JaM

BY John Warnock AND Martin Newell

Version of June 6, 1980

[Ivy]<JaM>JaM.bravo

PREFACE

This document gives a brief description of a Mesa based interactive programming environment called "JaM". In this document three general topics are discussed: what is JaM; what applications lend themselves to the use of JaM; and how JaM interacts with and is interfaced to Mesa.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road
Palo Alto California 94304

This document is for Xerox internal use only

Contents

Introduction

Basic Operation

Examples

JaM Command Catagories

Arithmetic Commands

Boolean and Relational Commands

Stack Manipulation Commands

Execution Control Commands

Dictionary Related Commands

Array Related Commands

Input/Output and Stream Related Commands

Attribute and Conversion Commands

Scanner and String Manipulation Commands

Graphic Commands

Introduction

This document gives a brief description of a Mesa based interactive programming environment called "JaM". In this document three general topics are discussed: What is JaM?; what applications lend themselves to the use of JaM; and how JaM interacts and is used with Mesa.(comment: only first and part of the second of these is done)

Basic Operation

To answer the question: "What is JaM?", several features of the system will be discussed.

1. JaM implements a stack oriented, virtual machine. This machine operates on entities called *objects*. In a common sense sort of way JaM can be thought of as an extremely flexible programmable Hewlett Packard type calculator that can handle a wide variety of objects in a powerful way.

2. An integral part of the JaM machine is a virtual memory that provides an address space of 2^{24} words for storage of all objects in the system.

3. An interactive user interface to the JaM machine is provided. The interface is implemented as a language having an extremely simple syntax. Strings in this language are interpreted by a *scanner* and transformed into a sequence of objects that are, in turn, executed by the JaM machine.

A JaM machine basically consists of three stacks, virtual memory, and the instruction set. The three stacks are: The operand stack, the dictionary (or context) stack, and the execution stack. Under most circumstances the user is concerned only with the operand stack. A description of the dictionary stack is put off until later, although the understanding of its function and use is important. The function of the execution stack need not be of direct concern to the user - it is required for the internal execution control of the machine.

When the JaM machine is initialized the operand stack is empty, the dictionary stack contains one dictionary called the system dictionary, and the machine is executing the keyboard stream (waiting for the user to type something.) Lines of input are passed to the JaM scanner to be parsed into *tokens*. Tokens are delimited by any mixture of spaces, tabs, carriage returns, and commas. The scanner recognizes 3 classes of tokens: *numbers*, *strings*, and *identifiers*. It converts each token into the relevant object, then passes it to *execution control* for execution by the JaM machine.

To understand what happens when the user types it is important to understand what the scanner recognizes, and what objects are generated for execution.

Numbers are of three types: integers, reals, and long integers.

```
 23   -465                          --Examples of integers.
-23.4   1.   0.00034        --Examples of reals.
-123456   1000000          --Examples of long integers.
```

Strings are any sequence of characters enclosed in balanced parentheses. For example:

```
(Hello. How are you)

(strings may consist
of more than one line
of information)

(strings may (contain nested (balanced) parentheses))
```

(strings may contain all manner of garbage @#$$$%%~~~&&**~%$$1233;;; except for unbalanced parens)

(Strings may contain as many as 32767 characters)

An identifiers is any token that is neither a number nor a string. For example:

.add
.print
abc
$$$
123a
abc.111
-.aaa

are all examples of valid identifiers.

Execution of objects has the following semantics:

number: the value is pushed onto the operand stack;

string: the string is pushed onto the operand stack;

identifier: the object *associated* with the identifier is *looked up* using the dictionary stack as a *context* (described later), and the associated object is in turn executed.

Examples

To see how this all works, several simple examples are given.

Example 1.

Consider the input:

123 456 .add .cvs .print

1. The scanner will recognize 123 as a number, convert it to an integer object and pass it to the JaM machine for execution, which will leave the integer value 123 on the operand stack.

2. In a similar way, 456 will be pushed onto the operand stack.

3. The scanner will recognize .add as an identifier. ".add" is known in the system dictionary as a *command* object which, when executed will replace two numbers on the top of the operand stack with their sum. The operand stack will now contain the integer object: 579.

4. The scanner will recognize .cvs as an identifier. ".cvs" is known in the system dictionary as a command object which, when executed will convert the object on the top of the operand stack into its string object equivalent. i.e. the integer 579 will be converted to the string "579", and put on the operand stack.

5. In a similar way, .print is a command object which will print the string found on the operand stack. In this case:

579

will appear on the screen.

Example 2.

The input:

```
(hello there  -- how are you) .print
```

will cause:

```
hello there  -- how are you
```

to appear on the screen.

Example 3.

When the JaM machine is first started, the system dictionary contains about one hundred definitions. Most of the definitions associate identifiers with command objects. These commands provide the primitive functions of the machine.

Included in the primitive operations are commands that allow new associations to be entered into dictionaries. The .def command makes associations in the *current* dictionary, which is the one on the top of the dictionary stack. For example:

The input:

```
(x) 100 23 .add .def
```

will create an association between the string object x and the integer object 123 in the current dictionary. Having made this definition, consider the input:

```
x 1 .add .cvs .print
```

The scanner will recognize x as an identifier, whose execution will yield the integer object 123, whose execution will, in turn, leave the integer 123 on the operand stack. When the rest of the line is executed, then:

```
124
```

will be printed.

Example 4.

The input:

```
(a)(hello -- how are you) .def
```

will define a to be the string object: hello -- how are you

The subsequent input:

```
a .print
```

will result in:

```
hello -- how are you
```

being printed.

Example 5.

All objects come in two varieties, *nouns* and *verbs*. This distinction is meaningful only to execution control. Execution of a noun object always results in that object being pushed onto the operand stack. Execution of a verb object depends on the type of the object, though for some objects, e.g. numbers, no distinction is made. Command objects are normally verb objects. Execution of a verb command object executes the command. Execution of a verb string causes the string to be scanned as input.

For example, in the input:

```
(average)(.add 2. .div) .cvx .def
```

the .cvx command will make the noun string object: .add 2. .div
into a verb ("eXecutable") before it is defined as the value of average.

Consequently, in the input:

```
123 456 average .cvs .print
```

execution of the identifier average will cause the string: .add 2. .div to be scanned and its tokens executed, rather than having it pushed onto the operand stack. The overall result will be that:

```
289.5
```

will be printed.

It can be seen from the above simple examples, that the user may assign values to identifiers, or assign procedures (executable objects) to identifiers. This ability to extend the JaM machine is the primary feature that allows complex systems to be built in the JaM environment.

To appreciate how this extensibility may be used, a more complete description of the primitive command set, and its relationship to the machine structure follows.

JaM Command Catagories

To describing the various JaM commands we will use a notation to indicate how each command works. Specifically, angle brackets enclosing an expression will be used to indicate a JaM object on the operand stack. For instance, <a><b><c> will indicate three objects on the stack, with <c> being the object on top, <b> next, and <a> beneath <b>. Certain letters will be used to indicate specific types of objects. <n> and <m> will represent numbers, <b> a boolean, <d> a dictionary, <a> an array, <s> and <t> will represent streams; and <x> and <y> will represent any kind of object. When a command returns elements, then is indicated by "=>" followed by the returned elements. The notation "--" is used to indicate that the command returns no elements. Commands normally remove their arguments from the operand stack, do the operation on the arguments, and then return results. Some commands, however, return their input arguments. The descriptions of the commands will indicate how each works.

There are roughly ten categories of JaM commands. These include:

Arithmetic Commands

The arithmetic commands provide for the basic arithmetic operations between mixed types of numbers. These commands take their operands from the operand stack, and leave the results on

the operand stack. Included are:

.add : <n><m> .add => <n+m>
.sub : <n><m> .sub => <n-m>
.mul : <n><m> .mul => <n*m>
.div : <n><m> .div => <n/m>
.neg : <n> .neg => <-n>
.cos : <n> .cos => <cos(n)>
.sin : <n> .sin => <sin(n)>
.atan : <y><x> .atan => <atan(y/x)>
.exp : <b><e> .exp => $<b^{e}>$
.log : <b><v> .log => $<\log_b v>$

Boolean and Relational Commands

These commands generate boolean constants by performing relational tests between numeric (or string) objects, and boolean operations between boolean objects. Included are:

.true : .true => <.true>
.false : .false => <.false>
.eq : <n><m> .eq => <.true> if n = m otherwise <.false> (also works on strings)
.gt : <n><m> .gt => <.true> if n > m otherwise <.false> (also works on strings)
.lt : <n><m> .lt => <.true> if n < m otherwise <.false> (also works on strings)
.not : <b> .not => <~b>
.and : $<b_1><b_2>$ .and => $<b_1$ AND $b_2>$
.or : $<b_1><b_2>$ .or => $<b_1$ OR $b_2>$
.xor : $<b_1><b_2>$ .xor => $<b_1$ XOR $b_2>$

Stack Manipulation Commands

These commands provide a set of functions that allow the user to manipulate the operand stack. These functions include facilities to duplicate portions of the stack, rearrange portions of the stack, eliminate portions of the stack, and count the entries on the stack. Included are:

.pop : <x> .pop => --
.copy : $<x_1><x_2> \ldots <x_i><i>$ .copy => $<x_1><x_2> \ldots <x_i><x_1><x_2> \ldots <x_i>$
.cntstk : $|<x_1><x_2> \ldots <x_i>$ .cntstk => $|<x_1><x_2> \ldots <x_i><i>$
.roll : $<x_1><x_2> \ldots <x_i><i><j>$ .roll => $<x_{(j+1) \bmod i}> \ldots <x_i><x_1> \ldots <x_{j \bmod i}>$
.dup : <x> .dup => <x><x>
.clrstk : $|<x_1><x_2> \ldots <x_i>$ .clrstk => |
.exch : <x><y> .exch => <y><x>

Stack Marking and Mark Manipulation Commands

These commands provide a set of functions that allow the user to mark the operand stack. These functions are used for keeping track of variable numbers of arguments on the operand stack. Included are:

.mark : .mark => <mark> (this command puts a Mark type object on the operand stack.)
.cnttomrk : $<mark><x_1><x_2> \ldots <x_i>$ .cnttomrk => $<mark><x_1><x_2> \ldots <x_i><i>$
.clrtomrk : $<mark><x_1><x_2> \ldots <x_i>$ .clrtomrk => <mark>

Execution Control Commands

This set of commands provides for control of execution. JaM has no "go to" type of command. Instead heavy use is made of "if - else", "looping", and "select" kinds of control mechanisms. Most execution control commands expect objects on the operand stack. These objects are executed as a function of other objects on the stack. For example: The .if command expects a boolean object and

any other object on the operand stack. If the boolean equals .true then the other object is executed, otherwise the .if command pops the object from the stack. The execution control commands include:

```
.exec       : <x> .exec => -- (executes object x)
.if         : <b><x> .if => -- (if b = .true then execute x)
.ifelse     : <b><x><y> .ifelse => -- (if b = .true then execute x else execute y)
.rept       : <i><x> .rept => -- (execute x -- i times)
.loop       : <x> .loop => -- (execute x until a  .exit command is executed)
.exit       : .exit => --  (exit from the current .rept, .loop, .dictforall, .arrayforall commands)
.stop       : .stop => -- (clear the execution stack.)
.singlestep : .singlestep => -- (put execution control into singlestep mode.)
.runfree    : .runfree => -- (take execution control out of singlestep mode.)
.quit       : .quit => -- (save virtual memory and exit to the operating system.)
```

Dictionary Related Commands

Dictionary objects are general symbol tables that may be used either as data structures or as part of the user's execution context.

As we learned earlier, the scanner, when it encounters an identifier, looks up the identifier in the dictionaries on the dictionary stack. What actually happens in this case is quite simple. The unknown identifier is first looked up in the dictionary on the top of the dictionary stack. If the identifier has a value in this dictionary, then the value is returned. If the identifier has no entry in this dictionary, then the search continues through each dictionary on the dictionary stack until the entry is found. At this point the value associated with the identifier is returned. The user has control over the contents of the dictionary stack via the .begin and .end commands, and therefore the user has control over his execution context. Later we will see how this control may be used.

When dictionaries are used as data structures, they are loaded, as objects, onto the operand stack. Various commands can then operate on these objects to store new definitions, or to retrieve definitions found in the dictionaries. Also a command exists for creating new dictionaries.

In the following definitions <k> and <v> can be any objects, but are used to denote the "key " and "value" respectively.

The dictionary related commands are:

```
.dict       : <i> .dict => <d> (dictionary with capacity of i entries)
.def        : <k><v> .def => -- (associates the value v with the key k in the current dictionary)
.del        : <d><k> .del => -- (deletes the key k from the dictionary d)
.load       : <k> .load => <v> (loads the value associated with k in the current dictionary)
.store      : <k><v> .store => -- (finds a definition of k in the current context and replaces that
                                   definition with value v. If no definition of k exists then the
                                   definition is placed in the current dictionary.)
.put        : <d><k><v> .put => -- (associates value v with key k in dictionary d.)
.get        : <d><k> .get => <v> (retrieves the value associated with k in dictionary d.)
.known      : <d><k> .known => <.true> if key k is in dictionary d <.false> otherwise.
.where      : <k> .where => <d><.true> if k is found in some dictionary d <.false> otherwise.
.clrdict    : <d> .clrdict => -- (clears all entries from dictionary d.)
.dictforall : <d><x>.dictforall => -- (puts <k><v> on the stack, and then executes <x>. This is done
                                   for every k,v pair in dictionary d)
.begin      : <d> .begin => -- (makes d the current dictionary on the dictionary stack.)
.end        : .end => -- (pops the current dictionary from the dictionary stack.)
.sysdict    : .sysdict => <systemdictionary>
.length     : <d> .length => <i> (replaces the dictionary with its current number of entries).
.maxlength  : <d> .maxlength => <i> (replaces the dictionary with its size).
```

Array Related Commands

Array objects are linear arrays of objects. Commands exist to create arrays, store into arrays, retrieve objects from arrays etc. Most of these commands either expect array objects on the

operand stack or return array objects on the operand stack.

In addition to their usefulness as data structures, execution of a verb array results in execution of each of its elements in turn. Procedures can be converted into an array form which, in some sense, corresponds to the compiled form of a procedure in other machines.

The array related commands consist of:

.array : <i> .array => <a> (new array of length i.)
.subarray : <a><i><j> .subarray => <a'> (a' is the subarray of a starting at position i and with length j.)
.aput : <a><i><v> .aput => -- (store v in the ith position of a.)
.aget : <a><i> .aget => <v> (get v from the ith position of a.)
.aload : <a> .aload => $<x_1><x_2>$ ... $<x_i><a>$
.astore : $<x_1><x_2>$ ... $<x_i><a>$ .astore => <a> (store x_1 ... x_i into array a of length i.)
.arrayforall : <a><x>.dictforall => -- (puts the contents of ai on the stack, and then executes <x>. This is done for every ai pair in array a).length : <a>
.length : <a> .length => <i> (replaces the array with its length).

Input/Output and Stream Related Commands

This catagory of commands deals primarily with string objects and stream objects. There exist primitive JaM commands to create streams, execute streams, read streams, write streams, and destroy streams. These commands include:

.print : <s> .print => -- (.prints the string s on the current output stream.)
.bytestream : <filename> <access> .bytestream => <bs> (this command creates a bytestream with the access characteristics represented by <access>. Here <access> = 1 for read, 2 for write, 4 for append -- or the sum of any of these. The created stream type object is left on the operand stack.)
.wordstream : <filename> <access> .wordstream => <bs> (this command creates a wordstream with the access characteristics represented by <access>. Here <access> = 1 for read, 2 for write, 4 for append -- or the sum of any of these. The created stream type object is left on the operand stack.)
.keystream : .keystream => <ks> (this command creates a keystream and leaves it on the operand stack.)
.killstream : <t> .killstream => (this command kills the given stream.)
.readline : <s> .readline => <line from stream><.true> (this command reads a line from the stream)
.readitem : <s> .readitem => <item from stream><.true> (this command reads a item from the stream) <.false> (if no more items in the stream)
.writebytes : <t><s> .writebytes => -- (write bytes in string s appended to stream t.)
.loadbcd : <filename> .loadbcd => -- (load mesa bcd and start.)

Attribute and Conversion Commands

These commands allow the user to determine the types of objects and to convert from one object type to another. This command set is not complete as yet, but the currently provided commands include:

.type : <x> .type => <NameOfType> (deliver the name of the type on top of operand stack. Current types include. .nulltype, .intergertype, .longintegertype, .realtype, .booleantype, .stringtype, .streamtype, .arraytype, .dicttype, .commandtype, .stacktype, .frametype, .marktype.)
.itype : <x> .itype => <typenumber> (deliver the number of the type on top of the operand stack. Current number assignments are:
nulltype = 0
integertype = 1
longintegertype = 2
realtype = 3
booleantype = 4

stringtype = 5
streamtype = 6
commandtype = 7
dicttype = 8
arraytype = 9
stacktype = 10
frametype = 11
marktype = 12

.length : <x> .length => <i> (length of: string (in characters); array (in elements); dictionary (in entries).)
.cvs : <x> .cvs => <s> (convert to string equivalent.)
.cvis : <x><s> .cvis => <s> (convert into given string space. This command will use s for all number and boolean conversions.)
.cvrs : <n><rdx> .cvrs => <s> (convert with radix to string equivalent.)
.cvirs : <n><rdx><s> .cvirs => <s> (convert with radix into given string space. This command will use s for all number and boolean conversions.)
.litchk : <x> .litchk => <.true> if a noun otherwise <.false>
.cvx : <x> .cvx => <x'> (convert into executable equivalent.)
.cvlit : <x> .cvlit => <xl> (convert into literal form -- works for strings, and arrays.)

Scanner and String Manipulation Commands

Commands in this group allow for string searching and manipulation. Also an interface to the JaM scanner exists in the ".token" command.

.token : <s> .token => <t><sr><.true>
<.false> (if token present then return .true and strip first token from given stream or string. Return remainder and token on the stack. If no token, then return only a .false.)
.string : <i> .string => <s> (<s> is a string of length i).
.length : <s> .length => <i> (replaces the string with its length in characters).
.substring : <s><i><j> .substring => <s'> (<s'> is a substring of s starting at position i for j characters.)
.putstring : <t><i><s> .putstring => <t'> (<t'> is the same as <t> except for the substring s starting at position i.)
.search : <t><s> .search => <t_{end}><t_{match}><t_{begin}><.true>(if there is a substring of t matching s)
<t><.false> (if no substring of t matches s).

.asearch : <t><s> .search => <t_{end}><t_{match}><.true>(if there is a starting substring of t matching s)
<t><.false> (if no starting substring of t matches s).

Graphics Commands

Commands in the Graphics group are supplimentary commands and are enabled by loading jamgraphics.bcd. Within this mesa module a display context stack is maintained so that transformations and other state information can easily be saved and restored. The .pushdc, .popdc and .initdc commands control this stack.

State information concerning the graphics device is held in an entity called the diplay context. and most transformation, clipping, and painting commands alter this state. Most drawing commands both alter state and use state.

.initdc :.initdc (Initializes the stack of display contexts.)
.pushdc :.pushdc (Pushes a copy of the current display context onto the display context stack.)
.popdc :.pushdc (Pops the current display context from the display context stack.)
.setview :
.translate :<x><y> .translate=>-- (Concatenates translation matrix with current transformation matrix.)
.scale :<sx><sy> .scale=>-- (Concatenates scale matrix with current transformation matrix.)
.rotate :<theta> .rotate=>-- (Concatenates rotation matrix (theta in degrees) with current transformation matrix.)
.drawto :<x><y> .drawto =>-- (Draws from the current position to the given position.)
.rdrawto :<x><y> .rdrawto =>-- (Adds x,y to the current position and draws from the current position to the computed position.)

.moveto :<x><y> .moveto =>-- (Sets the current position to the given position.)
.rmoveto :<x><y> .rmoveto =>-- (Adds x,y to the current position and sets the current position to the computed position.)
.drawbox :<llx><lly><urx><ury>.drawbox =>-- (Draws a box with the given lower left corner and upper right corner. The draw position is left at the lower left corner.)
.drawboxarea :<llx><lly><urx><ury>.drawboxarea =>-- (Draws a filled box with the given lower left cornerand upper right corner. The drawposition is left at the lowerleft corner.)
.drawspline :<x0><y0>...<xn><yn><n>.drawspline =>--(Draws an open ended curve through n given points. This curve is a natural spline consisting of n cubics.)
.drawcspline :<x0><y0>...<xn><yn><n>.drawcspline =>--(Draws a closed curve through n given points. This curve is a natural spline consisting of n+1 cubics.)
.drawblob :<x0><y0>...<xn><yn><n>.drawblob =>--(Fills a closed curve through n given points. This curve is a natural spline consisting of n+1 cubics.)
.drawcubic :<cx0><cy0><cx1><cy1><cx2><cy2><cx3><cy3>.drawcubic =>-- (Draws a cubic with the given coefficients.)
.beziertocubic :<x0><y0><x1><y1><x2><y2><x3><y3>.beziertocubic => <cx0><cy0><cx1><cy1><cx2><cy2><cx3><cy3> (Converts four Bezier control points to a parametric form of a cubic.)
.startpath :.startpath =>-- (Starts a path for the area generation machinery.)
.starteopath :.starteopath =>-- (Starts a path for the area generation machinery except that even/odd parity is used to determine interior.)
.enterpoint :<x><y>.enterpoint =>-- (Enters the given point into the current path.)
.entercubic :<cx0><cy0><cx1><cy1><cx2><cy2><cx3><cy3>.entercubic =>-- (Enters the given cubic into the current path.)
.newboundary :.newboundary => -- (Starts a new boundary in the current path.)
.drawarea :.drawarea => -- (Fills the interior of the areas comprising the current path. Also deletes the path.)
.erase :.erase => -- (Erases all area inside the current clipping regions.)
.setclip :<llx><lly><urx><ury>.setclip => --(Sets the current clipping region to be the given rectangle.)
.addclip :<llx><lly><urx><ury>.setclip => --(Adds the given rectangle to the set of clipping regions.)
.resetclip :.resetclip => -- (Resets the list of clipping rectangles to nil.)
.enableclip :.enableclip => -- (Enables the clipper.)
.disableclip :.disableclip => -- (Disables the clipper.)
.testnoclip :.testnoclip => .true if clipping disabled else .false.
.initboxer :.initboxer =>-- (Initializes the boxing machinery.)
.readboxer :.readboxer=> <llx><lly><urx><ury><dx><dy><dx><dy> (Reads the current box values.)
.boxcorners :<llx><lly><urx><ury><dx><dy><dx><dy>.boxcorners => <llx><lly><llx><ury><urx><ury><urx><lly> (converts box parameters to four corners)
.stopboxer :.stopboxer=>-- (Stops the current boxing calculation)
.testbox :<llx><lly><urx><ury><dx><dy><dx><dy>.testbox =>-- (tests a given box against the current clipping regions)
.testboxin :.testboxin=> .true if tested box is in.
.testboxout :.testboxout=> .true if tested box is out.
.paint :<paintfunction> .paint => -- sets the painting function (see bitblt)
.texture :<texture> .texture =>-- sets texture (4x4 bit pattern in low half of integer).
.touch :.touch=> <x><y> (yields x,y position of mouse at time when the red button is lifted. The returned values are in the current coordinate system.)
.mouse :.mouse=> <x><y> (yields x,y position of current mouse position. The returned values are in the current coordinate system.)
.setfont :<al-fontfile>.setfont=> (sets the current font.)
.drawchar :<c> .drawchar => (Draws a character at the current drawposition and updates the draw position to reflect the width of the character.)
.erasechar :<c> .erasechar => (erases a character at the current drawposition and updates the draw position to reflect the negative width of the character.)
.drawstring :<s> .drawstring => (Draws a string at the current drawposition and updates the draw position to reflect the length of the string.)
.getstringbox :<s> .getstringbox =><llx><lly><urx><ury><dx><dy><dx><dy> (Returns the bounding box of the given string.)
.getstringwidth :<s> .getstringwidth =><w> (Returns the length of the given string.)

Error and User Definable Commands

Commands in this group are used in the JaM system but either have no definitions or are given some default definition which is meant to be changed. Included in this group are all JaM errors, and some miscellaneous debugging commands.

When an error occurs in the JaM machine (e.g. stackunderflow), the JaM machine executes a fixed identifier for that error. It is the intent of this mechanism that the user define these names to be procedures that are appropriate to the user's application. The current list of error names is:

.undefkey : A dictionary lookup failed. when this identifier is executed the offending name is on the operand stack.
.longname : A file name passed to .run or .bytestream is too long (256 chars). The contents of the stack contains the offending command plus its input parameters.
.badname : is generated from an attempt to create a stream with the given name. This error can come from either the .run command or the .bytestream command.
.typechk : a command is executed which has been passed objects of the wrong type on the stack. Both the command and the original arguments are on the operand stack when .typechk is executed.
.dictfull : an attempt to define a new entry into a full dictionary has occurred. The offending command and the argument prior to the call are on the stack.
.stkundflw : an attempt to retrieve an argument from an empty stack has occured.
.syntaxerr : an isolated right paren has been found by the scanner.
.overflow : numeric overflow has occurred in scanning a number.
.stkovrflw : a stack has overflowed. This error may come from various parts of the JaM machine. When the error occurs, all stacks are stored into arrays and put on the operand stack in the order: opstack, dictstack, execstk.
.rangechk : some string or array operation is attempting to store out of bounds.

Certain debugging commands are provided in JaM. These commands are executed only in certain contexts. With these commands it is straightforward to build both tracing facilities and breakpointing facilities. The commands include:

.singlestep : This command puts execution control into single step mode. When in this mode, execution control puts each object (except commands), as encountered, onto the operand stack. It then executes the identifier ".step" which the user should have previously defined. After executing .step execution control executes the object that was on the operand stack, and continues.
.runfree : Takes execution control out of single step mode.
.interrupt : Is executed when the key (spare2--below the "_" key) is pressed. This command is defined to be equivalent to .stop. This command is used to get out of infinite loops.
.step : is used with .singlestep (see above) and has no default definition.

The Programming and Use of JaM

Because JaM is quite different from other programming languages, it is appropriate to give hints on how JaM is programmed and how to put JaM to good use. The following paragraphs will give some of these hints.

1. The JaM input language has very little syntax. This attribute is both good and bad. The lack of syntax is good because a uniform representation is attained. The lack of syntax is bad because the code is hopelessly unreadable. In JaM it is almost true that any line of code can do anything (given the appropriate redefinitions of identifiers). Because of this property of JaM code, it is desirable to do several things when programming. First, document each function as it is written (there are utility functions "/def" and "/xdef" to help with this). Second, use naming conventions for funtions that all belong to a given class (you have noticed the convention of starting each intrinsic command name with "."). The latter convention will allow a person reading the code to tell what catagory a function is in by its name.

2. Since JaM is a stack oriented machine the user must mentally keep track of the contents of the stack. It is easier to program JaM if each routine performs only one function and is very short. Normally a JaM routine should be at most one or two lines long. For example, suppose we wish to use the absolute value of a given number, then rather than introduce the code in line, it is desirable to write an "abs" function and then use that function e.g.

   (abs)(.dup 0 .lt (.neg) .cvx .if) .cvx .def

   Also if complex parameters are being passed to JaM control statements, it is better to name the parameters rather than have lines and lines full of parentheses. For example:

   (func)(-*some kind of test*- (---

```
---lots of stuff if true ---
---)
(---
---lots of stuff if false---
---).ifelse .cvx .exec).cvx .def
```

Should be written:

```
(func)(-some kind of test- trueaction falseaction .ifelse .cvx .exec).cvx .def
(trueaction)(--lots of stuff if true--).def
(falseaction)(--lots of stuff if false--).def
```

Using this style the person reading the program can visually capture the logic of the function without having to parse the branches of all the control statements.

3. The JaM machine saves its virtual memory when a normal exit is taken (by executing the ".quit" command). When JaM is reentered this virtual memory is restored and therefore all definitions are restored. This state saving property and the potential size of the virtual memory makes JaM quite different from other programming languages. Because of this difference JaM is used in ways that are unusual. For instance: It is quite reasonable to store permanent data in the JaM virtual memory. e.g. documentation, design information, notes to yourself, etc. This is done by creating and organizing dictionaries that deal with information in a way natural to the user. It is also easy to keep permanent procedural data bases in JaM which makes it ideal for design engineering applications. Large, complex data structures can be built and easily maintained within the JaM virtual memory.

4. The JaM virtual machine is written in mesa and is designed so new intrinsic commands, also written in mesa, can be easily interfaced (at run time) to JaM. This capability provides two advantages. The first is that the user can try ideas for functions in the JaM environment, and when performance becomes a problem recode the new functions in mesa and make them intrinsic. It also has an advantage in that mesa user's can interface suitable routines to JaM and use JaM as an executive for those routines. In the interface between JaM and Mesa, mesa programs can use the JaM virtual memory as well as the JaM interpreter and execution machinery. Some of these capabilities will be discussed in the last section of the manual.

(to be continued)