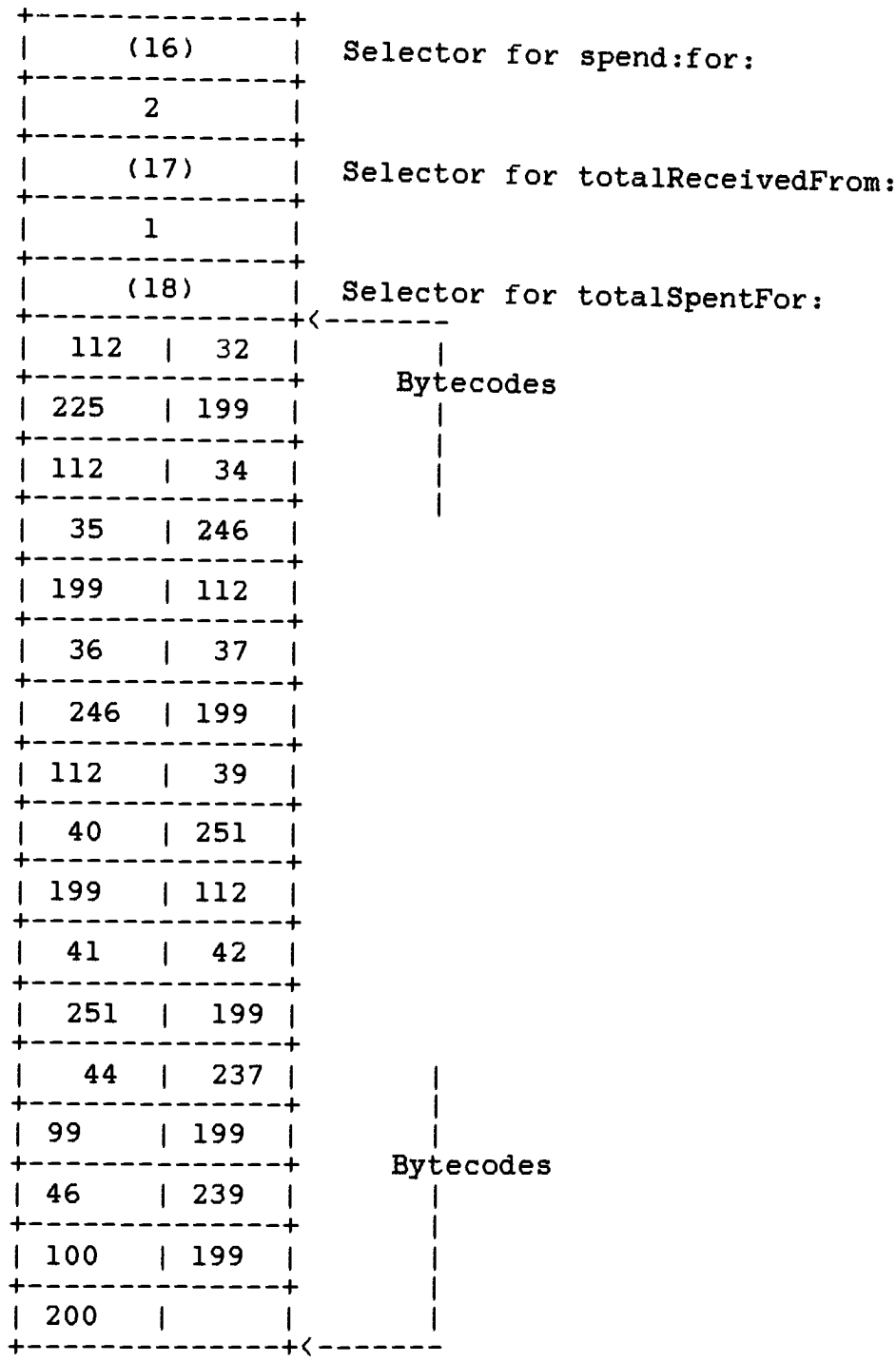
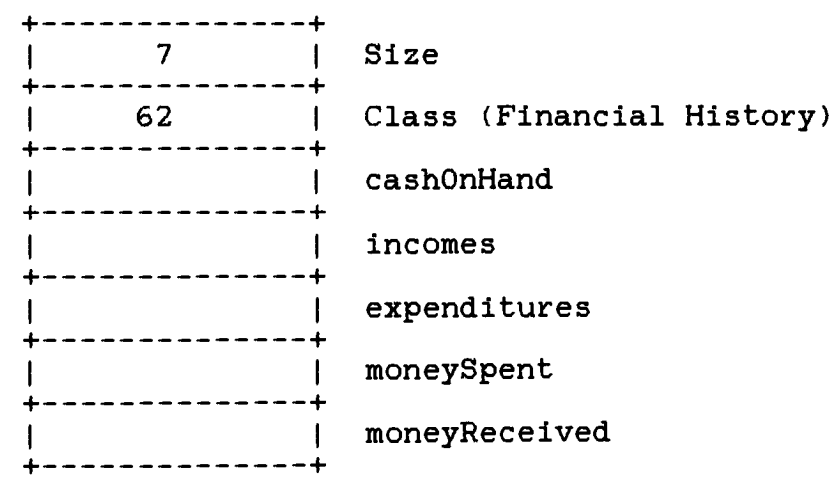

(16)
Selector for spend:for:
2
(17)
Selector for totalReceivedFrom:
1
(18)
Selector for totalSpentFor:
112 | 32
Bytecodes
225 | 199
112 | 34
35 | 246
199 | 112
36 | 37
246 | 199
112 | 39
40 | 251
199 | 112
41 | 42
251 | 199
44 | 237
99 | 199
Bytecodes
46 | 239
100 | 199
200
Method someDemo
Object Pointer = 88

Figure 18

Method "someDemo" shown in figure 18, sends various messages to each of the above methods (described earlier) to perform actions. Here in all cases the receiver is someDemo.

Appendix D shows dumps by the interpreter of the contents of various methods, the receiver variables, etc. Bytecodes 199 and 200 are used for dumping the receiver's objects and its contents, other than that it has no significance in these routines. The receiver object initially looks like (Figure 19):



Receiver's instance variables
Object Pointer = 86

Figure 19

All its instance variable fields "cashOnHand", "incomes" "expenditures", "moneySpent", and "moneyReceived" are set to nil.

The dictionary associated with the Financial History Class is shown in figure 21.

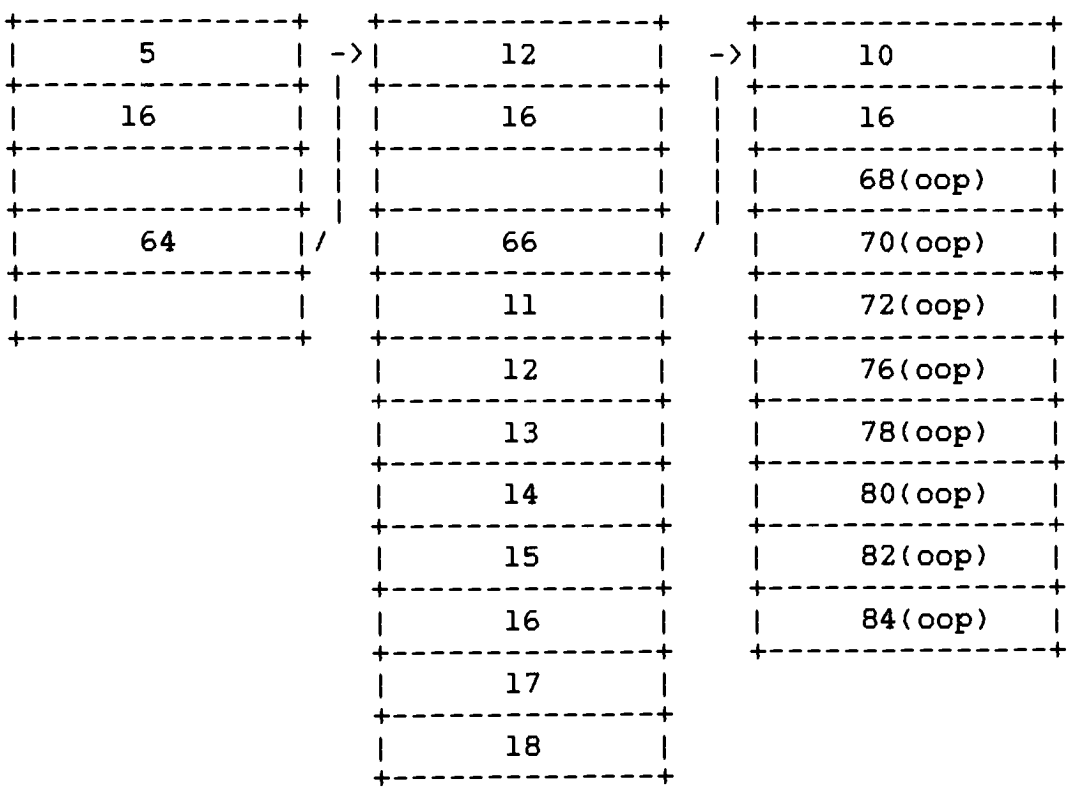


Figure 21

The selectors are chosen arbitrarily and in the Methods the object pointer of each of the methods are stored. These objects were created during the initialization of the Smalltalk system.

The selectors and its associated numbers used in the dictionary lookup are shown in Figure 22:

Selector	Selector No
new:	11
at:	12
at:put:	13
initialBalance:	14
receive:from:	15
spent:for:	16
totalReceivedFrom:	17
totalSpentFor:	18

Figure 22

The object pointers for the methods are shown in figure 23. These object pointers are held in a method array (see figure 21). When the selector is found in the selectors array, the associated method's object pointer is returned.

methods	oop
new:	68
at:	70
at:put:	72
initialBalance:	76
receive:from:	78
spend:for:	80
totalReceivedFrom:	82
totalSpentFor:	84

Figure 23

The methods for "initialBalance:", "receive:from:", "spent:for", "totalReceivedFrom:" and "totalSpentFor:" were described earlier. The selectors "new:", "at:", "at:put:" are primitive methods and the interpreter do not perform context switch to execute these methods.

The methods (Figure 24) for these objects have a special flag set which indicates to the interpreter that it could respond directly. Usually these 3 selectors are stored in a special selector array. However in our example these are put in the dictionary associated with Financial History Class.

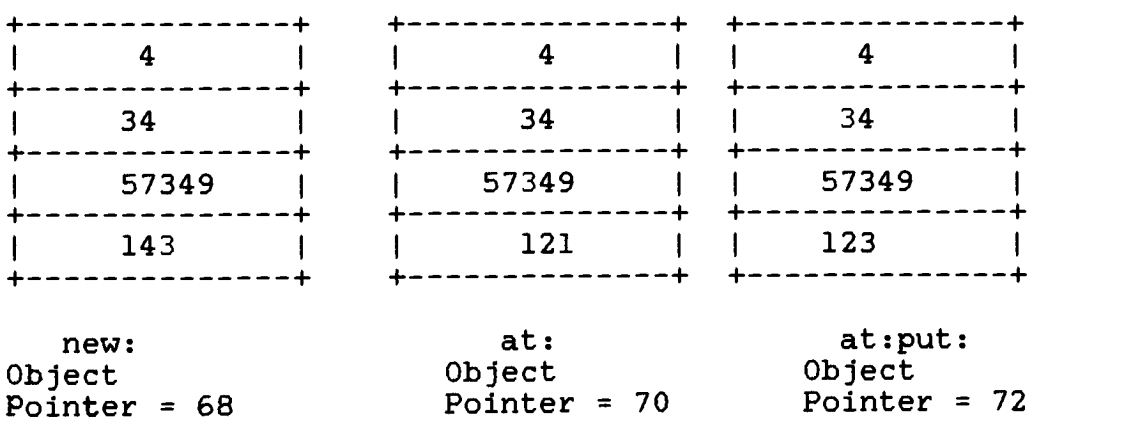
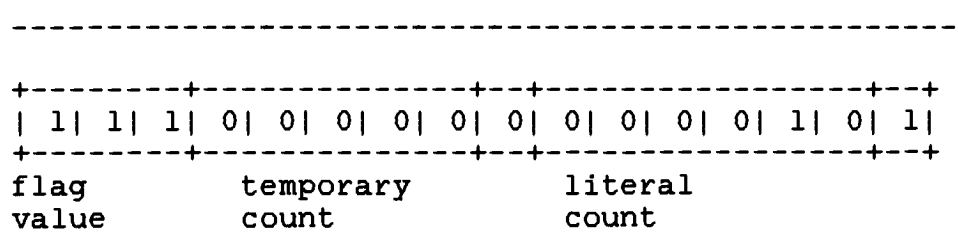


Figure 24

The header field of the Compiled Method "new:" shown in figure 25, which contains 57349 (actually the flag value

part), indicates to the interpreter that this method can be executed primitively.

The example shows the header information of the "new:" method. The field which contains 143 (actually the header extension part) indicates the primitive index number.



The header extension for the primitive index 71 is:

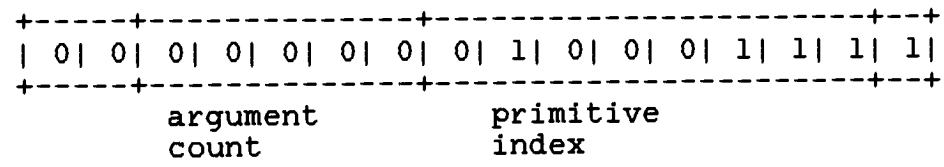


Figure 25

The interpreter checks the method header first to see if a primitive method is indicated. In this case the flag value contains 7, which means that a primitive method exists. From the header extension part the primitive number is fetched. In this example it is 71, which is the primitive index for the method "new:". From this method the interpreter will execute its next bytecode.

When the interpreter is executing the bytecodes in the current method (Its object pointer is 88 and the receivers object pointer is 86) i.e in showDemo, the Compiled Method (also known as the home context as well as the active context) looks like (Figure 25.1):

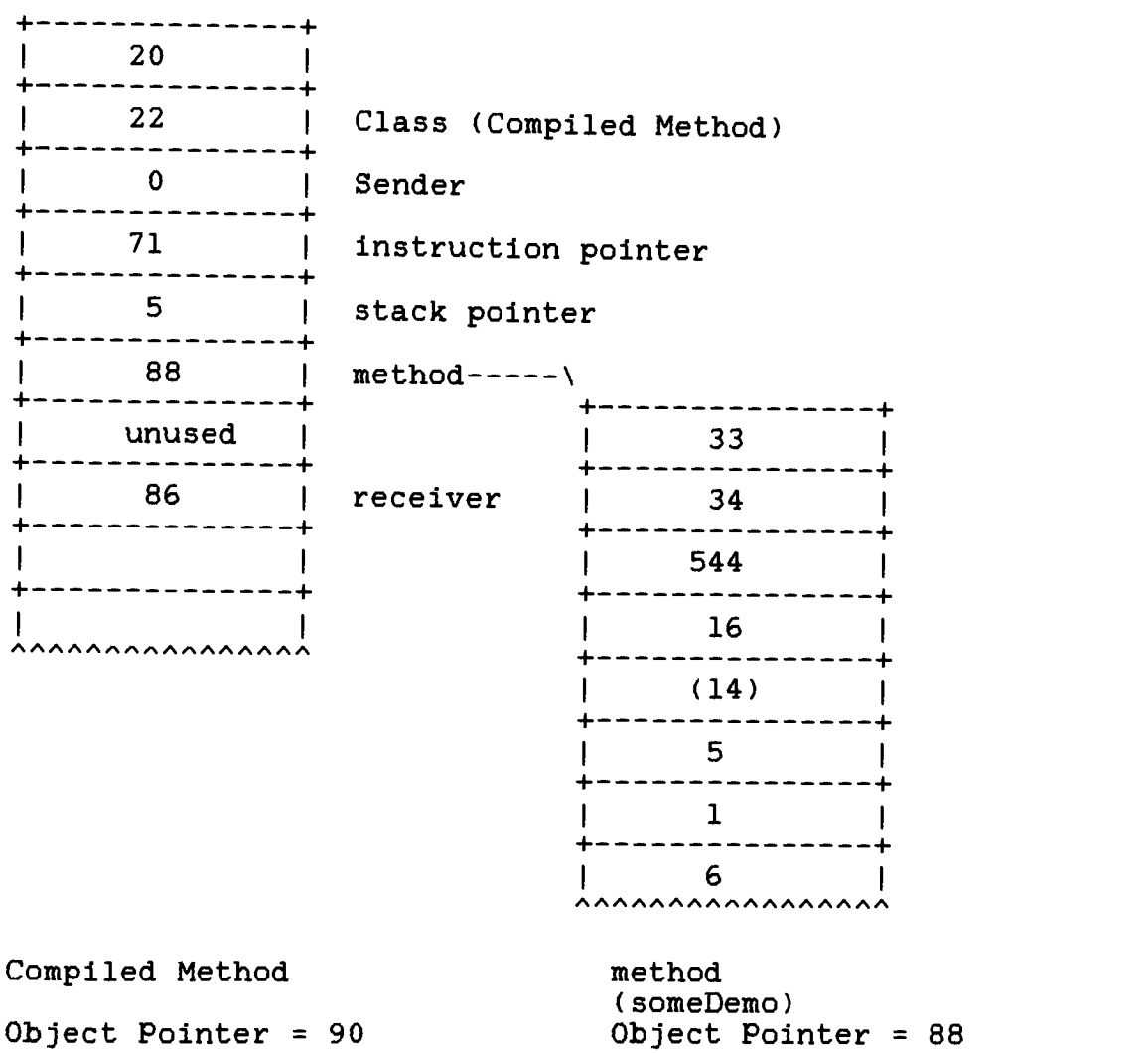


Figure 25.1

Intitally the home context and the active context are the same (they both have the same object pointers). Bytecode 112 instructs the interpreter to push the receiver (object pointer is 86) into the stack. Bytecode 32 pushes the argument 16 (initialBalance:16) into the stack. When bytecode 225 is encountered for the message "initialBalance : 16", the interpreter searches the selector associated with "initialBalance:" in the dictionary of the Financial History class. It finds the method associated with the selector for initialBalance:" and returns the object pointer for the method. The interpreter first updates the stack pointer and instruction pointer of the current context (its initial values are replaced with new values), then makes the new method as the current method to be executed. The current context is saved in the current context stack. The new context becomes the active context, however the previous home context/active context becomes the home context. If access to global variables are required they are accessed from the home context.

The new context is shown in figure 25.2. Notice, on this active context the sender's field contains the object pointer of the previous active context. This will be used by the interpreter to return to its caller correctly.

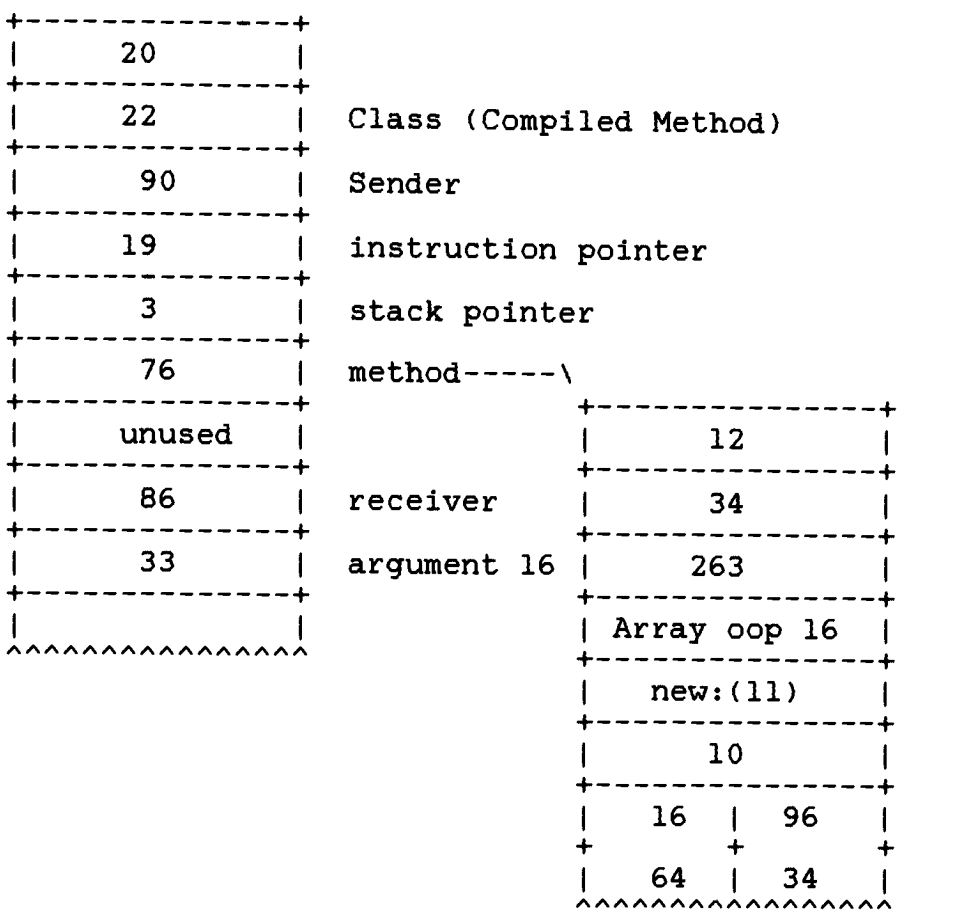


Figure 25.2

So when the execution of this Compiled Method is complete and the bytecode 124 is encountered control will return to the correct context. Using the selector for "initialBalance:" (14), the Compiled Method for "initialbalance:" is fetched (object pointer 76) from the

dictionary, and a new context is created to execute this method and this becomes the current context. From here the interpreter fetches its next bytecode to execute.

Bytecode 16, 96 stores the amount in the cashOnHand object. Bytecode 64 pushes the value of the shared variable (Array) in the stack; 34 pushes the argument 33 (16 as small integer) to the message "new:" in the stack. When bytecode 225 is encountered (i.e. message "new:"), a dictionary lookup is performed to find the method associated with the selector "new:". In this case however no context switch occurs; since "new:" is a primitive method (see figure 24). The interpreter invokes the primitive directly (primitiveNew function in C program). It executes the next bytecode 97. This stores the object pointer associated with the newly created object in the incomes instance variable of the receiver.

When the interpreter reaches and executes the last bytecode, 120, the interpreter returns the receiver as the value of the message. The suspended context is made active, and the interpreter executes the next bytecode, which is 199 in the "showDemo" method whose object pointer is 88. This bytecode dumps the receiver's object contents. This allows us to look at the contents of the receiver's instance variables and shows that the cashOnHand instance variable in the receiver has the value 16. It also shows that the incomes

and expenditures instance variables have object pointer for the new indexable objects that were created with the message "new:".

Bytecode 112 pushes the receiver onto the stack. Bytecodes 34 and 35 pushes the arguments to the message "receive: from:" onto the stack. When bytecode 246 is encountered and executed, a context switch occurs. In this case the interpreter cannot respond directly to the message "receive:from:" as the method header indicates that no primitive method is available. The current context is suspended and the method associated with the "receive: from:" (selector number 15 in this case) is made the current context. The interpreter fetches the first bytecode, 1, from this context and executes it. The two arguments amount and source are pushed onto the stack. The next bytecode 240 associated with the message "at:put:" is fetched. Again the selector (13 in this case) associated with the message "at:put:" is searched in the message dictionary. Since the method indicates that it is a primitive index, no context switch occurs and the interpreter executes the primitive directly.

When the interpreter executes the last bytecode in the current context it returns back to the receiver, whose context is made active again. The interpreter fetches the next bytecode, 199, and dumps the receiver's instance

variables; it then fetches the next bytecode, 112, and so on. This process continues until the last bytecode, 200, is encountered; in this implementation bytecode 200 indicates to the interpreter that the execution should stop, and the bytecode is not dispatched.

5.0 MEMORY MANAGEMENT OVERVIEW

Two kinds of memory management implementations are possible in Smalltalk-80:

1. Real memory - In this approach all the objects in the environment always reside in primary memory.

2. Virtual memory - In this approach objects may reside in more than one level of a memory hierarchy and are moved when needed.

In this implementation the real memory implementation was chosen. The actual details of the implementation are described in a later section of the thesis. Memory for objects in real memory is allocated from a heap. Once all the free space is used up the memory manager tries to regenerate free space by compaction; moving all objects that are in use to contiguous address at one end of the heap. The free space is moved to another end of the heap for later reallocation. If compaction does not regenerate any free space the memory manager invokes the garbage collector to regenerate free space from dangling objects.

Everything in a Smalltalk system is an object. From a storage point of view, memory is divided into blocks (one for each object) plus a pool of memory that has not yet been allocated. Every time a new object is created, a block of the appropriate size is found for that object. When objects

are no longer needed, the blocks associated with them may be returned to the pool.

A special entity called an object pointer is assigned to each object. Each object has a unique object pointer. Object pointers are 16-bits long, with formats is shown in Figure 21.1.

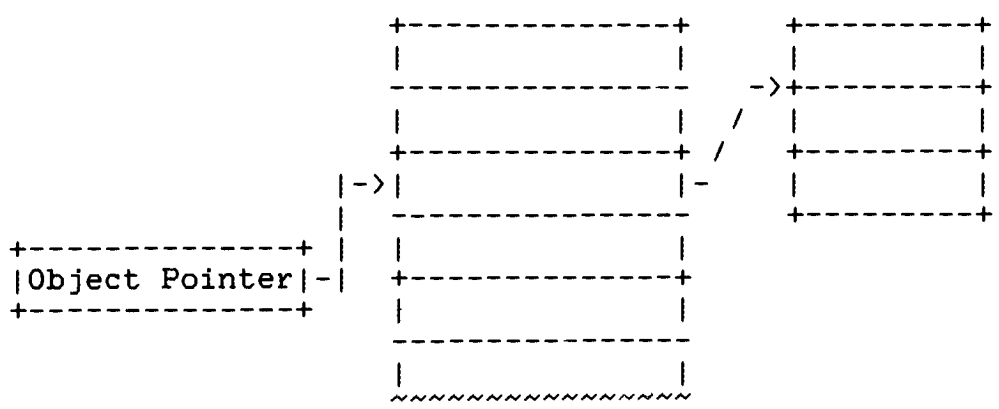


Figure 21

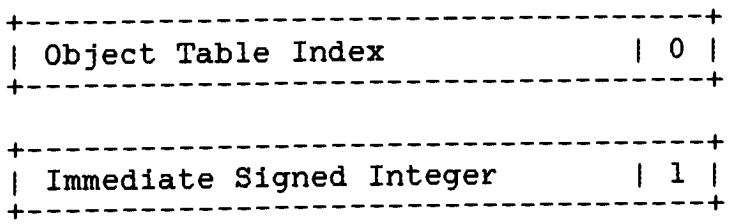


Figure 21.1

If the object pointer were the actual address in memory of the object, then there would be fast access to an object given its pointer. However, in the Smalltalk system an object pointer is an indirect pointer to the object, through a table kept by the memory manager. This allows the memory manager to move an object around in memory without affecting any object that refers to it. It also insures that the memory manager is the only entity in the system concerned with (and allowed to change) memory. In the Smalltalk Virtual Image, object pointers are single 16-bit words. This allows for up to 64k objects in the system; these objects may take up much more than 64k of memory.

GARBAGE COLLECTION

In Smalltalk-80, new objects are allocated explicitly, but there is no explicit language construct that causes an object to be deallocated. Because there are no explicit deallocation routines to notify the memory manager that an object is being abandoned, the memory manager must identify objects that have become inaccessible, and deallocate them to reclaim memory for later allocation.

During memory regeneration, the first problem is to determine which objects are being used and which are not. There are two widely used methods:

1. Marking - In this approach the garbage collector performs an exhaustive search in memory for accessible objects and marks them. Marked objects are then moved into one contiguous block of memory. Those objects that were unmarked are freed and the free space is reclaimed.

2. Reference Counts - In this approach reference counts are used to determine which objects are active and which are not. Whenever a reference to an object is made the object's reference count is incremented; whenever a reference to an object is eliminated, the object's reference count is decremented by one. The object is freed when its reference count reaches zero.

COMPARISON

Both marking and reference counting have advantages and disadvantages. In using marking for memory regeneration, the principal disadvantage is that there may be lengthy interruptions during space reclamation, and this may be unsuitable for interactive applications especially if memory is large. Several advantages exists for marking: programs usually run faster on marking system than on systems using reference counts; recursive structures are always reclaimed; and programs usually run faster since the garbage collector is invoked only when storage regeneration is necessary.

In using reference counts, the most important advantage is that liberation of free objects is incremental; i.e., there is never a single interval of time devoted to garbage collection. This is well suited for interactive applications, since the system is not going to have any lengthy interruptions during garbage collection. The disadvantages to reference counts are, extra space must be provided for reference counts. Whenever object references or dereferences other objects, then the reference counts for the objects must be updated. When an object's reference count reaches zero and is to be deallocated all other object's reference count referenced from that object must also be decremented, and during decrementing if they also reach zero they must be deallocated. This requires extra execution time, also if a recursive structure is created, that structure may never get reclaimed.

In some cases the above two approaches to garbage collection can be combined. Normally reference counting will be employed to reclaim space; at intervals, marking will be used to reclaim recursive structures.

These two are not the only approaches to garbage collection. A variation of these approaches called "Bakers Garbage Collector" was proposed by Baker [3]. This was implemented by the Digital Group in their Smalltalk-80 implementation. Here the process of collecting garbage is interleaved with

the process of allocating space in such a way that the time required for collecting the garbage and compacting space is distributed fairly smoothly over time.

5.1 Object Memory Implementation

Memory management in Smalltalk-80 system is done using the "real memory" implementation (i.e. all the objects in the Smalltalk-80 environment are directly addressable). Objects are not shuffled between secondary storage and primary memory by the Smalltalk-80 system. Since the memory manager is implemented on top of the UNIX operating system, it is possible that objects are swapped in and out by UNIX; however, this process is transparent to the Smalltalk Virtual Machine.

When the Smalltalk system is brought up, memory is allocated as segments, using the C function "Calloc". Each segment is 64k words long, and each word is 16 bits in length and the segment is word indexed (Figure 30).

Currently four segments are allocated. Segment zero is reserved for the object table; segments one through three are used by the Smalltalk-80 system for object space. Space for objects is allocated from these segments. The first 21 words of segment one through three is reserved by the system, and is used to maintain the free lists.

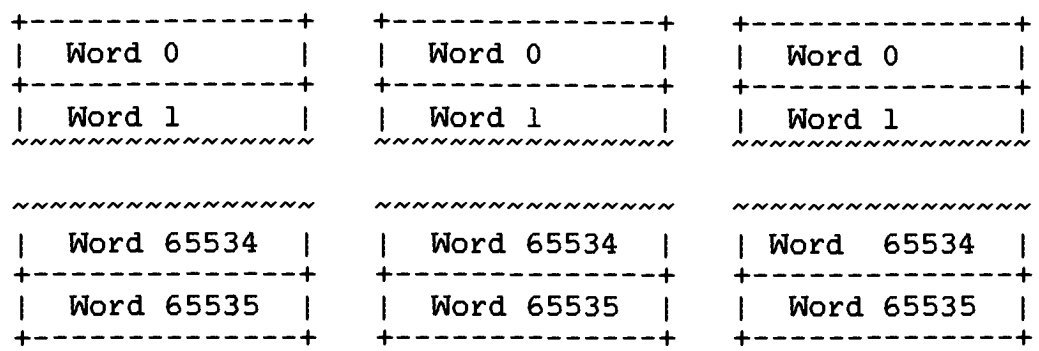


Figure 30

The size of an object pointer determines the maximum number of objects a Smalltalk-80 system can contain. In our implementation we choose the method described by the implementors [1]. Since 16-bit object pointers are used, we can refer to a maximum of 65535 objects (this is not strictly correct, this depends on the actual size of the object table).

When the Smalltalk-80 system is brought up and initialized, some 26 object pointers and associated heap areas in segment one are initialized with system defined objects (i.e, TRUE, FALSE, NIL, ARRAY, INTEGER, etc). Most of these objects do not take up more than 2 words of heap storage.

5.1.1 Object Table

The object table is stored in segment 0. Each entry in the object table requires two 16-bit words. Upto 32K objects can be addressed.

All references to an object are made indirectly through an object pointer via the object table. The actual pointer to the object is stored in the object table. An object pointer to that object table entry is used in reference to the object. Only one pointer to an objects memory location is allowed (Figure 31). The object table is indexed by zero relative indexing. Object table entry 0 is reserved for use by the system in maintaining object table free list.

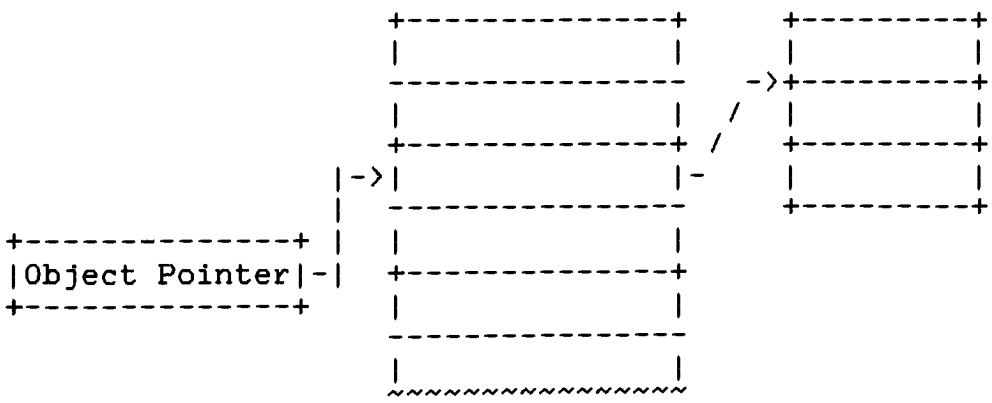
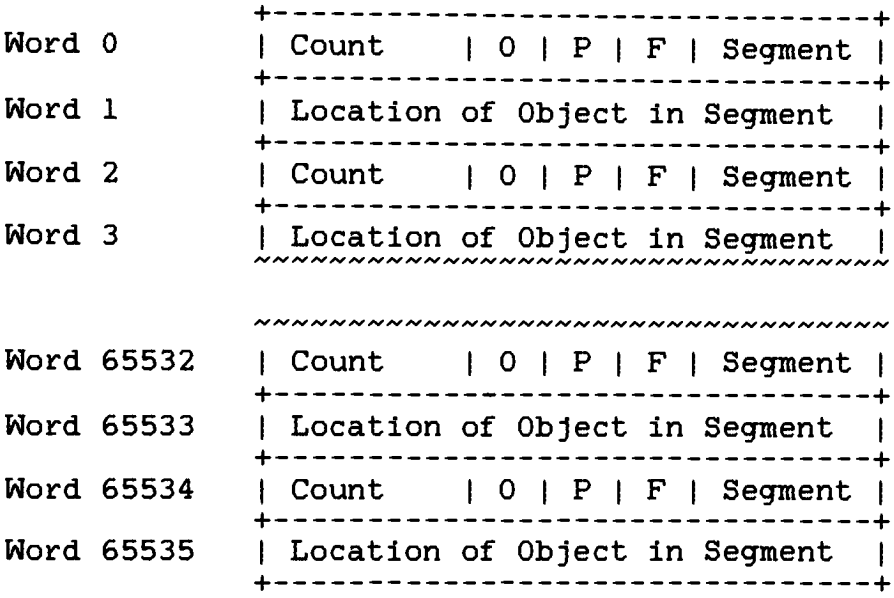


Figure 31

An object pointer (Figure 21.1) can either be an index to the object table or an immediate signed integer. The interpreter distinguishes an object pointer from an immediate signed integer by looking at its low order bit. A value of zero signifies that the rest of the bits hold an object table index; a one signifies that the rest of the bits hold an immediate signed integer. The idea of storing integers in object pointers is to decrease the execution overhead of the interpreter, since they come and go with high frequency during arithmetic and many other operations. The interpreter tests the object pointer every time a reference to an object is made to determine whether it is actually an object pointer or an encoded small integer. The distinction between objects that contain object pointers and those contain integer values is never visible to the Smalltalk-80 programmer.

The format of the object table entry is shown in Figure 33. The Count field, which is eight bits long, is used by the memory manager for reference counting and marking. The count field of the first 26 system defined objects are set to 140 during the system initialization phase, as these objects never gets deallocated. The Odd bit field (O) is used to determine whether to allocate an extra word of storage. When request for object space exceeds 255 words an extra word is allocated. This extra word is used by the garbage collector. The Pointer bit field (P) is used to

indicate if data in the heap are object pointers or not. When the Pointer Bit is one, the data consists of object pointers; when the pointer bit is zero, the data contains 8- or 16-bit integers.



Object Table

Figure 33

The Segment field is four bits long and is used to select the segment holding the object. The location field, which is 16 bits long specifies the start of the space in the segment that is owned by the object table entry. If the Free Bit (F) is on, that object table entry is free; if it is off, the entry is being used.

Object table entries are also used to maintain the free spaces in heap segments one through three. Thus when the Free bit is off in an object table entry, the entry is either in use by an object or is an entry describing a free heap chunk.

All unallocated free entries in the object table entry are kept on a linked list. The link from one free entry to another free entry is kept in the location field (Figure 34). When an object is deallocated its object pointer is not returned to the free list. Instead it is flaged as free (the count field is set to zero), and put on the free heap segment list.

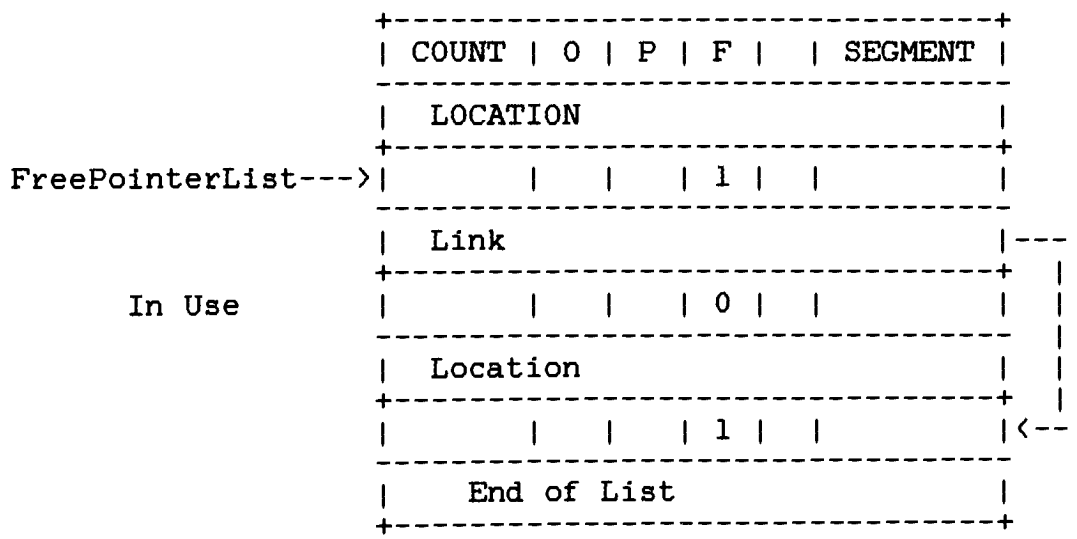


Figure 34

5.1.2 Object Space

Unallocated space in segments one through three are grouped into free chunks of assorted sizes and each of these free chunks is assigned an object table entry. Free chunks are linked together into free lists, each list containing chunks of the same size. The first 21 words of segments one through three are reserved by the memory manager to maintain the free lists. For example, all free spaces in segment one of size two are linked at word two in Segment 1, free spaces of size three in word three and so on. All free chunks bigger than 20 words are linked into a single list attached to word 21 of segment one (Figure 35).

A separate free list is maintained for each segment, but only one free pointer list is maintained for the object table. An object table entry associated with a free chunk is distinguished from an allocated chunk by setting the count field of the object table entry to zero for a free chunk, and nonzero for an allocated chunk.

Initially, when no space is allocated by the memory manager there will be three object table entries set (Free Bit is zero) pointing to the three segments one through three. Since all segments will be greater than 21 words long, word 21 of each segment will hold the object table entry pointing to that segment.

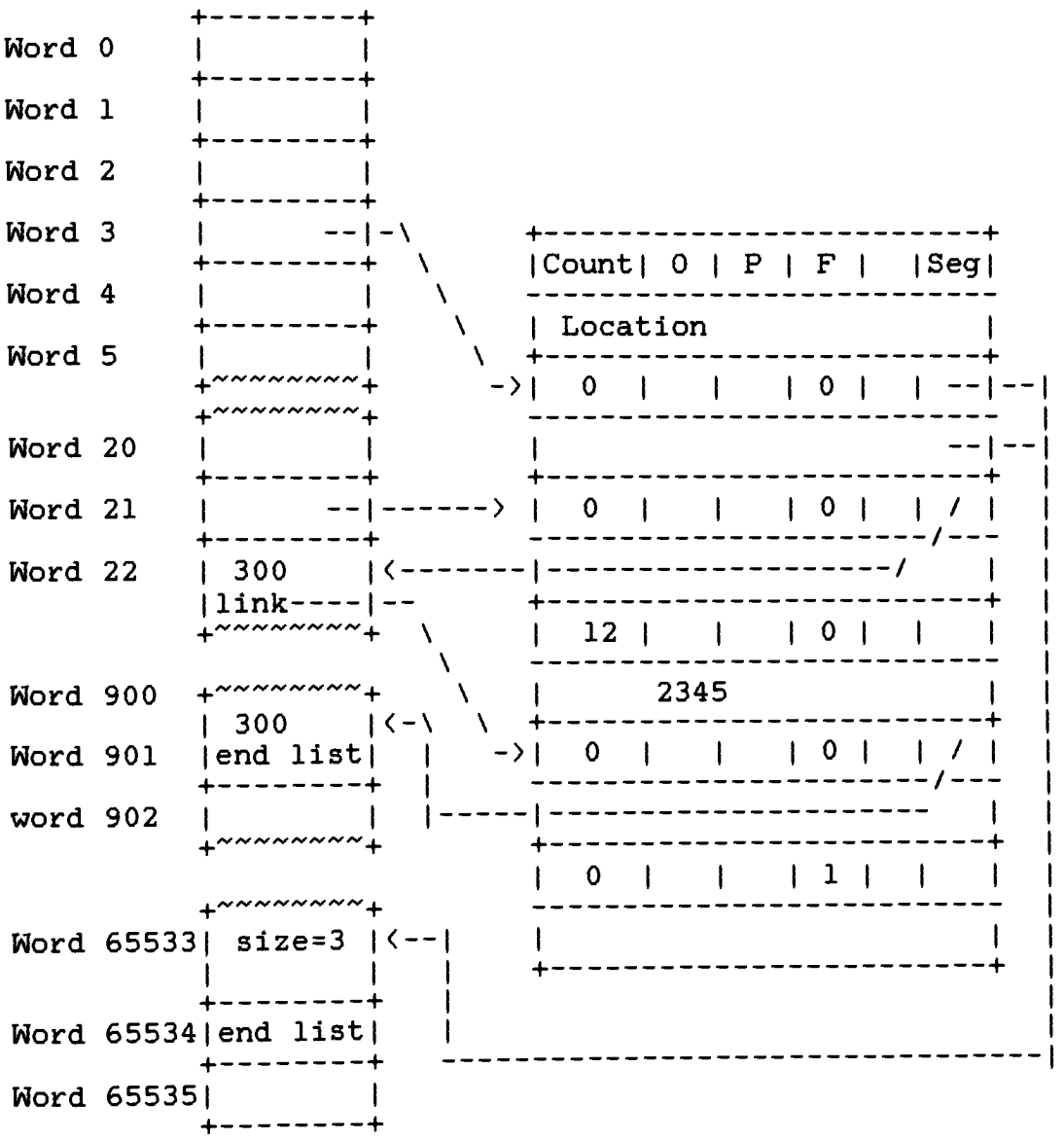

Word 0
Word 1
Word 2
Word 3
Word 4
Word 5
Word 20
Word 21
Word 22
Word 900
Word 901
word 902
Word 65533
Word 65534
Word 65535
size=3
end list
Count
Location
Seg
2345

Figure 35

5.1.3 Allocation

A new object is created by obtaining contiguous space from a heap segment. The actual data of the object is preceded by a two word header (Figure 36).

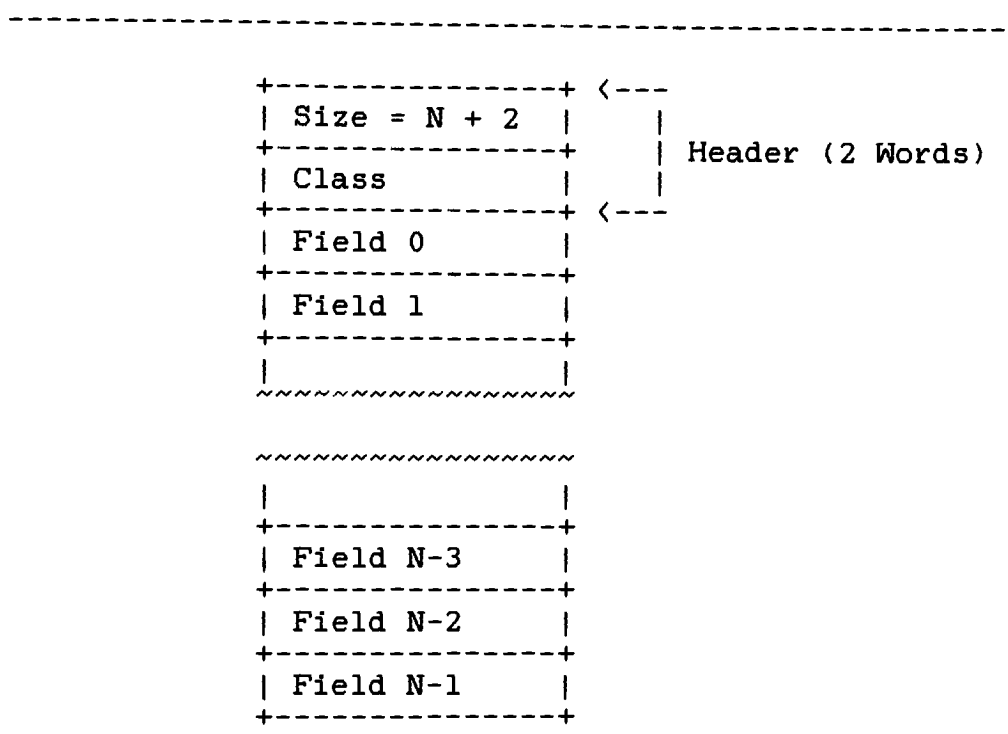


Figure 36

The Size field indicates the number of words that object occupies including the header, and the Class field identifies the class of the object.

To allocate space for an object the memory manager uses the following algorithm [2, 4]:

1. If a request is made for object space of size < 21 words, the requested size is used as an index in the first 21 words of the current segment. If there is a free chunk, it is returned and the request is satisfied. If there is no free chunk in that segment, the next segment is searched. This continues for all the remaining segments. If still no space is found then the Memory Manager uses the algorithm for space larger than 21 words, which is described next.

2. If a request is made for object space greater than 21 words, or if there were no spaces available of size < 21 words, then word 21 of the segment is used as index to the free list. If the free chunk is larger than the requested size, the chunk is split, the requested space is returned, and the unused space is back onto on a free list.

If no chunk of sufficient size is found in the current segment, the memory manager looks at the next segment. The memory manager first compacts the segment to improve the chances of finding sufficient space. During compaction all the allocated objects are moved to one end of the segment and free space is moved to the other end. After compaction the memory manager tries to satisfy the allocation request. If still no space of the requested size is found the Smalltalk System terminates.

5.1.4 Deallocation

Since there is no explicit language construct to deallocate objects when no longer needed, the memory manager performs the necessary housekeeping to track and release unneeded object space. When an object is deallocated, its space is placed on the list of free chunks of the appropriate size. No coalescing is performed by the memory manager when an object is deallocated.

5.1.5 Compaction

To perform memory compaction, the memory manager first scans the object table for every in-use entry for objects in the same segment. The pointer from the object table to the object is reversed. The object's own pointer is stored in the first word of its header which causes the header to point to the object table entry. Before reversing the pointer, the size field of the header is saved in the location field in the object table entry (the location field is recomputed by the memory manager when the object is moved to another location in the heap). When an allocated object is found it is moved as far down in the segment as possible without overwriting other allocated objects. After moving an object, the corresponding object table entry for that object is updated (i.e, its location field is changed to point to the new location). The free chunks found are now

coalesced into a big chunk which is linked on the appropriate size field in the heap. It also links the object table entries reclaimed from the free chunks. After the object is moved the object table pointer is reversed. If no free chunks are found in the current segment, the memory manager will not move any objects.

The object table is never compacted. The Smalltalk interpreter is halted during memory compaction.

5.2 GARBAGE COLLECTION

5.2.1 Reference Counting

Since there is no explicit language construct in Smalltalk to deallocate objects when they are no longer needed, in our implementation the interpreter uses the reference counting mechanism to reclaim spaces from unused objects. The algorithm used is similar to the one described by the Smalltalk-80 designers [1, 2, 4].

The reference count of an object is recorded in the count field of its object table entry. The memory manager manipulates reference counts during the object store operation. When an object pointer referencing object X is stored into a location that formerly contained an object pointer referencing object Z, the count field of X in the object table entry is incremented and the count field of Z

is decremented. Also since the count field of an object table is only eight bits long, it is possible for an incremented count to overflow (the high order bit of the count field serves as an overflow bit), so when the count field reaches 128 it stays there, it is neither incremented or decremented.

When the count field is decremented of an object, the memory manager checks if it is zero. If so first the count field of every object referenced from that object is decremented and then the object is deallocated.

5.2.2 Marking

One of the biggest disadvantages of reference counting is that it is not possible to reclaim objects from recursive structures. Such a structure is said to occur when an object references itself directly or when an object references itself indirectly via other objects that references it. In a reference counting system, when a recursive structure becomes inaccessible to a program, it will have non-zero reference counts, and therefore the memory manager will not be able to detect and deallocate that object.

To resolve this problem, a second algorithm involving marking is used. During marking, the garbage collector

marks all accessible objects starting with the first object table entry in use. It first sets the count field of every object table entry to zero. When marking starts it sets the counts field to one for every marked objects. After marking is done the object table entry is searched for unmarked entries. If an entry is unmarked, its heap space is freed and added to the appropriate free list, and its object table entry is freed and added to the free list of the object table.

Because marking stops the interpreter from doing anything, it is only called when the memory manager cannot satisfy request for object space from any segments after they have been compacted. Ideally, the marking phase will find unused but not deallocated recursive structures, will reclaim some space and thereby satisfy the allocation request (currently this algorithm is disabled in our system, since our system is incapable of executing the Virtual Image provided by XEROX. For a discussion, see Chapter 6).

6.0 CONCLUSION

The Smalltalk-80 system is powerful and comprehensible, in part because almost everything in the system is treated in a uniform way, as an object. Some uniformity has been sacrificed for the sake of execution speed. For example small integers are not represented as objects with object pointers; they are instead encoded directly in the object pointer, since they are used with high frequency in many operations and the cost of representing them as ordinary objects in the system would be high. However to save this extra cost in their efficient representation, the interpreter performs test to determine whether they are small integers or object pointers, by testing the low order bit. If the low order bit is 0 it is an object pointer, if it is 1 then it's a small integer.

Another area where uniformity is sacrificed is in primitive routines. As discussed earlier, when a message is sent, the interpreter usually responds by executing a Smalltalk Compiled Method. This involves creating a new Method Context for that Compiled Method and executing its bytecodes. For commonly used arithmetic operations though, this type of method lookup would be expensive; to save the overhead of message lookup, these messages are responded to directly by the interpreter without the creation of a new context or execution of any other bytecodes.

In the current implementation the behaviour of the Smalltalk-80 system is identical to the behaviour of a real memory implementation, even though the the underlying object memory is virtual (i.e, managed by the UNIX system). This difference is totally transparent to the Smalltalk-80 Virtual Machine.

The Smalltalk-80 Virtual Machine specification as it appears in "Smalltalk-80: The Language and Its Implementation" [1], do not specify any initial state of the Smalltalk-80 system for the implementors. Therefore an implementor has no way of judging how an initialized Smalltalk system would behave as they develop their interpreter. Even though this may not be very crucial for an implementor whose implementation makes use of the Virtual Image provided by XEROX, it still creates some problems. The implementor has to decide what initial value the register instruction pointer will have, what values the registers active context and home context will have, etc.

In the current implementation we did not make use of the Virtual Image provided by XEROX and therefore, a great degree of effort was spent in deciding what state some objects will be in, the values of some registers used by the interpreter (i.e., state of the instruction pointer, active context, home context etc). In some cases these were initialized by trial and error. Also we had no way of

executing the test routines provided by XEROX since there is no graphics support in our system.

It is difficult to quantify the performance of the Smalltalk-80 Virtual Machine on a time-sharing system without the graphics kernel. Currently there is no graphics support. Much has been said [4] about the poor performance of the Smalltalk-80 System in a time-sharing environment. It is felt that this is not due to the performance of the interpreter executing bytecodes; rather, it is due to the support required by the graphics kernel. The system as proposed by the Smalltalk-80 implementors [1] was not designed to run on a time-sharing system, since there are a large number of places where the code loops waiting for input from the user. (This is shown well by the DEC implementation [4]. In the DEC implementation the system was converted into an event driven system where the system would either sleep or perform garbage collection when it was looping).

Appendix C shows a performance log produced by the UNIX "prof" execution profiler program. This program monitors program execution and generates statistics. It shows an average of 10 object memory references for each bytecode executed, and that for every fetch a check was made to see if the object fetched was an object pointer or a small integer.

Indirection to object space via the object table is costly. This can be improved in some places by removing the indirection. Since some system objects are always present (i.e., they are not dynamically created and released), these objects could be stored in a separate heap segment. No indirection via the object table would be needed. When needed they will be fetched directly, and no garbage collection would be performed on this heap segment.

A unique feature of the Smalltalk-80 language is the dynamic binding of methods to a message based on the class of its receiver. However this dynamic binding requires a lookup of the selector in the message dictionaries of the superclass chain for the receiver at execution time. This is a very time consuming operation since a large number of bytecodes involve method lookup. Caching the results of the whole lookup process has been suggested [3], to reduce this overhead.

Since our goal was to learn about the Smalltalk-80 system, in particular the implementation of the Virtual Machine, (we had no previous experience with object-oriented system) no effort was made in the implementation phase to increase the performance of the Virtual Machine.

7.0 FUTURE WORK

The current Smalltalk-80 Virtual Machine is not complete. There is no support for graphics and therefore it is impossible to execute the Virtual Image as supplied by XEROX.

To make the Smalltalk-80 Virtual Machine complete the graphics kernel must be added and the required primitives implemented.

The current Smalltalk-80 Virtual Machine code is not optimized. For example, the access routines used by the interpreter such as (fetchPointerofObject (...), storeIntegerofObject (...), etc) are performed by independent routines. These routines are called regularly by the interpreter and the frequency of these calls is high. Rather than coding these routines as separate functions, these routines may be modified and replaced by inline code in the interpreter, so that the cost of function calls in C can be avoided.

The instantiation (creation) of objects is also expensive because several function calls are required in C. This can be reduced by collapsing several routines into one. Also instead of using the interface routines to instantiate new objects, the memory manager can directly try to acquire new object space from the free list.

Currently since object pointers are 16 bits long, the number of objects that the system can have is limited to 65536. If 32-bit Object Pointers are used this limitation can be avoided and a larger number of objects can be addressed.

A large amount of storage is required for the Virtual Image (the current size of the Virtual Image requires at least one half megabyte of memory). In a real-time system this Image would be divided up into pages by the operating system; therefore there would be large number of page faults if there are other processes in the system besides Smalltalk. This problem needs to be addressed in a time-sharing system by carefully dividing the object space, and keeping the frequently used objects resident in the system.

Other areas where improvements can be made in a real-time system is to have the Virtual Machine functions divided up into independent processes. Separate processes can be created to handle graphics display while the interpreter is doing other work, such as memory compaction. This will require changes to the structure of the interpreter; for example, it will require that the graphics routines do not access the memory area while the memory manager is performing compaction on that area.

8.0 APPENDIX A

8.1 The Smalltalk-80 Bytecodes

Range	Bits	Function
0-15	0000iiii	Push Receiver Variable #iiii
16-31	0001iiii	Push Temporary Location #iiii
32-63	001iiiii	Push Literal Constant #iiiii
64-95	010iiiii	Push Literal Variable #iiiii
96-103	01100iii	Pop and Store Receiver Variable #iii
104-111	01101iii	Pop and Store Temporary Location #iii
112-119	01110iii	Push (receiver, true, false, nil, -1, 0, 1, 2) [iii]
120-123	011110ii	Return (receiver, true, false, nil) [ii] From message
124-125	0111110i	Return Stack Top From (Message, Block) [i]
126-127	0111111i	unused
128	10000000 jjkkkkkk	Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable) [jj] #kkkkkk
129	10000001 jjkkkkkk	Store (Receiver Variable, Temporary Location, Illegal, Literal Variable) [jj] #kkkkkk
130	10000010 jjkkkkkk	Pop and Store (Receiver Variable, Temporary Location, Illegal, Literal variable [jj] #kkkkkk
131	10000011 jjjkkkkk	Send Literal Selector #kkkkk with jjj arguments
132	10000100 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk with jjjjjjjj arguments
133	10000101 jjjkkkkk	Send Literal Selector #kkkkk to Superclass with jjj arguments.
134	10000110 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk to Superclass with jjjjjjjj arguments.
135	10000111	Pop Stack Top
136	10001000	Duplicate Stack Top
137	10001001	Push Active Context
138-143		Unused
144-151	10010iii	Jump iii+1 (1 through 8)
152-159	10011iii	Pop and Jump on False iii+1 (1 through 8)
160-167	10100iii jjjjjjjj	Jump (iii-4)*256+jjjjjjjj
168-171	101010ii jjjjjjjj	Pop and Jump on True ii*256+jjjjjjjj
172-175	10101iii jjjjjjjj	Pop and Jump on False ii*256+jjjjjjjj
176-191	1011iiii	Send Arithmetic Message #iiii

192-207	1100iiii	Send Special Message #iiii
208-223	1101iiii	Send Literal Selector #iiii with no arguments
224-239	1110iiii	Send Literal Selector #iiii with 1 argument
240-255	11111iii	Send Literal Selector #iiii with 2 arguments

9.0 APPENDIX B

9.1 Interface Between The Interpreter And Object Memory

The memory manager provides necessary interface routines allowing the interpreter access to object memory. The interpreter cannot directly manipulate the object memory; it must make its requests to the memory manager through these interface routines to access the object memory.

All the interface routines uses zero-relative integer indices to indicate an object's fields. The routines described below were converted from Smalltalk-80 routines to C routines [1]. There are 21 interface routines, providing access to information in several ways.

For Object Pointer access there are two routines:

fetchPointer_ofObject (fieldIndex, objectPointer)

This routine returns the object pointer found in the field indexed by fieldindex of the object associated with objectPointer.

storePointer_ofObject_withValue (fieldIndex, objectPointer, valuePointer)

This routine is the inverse of the above. It stores the value pointer in the field specified by fieldIndex in the object specified by the objectPointer.

For Word access there are two routines:

fetchWord_ofObject (fieldIndex, objectPointer)

This routine returns the 16-bit value found in the field

specified by the fieldIndex of the object pointed by the objectPointer.

storeWord_ofObject_withValue (fieldIndex, objectPointer, valueWord)

This routine stores a 16-bit value valueWord in the field specified by fieldIndex of the object pointed by the objectPointer.

For Byte access there are two routines:

fetchByte_ofObject (byteIndex, objectPointer)

This routine returns an 8-bit value found in the byte numbered byteIndex of the object pointed by the objectPointer.

storeByte_ofObject_withValue (byteIndex, objectPointer, valueByte)

This routine stores an 8-bit value valueByte in the byte numbered byteIndex of the object pointed by the objectPointer.

For reference counting there are two routines:

increaseReferencesTo (objectPointer)

Increase the reference count of the object pointed by the objectPointer.

decreaseReferencesTo (objectPointer)

Decrease the reference count of the object pointed by the objectPointer.

For class pointer access there is a single routine:

fetchClassOf (objectPointer)

Return the class of the object pointed by the objectPointer as an object Pointer.

For length access there are two routines:

fetchWordLengthOf (objectPointer)

Return the number of word fields of the object associated with the objectPointer.

fetchByteLengthOf (objectPointer)

Return the number of byte fields of the object associated with the objectPointer.

For object creation there are three routines:

instantiateClass_withPointers (classPointer, instanceSize)

Create a new object and return the object pointer. This object will only contain pointers. The class field of the new object will be specified by the classPointer. InstanceSize specifies the number of pointers the object will have.

instantiateClass_withWords (classPointer, instanceSize)

Create a new object and return the object pointer. The object will contain 16-bit numerical values. The class field of the new object will be specified by the classPointer. InstanceSize specifies the number of words the object will have.

instantiateClass_withBytes (classPointer, instanceByteSize)

Create a new object and return the object pointer. This object only contain 8-bit numerical values. The class field of the new object will be specified by the classPointer. InstanceSize specifies the number of 8-bit values the object will have.

For instance enumeration there are two routines:

initialInstanceOf (classPointer)

Return the object pointer of the first instance of the class whose object pointer is classPointer in the defined ordering (e.g., the one with the smallest object pointer).

instanceAfter (objectPointer)

Return the object pointer of the next instance of the same class as the object whose object pointer is objectPointer, in the defined ordering (e.g., the one with the next larger object pointer).

For pointer swapping there is a single routine:

swapPointersOf_and (firstPointer, secondPointer)

Swap pointer between two objects. Make the first object point to the second object and the second object to the first object.

For integer access there are four routines:

integerValueOf (objectPointer)

Return the integer value of an object pointer which contains a small integer. The integer value is encoded directly in the objectPointer.

integerObjectOf (value)
Encode a small integer as an object pointer, and return the object pointer.

isIntegerObject (objectPointer)
Return true if the object pointer is encoded as a small integer, false otherwise.

isIntegerValue (value)
Return true if value can be represented as a small integer (value >= -16384 and value <= 16384), false otherwise.

10.0 APPENDIX C

10.1 Program Execution Statistics

%time	cumsecs	#call	ms/call	name
20.0	1.25	1043	1.01	__doprnt
19.0	2.05			mcount
4.8	2.30	5511	0.05	_isIntegerObject
4.8	2.55	2730	0.09	_extractBits_to_of
4.4	2.78	3107	0.08	_power
4.0	2.99	4916	0.04	_canBeIntegerObject
3.2	3.16	8241	0.02	_bitAnd
3.2	3.33	2462	0.07	_segment_word_bits_to
3.0	3.48	3576	0.04	_segment_word
2.9	3.63			_monstartup
2.4	3.76	2462	0.05	_ot_bits_to
1.9	3.86	1391	0.05	_locationBitsOf
1.6	3.94	2778	0.03	_bitShift
1.6	4.03	210	0.10	_heapChunkOf_word
1.5	4.11	1321	0.05	_segmentBitsOf
1.3	4.18			_bitXor
1.3	4.24	377	0.16	_countBitsOf
1.3	4.31	1043	0.06	_printf
1.3	4.38	4	16.67	_write
1.0	4.43	48	1.04	_integerObjectOf
1.0	4.48	2	25.00	_primitiveAdd
0.8	4.52	1120	0.04	_segment_word_put
0.6	4.55	1	33.34	_activateFakeContext
0.6	4.58	6	5.56	_dispatchSubscriptAndStreamPrimitives
0.6	4.62	51	0.63	_dumpObject
0.6	4.65	44	0.76	_obtainPointer_location
0.6	4.68	198	0.17	_storePointer_ofObject_withValue
0.5	4.71	219	0.11	_countDown
0.5	4.73	377	0.07	_ot_bits_to_put
0.3	4.75	1	16.67	_financialHistory
0.3	4.77	49	0.34	_allocate_odd_pointer_extra_class
0.3	4.78	50	0.33	_attemptToAllocateChunkInCurrentSegment
0.3	4.80	1	16.67	_createDictionary
0.3	4.82	15	1.11	_executeNewMethod
0.3	4.83	66	0.19	_fetchByte_ofObject
0.3	4.85	30	0.56	_fetchInteger_ofObject
0.3	4.87	65	0.26	_freeBitOf_put
0.3	4.88	120	0.14	_headOfFreeChunkList_inSegment
0.3	4.90	68	0.19	_heapChunkOf_byte
0.3	4.92	701	0.02	_heapChunkOf_word_put
0.3	4.93	21	0.79	_increaseReferencesTo
0.3	4.95	1	16.67	_initSmallHistory
0.3	4.97	1	16.67	_initializeObjectTable
0.3	4.98	25	0.67	_instantiateGuaranteedPointers
0.3	5.00	18	1.11	_isBlockContext
0.3	5.02	30	0.56	_literal
0.3	5.03	16	1.04	_literalCountOfHeader
0.3	5.05	3	3.33	_malloc
0.3	5.07	51	0.33	_oddBitOf
0.3	5.08	49	0.34	_pointerBitOf_put
0.3	5.10	7	2.38	_pop
0.3	5.12	64	0.26	_push
0.3	5.13	377	0.04	_segment_word_bits_to_put
0.3	5.15	96	0.19	_segment_word_byte
0.3	5.17	15	1.11	_sendLiteralSelectorBytecode
0.3	5.19	64	0.31	_stackByteCode
0.3	5.20	70	0.24	_storeByte_ofObject_withValue
0.3	5.22	158	0.11	uram
0.2	5.23	252	0.03	_countUp
0.2	5.23	46	0.13	_isIntegerValue
0.2	5.24	124	0.07	_sizeBitsOf
0.2	5.25	143	0.06	_sizeBitsOf_put

```
 0.0     5.25       5     0.00  __flsbuf
 0.0     5.25       1     0.00  _abandonFreeChunksInSegment
 0.0     5.25       7     0.00  _activateNewMethod
 0.0     5.25      49     0.00  _allocateChunk
 0.0     5.25       4     0.00  _arithmeticSelectorPrimitive
 0.0     5.25      49     0.00  _attemptToAllocateChunk
 0.0     5.25       5     0.00  _calloc
 0.0     5.25       6     0.00  _checkIndexableBoundsOf_in
 0.0     5.25      47     0.00  _classBitsOf
 0.0     5.25      59     0.00  _classBitsOf_put
 0.0     5.25       1     0.00  _compactCurrentSegment
 0.0     5.25     165     0.00  _countBitsOf_put
 0.0     5.25       8     0.00  _deallocate
 0.0     5.25      21     0.00  _decreaseReferencesTo
 0.0     5.25      80     0.00  _dispatchOnThisBytecode
 0.0     5.25       8     0.00  _dispatchPrimitives
 0.0     5.25       2     0.00  _dispatchStorageManagementPrimitives
 0.0     5.25       1     0.00  _doAction
 0.0     5.25      88     0.00  _fetchByte
 0.0     5.25      39     0.00  _fetchClassOf
 0.0     5.25      15     0.00  _fetchContextRegisters
 0.0     5.25     386     0.00  _fetchPointer_ofObject
 0.0     5.25      21     0.00  _fetchWordLengthOf
 0.0     5.25       2     0.00  _fetchWord_ofObject
 0.0     5.25      14     0.00  _fixedFieldsOf
 0.0     5.25      22     0.00  _flagValueOf
 0.0     5.25     162     0.00  _freeBitOf
 0.0     5.25       1     0.00  _fstat
 0.0     5.25       1     0.00  _gtty
 0.0     5.25      34     0.00  _headOfFreeChunkList_inSegment_put
 0.0     5.25     102     0.00  _headOfFreePointerList
 0.0     5.25     102     0.00  _headOfFreePointerListPut
 0.0     5.25       8     0.00  _headerExtensionOf
 0.0     5.25      53     0.00  _headerOf
 0.0     5.25      70     0.00  _heapChunkOf_byte_put
 0.0     5.25      12     0.00  _initPrimitive
 0.0     5.25       1     0.00  _initialBalanceMethod
 0.0     5.25       8     0.00  _initialInstructionPointerOfMethod
 0.0     5.25       1     0.00  _initializeGuranteedPointers
 0.0     5.25       1     0.00  _initializeHeaps
 0.0     5.25       1     0.00  _initializeMemory
 0.0     5.25       1     0.00  _initializeSmalltalk
 0.0     5.25      38     0.00  _instanceSpecificationOf
 0.0     5.25      35     0.00  _instantiateClass_withPointers
 0.0     5.25      14     0.00  _instantiateClass_withWords
 0.0     5.25      15     0.00  _instructionPointerOfContext
 0.0     5.25      50     0.00  _integerValueOf
 0.0     5.25       1     0.00  _ioctl
 0.0     5.25       2     0.00  _isIndexable
 0.0     5.25       8     0.00  _isPointers
 0.0     5.25      14     0.00  _isWords
 0.0     5.25       1     0.00  _isatty
 0.0     5.25       7     0.00  _largeContextFlagOf
 0.0     5.25       6     0.00  _lengthOf
 0.0     5.25      16     0.00  _literalCountOf
 0.0     5.25      38     0.00  _literal_ofMethod
 0.0     5.25     137     0.00  _locationBitsOf_put
 0.0     5.25      15     0.00  _lookupMethodInClass
 0.0     5.25      15     0.00  _lookupMethodInDictionary
 0.0     5.25       1     0.00  _main
 0.0     5.25       7     0.00  _newActiveContext
 0.0     5.25       7     0.00  _nilContextFields
 0.0     5.25       1     0.00  _objectArray
 0.0     5.25      49     0.00  _oddBitOf_put
 0.0     5.25      49     0.00  _ot_put
 0.0     5.25      51     0.00  _pointerBitOf
```

```
 0.0     5.25       8      0.00  _
 0.0     5.25      39      0.00  _popInteger
 0.0     5.25       2      0.00  _popStack
 0.0     5.25      12      0.00  _positive16BitIntegerFor
 0.0     5.25       2      0.00  _positive16BitValueOf
 0.0     5.25       4      0.00  _primitiveAt
 0.0     5.25      15      0.00  _primitiveAtPut
 0.0     5.25       2      0.00  _primitiveIndexOf
 0.0     5.25      15      0.00  _primitiveNewWithArg
 0.0     5.25       2      0.00  _primitiveResponse
 0.0     5.25       1      0.00  _primitiveSubtract
 0.0     5.25       4      0.00  _profil
 0.0     5.25      13      0.00  _pushInteger
 0.0     5.25      13      0.00  _pushLiteralConstant
 0.0     5.25       2      0.00  _pushLiteralConstantBytecode
 0.0     5.25       2      0.00  _pushLiteralVariable
 0.0     5.25       5      0.00  _pushLiteralVariableBytecode
 0.0     5.25      10      0.00  _pushReceiverBytecode
 0.0     5.25      10      0.00  _pushReceiverVariable
 0.0     5.25      15      0.00  _pushReceiverVariableBytecode
 0.0     5.25      15      0.00  _pushTemporaryVariable
 0.0     5.25       1      0.00  _pushTemporaryVariableBytecode
 0.0     5.25       1      0.00  _receive_fromMethod
 0.0     5.25      48      0.00  _releasePointer
 0.0     5.25      75      0.00  _removeFromFreeChunkList
 0.0     5.25      20      0.00  _removeFromFreePointerList
 0.0     5.25       7      0.00  _resetFreeChunkList_inSegment
 0.0     5.25       7      0.00  _returnBytecode
 0.0     5.25       7      0.00  _returnToActiveContext
 0.0     5.25       1      0.00  _returnValue_to
 0.0     5.25       5      0.00  _reverseHeapPointersAbove
 0.0     5.25      49      0.00  _sbrk
 0.0     5.25      70      0.00  _segmentBitsOf_put
 0.0     5.25      19      0.00  _segment_word_byte_put
 0.0     5.25      15      0.00  _sendBytecode
 0.0     5.25      15      0.00  _sendSelectorToClass
 0.0     5.25       4      0.00  _sendSelector_argumentCount
 0.0     5.25       7      0.00  _sendSpecialSelectorBytecode
 0.0     5.25       8      0.00  _sender
 0.0     5.25       4      0.00  _spaceOccupiedBy
 0.0     5.25       1      0.00  _specialSelectorPrimitiveResponse
 0.0     5.25      15      0.00  _spent_forMethod
 0.0     5.25      19      0.00  _stackPointerOfContext
 0.0     5.25       9      0.00  _stackValue
 0.0     5.25       7      0.00  _storeAndPopReceiverVariableBytecode
 0.0     5.25      15      0.00  _storeContextRegisters
 0.0     5.25      42      0.00  _storeInstructionPointerValue_inContext
 0.0     5.25      15      0.00  _storeInteger_ofObject_withValue
 0.0     5.25      31      0.00  _storeStackPointerValue_inContext
 0.0     5.25       2      0.00  _storeWord_ofObject_withValue
 0.0     5.25       4      0.00  _subscript_with
 0.0     5.25      30      0.00  _subscript_with_storing
 0.0     5.25       1      0.00  _success
 0.0     5.25      15      0.00  _sweepCurrentSegmentFrom
 0.0     5.25       8      0.00  _temporary
 0.0     5.25       9      0.00  _temporaryCountOf
 0.0     5.25      27      0.00  _toFreeChunkList_add
 0.0     5.25       1      0.00  _toFreePointerList_add
 0.0     5.25       1      0.00  _totalReceivedFromMethod
 0.0     5.25       7      0.00  _totalSpentForMethod
 0.0     5.25     163      0.00  _transfer_fromIndex_ofObject_toIndex_ofObject
 0.0     5.25       1      0.00  udiv
```

11.0 APPENDİX D

11.1 Test Program Listings And Results

```
****ex6.c
#include "deftype.h"
#include "igp.h"

INT16  bankClass;
GLOBAL INT16 currentBytecode;
GLOBAL INT16  method;
GLOBAL INT16  receiver;
GLOBAL INT16  homeContext;

PROCEDURE initBankHistory();
PROCEDURE FinancialHistory();

PROCEDURE doAction()
{
   initBankHistory();
   FinancialHistory();
   activateFakeContext();
   dumpObject (homeContext);

   while (TRUE)
   {
      currentBytecode = fetchByte();
      printf("CurrentBytecode = %d\n",currentBytecode);
      if (currentBytecode == 200)
         exit(0);
      if (currentBytecode == 199)
      {
         dumpObject(receiver);
         dumpObject(fetchPointer_ofObject(1, receiver));
         dumpObject(fetchPointer_ofObject(2, receiver));
      }
      else
         dispatchOnThisBytecode();
   } /* while */

} /* end doAction */
```

```
PROCEDURE initialBalanceMethod();
PROCEDURE receive_fromMethod();
PROCEDURE spent_forMethod();
PROCEDURE totalReceivedFromMethod();
PROCEDURE totalSpentForMethod();
PROCEDURE objectArray();
PROCEDURE createDictionary();

INT16  oop, dict, methodArray;

PROCEDURE initBankHistory()
{
   createDictionary();
   objectArray();
   initialBalanceMethod();
   receive_fromMethod();
   spent_forMethod();
   totalReceivedFromMethod();
   totalSpentForMethod();
   printf("\nFinancialHistory Class\n");
   dumpObject(bankClass);
   printf("dict\n");
   dumpObject(dict);
   printf("methodArray\n");
   dumpObject(methodArray);
} /* end initBankHistory */
```

```
PROCEDURE FinancialHistory()
{
    receiver = instantiateClass_withWords (bankClass, 5);
    printf ("\nreceiver = %d\n", receiver);
    dumpObject (receiver);

    method = instantiateClass_withWords(COMPILEDMETHOD, 34);
    printf ("\nmethod = %d\n", method);
    storeWord_ofObject_withValue (0, method, 545); /* header */
    storeInteger_ofObject_withValue (1, method, 16);
    storeWord_ofObject_withValue (2, method, 14);
    storeInteger_ofObject_withValue (3, method, 5);
    storeInteger_ofObject_withValue (4, method, 1);
    storeInteger_ofObject_withValue (5, method, 6);
    storeInteger_ofObject_withValue (6, method, 2);
    storeWord_ofObject_withValue (7, method, 15);
    storeInteger_ofObject_withValue (8, method, 3);
    storeInteger_ofObject_withValue (9, method, 10);
    storeInteger_ofObject_withValue (10, method, 4);
    storeInteger_ofObject_withValue (11, method, 1);
    storeWord_ofObject_withValue (12, method, 16);
    storeInteger_ofObject_withValue (13, method, 2);
    storeWord_ofObject_withValue (14, method, 17);
    storeInteger_ofObject_withValue (15, method, 1);
    storeWord_ofObject_withValue (16, method, 18);
```

```
PROCEDURE initialBalanceMethod()
{
   oop   = instantiateClass_withWords (COMPILEDMETHOD, 10);
   storePointer_ofObject_withValue (3, methodArray, oop);   /* Add in dict */
   storeWord_ofObject_withValue (0, oop, 263);
   storePointer_ofObject_withValue (1, oop, CLASSARRAYPOINTER);
   storeWord_ofObject_withValue (2, oop, 11);
   storeInteger_ofObject_withValue (3, oop, 10);
   storeByte_ofObject_withValue (8, oop, 16);
   storeByte_ofObject_withValue (9, oop, 96);
   storeByte_ofObject_withValue (10, oop, 64);
   storeByte_ofObject_withValue (11, oop, 34);
   storeByte_ofObject_withValue (12, oop, 225);
   storeByte_ofObject_withValue (13, oop, 97);
   storeByte_ofObject_withValue (14, oop, 64);
   storeByte_ofObject_withValue (15, oop, 34);
   storeByte_ofObject_withValue (16, oop, 225);
   storeByte_ofObject_withValue (17, oop, 98);
   storeByte_ofObject_withValue (18, oop, 120);
   printf("\nMETHOD   initialBalance:\n");
   dumpObject(oop);
} /* end initialBalanceMethod */
```

```
PROCEDURE receive_fromMethod()
{
   oop   = instantiateClass_withWords (COMPILEDMETHOD, 7);
   storePointer_ofObject_withValue (4, methodArray, oop);   /* Add in dict */
   storeWord_ofObject_withValue (0, oop, 515);
   storePointer_ofObject_withValue (1, oop, 13);
   storeByte_ofObject_withValue (4, oop, 1);
   storeByte_ofObject_withValue (5, oop, 17);
   storeByte_ofObject_withValue (6, oop, 16);
   storeByte_ofObject_withValue (7, oop, 240);
   storeByte_ofObject_withValue (8, oop, 0);
   storeByte_ofObject_withValue (9, oop, 16);
   storeByte_ofObject_withValue (10, oop, 176);
   storeByte_ofObject_withValue (11, oop, 96);
   storeByte_ofObject_withValue (12, oop, 120);
   printf("\nMETHOD : received:from:\n");
   dumpObject(oop);
} /* end receive_fromMethod */
```

```
PROCEDURE spent_forMethod()
[
   oop   = instantiateClass_withWords (COMPILEDMETHOD, 7);
   storePointer_ofObject_withValue (5, methodArray, oop);   /* Add in dict */
   storeWord_ofObject_withValue (0, oop, 515);
   storePointer_ofObject_withValue (1, oop, 13);
   storeByte_ofObject_withValue (4, oop, 2);
   storeByte_ofObject_withValue (5, oop, 17);
   storeByte_ofObject_withValue (6, oop, 16);
   storeByte_ofObject_withValue (7, oop, 240);
   storeByte_ofObject_withValue (8, oop, 0);
   storeByte_ofObject_withValue (9, oop, 16);
   storeByte_ofObject_withValue (10, oop, 177);
   storeByte_ofObject_withValue (11, oop, 96);
   storeByte_ofObject_withValue (12, oop, 120);
   printf("\nMETHOD = spent:for:\n");
   dumpObject(oop);
} /* end spent_forMethod */
```

```
PROCEDURE totalReceivedFromMethod()
{
   oop   = instantiateClass_withWords (COMPILEDMETHOD, 4);
   storePointer_ofObject_withValue (6, methodArray, oop);   /* Add in dict */
   storeWord_ofObject_withValue (0, oop, 259);
   storePointer_ofObject_withValue (1, oop, 12);
   storeByte_ofObject_withValue (4, oop, 1);
   storeByte_ofObject_withValue (5, oop, 16);
   storeByte_ofObject_withValue (6, oop, 224);
   storeByte_ofObject_withValue (7, oop, 124);
   printf("\nMETHOD = totalReceivedFrom:\n");
   dumpObject(oop);
} /* end totalReceivedFromMethod */
```

```
PROCEDURE totalSpentForMethod()
{
   oop   = instantiateClass_withWords (COMPILEDMETHOD, 4);
   storePointer_ofObject_withValue (7, methodArray, oop);   /* Add in dict */
   storeWord_ofObject_withValue (0, oop, 259);
   storePointer_ofObject_withValue (1, oop, 12);
   storeByte_ofObject_withValue (4, oop, 2);
   storeByte_ofObject_withValue (5, oop, 16);
   storeByte_ofObject_withValue (6, oop, 224);
   storeByte_ofObject_withValue (7, oop, 124);
   printf("\nMETHOD - totalSpentFor:\n");
   dumpObject(oop);
} /* end totalSpentFor */
```

```
PROCEDURE objectArray()
{
   oop    = instantiateClass_withWords (bankClass, 3); /* Array association */
   countBitsOf_put (oop, 140);
   storePointer_ofObject_withValue (1, CLASSARRAYPOINTER, oop);
   storePointer_ofObject_withValue (1, oop, dict);
   storeWord_ofObject_withValue (2, oop, 24576);
   printf("\nArray object and its association\n");
   dumpObject(CLASSARRAYPOINTER);
   dumpObject(oop);
} /* end objectArray */
```

```
PROCEDURE createDictionary()
{
   INT16  i;

   bankClass = instantiateClass_withPointers (CLASSARRAYPOINTER, 3);
   countBitsOf_put (bankClass, 140);
   dict = instantiateClass_withPointers (CLASSARRAYPOINTER, 10);
   countBitsOf_put (dict, 140);
   storePointer_ofObject_withValue (1, bankClass, dict);
   methodArray = instantiateClass_withPointers (CLASSARRAYPOINTER, 8);
   countBitsOf_put (methodArray, 140);
   storePointer_ofObject_withValue (1, dict, methodArray);
   for (i = 2; i <= 9; i++)
      storeWord_ofObject_withValue (i, dict, i + 9);

   oop = instantiateClass_withWords (COMPILEDMETHOD, 2); /* method new:    */
   countBitsOf_put (oop, 140);
   storeWord_ofObject_withValue (0, oop, 57349);
   storeWord_ofObject_withValue (1, oop, 143);          /* primitive 71   */
   storePointer_ofObject_withValue (0, methodArray, oop);
   printf("\nMETHOD (primitive) - new:\n");
   dumpObject(oop);
   oop = instantiateClass_withWords (COMPILEDMETHOD, 2); /* method at      */
   countBitsOf_put (oop, 140);
   storeWord_ofObject_withValue (0, oop, 57349);
   storeWord_ofObject_withValue (1, oop, 121);          /* primitive 60   */
   storePointer_ofObject_withValue (1, methodArray, oop);
   printf("\nMETHOD (primitive) - at:\n");
   dumpObject(oop);
   oop = instantiateClass_withWords (COMPILEDMETHOD, 2); /* method at:put: */
   countBitsOf_put (oop, 140);
   storeWord_ofObject_withValue (0, oop, 57349);
   storeWord_ofObject_withValue (1, oop, 123);          /* primitive 61   */
   storePointer_ofObject_withValue (2, methodArray, oop);
   printf("\nMETHOD (primitive)   at:put:\n");
   dumpObject(oop);
}
```

```
****ex6a.lis

METHOD (primitive) = new:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  68        140          0          0          0          1         114
 114 -      4 115 =     34 116 - 57349 117 -   143

METHOD (primitive) = at:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  70        140          0          0          0          1         110
 110 =      4 111 =     34 112 - 57349 113 -   121

METHOD (primitive) = at:put:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  72        140          0          0          0          1         106
 106 =      4 107 =     34 108 - 57349 109 -   123

Array object and its association
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  16        130          0          1          0          1         181
 181 =      5 182 =     16 183 =     2 184 -    74 185 -     2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  74        140          0          0          0          1         101
 101 =      5 102 -     62 103 =     0 104 =    64 105 = 24576

METHOD = initialBalance:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  76          1          0          0          0          1          89
  89 =     12  90 =     34  91 =   263  92      16  93      11  94 -     21
  95 =   4192  96 =  16418  97 = 57697  98   16418  99 = 57698 100 =  30720

METHOD = received:from:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  78          1          0          0          0          1          80
  80 =      9  81 =     34  82 -   515  83      13  84 -   273  85 =  4336
  86 =     16  87    45152  88    30720

METHOD = spent:for:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  80          1          0          0          0          1          71
  71 -      9  72 =     34  73 =   515  74 =    13  75 -   529  76    4336
  77       16  78 - 45408  79 =  30720

METHOD = totalReceivedFrom:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  82          1          0          0          0          1          65
  65 -      6  66 -     34  67     259  68      12  69 =   272  70   57468

METHOD = totalSpentFor:
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  84          1          0          0          0          1          59
  59 =      6  60 -     34  61     259  62      12  63     528  64   57468
```

```
FinancialHistory Class
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  62        140          0          1          0          1         140
 140 =     5 141 -    16 142 =     2 143 -    64 144 =     2
dict
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  64        140          0          1          0          1         128
 128 =    12 129 =    16 130 -     2 131 =    66 132 =    11 133 =    12
 134 =    13 135 =    14 136 -    15 137 =    16 138      17 139 -    18
methodArray
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  66        140          0          1          0          1         118
 118 =    10 119 =    16 120 =    68 121 -    70 122      72 123 -    76
 124 =    78 125 =    80 126      82 127 =    84

receiver = 86
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86         0          0          0          0          1          52
  52 =     7  53 =    62  54 =     0  55 -     0  56      0  57 =     0
  58 =     0
In compactCurrentSegment

method = 88
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  88         0          0          0          0          2         164
 164 =    36 165 =    34 166 -   545 167 =    33 168 -    14 169 -    11
 170 =     3 171 =    13 172 -     5 173 =    15 174 -     7 175 -    21
 176 -     9 177 =     3 178 -    16 179      5 180 =    17 181 -     3
 182 =    18 183 - 28704 184 = 57799 185 = 28706 186    9206 187   51056
 188 =  9253 189 - 63175 190 - 28711 191 - 10491 192 - 51056 193 = 10538
 194 = 64455 195 = 11501 196 - 25543 197 = 12015 198 - 25799 199 = 51200
homeContext= 90
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90         1          0          0          0          2         144
 144 =    20 145      22 146 -     0 147     71 148 =     5 149 -    88
 150 =     0 151 -    86 152 -     0 153 -     0 154      0 155      0
 156       0 157 -     0 158 -     0 159 =     0 160 -     0 161 -    0
 162 =     0 163 =     0
CurrentBytecode = 112
CurrentBytecode - 32
CurrentBytecode   225
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  92         0          0          1          0          2         124
 124 -    20 125 =    22 126     90 127     19 128      3 129 -    76
 130 -     2 131 -    86 132 =   33 133 -     2 134      2 135      2
 136 =     2 137 =     2 138      2 139      2 140      2 141      2
 142 =     2 143 =     2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90         2          0          0          0          2         144
 144 =    20 145 =    22 146      0 147     77 148 =     5 149 =    88
 150 =     0 151 -    86 152      0 153 -     0 154 -     2 155      2
 156       0 157 =     0 158 =    0 159      0 160       0 161 -    0
 162       0 163       0
CurrentBytecode - 16
```

```
CurrentBytecode = 96
CurrentBytecode = 64
CurrentBytecode = 34
CurrentBytecode = 225
CurrentBytecode = 97
CurrentBytecode = 64
CurrentBytecode = 34
CurrentBytecode = 225
CurrentBytecode = 98
CurrentBytecode = 120
CurrentBytecode = 199
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86          3          0          0          0          1          52
  52 =     7  53 =    62  54 -    33  55      94  56 -    96  57 -     0
  58 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94          1          0          0          0          2         112
 112 =    12 113 -    74 114 =     0 115 =     0 116 =     0 117 -     0
 118 -     0 119 =     0 120 -     0 121 =     0 122 =     0 123       0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  96          2          0          0          0          2         100
 100 =    12 101 =    74 102 =     0 103 -     0 104 -     0 105 -     0
 106 =     0 107 =     0 108 =     0 109 =     0 110 -     0 111       0
CurrentBytecode = 112
CurrentBytecode = 34
CurrentBytecode = 35
CurrentBytecode = 246
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  98          0          0          1          0          2          80
  80 -    20  81 =    22  82 -    90  83 =    11  84       5  85 -    78
  86 -     2  87 =    86  88 -    11  89 -     3  90       2  91       2
  92 -     2  93 =     2  94 -     2  95 -     2  96 -     2  97 -     2
  98 -     2  99 =     2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90          2          0          0          0          2         144
 144 =    20 145 =    22 146 -     0 147 =    87 148       7 149 =    88
 150 =     0 151 =    86 152 =     0 153 =     0 154 =    86 155 -     2
 156 =     2 157 -     2 158 -     0 159 =     0 160 -     0 161       0
 162 =     0 163 -     0
CurrentBytecode = 1
CurrentBytecode = 17
CurrentBytecode = 16
CurrentBytecode = 240
CurrentBytecode = 0
CurrentBytecode = 16
CurrentBytecode = 176
CurrentBytecode = 96
CurrentBytecode = 120
CurrentBytecode = 199
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86          5          0          0          0          1          52
  52 =     7  53 -    62  54 =    43  55      94  56      96  57       0
  58 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94          1          0          0          0          2         112
```

```
 112 =     12 113 =     74 114 =      5 115 =      0 116 =      0 117 =      0
 118 =      0 119 =      0 120 =      0 121 =      0 122 =      0 123 =      0
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
  96           2           0           0           0           2          100
 100 =     12 101 =     74 102 =      0 103 =      0 104 =      0 105 =      0
 106 =      0 107 =      0 108 =      0 109 =      0 110 =      0 111 =      0
CurrentBytecode = 112
CurrentBytecode = 36
CurrentBytecode = 37
CurrentBytecode = 246
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
 100           0           0           1           0           2           60
  60 =     20  61 =     22  62 =     90  63 =     11  64 =      5  65 =     78
  66 =      2  67 =     86  68 =     13  69 =      5  70 =      2  71 =      2
  72 =      2  73 =      2  74 =      2  75 =      2  76 =      2  77 =      2
  78 =      2  79 =      2
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
  90           2           0           0           0           2          144
 144 =     20 145 =     22 146 =      0 147 =     97 148 =      9 149 =     88
 150 =      0 151 =     86 152 =      0 153 =      0 154 =     86 155 =     86
 156 =      2 157 =      2 158 =      2 159 =      0 160 =      0 161 =      0
 162 =      0 163 =      0
CurrentBytecode = 1
CurrentBytecode = 17
CurrentBytecode = 16
CurrentBytecode = 240
CurrentBytecode = 0
CurrentBytecode = 16
CurrentBytecode = 176
CurrentBytecode = 96
CurrentBytecode = 120
CurrentBytecode = 199
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
  86           7           0           0           0           1           52
  52 =      7  53 =     62  54 =     55  55 =     94  56 =     96  57 =      0
  58 =      0
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
  94           1           0           0           0           2          112
 112 =     12 113 =     74 114 =      5 115 =      6 116 =      0 117 =      0
 118 =      0 119 =      0 120 =      0 121 =      0 122 =      0 123 =      0
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
  96           2           0           0           0           2          100
 100 =     12 101 =     74 102 =      0 103 =      0 104 =      0 105 =      0
 106 =      0 107 =      0 108 =      0 109 =      0 110 =      0 111 =      0
CurrentBytecode = 112
CurrentBytecode = 39
CurrentBytecode = 40
CurrentBytecode = 251
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object    CountBit      OddBit    PointerBit   FreeBit   SegmentBit   LocationBit
 102           0           0           1           0           2           40
  40 =     20  41 =     22  42 =     90  43 =     11  44 =      5  45 =     80
  46 =      2  47 =     86  48 =      7  49 =     21  50 =      2  51 =      2
  52 =      2  53 =      2  54 =      2  55 =      2  56 =      2  57 =      2
  58 =      2  59 =      2
```

```
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90          2          0          0          0          2         144
 144 =    20 145 =    22 146 =     0 147 =   107 148       11 149 -    88
 150 =     0 151 -    86 152 =     0 153 -     0 154 =     86 155      86
 156 =    86 157 =     2 158 -     2 159 =     2 160 =      0 161 =     0
 162 =     0 163 =     0
CurrentBytecode - 2
CurrentBytecode  17
CurrentBytecode - 16
CurrentBytecode - 240
CurrentBytecode - 0
CurrentBytecode  16
CurrentBytecode = 177
CurrentBytecode - 96
CurrentBytecode = 120
CurrentBytecode  199
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86          9          0          0          0          1         52
  52 -     7  53 =    62  54 =    49  55 -    94  56 =    96  57 -     0
  58 =     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94          1          0          0          0          2         112
 112 =    12 113      74 114 =     5 115 =     6 116 =     0 117 -     0
 118 =     0 119 =     0 120 =     0 121 =     0 122 -     0 123 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  96          2          0          0          0          2         100
 100 =    12 101 -    74 102 -     0 103 =     0 104 -     0 105 -     0
 106 =     0 107 -     0 108       0 109 =     0 110       0 111       3
CurrentBytecode = 112
CurrentBytecode = 41
CurrentBytecode = 42
CurrentBytecode = 251
In compactCurrentSegment
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
 104          0          0          1          0          3         180
 180 =    20 181 =    22 182 =    90 183 -    11 184       5 185 -    80
 186 -     2 187 -    86 188 -     9 189 =     3 190       2 191       2
 192 =     2 193      2 194 =     2 195       2 196       2 197       2
 198 =     2 199 -     2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90          2          0          0          0          2         144
 144 -    20 145 -    22 146 -     0 147 -   117 148       13 149 -    88
 150 -     0 151 =    86 152       0 153 -     0 154 -     86 155      86
 156 -    86 157 -    86 158       2 159       2 160 -     2 161       0
 162 -     0 163 -     0
CurrentBytecode   2
CurrentBytecode - 17
CurrentBytecode - 16
CurrentBytecode   240
CurrentBytecode - 0
CurrentBytecode - 16
CurrentBytecode   177
CurrentBytecode   96
CurrentBytecode   120
CurrentBytecode   199
Dump of Object Table and Heap
```

```
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86         11          0          0          0          1          52
  52 =     7  53 =    62  54 =    41  55 =    94  56 -    96  57 =     0
  58 =     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94         1           0          0          0          2         112
 112 =    12 113 =    74 114 =     5 115 =     6 116 =     0 117 -     0
 118 =     0 119 =     0 120 =     0 121 =     0 122 -     0 123 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  96         2           0          0          0          2         100
 100 =    12 101 =    74 102 -     4 103 =     0 104 =     0 105 -     0
 106 =     0 107 =     0 108 =     0 109 =     0 110 =     0 111 -     3
CurrentBytecode - 44
CurrentBytecode = 237
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
 106         0           0          1          0          3         160
 160 =    20 161 =    22 162 -    90 163 =    11 164 =     3 165 =    82
 166 =     2 167 =    86 168 =     5 169 =     2 170 =     2 171 -     2
 172 =     2 173 =     2 174 =     2 175 =     2 176       2 177 -     2
 178 =     2 179 -     2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90         2           0          0          0          2         144
 144 =    20 145 =    22 146 =     0 147 =   123 148      13 149 -    88
 150 =     0 151 =    86 152 =     0 153 -     0 154      86 155 =    86
 156 -    86 157 =    86 158 -     2 159       2 160 =     2 161       0
 162 =     0 163 =     0
CurrentBytecode - 1
CurrentBytecode - 16
CurrentBytecode - 224
CurrentBytecode   124
CurrentBytecode = 99
CurrentBytecode = 199
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86         11          0          0          0          1          52
  52 =     7  53 -    62  54 -    41  55 -    94  56 -    96  57 -    13
  58 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94         1           0          0          0          2         112
 112 =    12 113 =    74 114 -     5 115 =     6 116 =     0 117 =     0
 118 =     0 119 -     0 120 =     0 121 =     0 122 =     0 123 =     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  96         2           0          0          0          2         100
 100 =    12 101 =    74 102      4 103      0 104 -     0 105 -     0
 106 -     0 107 =     0 108 -     0 109      0 110       0 111       3
CurrentBytecode   46
CurrentBytecode   239
In activateNewMethod.. new context is
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
 108         0           0          1          0          3         140
 140 -    20 141 -    22 142 -    90 143      11 144      3 145 -    84
 146 -     2 147 =    86 148 -     3 149      2 150 =     2 151      2
 152 -     2 153      2 154 -     2 155      2 156 -     2 157      2
 158       2 159 =    2
Dump of Object Table and Heap
```

```
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  90          2          0          0          0          2         144
 144 -    20 145 =    22 146 -     0 147 -   131 148 -    11 149 -    88
 150 =     0 151 =    86 152 -     0 153 -     0 154 -    86 155 =    86
 156 =    86 157 =     2 158 =     2 159 =     2 160 -     2 161 -     0
 162 =     0 163 =     0
CurrentBytecode = 2
CurrentBytecode = 16
CurrentBytecode = 224
CurrentBytecode = 124
CurrentBytecode = 100
CurrentBytecode = 199
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  86         11          0          0          0          1          52
  52 =     7  53 =    62  54 =    41  55 =    94  56 =    96  57 -    13
  58 =     9
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  94          1          0          0          0          2         112
 112 =    12 113 -    74 114 =     5 115 =     6 116 -     0 117 -     0
 118 =     0 119 =     0 120 =     0 121 =     0 122 -     0 123 -     0
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  96          2          0          0          0          2         100
 100 =    12 101 =    74 102 -     4 103 =     0 104 =     0 105 =     0
 106 =     0 107 =     0 108 -     0 109 =     0 110 =     0 111 -     3
CurrentBytecode = 200
```

```
****ex7.c

#include "deftype.h"

PROCEDURE doAction()
{
   INT16 oop[8], i;

   /* Test of the Compaction Algorithm */

   oop[0] = instantiateClass_withWords (33, 121);
   oop[1] = instantiateClass_withWords (34, 176);
   oop[2] = instantiateClass_withWords (35, 176);

   printf ("Demonstration of Memory Compaction using segment 4\n\n");
   printf ("Allocation request for 50 and populated with 1\n");
   oop[3] = instantiateClass_withWords (5, 50);
   for (i = 0; i < 50; i++)
      storeWord_ofObject_withValue (i, oop[3], 1);
   dumpObject (oop[3]);

   printf ("Allocation request for 20 and populated with 2\n");
   oop[4] = instantiateClass_withWords (6, 20);
   for (i = 0; i < 20; i++)
      storeWord_ofObject_withValue (i, oop[4],  2);
   dumpObject (oop[4]);
   printf ("Deallocation this object immediately\n");
   deallocate (oop[4]);

   printf ("Allocation request for 50 and populated with 3\n");
   oop[5] = instantiateClass_withWords (7, 50);
   for (i = 0; i < 50; i++)
      storeWord_ofObject_withValue (i, oop[5], 3);
   dumpObject (oop[5]);

   printf ("Allocation request for 30 and populated with 4\n");
   oop[6] = instantiateClass_withWords (8, 30);
   for (i = 0; i < 30; i++)
      storeWord_ofObject_withValue (i, oop[6], 4);
   dumpObject (oop[6]);

   printf ("Allocation request for 18 and populated with 5\n");
   oop[7] = instantiateClass_withWords (9, 18);
   for (i = 0; i < 18; i++)
      storeWord_ofObject_withValue (i, oop[7], 5);
   dumpObject (oop[7]);
```

```
    dumpObjectTable();
    dumpHeap();
    dumpFreeHeap();

    printf ("As seen from the Above Dump there is enough mamory \n");
    printf ("but fragmented. So compaction will recover and satisfy\n");
    printf ("this allocation request\n");
    printf ("Allocation request for 18 and populated with 51\n");
    oop[4] = instantiateClass_withWords (6, 20);
    for (i = 0; i < 20; i++)
       storeWord_ofObject_withValue (i, oop[4], 51);
    dumpObject (oop[4]);

    printf("A fresh dump of the object table and allocated objects follow\n");
    dumpObjectTable();
    dumpHeap();
    dumpFreeHeap();
    dumpObject (oop[3]);
    dumpObject (oop[4]);
    dumpObject (oop[5]);
    dumpObject (oop[6]);
    dumpObject (oop[7]);

} /* end doAction */
```

```
****ex7.lis
In compactCurrentSegment
In compactCurrentSegment
Demonstration of Memory Compaction using segment 4

Allocation request for 50 and populated with 1
In compactCurrentSegment
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  62          0          0          0          0          4         148
 148 =     52 149 =      5 150 =      1 151 -      1 152        1 153        1
 154 =      1 155 -      1 156 =      1 157 =      1 158 =      1 159        1
 160 -      1 161 -      1 162 =      1 163 =      1 164 =      1 165 -      1
 166 =      1 167 =      1 168 =      1 169 =      1 170 =      1 171        1
 172 =      1 173 -      1 174 -      1 175 -      1 176 =      1 177 -      1
 178 -      1 179 =      1 180 =      1 181 -      1 182 -      1 183 -      1
 184 -      1 185 =      1 186 =      1 187 =      1 188 =      1 189        1
 190 -      1 191 =      1 192 =      1 193 =      1 194 =      1 195        1
 196 =      1 197 =      1 198 =      1 199 =      1
Allocation request for 20 and populated with 2
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  64          0          0          0          0          4         126
 126 -     22 127 =      6 128        2 129 =      2 130 =      2 131 =      2
 132 =      2 133 =      2 134 =      2 135 =      2 136 -      2 137 -      2
 138 =      2 139 =      2 140 =      2 141 =      2 142 -      2 143        2
 144 =      2 145 =      2 146 =      2 147 -      2
Deallocation this object immediately
Allocation request for 50 and populated with 3
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  66          0          0          0          0          4          74
  74 -     52  75 -      7  76        3  77 -      3  78 -      3  79        3
  80 -      3  81        3  82 =      3  83        3  84 =      3  85 -      3
  86        3  87 -      3  88 =      3  89 -      3  90 =      3  91        3
  92        3  93 -      3  94 =      3  95 =      3  96 -      3  97        3
  98 =      3  99 -      3 100 -      3 101 -      3 102        3 103 -      3
 104 =      3 105 =      3 106 -      3 107 =      3 108        3 109 -      3
 110 =      3 111 =      3 112 -      3 113 -      3 114 -      3 115 -      3
 116 =      3 117 =      3 118 -      3 119 -      3 120        3 121 -      3
 122 -      3 123 =      3 124 -      3 125 -      3
Allocation request for 30 and populated with 4
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  68          0          0          0          0          4          42
  42 =     32  43 =      8  44        4  45 -      4  46 -      4  47 -      4
  48 =      4  49 -      4  50        4  51        4  52 =      4  53        4
  54        4  55        4  56 =      4  57 =      4  58 =      4  59        4
  60 -      4  61        4  62        4  63 -      4  64 =      4  65 -      4
  66 -      4  67 -      4  68 -      4  69 =      4  70 =      4  71        4
  72 -      4  73 -      4
Allocation request for 18 and populated with 5
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  70          0          0          0          0          4         128
 128 =     20 129 -      9 130        5 131 -      5 132 -      5 133        5
 134 -      5 135 -      5 136 -      5 137 -      5 138 -      5 139        5
 140 =      5 141 =      5 142        5 143 -      5 144        5 145 -      5
 146 =      5 147 =      5
Word     CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
   0          0          0          0          1          0          72
   2        130          0          0          0          1         198
   4        130          0          1          0          1         196
```

6	130	0	1	0	1	194
8	130	0	1	0	1	192
10	130	0	1	0	1	190
12	130	0	1	0	1	188
14	130	0	1	0	1	186
16	130	0	1	0	1	181
18	130	0	1	0	1	179
20	130	0	1	0	1	177
22	130	0	1	0	1	175
24	130	0	1	0	1	173
26	130	0	1	0	1	171
28	130	0	1	0	1	169
30	130	0	1	0	1	167
32	130	0	1	0	1	165
34	130	0	1	0	1	163
36	130	0	1	0	1	161
38	130	0	1	0	1	159
40	130	0	1	0	1	157
42	130	0	1	0	1	155
44	130	0	1	0	1	153
46	130	0	1	0	1	151
48	130	0	1	0	1	149
50	130	0	1	0	1	147
52	130	0	1	0	1	145
54	0	0	0	0	1	22
56	0	0	0	0	2	22
58	0	0	0	0	3	22
60	0	0	0	0	4	22
62	0	0	0	0	4	148
64	0	0	0	0	4	126
66	0	0	0	0	4	74
68	0	0	0	0	4	42
70	0	0	0	0	4	128
72	0	0	0	1	0	74
74	0	0	0	1	0	76
76	0	0	0	1	0	78
78	0	0	0	1	0	80
80	0	0	0	1	0	82
82	0	0	0	1	0	84
84	0	0	0	1	0	86
86	0	0	0	1	0	88
88	0	0	0	1	0	90
90	0	0	0	1	0	92
92	0	0	0	1	0	94
94	0	0	0	1	0	96
96	0	0	0	1	0	98
98	0	0	0	1	0	100
100	0	0	0	1	0	102
102	0	0	0	1	0	104
104	0	0	0	1	0	106
106	0	0	0	1	0	108
108	0	0	0	1	0	110
110	0	0	0	1	0	112
112	0	0	0	1	0	114
114	0	0	0	1	0	116
116	0	0	0	1	0	118
118	0	0	0	1	0	65535

Dump of the Heap Free List

0	65535	0	65535	0	65535	0	65535
1	65535	1	65535	1	65535	1	65535
2	65535	2	65535	2	65535	2	65535
3	65535	3	65535	3	65535	3	65535
4	65535	4	65535	4	65535	4	65535

```
   5    65535      5    65535      5    65535      5    65535
   6    65535      6    65535      6    65535      6    65535
   7    65535      7    65535      7    65535      7    65535
   8    65535      8    65535      8    65535      8    65535
   9    65535      9    65535      9    65535      9    65535
  10    65535     10    65535     10    65535     10    65535
  11    65535     11    65535     11    65535     11    65535
  12    65535     12    65535     12    65535     12    65535
  13    65535     13    65535     13    65535     13    65535
  14    65535     14    65535     14    65535     14    65535
  15    65535     15    65535     15    65535     15    65535
  16    65535     16    65535     16    65535     16    65535
  17    65535     17    65535     17    65535     17    65535
  18    65535     18    65535     18    65535     18    65535
  19    65535     19    65535     19    65535     19    65535
  20    65535     20    65535     20    65535     20    65535
  21    65535     21    65535     21    65535     21       64
  22      123     22      178     22      178     22       20
Dump of Free Heap Segment Chunks
Segment Dump =  4  of size = 21
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  64          0          0          0          0          4         126
 126 =     2 127 -     60
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  60          0          0          0          0          4          22
  22 =    20  23 = 65535  24 -     0  25 -     0  26 -     0  27 -     0
  28 =     0  29 -     0  30 =     0  31 -     0  32 -     0  33 -     0
  34 =     0  35 =     0  36 =     0  37 -     0  38 =     0  39 =     0
  40 =     0  41 -     0
As seen from the Above Dump there is enough mamory
but fragmented. So compaction will recover and satisfy
this allocation request
Allocation request for 18 and populated with 51
In compactCurrentSegment
In compactCurrentSegment
In compactCurrentSegment
In compactCurrentSegment
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  60          0          0          0          0          4         178
 178 =    22 179 -     6 180 -    51 181     51 182     51 183 =    51
 184 -    51 185 =    51 186 -    51 187     51 188 =    51 189 =    51
 190 -    51 191     51 192 -    51 193     51 194     51 195 =    51
 196 =    51 197 -    51 198     51 199     51
A fresh dump of the object table and allocated objects follow
Word     CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
   0          0          0          0          1          0          64
   2        130          0          0          0          1         198
   4        130          0          1          0          1         196
   6        130          0          1          0          1         194
   8        130          0          1          0          1         192
  10        130          0          1          0          1         190
  12        130          0          1          0          1         188
  14        130          0          1          0          1         186
  16        130          0          1          0          1         181
  18        130          0          1          0          1         179
  20        130          0          1          0          1         177
  22        130          0          1          0          1         175
  24        130          0          1          0          1         173
  26        130          0          1          0          1         171
  28        130          0          1          0          1         169
```

30	130	0	1	0	1	167
32	130	0	1	0	1	165
34	130	0	1	0	1	163
36	130	0	1	0	1	161
38	130	0	1	0	1	159
40	130	0	1	0	1	157
42	130	0	1	0	1	155
44	130	0	1	0	1	153
46	130	0	1	0	1	151
48	130	0	1	0	1	149
50	130	0	1	0	1	147
52	130	0	1	0	1	145
54	0	0	0	0	1	22
56	0	0	0	0	2	22
58	0	0	0	0	3	22
60	0	0	0	0	4	178
62	0	0	0	0	4	126
64	0	0	0	1	4	72
66	0	0	0	0	4	54
68	0	0	0	0	4	22
70	0	0	0	0	4	106
72	0	0	0	1	0	74
74	0	0	0	1	0	76
76	0	0	0	1	0	78
78	0	0	0	1	0	80
80	0	0	0	1	0	82
82	0	0	0	1	0	84
84	0	0	0	1	0	86
86	0	0	0	1	0	88
88	0	0	0	1	0	90
90	0	0	0	1	0	92
92	0	0	0	1	0	94
94	0	0	0	1	0	96
96	0	0	0	1	0	98
98	0	0	0	1	0	100
100	0	0	0	1	0	102
102	0	0	0	1	0	104
104	0	0	0	1	0	106
106	0	0	0	1	0	108
108	0	0	0	1	0	110
110	0	0	0	1	0	112
112	0	0	0	1	0	114
114	0	0	0	1	0	116
116	0	0	0	1	0	118
118	0	0	0	1	0	65535

Dump of the Heap Free List

0	65535	0	65535	0	65535	0	65535
1	65535	1	65535	1	65535	1	65535
2	65535	2	65535	2	65535	2	65535
3	65535	3	65535	3	65535	3	65535
4	65535	4	65535	4	65535	4	65535
5	65535	5	65535	5	65535	5	65535
6	65535	6	65535	6	65535	6	65535
7	65535	7	65535	7	65535	7	65535
8	65535	8	65535	8	65535	8	65535
9	65535	9	65535	9	65535	9	65535
10	65535	10	65535	10	65535	10	65535
11	65535	11	65535	11	65535	11	65535
12	65535	12	65535	12	65535	12	65535
13	65535	13	65535	13	65535	13	65535
14	65535	14	65535	14	65535	14	65535
15	65535	15	65535	15	65535	15	65535
16	65535	16	65535	16	65535	16	65535

```
  17    65535     17    65535     17    65535     17    65535
  18    65535     18    65535     18    65535     18    65535
  19    65535     19    65535     19    65535     19    65535
  20    65535     20    65535     20    65535     20    65535
  21    65535     21    65535     21    65535     21    65535
  22      123     22      178     22      178     22       32
Dump of Free Heap Segment Chunks
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  62          0          0          0         0          4          126
 126 =     52 127 =      5 128 =      1 129 -      1 130 =      1 131          1
 132 -      1 133 =      1 134 -      1 135 =      1 136 =      1 137 =        1
 138 -      1 139 =      1 140 =      1 141 -      1 142 =      1 143 -        1
 144 =      1 145 -      1 146 =      1 147 -      1 148 -      1 149 -        1
 150 -      1 151 -      1 152 -      1 153 =      1 154 =      1 155 =        1
 156 =      1 157 -      1 158        1 159 =      1 160 -      1 161          1
 162 =      1 163 -      1 164 -      1 165        1 166 -      1 167 -        1
 168 -      1 169 -      1 170 -      1 171 -      1 172 -      1 173 =        1
 174 =      1 175 =      1 176 -      1 177 =      1
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  60          0          0          0         0          4          178
 178 =     22 179 =      6 180 =     51 181 =     51 182 =     51 183 -      51
 184 =     51 185 -     51 186 =     51 187 =     51 188 =     51 189         51
 190 =     51 191 =     51 192 =     51 193 =     51 194 -     51 195 -      51
 196 -     51 197 -     51 198 =     51 199 -     51
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  66          0          0          0         0          4          54
  54 =     52  55 -      7  56 -      3  57 =      3  58 =      3  59 =        3
  60 =      3  61 =      3  62 =      3  63 -      3  64 =      3  65 -        3
  66 =      3  67 =      3  68 =      3  69 =      3  70 =      3  71          3
  72 =      3  73 =      3  74 =      3  75 =      3  76 =      3  77 -        3
  78 =      3  79 =      3  80 =      3  81 -      3  82 =      3  83 -        3
  84 =      3  85 =      3  86 =      3  87 -      3  88 -      3  89 -        3
  90 =      3  91 -      3  92 =      3  93 =      3  94 =      3  95 =        3
  96 =      3  97 =      3  98 =      3  99 -      3 100 =      3 101 -        3
 102 =      3 103        3 104 =      3 105 -      3
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  68          0          0          0         0          4          22
  22 =     32  23 -      8  24 -      4  25         4  26 =      4  27 -        4
  28 =      4  29 -      4  30 =      4  31         4  32 =      4  33          4
  34 =      4  35 =      4  36 -      4  37 -      4  38 =      4  39          4
  40 =      4  41 =      4  42 =      4  43 -      4  44 =      4  45          4
  46 -      4  47 -      4  48 -      4  49 -      4  50         4  51          4
  52 =      4  53 -      4
Dump of Object Table and Heap
Object   CountBit     OddBit   PointerBit  FreeBit  SegmentBit  LocationBit
  70          0          0          0         0          4          106
 106 -     20 107 -      9 108        5 109        5 110 -      5 111 -        5
 112 -      5 113 -      5 114        5 115        5 116        5 117          5
 118 -      5 119 -      5 120        5 121        5 122 -      5 123          5
 124 -      5 125 -      5
```

BIBLIOGRAPHY

CITED:

1. Goldberg, A., Robson, D., "Smalltalk-80: The Language and its Implementation," Addison-Wesley Publishing Company, 1983.

2. Kaehler, T., "Virtual Memory for an Object-Oriented Language," BYTE, Volume 6, Number 8, August 1981.

3. Krasner G., "Smalltalk-80: Bits of History, Words of Advice," Addison-Wesley Publishing Company, 1983.

4. Krasner G., "The Smalltalk-80 Virtual Machine," BYTE, Volume 6, Number 8, August 1981, pp 300-320.

5. Ingalls, Daniel, H.H., "The Smalltalk-76 Programming System: Design and Implementation," Conference Record, Fifth Annual ACM Symposium on Principles of Programming Language, January 1978.

6. Ingalls, D, H.H., "Design Principles Behind Smalltalk," BYTE, Volume 6, Number 8, August 1981, pp 286-298.

7. McCall, K., "TinyTalk, a Subset of Smalltalk-76 for 64KB Microcomputers," Sigsmall Newsletter, September 1980.

UNCITED:

1. Aho, A.V., Ullman, J.D., "Principles of Compiler Design," Addison-Wesley Publishing Company, 1977.

2. Cox, J.B., "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE Software, January 1984.

3. Horowitz, E., Sahni, S., "Fundamentals of Data Structures," Computer Science Press, 1977.

4. Kernighan, B.W., Ritchie, D.M., "The C Programming Language," Prentice-Hall Inc., 1978.

5. Kernighan, B.W., Pike, R., "The Unix Programming Environment," Prentice-Hall Inc., 1984.

6. Pratt, T.W., "Programming Languages: Design and Implementation," Prentice-hall Inc., Englewood, 1975.

7. Wirth, N., Algorithms + Data Structures = Programs," Prentice-Hall Inc., Englewood, 1976.