D
(A Manual)

Wolfgang Nonner and Alex Peyser

September 27, 2011

Contents

List of Tables

Chapter 1

INTRODUCTION

The *D machine* is a general purpose computer and uses a native language that we call *D*. The D machine is a virtual computer implemented by a program that is coded in C following the 2001 Posix standard. D machines cooperate in various configurations across a network or cluster of physical hosts.

1.1 A computer of objects

A D machine works with quanta of information that we shall call 'objects' (adding a usage to an existing term, rather than inventing something like 'quarks'). An object comprises a **value** (the essence, from your point of view) and a **description** that carries a thorough specification (transparently used by the machine, but also accessible to you). Objects hold pieces of information that are familiar to you and 'understood' by the machine. Objects thus provide a common ground.

D sets out from a few varieties of **simple object**. Among them, *name*, *numeral*, and *operator* are the workhorses. Simple objects serve as the building materials of **composite objects**. There are several composite varieties, such as the *list* and the *dictionary*. Composite objects themselves can become elements of higher composite objects, and so on. There is no logical limit to the internal complexity of composite objects that you

create. With regard to quantity, anything from a single measured sample to a data base of the accumulated experimental results of several years can become one object to the D machine.

Objects can be worked on as data, or they can instruct the machine how to work on objects. 'Datum' or 'instruction' is a changeable attribute of information, not a fundamental distinction. One and the same form fits datum and instruction. Hence, these two varieties of information can be combined into composite objects that represent both **passive and active properties** of the models of reality that you construct in your computers.

D retrieves objects by **association**. Collections of objects that are mutually associated in some sense can be created in the form of composite objects. Composite objects that hold associations of objects are **lists** and **dictionaries**. Lists hold a linear array of objects that are accessed randomly or sequentially through a numerical index. Dictionaries hold an array of paired entries, of which one is a name, and the other any object that is associated with the name. References within dictionaries are made through the name. Lists and dictionaries are dynamic: entries are made and re-defined at any time.

Albeit a logical entity, the value and the description of a composite object are physically distinct. When you logically designate a composite object, you physically refer to a description of the object. By physically duplicating only the description of a composite object, multiple logical copies of the composite object can be created economically and used in multiple associations.

D lets you create composite objects that comprise a subset of the value of a parent composite object. These **children objects** do not receive a duplicate of the parent's value: they share one and the same original. Children objects are combined with other objects (including children of other composite objects) into new composite objects in unlimited cut-and-paste operations. Altogether, these techniques, transparently based on object descriptions as the representatives of values, let you create multiple specific access schemes to a body of information. Thus the capability of building data bases is innate to D.

The most common method for the retrieval of a D object is through an associated name, a time-honoured practice. Upon every reference to a

name, however, the D machine determines the **currently associated object** through an ad hoc search of the vocabulary in use. This vocabulary is subject to change. Names can be defined or re-defined in each dictionary, and dictionaries as a whole can be moved in and out of use. The machine can learn new terms. It can react intelligibly to terms, much like craftsmen of different trades act upon their specific readings of a blueprint. Furthermore, by switching among contexts defined by dictionaries, the machine can randomly attend to a variety of tasks without confusing their matters. The instructions to control these capabilities are very simple.

A D machine at work *feeds on objects*, consuming them one at a time. Every submitted object is dealt with immediately according to the rules of the **reverse Polish notation** (RPN, as familiar from Hewlett-Packard calculators). A submitted object either is transferred to an operand stack to serve as operand, or initiates one or several operations. An operation consumes some or all of the objects accumulated on the operand stack, and in turn may push result objects on the stack, ready for being processed by subsequent operations. By an ironclad design rule, operations are unaware of their precursors or followers: that is your exclusive privilege. Information passed from one operation to another by the implicit use of the stack needs not be specified over and over. This keeps the code lean and the focus on the action. Perhaps the most important windfall of RPN is that, for any single object that you feed the machine, you know the exact consequences and the exact instant when these occur. No high-level syntax, however well you keep abreast with its twists, can substitute for this knowledge.

The D machine accepts objects in two forms: **text** or **binary**. It recognizes automatically which form is being presented (by inspecting the description of the composite source object). When working from text, D automatically translates text tokens into binary objects before submitting them to its internal mill of binaries. This front end makes obsolete intermediaries such as command line interpreters, batch processors, compilers, linkers, or loaders of executable files, and spreadsheat and data base front ends, together with their many idioms. (There goes an industry!) Binary objects are readily converted into their text form and passed around among different platforms. In effect, one *lingua franca*, D, serves to concisely express and communicate all kinds of datum or instruction.

As a language of action, D thrives on verbs, here called *operators* and *procedures*. **Operators** invoke hardware code. Some operators do jobs as small as those of hardware instructions of the host machine, but the majority provide services that require many host instructions. Polymorphic D operators accept diverse kinds of object to work on, and tune themselves using information in the object descriptions. For instance, the *add* operator takes any combination of numeral types, performs the addition in several possible forms of arithmetic or precision, and accommodates all permutations of scalar and array. Old and new undefined values are recognized and propagated into the results without need for explicit exception handling. Furthermore, these operators make full use of your hardware. When multiple processors exist in a computer, they all can be used without writing extra code. Some operators can use an entire cluster of host computers, again without extra programming. In these ways, your plan can be formulated in D without attracting the usual cloud of confusing, albeit necessary, second thoughts.

Versatile operators keep D programs concise and comfortable to write. Furthermore, D lets you define new operators in the form of **procedures** (lists of objects to be executed). The overhead of composition is minimal. In the text representation, a pair of { } brackets wrap a set of objects that constitute the procedure (which is an active list); a subsequent operation usually gives the new tool a name. Procedures being executed can use the stack for their operands and results as do the operators. Thus, there is no formal difference between references to an operator and those to a procedure. This continuity fosters the fine-grain decomposition of instructions, and, together with explicit and well-thought names, produces structured code that is intelligible to person and machine.

While executing objects, the machine **controls its own operation** much like hardware that executes its native code. Hence, D can implement functions of operating systems, shells, graphical interfaces, debuggers, menus, tasks, or whatever layers of instruction are desirable. Much of D's capacity of self-organization flows from a capacity sheepishly excluded from conventional programming models: data and instructions are formed from one and the same set, objects. Thus the D machine can compute 'instructions' like it computes 'data'. This capacity is a trademark of life, from the cell chemistry to the human ability to reflect on your own thinking. No matter what your problems are: computed in-

structions offer solid, elegant, and sometimes unique solutions to problems that range from unspeakably boring to hair raising.

Defining a problem in D involves developing a specific vocabulary of objects, and in this sense a new language. D may be paraphrased as **a language to make languages**. Diversified vocabularies are one strength of the D machine; a strong stomach is another. Almost never will you have to write an ad hoc interpreter of your new idiom, because the secondary language code can go right through the existing D mill. D code resembles the 'vectors' constructed by a molecular biologist: it achieves the desired product not by creating a new, but by exploiting an existing, machinery for expression.

D likely differs from most programming techniques you have met because it does not use a 'high-level' syntax. Rather than rules, D gives you verbs (*operators*) to learn. D, in essence, is an unlimited collection of verbs. You can work through D's resident verbs by installment, ignoring those you do not need, gearing up as you go, and winding down as your project matures. You can invent your own verbs, use them, and discard them as you move on. Learning and re-learning of verbs and their usage is eased by the extreme simplicity of form. D code flows smoothly because it uses human words rather than computer expletives. The other surprise may be that, underneath its easy manners, D hides a Laconian.

D is **robust**, because it is simple and because it works with objects that include a thorough specification. Its operators, for example, will choose either to adapt to their operand objects or to reject the given objects as unsuited for the attempted use (this summarizes most of the safeguards necessary to let the machine steer clear of crashes). An able defender of its own integrity, the machine abstains from questioning your designs in terms of selfish syntactical rules. Instead, D respects you: it accepts every word that you present and executes this word **verbatim**. This may remind you of the ways of a fool, but it is also reminiscent of the method of Socrates: the machine dares you to follow through what you conceive. You are rewarded by solutions that work, convince, and are delivered with egg-laying promptness.

1.2 Talking – Thinking – in Objects

The widely used languages, Fortran, Basic, C, or Pascal, use a multi-level syntax: tokens form expressions, expressions statements, statements functions, and functions programs. D uses a syntax of objects and requires no rules of composition beyond those of constructing objects.

You may wonder how D organizes objects into instructions. A comparison of phrases in C and D shows that the method of D is simple, consistent, and very generally useful.

The C statement: **y = 9.8 * exp(-x/tau);**

could read in D: **/y 9.8 x neg tau div exp mul def**

The C code assigns the result of an algebraic expression to a variable and neatly concludes the statement with a semicolon. The D code is an open-ended string of objects. Like you walk by putting one foot in front of the other, the D machine computes by executing one object in the string after the other:

/y pushes a name object of value 'y' onto the operand stack; the '/' gives the name the *passive* attribute, designating it for use as an operand (a literal)

9.8 pushes a numeral object of value '9.8' onto the operand stack (thereby 'executing' the numeral)

x looks up the object associated with the name 'x' (a numeral in our case) and pushes a copy of it onto the operand stack (a name without the '/' prefix receives the *active* attribute, designating it for use in a dictionary search)

neg looks up the object associated with the name 'neg', which is an operator; invokes the operator, which negates the value of the top element of the operand stack (the copy of x)

tau pushes the object associated with 'tau' onto the operand stack (you got the idea)

div resolves to an operator, which divides the next-to-top element of the operand stack by the top element, removes the divisor and divident from the stack, pushes the quotient

exp another operator, which replaces the top element of the operand stack by its exponential

mul an operator, which multiplies the top two elements of the operand stack (9.8 and the exponential) with one another, replaces the factors by the product

def this name resolves to an essential operator, which associates the object at the top of the stack (the numeral result of the expression) with the name ('y', now the next-to-top element of the stack), and places the new association pair into the current dictionary for future reference.

D code thus builds on operands and operators held together by an invisible glue, the *reverse Polish notation*. The rule is minimal: an operator expects that the operand stack contain a sufficient number of suitable operands. Beyond that, there are *no other formal constraints* on the order or choice of objects that form a D script.

The C code has the appeal of high-school algebra: sort of familiar. D uses the more elegant and general concept of operators. Since elegance can be measured by the frugality of means, let's count: the C example

needs twelve tokens, two more than D; C uses six kinds of token, D three. The advantage is on D, albeit small. We will note below that the distance grows steeply even with mild increases of difficulty. D travels on foot where C gets only with heavy gear.

The C code in this example is shorter to type than the D code, because it employs special characters as shorthands. Good D style generally abstains from shorthand in order to keep things clear and consistent (shorthand systems have a knack to befuddle their inventors). You will see below that D lets you spell out in full what is needed for clarity, and nevertheless allows you to be more concise than in shorthand C code.

Having looked at algebra, we may ask: how does D express a control statement? For instance, C implements a *for* loop through a special grammatical construct called a *for-statement*. D builds a *for* loop through an operator and does so without grammatical ado. Thus, to form the sum of the integers between 0 and 100 by brute force, you may write

In C: `for (k = sum = 0; k <= 100; k++) sum += k;`

In D: `/sum 0 0 1 100 { add } for def`

The *sum* is defined as the effect of the *for* operator and its operands (i.e. 0 1 100 { add }) onto a value seeded on the stack, 0. *for* executes the loop body, provided in the form of the procedure, { add }, once for each value from the initial 0, by steps of 1, to the limit 100. The current count is passed to the procedure by pushing it on the operand stack. The procedure in our example contains a single operator, *add*, which adds the current count to the running sum maintained on the stack.

The C *for* statement is compact, as it bristles of shorthands. Yet it needs 20 tokens, whereas the D statement has 11. The C code uses nine different kinds of token, the D code, four.

To prime an array x of n real numbers with zeroes you may write

In C: `for (k=0; k<n; k++) x(k) = 0.0;` (20 tokens)

In D: `0.0 x copy` (3 tokens)

There is no need not look for the champion here. D uses an intelligent operator that determines automatically the dimension of the object it is working with and controls an internal loop accordingly. In the C construct, you are responsible for the count, and an error in the limit variable can cause a crash; you also have to reckon with C's expletives, which please a compiler but hardly a human.

This example raises the question of how fast the intelligent D operators execute. Obviously, they have to do a lot of object checking. On the other hand, D operators absorb the innermost loop when working on whole composite objects such as arrays, and then execute as fast as host machine code can do. Only small-grain number crunching will be significantly slower. But does this really matter? Science is change, and the economy of instructing a computer for a new twist of science precedes considerations regarding the speed of computation (consider: programming costs **your** prime time, whereas execution ties up a computer and, when numerically extensive, often can use many boxes around the clock). Furthermore, after stable D code has evolved, it always can be speeded by replacing bottlenecks by fast ad hoc operators written in C (there exists a mechanism for doing just that). This still is by far less work than developing the entire project in a language like C.

Moving up in syntax, let's cast the exponential expression from before as a function, such that x and tau are submitted and the expression value returned:

In C:
```
float myexp(tau,x)
float tau,x;
{
return(9.8 * exp(-x/tau));
}
```
(27 tokens)

In D:
```
/myexp {
div neg exp 9.8 mul
} def
```
(10 tokens)

You associate the name *myexp* with a procedure, whose body is enclosed between {}. When you enter this code, the objects in the procedure body are not executed: they are translated into their internal form and stored for later invocation (the D analog of 'compilation'). Invocations of this function/procedure could read:

In C:
```
y = myexp(x,tau);
```

In D:
```
/y x tau myexp def
```

Both languages provide the means for partitioning code into re-usable tools (functions in C, procedures in D). Are things in C and D really that similar? — They are not.

Both languages use a stack to transfer arguments. C functions can take many arguments and return at most one value, whereas D procedures can return many values. What might seem to be a small limitation of C opens a door for confusion because multiple results are communicated via pointer arguments whose direction of use is not obvious from the syntax. D deals with the two directions of data travel orthogonally.

The compiled C function is linked into a program. The function thenceforth stays as is, subserves the program, and can be invoked only from the context of the program. The D procedure, in contrast, is assimilated as an individual object into the machine. Any D code already present, passing through, or assimilated thereafter can use the procedure or be used by it. Moreover, the procedure thus assimilated is not forever cast as

is, for two reasons: (1) because you can edit the procedure in memory, (2) because you can replace the entire procedure associated with a name by a new one: the name-object associations are changeable. (Even intrinsic operators do not 'own' their names; hence, there are no 'reserved words' in D and, conversely, you can substitute or expand system operators by procedures that you associate with the former operator names). In effect, whereas the elements of a C program are rigid and welded together, those of a D process remain ductile and able to form new connections.

Although D objects can assemble ad hoc in the machine, their relationships are tightly controllable. The method is encapsulation. The following code invokes a procedure twice, each time providing it with a different context by executing it with different current dictionaries:

```
dict_A begin that_procedure end
dict_B begin that_procedure end
```

that_procedure can retrieve objects from all dictionaries currently on the dictionary stack. If it defines or re-defines objects in the course of its execution, these changes are made exclusively in the dictionary placed on top of the dictionary stack through the *begin* operator (and later removed by *end)*); this holds true for the entire dynamic context of *that_procedure*. The current dictionary, hence, provides a semi-permeable capsule around the procedure. The invoking program can put objects to be used by the procedure into the capsule, read results deposited in the capsule, or may leave the capsule alone as a private space of the procedure. Since any number of capsules can be maintained, the procedure can be used randomly in varying contexts without a risk of confusion.

Encapsulation is a responsibility of the caller. This greatly simplifies the writing of procedures, because no attention needs be given to the scope of their object names: you can write along as if there was no possibility of interference with other code. This technique is the opposite to C's approach to control access, where the lexical scope is determined by declarations contained in each function. The encapsulation technique is used by operating systems, to isolate the system itself from user processes and one user process from another. It is the method of choice in all systems that do not *a priori* restrict the players.

D extends encapsulation to the flow of control. You can execute code with the provision that control can return immediately to the calling code, cutting through a nest of pending procedures or loops. Consider as an example:

```
{ my_program } stopped
```

The operator *stopped* invokes the procedure operand, which contains a reference to a user program. If *my_program* executes the operator *stop* upon recognizing a severe problem anywhere within its dynamic context, execution resumes with the object following *stopped* (a boolean object is returned on the operand stack to inform the caller about the kind of termination). D provides a hierarchy of such escape mechanisms by which unpredicted events trigger orderly retreats to prepared positions. Most procedures can be written on the assumption that things go well, because surprises are passed from their discoverer straight to the supervisor of the context rather than bubble back through a chain of command where they would require attention at every intermediate level.

We now turn to composite objects. Such objects (and nests thereof) are not 'declared' like a C function or structure. D operators let you *build* such objects from constituent objects, or, conversely, *dissect* a composite object into smaller entities for specific uses. D objects, hence, are inherently dynamic. The following examples produce some composite objects:

```
1: 100 /w array
2: (This is a string)
3: <s 120. 40. 28. 1e5 29.546 1 -99>
4: ( a b 100 (HOHOHO) )
5: 12 dict 130000 list
6: { (Honni soit qui mal y pense\n) toconsole }
```

The objects are: (1) an array of 100 16-bit integers holding unpredictable initial values; (2) an initialized string (i.e. array of byte integers); (3) an initialized array of single-precision floating-point numbers; (4) a list of the objects created by the D code enclosed between brackets; (5) a virgin dictionary for up to 12 associations, and a list initially containing 130,000

null objects; (6) a procedure that when executed writes a bonmot on the console screen. The composite objects thus created populate the top 7 positions of the operand stack.

Composite D objects include arrays (a set of numeral values that share the same characteristics), and lists (a set of arbitrary objects). Lists and arrays can be created in a fashion that defines all elements of their value, or with initially undefined or null elements. Elements of these objects are accessed through an index. Alternatively, a composite D object is built in the form of a dictionary, where each element is associated with a name for reference. Dictionaries are created empty and are filled explicitly by subsequent operations.

As an example of composite object dynamics in D, consider a family of recorded traces from an electrophysiological experiment. Some subsection of each record was obtained while an agent was being applied and thus reports the time course of the agent's action. Assume that in one record the block of 200 samples that starts at index 1750 contains the episode of interest. To make a new object from this block, use

```
/onresponse trace 1750 200 getinterval def
```

The *getinterval* operator creates a new array object that represents the specified subset of the values contained in the array associated with *trace*; we associate the subset object with *onresponse*. Objects representing time courses of drug action from many records are grouped together into a family object by:

```
/responsefamily [ onresponse ... ] def
```

The family then may be submitted as an entity to analysis or picture taking. Note that the included episodes need not occupy the same subset of samples in each record: they can start anywhere and be of arbitrary lengths, because the children objects formed about them automatically inherit this information and carry it on to operators, which will use it to focus their effects. These objects and their list form a new and more specific system of reference for your data.

On another occasion, you have received, as mail sent by a colleague, a text file of one-column tables of floating point numbers. You want to submit this material to an analysis that you have developed on your D machine. All you need to do is use the text editor that is associated with your D machine (*emacs*) to put 'vectors' like

/RateConstants <**s . . .** > **def**

around each table that you wish to package. Then, back from the editor, you use a browser window (called *TheEye* and established when the D machine is brought up) to select the edited file, and a command window (called *DVT macros*) to select the action *Load*, and hit the function key 'F1' on your keyboard. The selected file is read into a temporary string buffer and the string is interpreted by the D machine. The result is that your current dictionary receives a new entry, *RateConstants* that is associated with a single-precision array of floating point numbers that is stored in the memory of the D machine.

Let's step back and re-consider what happened here. We invented a notation to present data to the computer, a language so to speak. That language happens to be D. By this trick, we assimilated the data into the machine without having written a single line of classical I/O instructions. The technique is attractive because the grammatical overhead of D is so small.

Now, assume that these data came in non-standard units related to physical quirks of the colleague's recording apparatus. For re-calibration, you include, after the data and in the same text file, some D code that transforms the raw arrays into the proper units, for example:

(RateConstants . . .) { **1.745e-6 mul pop** } **forall**

The *forall* operator applies the calibration procedure to all arrays whose names are included in the list (enclosed between [ ]), and the multiplication operator in the procedure scales all elements of each array (evidently, D is not verbose).

Whenever you will go back to this data file, you will get properly cali-

brated data without sacrificing the original or losing track of your additional calibrations. With minimal ado, you have converted this data set into something that takes care of itself: an object comprising both 'data' and 'instruction', self-calibrating data, so to speak.

A natural form of organizing larger collections of objects is the **tree**. A tree starts from a dictionary or list that holds simple or other composite objects; these composite objects, again, can hold composite objects, and so on to any level of nesting. Since trees are a very useful form of organizing objects, D provides operators to move entire trees among media. For instance,

```
SolutionsBook (path) (filename) writeboxfile
```

puts the tree that roots in the list *SolutionsBook* into a file. This list contains a collection of all solution descriptions ever used in your lab, probably in the form of one dictionary per solution; each dictionary has a number of standardized entries, including lists of stock solution names, their indices in the book, and pipetted volumes. Altogether, moving this whole structure piecewise would be very cumbersome, and moving only changed or added pieces would be error prone. The *writeboxfile* operator packs a copy of the binary objects (including value) of the entire tree into a box object and saves it as a whole. In order to load this tree into a D machine for further work, use

```
/SolutionsBook (path) (filename) readboxfile def
```

This unwraps the tree of objects and returns the root object of the tree that was folded into the box, and associates it with a name in the current dictionary. You now can look up individual solutions or add new entries to the book.

A procedure (provided in a general tool library), *tofile*, lets you translate a tree into its text equivalent and save that text in a file. The converse procedure, *fromfiles*, executes the D code contained in a text file and thus creates in the VM all objects that are defined in the text file. You can also use *tofile* to prepare a file containing a collection of D objects that

you wish to submit in toto to a PostScript engine for a picture. Thus, objects organized as trees can be moved and converted efficiently with a few smart operators or procedures.

In the recent examples, no equivalent C phrases have been listed, because C lacks the intrinsic means to hold its ground. The means, though, can be created in C, by composing a virtual D machine. This explains how D came into existence – and to a name.

This concludes our first tour of D. The following chapters will give a reference of the machine model and operators. The best way to learn D, as always, is by example. The D machine comes with a stock of useful D objects in text files. These implement basic utilities that you likely will use in most work on the D machine. They include operator-like extensions of the D machine in the form of procedure libraries, including a mouse-operated universal browser that lets you inspect and select any information organized by the D machine (called 'TheEye'). These use some of the capacities that set D apart from other language models, and thus may serve you also as a grab bag.

A D machine is embedded in the operating system of its host. In particular, it cooperates with a resident editor (*emacs*) and countless other useful programs of the host that you do not want duplicate in D. The current D machine is designed for use in a networked cluster of hosts. Multiple D machines can exist in each host and communicate with one another and across hosts for distributed processing. In fact, there are three flavors of D machine, one slightly specialized to communicate with you (the D virtual terminal, *dvt*), one slightly specialized to work with other D machines (called a D node, *dnode*), and one specialized to do linear algebra using many machines in a cluster (*dpawn*.

Chapter 2

THE MACHINE

We describe here the concepts and parts that define the generic D machine. Actual D machines have additional features by which they specialize for different tasks within a group of D machines that you set up for your work. The specializations of the D machine and their cooperation in a group will be described in Chapter 3.

2.1 The Objects

Simple objects	**null**
	numeral
	operator
	name
	mark
	boolean
Composite objects	**array**
	list
	dictionary
	box

Simple objects are unique, self-contained quanta of information. Com-

posite objects may be or may not be unique, and always consist of two separate parts: information that describes the object and information that represents the value of the object. Simple objects are analogous to small pieces of merchandise that can be traded over the counter. Composite objects are analogous to real estate. When you pass a composite object via the operand stack to an operator, you pass the information that describes the object, you do not pass the value. Likewise, when you trade your house, you do not put it physically on the negotiation table: you bring a copy of the deed.

- The **null** object fills a space and serves as a void. Some types of null object serve as the vehicle of a simple value that the D machine does not operate on but that it needs to opaquely pass around among operators (e.g., the handle of a network socket).

- The **numeral** object comes in several types of number representation and range. The numeral types comprise four ranges of signed integer (*byte*, *word*, *long*, and *extended* of 8, 16, 32, and 64 bit), and two precisions of IEEE real, *single* and *double* (32-bit and 64-bit). Operators that work on numeral objects use the intrinsic type specification to determine what arithmetics and conversions are required. Where numeral types need to be specified explicitly (e.g. when creating an array), this is done by (passive) name objects whose initial character is a type mnemonic, e.g., /w or /Word.

- The **operator** object represents an operation intrinsic to D or provided in user-created libraries of extrinsic operators. An operator's value is an array of host machine code. For reasons of sanity, D does not *provide* operators that let you operate on an operator's value. (For reasons of principle, D does not *prevent* you from writing extrinsic operators that do just that.)

- The **name** object's value is a compacted string. Currently name values are restricted to maximally 29 significant characters. Allowed characters are letters (upper and lower cases are distinguished), digits, and *underline*. The first character of a name must not be a digit. When a name object carrying the *active* attribute is executed, the current vocabulary is searched for an object associated with this name.

The vocabulary is defined by the dictionaries currently held on the dictionary stack. A fast search algorithm, employing hashing, performs this critical task (most searches in a dictionary conclude after a single name comparison regardless of the size of the dictionary). Name objects carrying the *passive* attribute provide their value as a literal (for instance, for associating the name with some object).

- The **mark** object is explained by its name.
- The **boolean** object's value is either 'true' or 'false'.
- An **array** value is formed by a linearly ordered set of numeral values of identical types. A **string** is an array of byte-integers (note that character codes are *signed* byte integers). Elements of arrays are referred to by index; the index of the first array element is zero.
- A **list** value is formed by a linearly ordered, generally mixed set of (whole) simple objects and/or (descriptions of) composite objects. **Procedures** are *active* lists (see below), of objects to be executed. Because the list value, again, may contain lists or dictionaries, you can use lists to build nested high-order structures. Value elements of lists are referred to by index; the index of the first list element is zero.

  Note the difference between arrays and lists: arrays pack numeral values of the same type, not objects, whereas lists contain objects. Although you can make a list of exclusively numeral objects, an array will be preferable for organizing numerals of identical types, because the array stores the numerical values in a more compact form and because mathematical operators deal much more efficiently with array than with list values. In particular, many mathematical operators when working on arrays optimize cache memory use of the host machine and spread the work over multiple threads to use all processors in multi-processor Linux boxes; these services, which greatly speed numerical computations, are provided transparently.

  When you duplicate an array or list object under another name or when building a list, or when you create a child object that represents a subset of an array or list value, the respective operators

create a new description, but they do not duplicate elements of the value. Hence the new objects share their values with the original. Changes made to a shared value (by modifying the value of one of the sharing objects) will affect all sharing objects.

- A **dictionary** value is formed by a set of association pairs, of which one partner is a name, and the other partner, any object. If the association is with a composite object, that object is represented by a description rather than the value. Associations are constructed dynamically, through operators. Dictionary entries are always made or retrieved through a name key, rather than a numerical index. The information contained in a dictionary is organized for quick retrieval by name. This organization precludes access via an index or subpartitioning into child dictionaries.

- A **box** value is composite but its elements are not accessible for peaking and poking. A box can be marked by operators in VM space around successively created objects; the objects in this box can be selectively discarded to free their VM space. Specially typed boxes in VM *can* be maintained for composite values that are privately created and used by extrinsic operators, but are inaccessible to intrinsic D operators (mind *Pandora*). A single operator folds a tree of VM objects into a temporary box and saves the box in a binary file; another operator retrieves the tree and appends it to the objects in VM.

Objects carry several **attributes**, some of which are changeable by operators. In composite objects, these attributes are part of the description; objects that share a value thus need not have identical attributes. An object is either *passive* or *active*. This attribute is changeable and directs the use of the object, as datum or instruction. The *tilde* attribute is converted to the *active* attribute when tilded *name*object is executed. The value of a composite object given the *readonly* attribute can only be read, but not modified (the *readonly* attribute cannot be reversed, but you can have multiple descriptions of the composite object, some carrying *readonly* and some not carrying the attribute). If the creation of a composite object involves the creation of a value, this object receives the *parent* attribute; objects created to share a subset of a parent object value receive the *child* attribute.

2.2 The Text Representation of Objects

Objects are formulated in text using the printable subset of the USASCII character set plus the control characters, 'space', 'newline', and 'carriage return'. 'Newline' and 'carriage return' are equivalent in terminating a line. Object tokens are separated by **white space**, a single or several successive control characters. **Comment**, a '|' character and its followers up to the end of the line, also act as white space (they are otherwise ignored by the machine). In addition to the separation by white space, objects are delimited by any of the special characters, ( ) [ ] { } < >, which serve to segregate enumerated or constructed contents of composite objects.

A **numeral** starts with a digit, sign, or '∗' character. An *integer* numeral consists of optional sign and a sequence of digits, optionally concluded by a type specifier. The range of the integer is determined by the specifier ('b/B' for byte, 'w/W' for word, 'l/L' for long, 'x/X' for extended). These integers are signed and stored as 8, 16, 32, or 64 bit values, respectively. A *real numeral* is distinguished from an integer by the presence of at least one of: a fractional part, an exponent, or a 's/S' or 'd/D' specifier. The exponential part consists of one of the characters 'e/E' immediately followed by an optionally signed sequence of exponent digits. Real numbers are stored with 32-bit (single) or 64-bit (double) precision. Numerals that do not carry a type specifier will by default be of type /X (when lacking real-number features), or else will be of type /D.

Numeral values can be *undefined*. Mathematical operators using an *undefined* value as an operand simply return *undefined* results. Thus, arrays that contain invalid data in a few positions can be submitted to computations without special attention to such exceptions, a little luxury that saves tedious programming. The text representation of an undefined numeral is an '∗', which may be followed by a decimal point (to indicate an undefined real numeral), or by one of the type specifiers (if that helps make a point).

A **name** is a sequence of letters, digits, or 'underline' characters (the leading character must not be a digit). Names are limited to 29 significant characters (longer names are silently truncated). Upper and lower case are distinguished. A name preceded by '/' is given the *passive* attribute; otherwise the attribute is *active*. If you are familiar with PostScript, please

note the difference that D does not let you use characters like '?!@#&$' in names.

Simple objects other than numeral or name have no direct text representation. Such objects are specified through associated names: 'null' for the **null** object, 'true' or 'false' for a **boolean**, and names such as 'add' for **operator** objects. The **mark** object and a complementary list operator are accessed through the '[' and ']' special characters, making list *construction* formally similar to array or procedure *enumeration*. Note, however, that the objects that are enclosed between the '[' and ']' operators are executed at the time when this text is interpreted; their *results* (which are accumulated on the operand stack) become the body of the list finally created by ']'. In this way passive-list contents are always computed rather than enumerated.

All classes of **composite** object can be created through operators, but only **arrays** and active lists (**procedures**) can be defined directly in text form by enumerating their elements.

Enumerated **arrays** are composed of numeral values enclosed between '<' and '>'; the array type is specified by a letter that immediately follows the '<' character and applies to all numerals in the array. Arrays of the byte type (**string**) may be defined also by alphanumerical enumeration enclosed between '(' and ')'; character codes that cannot be designated directly in such an enumeration are represented by backslash sequences:

\n	newline (line feed)
\r	carriage return
\(	left parenthesis
\)	right parenthesis
\\	backslash
\ddd	ASCII character code ddd (octal)
\ newline	no character - both are ignored

Elements of an enumerated **procedure** are embraced by '{' and '}'. Objects so embraced are not executed: they are translated into their internal representations and simply stored as the value of the procedure object (compiled, so to speak). In particular, no dictionary searches are per-

formed for active names at this time. The objects contained in the procedure value will be executed only through invocations of the procedure.

Rather than enumerating the value of a procedure between '{...}' you can construct a list and assign the list the 'active' attribute to make it a procedure. This is a method by which procedures are constructed on the fly by other D code. A shorthand is offered (with apologies) for such construction. It involves giving objects the attribute *tilde* through the '∼' prefix. The following long-hand and short-hand constructs have identical results:

```
( x /y mkact /add mkact ) mkact
∼( x ∼y ∼add )
```

They build a procedure that adds the value of *x* at the time of building to the value of *y* at the time of execution (note when the dictionary searches for *x* and *y* occur). When you refer to an active name during list construction, a dictionary search is done and the value of the matching object is placed into the list under construction. The *tilde* attribute suppresses the dictionary search at this time; instead, the active name itself is inserted into the list (the future procedure). The *tilde* attribute of the '[' operator tells the matching ']' operator to give the list the *active* attribute when closing up the list construction (rendering the passive list a procedure). There are other uses of the *tilde* attribute that will be described in 2.6.6.

A constraint. Arrays and procedures defined by enumeration have to be completely contained within a single portion of text submitted to the translator (such as a console phrase or a string whose content has been derived from a file). The translator is used implicitly by the D executive (mill) described below.

In summary, the text form of D code is translated with minimal grammatical ado into a linear sequence of internal objects whose fate is to enter a mill of binaries described below. Object tokens become either individual simple objects, or become incorporated into a composite object when they are enclosed between certain types of bracket.

2.3 Where Objects Live

Objects live on three stacks and in a large memory area referred to as the Virtual Memory (VM). Sizes of these storage facilities are set to fixed values when the D machine is brought up (*dvt*) or are created to a desired size by an operator (*dnodes*). The stacks hold whole simple objects or descriptions of composite objects. The VM holds a master description and the value of all parent composite objects (children objects are descriptions providing access to parts of parent object values, but there is no record of children objects per se in the VM).

- The **operand stack** holds objects to be consumed by operators or resulting from operators. D uses the *reverse Polish notation*, and the operand stack serves as the universal vehicle for passing operands from one operator to another. Objects accumulate on this stack until an operation consumes them as operands. While they remain on the stack, they can be explicitly manipulated there by stack operators.

- The **execution stack** holds a file of objects in execution. The top element is the next in line for execution, whereas elements below are suspended until they become the top element. An object carrying the *active* attribute can be submitted to execution by pushing it on the execution stack (by an operator or through reference to an associated active name). The execution stack will mostly be populated by active strings or procedures (active lists). After a string has been moved to the execution stack, the object tokens contained in it are successively translated and submitted to execution, until the string is exhausted and removed from the execution stack (this exposes the latest suspended object). With a procedure, the objects constituting its value are successively submitted to execution. Other objects pushed on the execution stack are consumed at once.

  The execution stack is transparently used by the operators that control the repeated execution of procedures (loop operators).

  Objects moved to the execution stack can be earmarked as dropback levels. If an object that is subsequently pushed on the stack for execution invokes a certain control operator, this operator drops the execution stack down to the topmost earmarked object. Execution

resumes with that object, cutting short the execution of execution stack objects between the current top of the stack and the dropback object. Activities can be swiftly terminated or aborted in this way, returning control to an embracing layer. The technique also serves to terminate loops from within. D is a structural purist's language: there is no 'goto' statement. The dropback mechanism, however, provides for all cases where a 'goto' might be desirable in a structured program.

- The **dictionary stack** holds dictionary objects currently in use. When an active name object is executed, the name is looked up in these dictionaries, beginning with the dictionary at the top and working toward the bottom. The first object found to be associated with the name substitutes for the name object. The dictionary stack is maintained through operators.

- The **VM** occupies a substantial amount of the host machine's RAM space. This memory is called 'virtual' because you can use it only through operators that establish a particular logical memory model.

  As composite objects are defined during D machine activity, they consume space in VM and eventually exhaust all available space. This is prevented by a mechanism involving three housekeeping operators, by which sets of VM objects are earmarked and selectively discarded from the VM space (references made to them in remaining objects are voided).

2.4 The Mill

This section describes the executive of the D machine, the *mill* for short. The mill accepts the objects that are handed to the machine and guides them to their targets, where they are stored, consumed, or transformed.

Two varieties of D machine, the D virtual terminal (*dvt*) and the D node (*dnode*), have slightly different mills that make the machines deal differently with external events and errors. These differences will be described in subsequent sections, which also tell you how to start up each variety

of D machine, or terminate its operation. Here we consider the aspects of mill operation that that apply to both the *dvt* and *dnode* machines.

When the program that implements the D machine is brought up, initializations are performed. These effect that:

- the dictionary stack contains the *system* dictionary (of intrinsic operators) and a *user* dictionary (the 'root' dictionary of the D machine).
- the execution stack holds a single object, a string. The string value is derived from a text file (called 'startup_dvt.d' or 'startup_dnode'; details are described in Chapter 3). It holds the first sequence of objects to be executed and, thus, is analogous to the 'bootstrap' code of an ordinary computer. The D objects imported from the startup file extend the operator set of the system by a library of procedures that are needed very often and may be viewed as operators that happen to be written in D.
- The VM is empty.

After this initialization, the mill will cycle through a succession of three phases. The ultimate goal of the mill's activity is to empty the execution stack by fair play. When the execution stack has been exhausted, the D machine is idle, waiting for an external event that delivers new objects to the execution stack (e.g., a console phrase).

Each turn of the D mill begins with the **test phase** evaluating conditions that require special attention. This is followed by the **fetch phase**, during which the next object in sequence for execution is determined. That object, then, is executed in the **execution phase**.

The **test phase** deals with external events that will typically push new objects onto the execution stack. The chief difference between the *dvt* and *dnode* is in their organizations of the test phase. The test phases of the two machine versions will be described separately in later sections.

In the **fetch phase**, the execution stack is inspected to derive the next object to be executed. This stack will only hold objects carrying the *active* attribute. A simple object will be popped and submitted directly to the execution phase. Of a procedure or string, instead, the first object

or translated token is extracted and submitted to the execution phase, whereas the remainder of the procedure or string remains on the execution stack until the entire procedure or string has been exhausted.

A fine point is that a procedure object actually is popped from the execution stack *prior* to the execution of its last value object (to save stack space during end-to-end recursion).

Any *passive* object passed to the **execution phase** is 'executed' by transfering it to the operand stack.

An *active* object passed to the execution phase is given special attention if it is a *name*. A *name* object is replaced by the currently associated object. The search for an associated object involves the dictionaries currently held on the dictionary stack and proceeds from the top of that stack to the bottom. The first association detected for the name is used. An active associated object is pushed on the execution stack, whereas a passive associated object is pushed on the operand stack. The mill starts a new turn.

Execution of other active objects also is directed by object class:

- A **null** object is discarded.
- An *operator* object is invoked, i.e. the host program module represented by the operator is executed. A large set of system operators has been built into the D machine and is accessible through the system dictionary *systemdict*, which is permanently present at the bottom of the dictionary stack. Libraries of additional operators (your personal extensions to the D operator set) can be loaded using certain operators and then become also incorporated into *systemdict*.
- Any other object is executed like a passive object, by transferring it to the top of the operand stack (thus the active/passive sttribute is ignored).

Two small points require a note here: (1) A passive null object is pushed on the operand stack, but an active null object effects nothing. (2) A procedure object encountered on the execution stack always serves as 'instruction' because of its *active* attribute; reference to a procedure through

an active name has the same effect. If a procedure is to be worked on as 'data', it needs to be transferred to the operand stack. This automatically happens when a procedure object is encountered in a stream of objects being executed (rather than being referred to by an active name). You can also use *get* to move a procedure associated with a name from a dictionary to the operand stack.

Many functions that are essential for the D machine, such as definition of name-object associations, are not of concern to the mill. The mill is concerned only with the orderly consumption of objects, whereas specific tasks belong to the operators. This strict division of labor keeps the plan simple and the D machine robust. Operators will be dealt with in 2.6.

Errors recognized by the mill, text translator, or operators lead to a consistent response in both types of D machine. The instance of discovery and a numeral representing the error are reported on the *dvt* that supervises the activity. Details of the reaction to errors will be described separately for each machine type.

In summary, the D mill executes the objects submitted to it via the execution stack, with effects that depend on the active/passive attribute and the class of the objects. Object associations by name are resolved. Executed objects eventually precipitate in objects pushed on the operand stack (to serve as 'data' for operators), or lead to the execution of host program modules (which perform the 'instructions' represented by D operators).

2.5 Working on a D Machine

When you are at the console of a D machine, the machine does not prompt you. It just sits there when it is not busy with executing D code. Simply type D code that you want to have interpreted. You can edit a phrase of D code as long as you have not concluded it with 'return'. Hitting 'return' submits your phrase to the D machine for interpretation.

How can you tell the machine is ready for more? This depends on the kind of D machine that your console is connected to (see Chapter 3 for details). In short, if your target is a *dvt*, you can simply keep typing because your code is buffered and successively executed in the order of

entry. If your target is a *dnode* or a group of *dnodes*, you can observe the 'TheHorses' window: upon delivery of your code to the targets, the respective window row or rows turn green while a target is busy and turn back to white when a target becomes ready for new work. If you do not respect the 'busy' status of the target(s), the *dvt* will reject your phrase of input with a 'Wait!' message, unless you overrule that protection by tagging console phrases with '!' (shout, so to speak).

If you have entered something like "{ } loop" the D machine will be running in a dead loop forever, unless you throw a wrench. Again, that wrench is different for the different kinds of D machine. For a *dvt*, the wrench is typing 'control-c' (twice when using an *emacs* shell). For a *dnode*, the wrench is entering (shouted) D code that will stop the loop activity, e.g. "!abort" if you want to make sure.

A D machine that has just started up has an initial vocabulary, of objects associated with names, residing in two dictionaries that permanently live as the bottommost two objects of the dictionary stack. The system dictionary (at the very bottom) references all intrinsic operators. If during a session you load a library of extrinsic operators, that set of operators will be merged with the operators already contained in the system dictionary (hence they are always accessible through their active names). The first entries in the user dictionary have been made by interpreting D code contained in the 'startup' file for the kind of D machine you are using. As you create new trees of D objects, reference their root objects in the user dictionary to maintain a permanent route of access (it is possible to 'lose' D objects in the machine, and retrieving them can be entertaining).

You should not redefine the objects handed to you in the user dictionary, unless you know exactly what you are doing. You will have to define additional objects in the user dictionary, but it is unwise to use the user dictionary as a working dictionary, littering it with random small objects. If you want to play around with the D machine as a desk calculator, define an empty dictionary for your new associations, reference it in userdict, and make it the current dictionary. The console phrase "100 dict dup /junk name begin" sees to that.

2.6 The Common Operators

This section describes the operators or procedures commonly implemented in all varieties of D machine and accessible via the system and user dictionaries that are permanently present on the dictionary stack. We will simply refer to them as 'operators' (those implemented as procedures are operators that happen to be written in D). The *dvt* and *dnode* varieties of D machine have additional machine-specific operators; these are described in Chapter 3. The section groups the operators into functional tool boxes (some tools are multi-functional and hence appear in several toolboxes).

In the operator tables that follow, items appearing on the left of the operator name (which is in **bold**) represent the operands and the order in which these are expected on the stack (the last operand is the topmost on the stack). The items on the right of the operator name are the results deposited on the stack (again, the last item is the topmost). These stack objects are designated by their class or type, or by names that reflect their usage; alternate choices of an operand are separated by '/'.

2.6.1 Feeling around

Whatever you do with a D machine, you want to be able to know what is in there. The following operators tell you. They do not take operands or return results: they rather list information on the console:

_	'show top object of operand stack in brief form'
v_	'show value of composite top object of operand stack'
s_	'show value of string top object of operand stack as text'
a_	'show all objects on operand stack in brief form'
d_	'show top object of dictionary stack in brief form'
da_	'show all objects on dictionary stack in brief form'
xa_	'show all objects on execution stack in brief form'
m_	'show current stack and VM usage'

To copy an object from the dictionary or execution stack to the operand stack (where you can further analyze it), use:

k **dg_** obj_k
k **xg_** obj_k

where k is the index of the targeted object, counting from the top down (the top element is $k = 0$).

The tool of choice for inspecting VM contents or file systems is a graphical browser called 'TheEye'. This mouse-driven browser, which is written in D, is described in a Chapter 3.

2.6.2 Manipulating objects on the operand stack

Whenever you give the D machine a passive object to interpret, it puts that object on the operand stack. When your console input invokes an operator, the operator takes its operands from the operand stack and leaves its results on the operand stack.

obj	**pop**	–
(obj_1 .. obj_n	**push**	–
obj_1 obj_2	**exch**	obj_2 obj_1
obj	**dup**	obj obj
$obj_1 \ldots obj_n$ n	**copy**	$obj_1 \ldots obj_n$ $obj_1 \ldots obj_n$
$obj_n \ldots obj_0$ n	**index**	$obj_n \ldots obj_0$ obj_n
$a_{n-1} \ldots a_0$ n j	**roll**	$a_{j-1 \bmod n} \ldots a_0$ $a_{n-1} \ldots a_{j \bmod n}$
‖ obj_1 obj_n	**clear**	–
‖ obj_1 obj_n	**count**	obj_1 obj_n n
–	**(**	mark
$obj_1 \ldots obj_n$	**)**	list
$obj_1 \ldots obj_n$	**cleartomark**	–
$obj_1 \ldots obj_n$	**counttomark**	mark $obj_1 \ldots obj_n$ n
–	**null**	null
–	**true**	true
–	**false**	false

Manipulation of objects on the operand stack becomes necessary, for in-

stance, when the results left on the stack by one operator do not match the order of operands expected by a subsequent operator. Simple stack operations remove (*pop*), exchange (*exch*), or duplicate (*dup*) the top element(s) of the stack. Others duplicate portions of the stack (*copy*), rotate stack elements in a chain (*roll*), or access the stack like an indexed array (*index*). The operand stack is cleared (*clear*), or its elements are counted (*count*). Operands are moved from the operand stack to the execution stack by the *push* operator; this is not intended for casual use, but as part of utilities for complicated and recursive D operators.

A word of caution: juggling objects on the stack can avoid making 'variables' (and thus a wicked state machine). On the other hand, stack acrobatics quickly turn unintelligible to the acrobat.

There are facilities to mark a position on the stack ([) and to count or delete the elements following the latest mark (*counttomark*, The main use of '[' will be in list construction, as is described later.

There is no text representation of the null or boolean objects. Associations of these objects with names are provided in the system or user dictionaries (**null**, **true**, **false**).

2.6.3 Mathematical operations

MONADIC operators are used in one of two possible ways:

num	**operator**	num (modified)
array	**operator**	array (modified)

where the operator is one of:

abs	'take absolute value'
neg	'negate'
sqrt	'square root'
cos	'cosine'
sin	'sine'
acos	'arccosine'

asin	'arcsine'
tan	'tangent'
atan	'arctangent'
exp	'exponential, base e'
ln	'logarithm, base e'
lg	'logarithm, base 10'
floor	'round down to nearest integer'
ceil	'round up to nearest integer'

DYADIC operators are used in one of four possible ways:

num_a num_b	**operator**	num (result)
array_a num_b	**operator**	array_a (modified)
num_a array_b	**operator**	num (result)
array_a array_b	**operator**	array_a (modified)

where the operator is one of:

add	'sum, $a + b$'
sub	'difference, $a - b$'
mul	'product, ab'
div	'quotient, a/b'
pwr	'power, a^b'
mod	'modulus, $a \mod b$'
acos2	'extended $\arccos(\frac{a}{\sqrt{a^2+b^2}})$'

The monadic operators take one operand. When given a numeral (scalar), they return a scalar. When given an array, they operate on all elements of the array, replace them with the results, and return the modified array. The dyadic operators take a pair of operands. With two scalar operands, they return a scalar result. With a pivotal scalar operand and a second array operand, they repeat the operation with each array element as the second operand and the scalar as the first operand; the scalar value is replaced by the running result. In the converse operand configuration, a pivotal array and a second scalar operand, the operation is performed on each array element as the first operand always using the scalar as the second operand; the results overwrite the original array elements. With two

arrays as operands, the operation is performed with the corresponding elements (array dimensions need to be matched) and the results overwrite the pivotal array.

acos2 computes the counterclockwise normalized arc (0 to 2π) corresponding to the Cartesian point (a, b) on the circle around $(0, 0)$. (It is a generalization of arccos, which takes a single argument and thus cannot give a unique result for arcs extending over more than a half circle.)

NOTA BENE: All mathemetical operators first convert their operands into the double-precision floating point IEEE format, then perform the operation, and finally convert the result back into the type of the (first) operand. All operators work with all numeral types for first or second operands. The mantissa of the longest format of integer (63 bits) exceeds that of the highest precision of the floating-point format (53 bits). Thus operations involving integers of type '/X' return results with at most 53 bits of mantissa accuracy. A result of type '/X' and of value greater than or equal to 2^{53} is returned as *undefined*.

The mathematical operators accept *undefined* operand values and automatically propagate *undefined* into the result values. If the operation itself fails to yield a valid result, it also returns *undefined*. No error is indicated when mathematical operators receive or generate *undefined* values, but other operators will generally reject *undefined* operand values. The *undefined* values of integers are expressed by the largest negative value of their ranges (e.g. -128 for /B). For real types *undefined* values comprise all INF and NAN patterns of the IEEE floating point standard; the '$*$' notation for expressing undefined numeral values always generates +INF. You can test whether a /S or /D type numeral value is *undefined* by comparing it through the *eq* or *ne* operator against '$*$'; both operators consider all possible INF and NAN patterns in this test as equal to '$*$' (+INF).

The mathematical operators described here are executed in different ways by the *dvt* and *dnode* versions of D machine. In a *dvt* a single thread of execution (processor) is used. The *dnode* can use multiple threads, up to the number of processors in the hardware of the host. The number

of threads used is controlled dynamically by operators. Except for the difference in execution time, the use of multiple threads is transparent to you.

2.6.4 Working with composite objects

Each class of composite object is created by a specific operator, *array*, *list*,], *dict*, and *save*. (Other operators can also create composite objects in VM; these include: *readboxfile*, *send* (in the target), *token*, *save*, *regex*, *regexi*, *fromscreen*, *loadlib*).

Several operators perform analogous tasks on composite objects of several classes. They inquire the dimension of the object (*length*), access or replace individual elements (*get*, *put*), physically copy the elements of the object value to another object (*copy*), or present the elements of the value successively to a procedure, which is invoked once for every element (*forall*). Arrays or lists (but not dictionaries) may be logically subdivided into subarrays or sublists (*getinterval*). Subarrays or sublists physically share their elements with their parents and siblings; hence, changes made to one member of the object family affect others that represent concerned elements of the value.

The following operators work on ARRAYS:

n /type	**array**	array
array	**length**	n
array index	**get**	num
num array index	**put**	–
array index n	**getinterval**	subarray
$array_1$/num $array_2$	**copy**	(sub)$array_2$
$array_1$ index $array_2$	**fax**	$array_1$ index
d-array idx count s-array	**tile**	d-array idx
d-array idx count first step	**ramp**	d-array idx
s-array idx iv d-array	**extract**	d-subarray
s-array d-array idx iv	**dilute**	d-array

s-array d-array idx iv	**dilute_add**	d-array
array n /type	**parcel**	subarray_2 subarray_1
oldmin oldmax array	**extrema**	newmin newmax
array bool	**ran1**	array
array proc	**forall**	–
string seek	**anchorsearch**	post match true or string false
string seek	**search**	post match pre true or string false
string	**token**	post obj true or string false
string regexpr	**regex**	post match pre (submatches) true string false
string regexpr	**regexi**	post match pre (submatches) true string false

Mathematical operators working with arrays have been described in 2.6.3. More mathematical operators for arrays are provided in a *dnode* and are described in Chapter 3.

copy moves array elements from one array into another, performing automatic numeral type conversion of the source array elements if the array types do not match. If the length of the destination array is greater than that of the source array, only the elements that receive source elements are changed. *copy* returns the destination array, or the subarray representing only the changed elements in the destination array.

To tile the contents of one array into successive subarrays of a larger array, use *tile*. Insert numbers generated with uniform increments into an array using *ramp*. *extract* copies every *iv*-th element of one array into successive elements of another, and *dilute* copies successive elements of one array into every *iv*-th element of another (*dilute_add* works like *dilute* but algebraically adds, rather than copies, elements to the destination array). These operators take any numeral type, but their operand arrays must match in type.

parcel subdivides an array into subarrays of arbitrary numeral type and thus facilitates multiple uses of arrays (Cave: no type conversion is done to the contents).

extrema updates the two running extrema given to it as numerals to comprise the extrema of the given array value. It takes any numeral and array types, and it returns the new extrema in the types of the old extrema. All numeral types encountered in *extrema* are converted to /D type prior to the operation.

ran1 generates a sequence of random numbers in an array of type '/S'. If the boolean operand is **true** the sequence starts from the seed (which is internally set and always the same to allow generation of pseudo-random sequences), else the sequence is continued. Random numbers are between 0.0 and 1.0 excluding the exact boundaries. The period of the generator is $\approx 10^8$ cycles. This is the function *ran1* described in Press et al. (page 280).

forall applies a procedure sequentially to all elements of an array, providing the current element to the procedure (as top element of the operand stack).

Five array operators specialize in strings (byte arrays). These dissect a string after seeking for a pattern that defines the breakpoint. *search* looks for the seek pattern anywhere in the string, whereas *anchorsearch* checks if the seek pattern leads the string. *token* parses the string using the rules of the D grammar and, when successful, returns the remainder of the string and the (first) D object found, in translated form. *regex*, and its case-insensitive kin *regexi*, also search a string for a matching pattern. They differ from *search* in that the match is defined not by a simple string literal, but is described by a *regular expression* (see C library manual page 'man:regex.7' for a definition). A regular expression can specify a great variety of patterns that are searched for in a single pass through the string. When a match is obtained, the returned list contains literal strings showing what was matched in second or later pieces of the search pattern; the string *match* includes the entire substring of *string* that generated the match, including characters that were not match criteria. The regular expression itself is an (interpreted) string (mind the composition rules for D string literals). *regex* and *regexi* allocate VM space for the list and strings of submatches.

The following operators deal with LIST objects:

n	**list**	list
–	**(**	mark
mark $obj_0 \ldots obj_{n-1}$	**)**	list
list index	**get**	obj
obj list index	**put**	–
list index n	**getinterval**	sublist
$list_1$ $list_2$	**copy**	(sub)$list_2$
$list_1$ index $list_2$	**fax**	$list_1$ index
oldmin oldmax list	**extrema**	newmin newmax
list proc	**forall**	–
list	**transcribe**	list

The square bracket operators, '[ ]', serve to construct a list, element for element, on the operand stack. First, a mark object is laid down by the '[' operator. Subsequent activity produces objects that accumulate on the stack following the mark. Finally, the ']' operator counts the objects down to the nearest mark, creates a list of matched size, copies all objects following that mark into the list's value, pops these stack objects including the mark, and pushes the list object. Note that the similarity with array or procedure enumeration is superficial: the objects enclosed between '[ ]' are *executed* like any other objects (they are NOT included literally into the body of the list as would be the case for objects enclosed between '{}').

extrema updates the two running extrema given to it as numerals to comprise the extrema of a list containing only numeral objects (see also the array version of *extrema*).

A list can give origin to an object tree, which comprises the value of the root list itself. If elements of the root value are composite objects themselves, also their values are part of the tree. Of the elements of these values, lists or dictionaries continue the tree (as new nodes). The operator *toboxfile* (see file operators) assembles, in a temporary VM buffer, a compact copy of the values of all composite objects that constitute a tree and saves the compact copy into one file. The complementary operator *fromboxfile* retrieves that tree into VM, again in one operation. These operators thus allow you to move large, logically connected collections of objects in a convenient and physically efficient manner.

A list, or any other object, submitted to *transcribe* will be replicated, and the replica will be returned. The replication is recursive and thus includes the entire tree of objects that originates from a composite original object.

The following operators deal with DICTIONARIES:

n	**dict**	dict
dict	**length**	num
dict	**used**	num
dict	**begin**	–
–	**end**	–
/name obj	**def**	–
obj /name	**name**	–
dict /name	**get**	obj
obj dict /name	**put**	–
dict /name	**known**	bool
/name	**find**	obj
$dict_1$ $dict_2$	**merge**	$dict_1$
dict	**cleardict**	dict
dict proc	**forall**	–
–	**systemdict**	dict
–	**userdict**	dict
–	**currentdict**	dict
–	**countdictstack**	num
list	**dictstack**	sublist
dict	**transcribe**	dict

The total capacity (*length*) or used capacity (*used*) of a dictionary can be inquired (as the number of association pairs). Other operators move a dictionary to and from the dictionary stack (*begin*, *end*) in order to control the lexical scope of name searches. Name-object associations are formed in the current dictionary (*def*, *name*) or in any dictionary explicitly specified (*put*). All entries of a dictionary are discarded by *cleardict*, returning an empty dictionary. You can determine whether a name is defined in a particular dictionary (*known*) or retrieve the associated object (*get*). All active dictionaries (i.e. those present on the dictionary stack) are searched

for an association by *find*. The associations in one dictionary are merged into another dictionary (*merge*, as by repeated *def* operations).

A dictionary, or any other composite object, submitted to *transcribe* will be replicated, and the replica will be returned. The replication is recursive and thus includes the entire tree of objects that originates from the original object.

Two dictionaries always present on the dictionary stack and the dictionary currently on top of the stack are accessed through associated names (*systemdict*, *userdict*, and the operator *currentdict*. The number of dictionaries present on the dictionary stack is inquired through *countdictstack*, and the current contents of the dictionary stack are copied into a list for diagnostic purposes by *dictstack*.

BOX objects are involved in the following operators:

–	**save**	box
box	**capsave**	–
box	**restore**	–
box	**length**	num (# of bytes)

save, *capsave*, and *restore* manage VM space. *save* creates a box object that represents all composite objects created following the execution of *save*. *capsave* caps the box object (objects created in VM following *capsave* are outside the box). *restore* removes the VM objects contained in the box from the VM, nulling all references made to these objects in remaining objects (no reference to objects in the box must exist on stacks by the time *restore* is applied). The VM is compacted, such that all remaining composite objects are stored contiguously. The size of the box object can be inquired by *length*. (Boxes containing a tree of objects can be stored in a file or retrieved from a file by *toboxfile*, *fromboxfile*.)

The VM management operators are used by a set of procedures (included in *userdict* via the startup files) in order to help you manage your D programs and their memory usage. The models established by this library are the *module* and the *layer*. A module is a tree starting from a dictionary. A layer is a caooed 'save' box of composite objects that modules work on.

Both modules and layers can be created, deleted, and re-created without wasting VM space.

MODULE and LAYER are implemented using the operators:

/module_name	**module**	/module_name savebox
/module_name savebox dict	**_module**	–
/module_name	**forgetmodule**	–
/module_name /snapshot_name	**savemodule**	–
/snapshot_name	**restoremodule**	–
/layer_name	**layer**	–
boolean /layer_name	**_layer**	boolean

To construct a module, put the name of the new module on the operand stack, execute *module*, define your module tree with a dictionary as root, put the dictionary on the operand stack, and execute *_module*. The construct is analogous to defining a procedure, using the *module* and *_module* operators in place of the delimitors '{...}'. *module* deletes a previously existing object matching *module_name* from the VM, and executes *save*. *_module* defines the dictionary under *module_name* in *userdict*, executes *capsave*, and saves the savebox and module_name in the module dictionary. Each time you define a new module under an existing module name, the old version is deleted from the VM so that dead wood will not accumulate.

A module may comprise the D code constituting your 'project'. It may alternatively contain a collection of tools used by many projects. The advantage of the module organization is that you can replace tools or projects by newer versions as you develop them while maintaining their D environment that you do not want to change at this time and while conserving VM space. *savemodule* preserves a replica ('snapshot') of a module in VM space. *savemodule* needs to be used with *userdict* as the current dictionary. *restoremodule* re-establishes the module by making a saved snapshot the module and discarding from VM the previous version of the module. You can in this way save the state of an entire project (including results that require lengthy computations) while you test new parts of your project and retrieve the previously accumulated results as

you debug the new parts of your project.

Sets of objects created during the execution of a module can be organized as a 'layer' and as such be automatically discarded from VM when another work cycle starts that constructs a new layer under the same name. Use *layer* and *_layer* to mark the beginning and ending points in your D code of the operations that define the objects of the layer (while using the module dictionary as the current dictionary). This stretch of code should also be wrapped as a *stopped* context so that its sucessful or failed completion is indicated by a boolean left on the operand stack. *_layer* passes this boolean through to the code of your module so that you can react to a failed construction of the layer.

2.6.5 Tests and logics

any_1 any_2	**eq**	bool
any_1 any_2	**ne**	bool
num_1 num_2	**gt**	bool
num_1 num_2	**ge**	bool
num_1 num_2	**lt**	bool
num_1 num_2	**le**	bool
$bool_1$/num_1 $bool_2$/num_2	**and**	bool/num
$bool_1$/num_1 $bool_2$/num_2	**or**	bool/num
$bool_1$/num_1 $bool_2$/num_2	**xor**	bool/num
bool/num	**not**	bool/num
–	**true**	true
–	**false**	false
num shift	**bitshift**	num

The relational operators compare two operands and return a boolean indicating whether or not the relation holds. Any two objects of matching class are tested for equality (*eq*, *ne*) of their values (simple objects) or identity of their values (composite objects). Only numerals are tested for the other relations (*ge*, *gt*, *le lt*). *eq* and *ne* accept numerals of value *undefined*, so they can test for the occurrence of this value.

Logical operations (*and*, *or*, *xor*, *not*) use booleans (then returning a boolean)

or work in a bitwise fashion on numerals (then returning a numeral). The internal representation of numeral values normally remains transparent, but here it is important for understanding the bitwise logical operations. These operators are intended to be used on integers, which are 8-, 16-, or 32-bit two's complement numbers; the *undefined* bit pattern is treated as a valid pattern (sign bit set, other bits reset). Real types are also accepted, but the interpretation of their results requires detailed knowledge of the IEEE floating-point formats. If two equal types are used in a binary logical operation, this operation is performed on all bits in parallel and the result is meaningful in all positions. If the second operand, however, is shorter than the first, the operands are aligned with respect to their least-significant bits and the shorter operand is expanded with zeroes. The operation is performed over the width of the first (longer) operand.

2.6.6 Controlling execution

~active	**start**	...
[obj ...	**push**	...
~obj	**exec**	...
bool ~active	**if**	...
bool ~active$_1$ ~active$_2$	**ifelse**	...
init incr limit ~active	**for**	...
count ~active	**repeat**	...
~active	**loop**	...
array/list/dict ~active	**forall**	...
—	**exit**	—
/name	**exitto**	—
~active /name	**exitlabel**	...
~active	**stopped**	... bool
—	**stop**	—
—	**abort**	—
—	**countexecstack**	num
list	**execstack**	sublist

Control operators modify the usual linear execution of objects. Many work with a procedure operand that they execute conditionally or re-

peatedly. In addition to the control operators listed in this section, other events, like those originating from communication with connected D machines or X Windows can also schedule objects for execution.

Any object carrying the *active* attribute is pushed onto the execution stack for immediate execution by **exec**. **start** does so, too, *after* popping the currently executing object from the execution stack. **push** transfers the objects follow the last mark onto the execution stack, in effect executing them in reverse order; this allows temporary hiding of a portion of the operand stack during execution of a given procedure and its automatic replacement on top afterwards. A procedure (or any active object provided in its place) is executed conditionally, dependent on the value of a boolean operand (**if**), or one of two procedures is executed dependent on a boolean (**ifelse**). Several operators execute a procedure repeatedly. **for** takes three numeral operands defining a range and step to control an internal count and executes a procedure for each count of the range, providing the procedure with the current count as the top element of the operand stack. **repeat** executes a procedure a prescribed number of times. **loop** repeats the execution of the procedure indefinitely (unless terminated from within). **forall** executes a procedure for each element of a composite object, providing the procedure with the current element as top element of the operand stack. Any loop operator can be terminated by executing one of the operators **exit**, **exitto**, **stop**, or **abort** within the dynamic context of the loop body. **exit** terminates the execution of the current loop and resumes execution of the object following the loop operator of the terminated loop. **exitto** is a generalized **exit** – it transfers control from the current point to the object following the enclosing matching **exitlabel** operator. Matching means that the ***/name*** matches.

stopped executes an object (usually a procedure) with a provision for two alternate kinds of termination, normal or stopped. Stopped termination (effected by the **stop** operator executed from within the dynamic context of the object) returns **false**, normal termination returns **true**, and execution continues execution with the object immediately following **stopped**. This construct helps prepare contexts that terminate on exceptional events and turn control over to a supervisory context. The exceptional termination removes from the execution stack all objects that belong to the stopped context (such as pending procedures and loops in execution); it provides a clean exit from a domain of structured code. **abort** drops the execu-

tion stack down to the most basic level (an **aborted** capsule in the *dvt* or the very bottom of the stack in the *dnode* variety of machine). The loop termination operator, **exit**, cannot move control outside the context of a stopped operator (the attempt produces an error), and **stop** cannot move control outside an **aborted** context in a *dvt*.

The operand described above as *active* typically is an active list (objects enclosed between '{ }'). You may supply instead any active object. A convenient method to supply an active name as operand to the control operator is to give the name with the *tilde* attribute. When the tilded object is executed it is converted to an *active* object and pushed on the operand stack, where the control operator will consume it.

2.6.7 Types, attributes, and their conversions

obj	**class**	/class
num/array	**type**	/type
obj	**readonly**	bool
obj	**active**	bool
obj	**parent**	bool
obj	**tilde**	bool
obj	**mkread**	obj
obj	**mkact**	obj
obj	**mkpass**	obj
num/array /type	**ctype**	num/array
array n /type	**parcel**	rest-of-array subarray
stringbuf index width obj	**text**	stringbuf index
stringbuf index width num format	**number**	stringbuf index
proc	**bind**	proc

The class, numeral type, and attributes of an object are inquired by specific operators. *class* or *type* return passive names such as '/nameclass' or '/b'. The attributes of an object are inquired by *readonly*, *active*, *parent*, or *tilde*. *mkread*, *mkact*, *mkpass* change an attribute; the *readonly* attribute cannot be reversed once established. An object carrying the *tilde* attribute is treated as passive object when it is executed and the *tilde* attribute is

automatically removed (a tilded name object would turn active thereby). Uses of *tilde* are described in 2.2.

The type of a numeral or array is converted into a different type by *ctype*, and an array of any type is parcelled into subarrays of the same or other type by *parcel*. Type conversion of a numeral object converts the type specification of the object and translates the value into the new binary representation. Type conversions involving arrays change only the type specification in the object description, but do not translate the value from the old to the new binary representation (hence, pre-existing values of such arrays generally become uninterpretable).

text or *number* translate an object into a text representation and insert the created text into a string buffer. Insertion into the buffer is controlled by the *width* operand (a numeral). The absolute value of *width* controls the number of characters (including padding spaces) that are inserted into the buffer. The sign of *width* controls the adjustment (negative: left; positive: right). If *width* is given with an *undefined* value ('$*$'), the minimal number of characters needed to represent the object is inserted (without leading or trailing spaces). The text is inserted into a string buffer, starting at the element designated by *index*; the index is updated to point to the next free element of the buffer. *text* interprets the operand *obj* as follows: numeral: the character coded by the numeral value is inserted; string: the string value is inserted; name: the text form of the name (with the passive attribute represented as a '/' prefix) is inserted; operator: the text form of the operator name is inserted. *number* translates only numeral objects *num* as specified by the numeral *format*. A negative sign of *format* selects fixed-point representation, a positive sign, floating-point representation. The value of *format* specifies the number of fractional digits to include for a non-integer numeral (with automatic rounding). An *undefined* value of *format* specifies the maximal number of fractional digits needed to represent *num*.

Subjecting a procedure to the *bind* operator replaces name objects that resolve to operators in the context of the current dictionary stack by the operator objects themselves. *bind* applies itself to the body of the operand procedure, and recursively to the bodies of all procedure objects nested therein. This has two consequences: (1) re-definition of current operator names will not affect the behavior of the bound procedure, and (2) exe-

cution will be speeded as bound operators are referred to directly rather than through an association.

2.6.8 Controlling VM resources

–	**vmstatus**	max used
–	**save**	box
box	**capsave**	–
box	**restore**	–
null or parent	**nextobject**	obj true or false

The values of internal objects that are created during a D session are accumulated in the virtual memory (VM), eventually filling all available physical space. *vmstatus* lets you examine the current capacity and usage of VM space (returned as number of bytes). *save*, *capsave*, and *restore* let you mark and reclaim used VM space.

save creates a box object in VM that marks the current level of used VM space. When following some further activities *restore* is applied to this box object, it discards all VM objects created after that instance of *save*. This could leave object descriptions in the machine (on the stacks or in composite objects remaining in the VM) that refer to discarded VM objects. Therefore, *restore* scans the stack for such references before it makes changes (it terminates with an error if it finds a reference). *restore* then discards the VM objects and scans all remaining VM objects for references to discarded objects. It replaces discarded objects in lists by **null** or purges associations with discarded objects from dictionaries.

When *capsave* is applied to a *save* box object, the range of discardable VM objects is restricted to those created between the executions of *save* and *capsave*. When *restore* is applied to a 'capped' save box, it deletes the enclosed range of VM objects and in addition moves all objects created after those in the discarded range to compact the used VM space (references to moved objects will be automatically corrected in remaining objects).

nextobject retrieves a composite object from VM. If used with a null object

as operand, the first object created in the VM is retrieved; with a VM object given as operand, the next object created after the operand object is retrieved. Applying *nextobject* repeatedly to each retrieved object will scan the VM objects in their order of creation. Note that only parent objects are accessible to *nextobject*, since no record of children objects is maintained in VM. In this way, you can locate a 'lost' object, for which you have not saved a reference. *nextobject* can also verify VM integrity as it will fail when running into a broken VM object.

2.6.9 File access

—	**getwdir**	substring
string	**setwdir**	—
(path) (filename) string	**readfile**	substring
string (path) (filename)	**writefile**	—
(path) (filename)	**readboxfile**	rootobj
rootobj (path) (filename)	**writeboxfile**	—
(path)	**findfiles**	filelist
(path) (filename)	**findfile**	false / filesize datetime attribute–bits true
string	**tosystem**	—
$string_1$	**fromsystem**	$string_2$
new–umask–integer	**umask**	old–umask–integer

The path operand of file operators is a directory tree following the Linux convention (final '/' is optional). **readfile** reads the file contents into the buffer *string* and returns the filled substring (if *string* cannot hold the entire file, an error results). Box files contain a folded tree of objects that is described to **writeboxfile** by the root object (a list or dictionary); **readboxfile** copies the objects of the tree to the VM and returns the root object of the tree. Box files could be written and read by different generations of D machine and/or D machines running on different platforms. For interconversion issues see 2.6.11.

getwdir and **setwdir** return the current directory or set a new current directory for subsequent file accesses.

findfiles scans the specified file directory and returns a sorted list of the

entries; each entry is described by a list containing: filename (string), file size (numeral, in bytes), compact date/time of last modification, and the file attributes (a /W numeral whose bitwise interpretation is:
TTTT---uuugggooo, where TTTT = 1100 - socket, 1010 - symbolic link, 1000 - regular file, 0110 - block device, 0100 - directory, 0010 - character device, 0001 - fifo, and where uuu, ggg, and ooo are bits defining the user (owner), group, and all (user, group,and all others) access privileges (read/write/execute)). The list of entries is sorted in the order directories, files, with each subgroup sorted alphabetically. **findfile** looks for a specific *filename*. If it doesn't exist, false is returned; otherwise it returns the values of the entry for that file from **findfiles** and true.

tosystem submits the string as a Linux command to the bash/emacs shell without waiting for completion. The operator is listed *here* because many file manipulations (like moving or deleting) are not supported by D operators, but are instead left to existing Linux commands. **tosystem** has unlimited other uses. **fromsystem** is a generalization of **tosystem**. It submits its operand string to the current Linux shell, blocks until the command has been completed, and returns a newly created string object holding the output of the invoked Linux command(s). (The returned string is created by **fromsystem**; use **save** and **restore** to discard such strings to prevent their accumulation.)

umask sets the umask for the process, and returns the old umask. The umask in an integer from 0 to 511 (777 in octal), where the bits are rwx|rwx|rwx: r = read, w = write and x = execute, the first group is user permissions, the second group is group permissions and the final is other permissions. Write operators such **writeboxfile** and **writefile** use a mask of rw|rw|r or 664 in octal; this mask is **and**ed with **not** umask to produce the final permission set. In most cases, the default umask is 022 in octal (|w|w) so group write is removed: $664 \ \& \ \neg 022 = 644$, in octal.

2.6.10 Time and date

— **gettime** time
— **gettimeofday** secs μsecs

time long–array–of–6	**localtime**	array
∼active x–array–of–8	**profiletime**	... x–array–of–8
seconds nanoseconds	**sleep**	—

gettime returns compacted Linux time as an extended numeral (seconds since something). **localtime** converts this long numeral into a local time and date and deposits the detailed result in the array; the entries are: year month day hour min sec. The time of day is the military version of the Babylonian, or whatever they call it. **gettimeofday** returns the "seconds since something" of **gettime** plus μseconds. **sleep** pauses for seconds plus nanoseconds. **profiletime** executes '∼active', storing the user time in 0 & 1 of array, the system time in 2 & 3 of array, the user time of children in 4 & 5 of array, and the system time of children in 6 & 7. The times are second & μsecond pairs; children in this context mean forked subprocesses.

2.6.11 Networking

(servername) port	**connect**	socket
socket	**disconnect**	—
socket string	**send**	—
socket comp_obj	**send**	—
—	**getsocket**	socket
socket sig	**sendsig**	—

A connection between two D machines is established when one of the machines establishes it (**connect**). The target of the connection is always a *dnode* described by the network name of its host and by the number of the port where that *dnode* is listening for connection requests (a *dvt* does not listen for connection requests). The port number is an offset into the range of user reserved ports, starting with 'o'; it is assigned to the *dnode* when the *dnode* process is started up by a shell command to the respective host. **connect** returns a socket number opaquely stored as the value of a typed **null** object (you cannot play with it, but you can pass it around in the D machine). The connection is closed by **disconnect**.

Two forms of message can be sent by connected machines to ane another (regardless of who established the connection). The first form submits a string; the receiving machine makes the string active and transfers it to the execution stack, so that the string is interpreted by the receiving machine. The second form of message folds the tree that originates from the composite operand object into a temporary box object and transmits the box object. The receiving machine executes **save**, unfolds the tree in the received box object into its VM, and transfers the root object with its active/passive attribute set to active onto the execution stack, thereby executing the root object. The composite object can be a list, dictionary, or array (except string). In other words, the difference between the two forms is that with a non-string composite operand object the receiving machine invokes **save** prior to executing the object, whereas with a string operand **save** is not invoked.

The example

(~dup ~capsave somedict ~begin ~proc-name ~end ~restore)

caps the save object that contains the downloaded tree (whose root is the list object), executes a procedure with a downloaded dictionary (a node of the tree) as the curent dictionary, and discards the tree from the VM of the receiving machine.

The receiving machine also performs necessary conversions when it receives D objects from a machine that invokes **send**. This is done during the unfolding of the received tree (**readboxfile** proceeds similarly when it recovers a tree from a box file). Interconversion of little and big Endian formats are done automatically when two different platforms communicate. On the other hand, the communicating D machines must use the 64-bit binary format for their objects (note: these D machines can be implemented on 32 and 64 bit platforms; the point is that the D object formats must be the same). Older D machines using the 32-bit object format do not communicate with the current 64-bit D machines (there exists, however, an undocumented operator to rescue box files that use the now obsolete 32-bit object format).

A *dvt* receives a **send** message through the **nextevent** operator; in a *dnode* it is the mill that receives and executes the message. A difference between the two machine types is that the *dvt* will not process a message before it has completed an ongoing activity, whereas a *dnode* will interrupt an ongoing activity (within a limited number of mill cycles) to attend to the message.

getsocket returns the socket number of the socket through which the most recent message was received. There is a grace period of 100 mill cycles before another message will be accepted. In this period the activity started by the message can execute **getsocket** with results that are guaranteed to be correct.

The operators described in this group effect point-to-point network communication between two D machines using a bi-directional stream protocol. Another group of network operators effect point-to-many-points communication among D machines using the MPI (Message Protocol Interface). The second group of network operators is available only to *dnodes* and *dpawns*, and is described in Chapter 3.

sendsig sends a 'signal' to a *dvt* or *dnode*. The signal is a platform independent integer, that maps to a platform-dependent POSIX signal number on the receiving end; these are mapped in `dm{signals.c` in the `sigmap` array. That mapping is accessible directly in *D* via the ***SIGNALS*** dictionary, which maps the names of signals (such as ***QUIT*** for `SIGQUIT` or `KILL` for `SIGKILL`) to the platform neutral *D* enumerations.

2.6.12 Configuration inquiries

- **getstartupdir** (path) 'where startup file is'
- **getmyname** string 'host name where we are'
- **getmyport** port# 'where we are listening (dnode)'
- **getmyfqdn** string 'hostname.domainname'
- **getconfdir** string 'typically /etc/dm'
- **gethomedir** string 'your home dir on this host'

Most of these operators return a readonly string object that is created once when the D machine is started and has a fixed value (changing only when *vmresize* is executed in a *dnode*).

2.6.13 Windows and graphics

Before you run D machines that use the X Window system, make sure that the X server is not only running on the *dvt* host (where windows will be displayed), but that this X server is configured so that it listens on its ports for requests from other machines (where your *dnodes* might live).

Hooking up X window services in a D machine cluster requires a ritual. We describe the ritual below without apologies (we did not invent it). This ritual is well hidden in 'startup' code so that you need not normally immerse in it – you may nevertheless read about it for an idea of what might go wrong.

— **Xwindows** bool

—	**Xdisplayname**	string
—	**screensize**	width height
xy (name) (iconname)	**makewindow**	window#
window# bool	**mapwindow**	—
window# width height	**resizewindow**	—
window# bool	**topwindow**	—
window#	**deletewindow**	—
RGB–array	**mapcolor**	color–index
window# xy color–index	**drawline**	—
window# xy color–index symbol# size	**drawsymbols**	—
window# xy color–index	**fillrectangle**	—
window# xy (text) [...]	**drawtext**	window# x y
vol	**bell**	—
—	**Xsync**	—

The *dnode* executes **Xconnect** to connect to the X server and use X windows services (see also X window operators for *dnodes* in 3.3.1). Authorizations can be done via ssh.

A *dvt* setting up a *dnode* also needs to inform the *dnode* which display name to use when connecting to the X server (that name is of the form 'hostname:number'). The *dvt* executes **Xdisplayname** to retrieve this information. In some instances, the hostname 'localhost' is inserted into the display name. This is not a problem when both the *dvt* and *dnode* run on one and the same host. It is a problem when the hosts are different. In this case, the *dvt* needs to edit the display name, replacing 'localhost' by the string returned by **getmyfqdn** (see 2.6.11).

Xwindows informs a D machine whether it is currently connected to an X window server. **screensize** returns the screen dimensions for the design of windows that you create with **makewindow**. **makewindow** takes three operands. The first operand *xy* specifies the position and dimensions of the window. The notation *xy* is shorthand for a number of options of how coordinate information can be presented to X window operators:

```
          < x y ... >
            [ x y ... ]
< x ... > < y ... >
  < x ... > [ y ... ]
  [ x ... ] < y ... >
    [ x ... ] [ y ... ]
```

The other operands of **makewindow** provide two strings for the header of the window (no more than 30 characters) and for the label of the iconized window (no more than 12 characters). A numeral *window#* is returned to serve as handle for specifying the window to other window operators.

When you create a window with **makewindow** you become also responsible for creating a dictionary that holds the procedures named *windowsize*, *drawwindow*, and *mouseclick*, and is included in *userdict* under the name formed by concatenating the letter 'w' with the window# returned by **makewindow**. The three procedures are designed for the usage:

width height	**windowsize**	– 'be notified of change in dimensions'
–	**drawwindow**	– '(re)draw window'
x y modifiers	**mouseclick**	– 'react to mouseclick'

windowsize responds by executing **resizewindow**. It uses the new dimensions found on the stack; or a modification thereof (e.g. maintaining a certain aspect ratio); or discards the new and re-uses the old dimensions (for a window that resists resizing). *drawwindow* draws the contents of the window. *mouseclick* receives as operands the coordinates of the mouse pointer and modifier numeral (see **nextevent** in 3.2.2 for details).

topwindow places a window on top of any others when used with a **true** boolean. **mapwindow** has more complex effects on the visibility of windows. If applied to a specified window, it raises the window to the top if the boolean operand is *true* or else hides the window. If **null** is specified in place of a window# either all windows associated with the X connection are raised to the top or hidden, dependent on *bool*. To delete a window use **deletewindow**.

mapcolor translates a color specified in an array of the form < red green

blue > (intensities are numbers from 0.0 to 1.0) into a color index, a simple numeral. Drawing operators use the color index to select colors.

fillrectangle fills a rectangular area of the window with uniform color. **drawline** interconnects a set of coordinate points by a line, following the order of the points in *xy*. Similarly, **drawsymbols** marks each coordinate point with a symbol selected by *symbol*, a numeral:

0 dot
1 stroked diamond
2 filed square
3 stroked square
4 stroked square with horizontal bar
5 cross
6 x
7 filled circle
8 stroked circle
9 stroked circle with horizontal bar
10 asterisk
11 filled up triangle
12 filled down triangle
13 filled right triangle
14 filled left triangle
15 vertical bar, centered
16 verical bar, bottom adjusted
17 vertical bar, top adjusted
18 horizontal bar, centered
19 horizontal bar, left adjusted
20 horizontal bar, right adjusted

drawtext draws text starting at the position given by *x* and *y* and returns the window# and coordinates (only *x* is modified) of the position following the last written character for a subsequent use of **drawtext**. Details are specified in the list operand: the string *font_spec* selects a font name in the X windows convention; *col_idx* a color; *h_align* and *v_align* an alignment (<0 - left (bottom); 0 - center; > 0 - right (top)). Note that X windows caches up to 10 fonts so that their alternating uses do not produce unreasonable delays.

bell dings the X window bell at relative volume *vol* (as a negative or positive percentage of the server settings). A -100 is silent, and a $+100$ is maximal volume, with 0 signifying the current bell setting.

The X window library buffers X window commands prior to transmitting them to the X server. **Xsync** transmits that buffer instantly.

2.6.14 Processes and File System Operators

There are a number of operators for accessing POSIX operating system process and filesystem values. Most of these operators are wrapped in `processes.d` into procedures for running sub-processes and handling file descriptors (in dictionary ***PROCESSES***). Two new objects are used here – file descriptors, which are heap allocated objects for writing and reading to and from files and pipes, and pids, which encapsulate processes id numbers as opaque objects on the stack.

—	**fork**	pid socket false socket true
—	**getppid**	pid
[(exec) (param)...]	**spawn**	—
pid int–sig	**killpid**	—
pid	**waitpid**	int–exit
pid	**checkpid**	false int–exit true
/name (val) /name null	**setenv**	—
/name	**getenv**	(val) null
int bool–readonly	**makefd**	fd
fd	**unmakefd**	int
fd	**readonlyfd**	bool
fd–src fd–target	**dupfd**	—
fd	**copyfd**	fd

—	**pipe**	fd–read fd–write
(dir) (file) int–flags	**openfd**	fd
fd	**suckfd**	(buffer)
fd (buffer)	**writefd**	fd
fd	**getfd**	byte *
fd	**ungetfd**	—
fd	**closefd**	—
fd	**closedfd**	bool
(buffer) fd	**readfd**	(buffer) fd true (sub–buffer) false
(buffer) fd char	**readtomarkfd**	(sub–buffer) fd true (sub–buffer) false
fd char	**readtomarkfd_nb**	(buffer) fd true (buffer) false
(dir) (prefix) null (prefix)	**tmpfile**	fd–read fd–write (dir) (prefixXXXXXX)
(dir) (subdir) null (subdir)	**tmpdir**	(dir) (subdirXXXXXX)
(dir) (path) null (path)	**rmpath**	—
(dir) (subdir) null (subdir) (dir) (subdir) null (subdir)	**finddir**	(dir/subdir) [(file)...] true false
∼active fd	**lockfd**	...
∼active fd	**unlockfd**	...
∼active fd	**trylockfd**	... true false

The **fork** operator forks the current process, duplicating the current *dm*. The child process receives a connecting socket to the parent and **true**. The parent receives a *pid* object representing the child's process id, a socket to the child and false. The socket object can be used to **send** procedures and objects between the parent and child. To get the parent pid in the child, call **getppid**. The processes have identical stacks and heaps, other

than the mentioned distinctions. The child can then replace itself with an executable image with **spawn**, which takes a list specifying the path to the executable (which can be found in the **PATH** environmental variable) and parameters; of course, it never returns.

Parent and child processes can be signaled with **killpid**, which takes the process id and the signal integer and returns nothing. The signal integer mapping are defined in ***SIGNALS*** – they are not the operating system signal integers, but a mapping of a subset of signals which are common across architectures, as used by **sendsig** and **rsendsig**; a raw, operating system specific signal can be sent by shifting it by 8 bits. Children can be reaped with **waitpid**, which takes a pid and return an exit value when the child exits (and therefore blocks). The exit value is the bottom 8 bits of the child processes exit value OR'd with any signal which caused an abnormal termination (shifted by 16 bits leftward if it's not a posix standard signal, or 8 bits after being converted to a ***SIGNALS*** value and or'd with 0x80). This value can be propagated directly by calling **die** on it. **checkpid** is the non-blocking version; it returns false if the child hasn't died, and true and the exit value if it has died (however, it still leaves the child unreaped, requiring an **waitpid** call on it).

Environmental variables can be accessed with **getenv**. The name of the variable as a *D* name is given to it; if such a variable exists, the value is returned as a (possibly empty) string, otherwise **null** is returned. **setenv** is the symmetric operator, setting **/name** variable to the value passed or unsetting the variable if **null** is passed to it.

makefd converts a raw integer representing a file descriptor into a read-only or write-only file descriptor object. **unmakefd** returns the integer representing a file descriptor. The raw integers are needed for creating stdin, stdout and stderr file descriptor objects. **readonlyfd** returns true if the file descriptor is a read descriptor, rather than a write descriptor. **dupfd** makes the target file descriptor point to the same underlying stream as the source file descriptor; they must both be write (or read) file descriptors which are open. Therefore **stdout stderr dupfd** makes anything written to **stderr** appear on **stdout**, even after **stdout** has been closed. **copyfd** creates a new file descriptor (with an arbitrary new file descriptor number) that points at the same underlying stream as the source file descriptor.

pipefd creates two new connected file descriptors. Writing on the write

fd produces that same sequence appear as input on the read fd, which is useful for creating arbitrary connections of processes. **openfd** opens a file for reading or writing: if the flag is 0, the file (which must exist) is opened for reading; if the flag is 1, the file (which may or may not exist) is truncated and opened for writing; or if the flag is 2, the file (which must exist) is opened for writing by appending. Read file descriptors can be consumed by **suckfd**, which reads the file descriptor until the end and allocates a string with everything read in; the file descriptor is then closed. **getfd** reads a single character, or returns undefined if the end of the file descriptor has been reached. The last character read can be unread with **ungetfd**. An open read or write file descriptor can be closed with **closefd**. The closed state can be tested with **closedfd**, which returns true if the file descriptor has had **closefd** called on it (or it's been **suckfd**'d dry or other read operators applied which close the stream).

readfd will fill a passed-in buffer with the contents of the file descriptor until the buffer is full. If the end of the file descriptor is reached before that, the file descriptor is closed and **false** is returned as the final value over the interval of the buffer that has been filled in. **readtomarkfd** works similarly, but reads up to a given character (such as a newline), and that character is discarded. **readtomarkfd_nb** is the non-blocking version of the operator; if the read would block, it returns everything read up to that point.

tmpfile creates a temporary file in a set directory, or the directory identified by the environmental variable ***TMPDIR***, or `/tmp` if no such variable is set, with the name `prefix` with six random characters appended to it. A read and write file descriptor is returned and the directory and name of the file. **tmpdir** creates a similar temporary directory. Neither is automatically cleaned up. **rmpath** will remove a file or empty directory. **finddir** will return the path of the directory, a list of the files in the directory, and true if the directory is non-empty; if the directory is empty, the call will just return false.

A file descriptor can be locked as a form of inter-process synchronization. **lockfd** blocks until an exclusive lock (as per POSIX **lockf**) is available, then locks the file descriptor while ∼active is executing. Inside that lock the file descriptor can be unlocked for a section of code via **unlockfd**. **trylockfd** will execute ∼active if the exclusive lock is available without locking, and

add **true** to the stack after executing it while fd is locked; otherwise, it does not execute ∼active and just returns **false**. These operators handle **stop**, **abort**, **exit** and **exitto** exits from the encapsulated contexts, automatically releasing (or re-taking) locks.

Chapter 3

THE MACHINES

D machines work in a group. The smallest group might include two D machines, both running on your desktop. One serves as a terminal, the other as the computational workhorse. You could extend this group to several workhorses, which might run locally in your (multi-processor) desktop or run in different physical hosts that are networked across your lab. In a variant, your D machine serving as terminal might run on your home computer and the other machines of the group on hosts in your lab. You might also recruit a group of D machines whose physical hosts are interconnected for high-speed communication to perform strongly coupled parallel computations (a 'cluster').

Three specializations of D machine let you build groups that suit your work:

1. the D Virtual Terminal (*dvt*)
2. the D Node (*dnode*)
3. the D Pawn (*dpawn*)

The *dvt* serves as a combination of text and graphical terminal between you and one or many *dnodes* that you have recruited to do computations for you.

A *dnode* includes an extended set of mathematical operators for solving numerical problems (the set can be augmented by extrinsic operators). A *dnode* can use all processors of its host for multi-threaded computations.

A *dnode* itself can create a set of *dpawns* for solving large numerical problems in parallel on a cluster of hosts (and, automatically, all processors of each host).

The *dvt* and *dnodes*, and *dnodes* among themselves, communicate point-to-point through a bi-directional stream protocol that runs on tcp/ip. A *dnode* and its *dpawns* communicate through the Multiple Message Protocol (mpi), which usually runs on tcp/ip. The *mpi* allows a *dnode* to control a cluster of *dpawns* using broadcasted messages.

Parallel computations in *dpawns* are supported by PETSc ('Portable, Extensible Toolkit for Scientific computation'). PETSc includes a large body of solvers for linear algebra problems. PETSc works on vectors and matrices that are stored across a machine cluster rather than in one machine. Work on different chunks of the vectors and matrices is divided amongst the different members of the cluster, so that much of the work can be done in parallel.

To do as much work as needed in parallel, not only must the parts directly supported by PETSc be done in parallel but also time-consuming work specific for the problem ('filling the matrix'). To allow you to do such specific work in parallel, the *dpawn* machine is laid out as a general purpose D machine (including all computational means of a *dnode*). The vectors and matrices that PETSc operators expect to work on are created in composite D objects, so that ordinary D operators can access them as well. These D objects are created in each *dpawn* of the cluster to hold exactly the chunks of the vectors and matrices that this *dpawn* is assigned to work on. These objects contain mapping information locating the local objects in the global vectors and matrices. You create the PETSc objects of each *dpawn* by code specific to this *dpawn* (this way parcelling the global vectors and matrices yourself rather than have PETSc operators assign the parcels). The code to fill in these objects and to do computations on them using PETSc can be the same for all *dpawns* provided you write the code for filling the parcels using the mapping information contained in the PETSc objects.

All phases of operating the *dpawn* cluster are supported by a library of procedures that is loaded with the 'startup' code of the *dnode*. The library very much makes the *dnode* look as the machine that solves the entire problem while the involvement of the *dpawn* cluster remains visible only in a small amount of overhead.

3.1 Overview of machine-specific operators

The following table summarizes the *operators* that are specific for one or more varieties of D machine, or whose behaviors differ among machines. Only actual operators are included. These specific operators are complemented by machine-specific procedures ('operators written in D') that the different D machines derive from their specific `startup_xxx.d` files.

Table 3.1: Operators for dvt, dnode and dpawn machines

operator	dvt	dnode	dpawn
connect	+	+	-
disconnect	+	+	-
send	+	+	-
sendsig	+	+	-
rsendsig	-	+	-
getsocket	+	+	-
getmyname	+	+	-
getmyfqdn	+	+	-
Xwindows	+	+	-
Xdisplayname	+	+	-
screensize	+	+	-
makewindow	+	+	-
deletewindow	+	+	-
mapwindow	+	+	-

continued on next page

continued from previous page

operator	dvt	dnode	dpawn
resizewindow	+	+	-
Xsync	+	+	-
mapcolor	+	+	-
drawline	+	+	-
drawsymbols	+	+	-
fillrectangle	+	+	-
drawtext	+	+	-
makewindowtop	+	+	-
bell	+	+	-
Xwindows_	-	+	-
Xconnect	-	+	-
Xdisconnect	-	+	-
nextevent	+	-	-
aborted	+	-	-
getmyport	-	+	-
setconsole	-	+	-
console	-	+	-
killsockets	-	+	-
socketdead	-	+	-
loadlib	-	+	+
nextlib	-	+	+
quit	+	-	+
die	+	+	+
lock	-	+	+
unlock	-	+	+
locked	-	+	+
serialize	-	+	+
threads	-	+	+
makethreads	-	+	+

continued on next page

continued from previous page

operator	dvt	dnode	dpawn
inter_lock	-	+	+
inter_unlock	-	+	+
inter_lock_set	-	+	+
tostderr	-	+	+
halt	-	+	+
continue	-	+	+
vmresize	-	+	+
getplugindir	-	+	+
getexecdir	+	+	+
getstartupdir	+	+	+
getconfdir	+	+	+
gethomedir	+	+	+
matmul_blas	-	+	+
decompLU_lp	-	+	+
backsubLU_lp	-	+	+
invertLU_lp	-	+	+
norm2	-	+	+
matvecmul_blas	-	+	+
triangular_solve	-	+	+
givens_blas	-	+	+
rotate_blas	-	+	+
rthreads	-	+	-
makerthreads	-	+	-
checkrthreads	-	+	-
rsend	-	+	+

continued on next page

continued from previous page

operator	dvt	dnode	dpawn
mpiprobe	-	-	+
mpiiprobe	-	-	+
mpisend	-	-	+
mpirecv	-	-	+
mpibarrier	-	-	+
mpibroadcast	-	-	+
mpirank	-	-	+
mpisize	-	-	+
groupconsole	-	-	+
petsc_vec_create	-	-	+
petsc_vec_copy	-	-	+
petsc_vec_copyto	-	-	+
petsc_vec_copyfrom	-	-	+
petsc_vec_syncto	-	-	+
petsc_vec_syncfrom	-	-	+
petsc_vec_max	-	-	+
petsc_vec_min	-	-	+
petsc_vec_destroy	-	-	+
petsc_mat_create	-	-	+
petsc_mat_copy	-	-	+
petsc_mat_copyto	-	-	+
petsc_mat_copyfrom	-	-	+
petsc_mat_syncto	-	-	+
petsc_mat_syncfrom	-	-	+
petsc_mat_destroy	-	-	+
petsc_mat_dup	-	-	+
petsc_mat_vecmul	-	-	+
petsc_ksp_create	-	-	+
petsc_ksp_destroy	-	-	+
petsc_ksp_tol	-	-	+
petsc_ksp_iterations	-	-	+

continued on next page

continued from previous page

operator	dvt	dnode	dpawn
petsc_ksp_solve	-	-	+

3.2 The D Virtual Terminal (*dvt*)

You can start a *dvt* process from a shell. Better support is provided to a *dvt* through a *dvt*-configured *emacs* environment. In this environment, you can start a *dvt* by the *emacs* command 'esc-x dvt'. In both shell environments, the *dvt* inherits the shell as its console. You therefore retain the editing capabilities of that shell when you work at the console of the *dvt*. In addition, the *emacs* environment provides many *dvt* supporting functions through bound keys. *emacs* itself will operate through an X Window system. The *dvt* then will bring up an additional, graphical user interface to itself consisting of three *dvt* windows.

3.2.1 The *dvt* mill

The test phase of the *dvt* mill is implemented in D code (cf. 2.4). This D code is loaded upon initialization when the file `startup_dvt.d` is interpreted. The D code of the outermost loop of the *dvt* uses a dvt-specific operator, **nextevent**, to determine, and react to, an event that requires the attention of the machine. This operator delivers requests made by you at the console of the *dvt*, requests made by windows associated with the *dvt*, requests made from *dnodes* over network connections, and an emergency signal. By this *modus operandi* a *dvt* will attend to new requests made to it only after its current activity is completed. (This is an important difference with regard to the behavior of a *dnode* or *dpawn*). A *dvt* is only interrupted upon sending it an emergency signal: pressing 'control-c' from *bash* or 'control-c control-c' from the *emacs* shell. Upon receiving this signal, the *dvt* will execute the **abort** operator. Likewise, when an error is discovered in a *dvt* activity itself, **abort** is executed after displaying an error message.

If you make the *dvt* perform major services (e.g., automate the coordination of a job involving many *dnodes*), you must organize your D code such that it can be executed in short bursts of mill cycles rather than, in the worst case, in an endless loop that prevents the outermost loop of the *dvt* from attending to external events through **nextevent**. If your *dvt* code being tested turns out such an obstinate customer, use the emergency signal from the keyboard to restore *dvt* responsiveness.

The operation of the *dvt* mill ceases when the operator **quit** is executed; this terminates the Linux process of the *dvt*.

Whenever a component of the *dvt* detects an error condition, it initiates a consistent error response: a string (revealing the instance of discovery, like the operator name) and a numeral (coding for the type of the error) are pushed on the operand stack, and the 'error' is executed. The operator **error** uses the string and numeral on the operand stack to formulate a console error message; after showing this message, **abort** is executed. This drops all stacks and resumes execution of the outermost loop of the *dvt*.

3.2.2 The *dvt* operators

The operator set of the *dvt* includes operators that either are unique to the *dvt* or are implemented there in a unique way. Many of these 'operators' actually are D procedures defined in the file `startup_dvt.d`.

stringbuf	**nextevent**	eventparams...
active–obj	**aborted**	—
instance–string error–num	**error**	—
instance–string error–num stringbuf	**errormessage**	message_substring
—	**quit**	—
return–int	**die**	—

nextevent takes a string buffer as operand and blocks until an event requiring *dvt* attention occurs. Several kinds of event are detected and cause **nextevent** to push an active name on the execution stack. These names and their significance are:

consoleline	`a phrase is available from the console keyboard'
nodemessage	`a message has been sent from a *dnode*'
windowsize	`request to change the size of an X window'
drawwindow	`request to (re)draw an X window'
mouseclick	`a mouse click into an X window has occurred'

Additional information relevant for the event is communicated by **nextevent** as follows:

consoleline The keyboard phrase is returned in a substring of the string buffer operand

nodemessage Dependent on the form of the message used in the **send** operator executed in the *dnode*, a **save** operation is performed, and/or the root object of a received object tree is pushed on the operand stack. In all usages of **send**, the received message string (substring of the string buffer) is pushed on the operand stack.

If the event originates from activity in an X window, the dictionary associated with the window name is pushed on the dictionary stack; this dictionary must be defined by you in **userdict** and be associated there with a properly formed name: the letter 'w' followed by digits specifying the window# (see 2.6.13). The dictionary must associate ***windowsize***, ***drawwindow***, and ***mouseclick*** with window-specific procedures. These procedures receive their operands from **nextevent**:

windowsize The width and height of the window are pushed on the operand stack

drawwindow No operands are passed

mouseclick The abscissa, ordinate, and modifiers describing the mouse event are pushed on the operand stack. Modifiers are generated by keyboard keys simultaneously held down when the mouse button is clicked, or by using the 2nd to 5th button of a multi-button mouse. The encoding of the modifiers is the standard encoding of the X Window library (see file `startup_dvt.d` for assignments used in the *dvt*).

Procedures named ***consoleline*** and ***nodemessage*** are defined in **userdict** when the file `startup_dvt.d` is interpreted. These procedures are part of the D code that implements the behavior of the *dvt* that we describe in this section.

aborted marks the current top object of the execution stack as the object to which the stack is dropped when **abort** is executed. Typically there is only one **aborted** context in a *dvt*. **error** is automatically invoked when

a *dvt* operator discovers an error condition. **error** expects two operands indicating the instance and cause of the error condition on the operand stack. It assembles a message string from this information (decoding the error number into a text message), prints the error message on the console screen, and forces the execution of **abort**. **errormessage** also composes an error message; it returns the message as substring of the designated string buffer and resumes normal execution.

quit terminates the *dvt* process. **die** will terminate the process like quit, but allows a return value. If the bits above the least significant byte are zero, then the lest significant byte is the return value. Otherwise, if the next byte is non-zero, then it's 7 least signifcant bits (top bit masked) refers to the architecture neutral signal (as in ***SIGNALS*** in `startup_common.dvt`. The last options is the remaining bits, which if non-zero when shifted right by 16 is the raw, operating system specific signal, as in *C*'s ***kill***.

Supervising of *dnodes*

When a *dvt* hooks up over a network with *dnodes*, it serves as the console of these D machines. The *dvt* provides a mechanism to switch the console among the various D machines that are hooked up to it. Initially, the console targets the *dvt*'s D machine itself.

—	***h_***	—
—	***hk_***	—
(hostname) port# group#	***_c***	—
(hostname) port# group# dim_array	***_csu***	—
—	***c_***	—
node#	***_dc***	—
node#	***_dx***	—
node#	***kill***	—
node#–or–group#	***_t***	—
node#	***_r***	—

h_ prints a help message summarizing all 'operators' that are provided to the *dvt* via the `startup_dvt.d` file.

hk_ prints a help message summarizing all keys that have been bound to *dvt*-supporting *emacs* functions when the *dvt* runs inside *emacs*.

_c opens a connection between the *dvt* and the *dnode* described by the name of its host and the D machine port where this *dnode* is listening (this port number is assigned to the *dnode* when the *dnode* is started from a shell (see the section on *dnodes*)). Port numbers are positive integers including 0. When the connection is established, the target *dnode* is instructed to use this *dvt* as its console and to send *X Window* communications to the *X Window* server on the host of this *dvt*. (There is a more fancy version of this command that tells also the *dnode* to color all its text communications with the *dvt* so that they appear as visually distinct.) The connected *dnode* is assigned a node number. Node numbers are positive integers (0 is reserved for the *dvt*) and are assigned and disassigned to *dnodes* as these are connected and disconnected (thus node 3 may be unused, whereas node 4 may be in use). Connected *dnodes* can be grouped by associating them with a group number (a negative integer). The *dvt* broadcasts console phrases to all *dnodes* of a group when a group is selected as the current target.

_csu includes the functions of ***_c*** but also sets up the storage capacities of the *dnode* to the dimensions specified in *dim_array* that has the entries: <1 opdssize excssize dictssize vmsize_in_MB userdictsize > ('opdssize', for instance, is the capacity, in objects, of the *dnode's* operand stack). *_csu* also instructs the *dnode* to read and interpret the `startup_dnode.d` file, which defines a library of procedures in the primed *dnode*.

c_ prints on the *dvt* console screen the node numbers of the dvt (which is 0), and of all currently connected *dnodes* together with their host name, port number, status ('ready' or 'busy'), and group number.

_dc disconnects the specified *dnode* from this *dvt*. Note that you can connect, disconnect, and reconnect *dnodes* as often as you wish without interfering with ongoing activity in the *dnode's* D machine: the *dnode* can continue to execute D code that constitutes a long job, independent of whether it is connected to a *dvt*. ***_dc*** does not tell the node to close *X Window* activity on the *dvt* host. You can continue to interact with the *dnode* using these windows, even though the *dnode* is no longer connected to a *dvt* on this host.

_dx acts like **_dc** but also instructs the *dnode* being disconnected from the *dvt* to close all *X Window* activities with the *dvt* host.

kill acts like **_dx**, and, in addition, tells the *dnode* being disconnected to resize its memory allocations to those of a dormant *dnode*. This terminates all ongoing activity in that *dnode*, except its listening to incoming connection requests.

_t selects a node or group of nodes as the target(s) for subsequent console input. (again, node #0 directs keyboard input to the *dvt* itself).

_r forces the 'ready' status of the node. When a *dnode* that is the currently selected target is given a console phrase to execute, the status of this node is set to 'busy', and the console phrase is delivered to the *dnode* together with a request to inform the *dvt* when the activity started by this console phrase has been completed (or stopped). When the *dvt* receives this completion message, it sets the status of the node to 'ready'. This mechanism ensures that activity initiated by a console phrase delivered to a *dnode* will be completed before you can give another console phrase to the *dnode*. (*dnodes* interrupt their activity to respond to requests made on their connections). If you try to give a console phrase to a 'busy' node, the *dvt* will refuse (discard) it and warn you by the console message "Wait!". You can override this mechanism by prefixing your console phrase with a tag (see below). Note that this mechanism belongs to the *dvt* — *dnodes* are unaware of its existence.

Many of the *dvt* operators described above can be invoked, in a more convenient interface, by selecting from an *X Window* windows entitled "TheHorses" and "DVT Macros" (described later in this section).

3.2.3 Tagged console phrases

Certain characters when appearing as the first character of a console phrase direct the *dvt* to treat the remainder of the console phrase in some special way. The effect of the tag will also depend on the currently selected target type, *dvt*, *dnode* or *dpawn*.

No tag *dvt* execute phrase as D code

	dnode	transmit phrase as D code to the *dnode*, observing the `ready/busy' rule
	dpawn	transmit phrase as D code to the *dnode*, which then retransmits phrase as D code to the D pawn, observing `ready/busy' rule for both
!	*dvt*	execute phrase as D code
	dnode	transmit phrase as D code to the *dnode*, disregarding the `ready/busy` rule
	dpawn	goes directly to *dnode*, as above; *dpawn* selection ignored.
$	*dvt*	submit phrase as Unix shell command to the host of the *dvt*
	dnode	transmit phrase and submit it as Unix shell command to the host of the *dnode*, observing the `ready/busy' rule
	dpawn	same as *dnode*, but directed at *dpawn* instead.
#	*dvt*	execute phrase in the D machine of the *dvt*, use results to build a Unix shell command, submit the shell command to the host of the *dvt*
	dnode	execute phrase in the D machine of the *dvt*, use results to build a Unix shell command, submit the shell command to the host of the *dnode*, observing the `ready/busy' rule
	dpawn	same as *dnode*, but directed at *dpawn* instead.
@	*dvt*	execute phrase in the D machine of the *dvt*, use results to build a Unix shell command, submit the shell command to the host of the *dvt*
	dnode	transmit the phrase to the *dnode* and execute it in the D machine of the *dnode*; make the *dnode* build a Unix shell command from the results, and submit it to the host of the *dnode*, observing the `ready/busy' rule
	dpawn	same as *dnode*, but directed at *dpawn* instead.
^	*dvt*	execute phrase as D code

	dnode	execute phrase as D code
	dpawn	execute phrase as D code on the *dnode*, disregarding *dpawn* selection
%	*dvt*	execute phrase as D code
	dnode	execute phrase as D code on the *dvt*, disregarding which *dnode* owns the keyboard
	dpawn	execute phrase as D code on the *dvt*, disregarding which *dnode* owns the keyboard – and any *dpawn* selection, of course

The @ and # tags have complex effects — you will hardly use them when typing console phrases (they rather are used in 'macros' produced by clicks into the "DVT Macros" window maintained by the *dvt* when it uses *X Window*. The ^ and % tags are also rarely used directly; their primary use is for "DVT Macros" commands that must be directed to the *dvt* or a *dnode*, bypassing temporarily any "TheHorses" or "ThePawns" selections.

3.2.4 Key bindings supporting the *dvt*

When a *dvt* is started through the *emacs* command 'esc-x dvt', the following key bindings with *dvt*-supporting *emacs* functions are created:

f1	execute the previous console phrase
f2	send 'continue' to the dvt/dnode
f3	send 'stop' to the dvt/dnode
f4	send 'abort' to the dvt/dnode
f5	pop up "TheHorses" window
f6	pop up all dvt/dnode windows
shift-f6	iconify all dvt/dnode windows
f7	raise the dvt emacs frame
f8	start a local dnode
control-h d	get this help
control-!	ignore busy state when sending to dnodes
control-1	same as control-!
control-c c	clear preceding text from emacs window
control-c control-a	send phrase wrapped in 'debug abort'
control->	send output to log file
control-c control-n	narrow dvt buffer
control-c control-w	widen dvt buffer

If you wish to see also the names of the corresponding emacs Lisp functions, use 'control-h d'.

Another service provided by the *emacs* environment is language support for D. When you edit D code in a edit buffer or the shell buffer you receive editing support (e.g. matching parentheses are indicated). The different D objects are highlighted in color (you can toggle this feature off or on using the standard *emacs* command 'esc-x fontlockmode').

3.2.5 The graphical user interface of the *dvt*

When the *dvt* process is running on a host that provides an *X Window* server, it will create and maintain three windows, two of which are *dvt*-specific and one of which is also implemented by *dnodes*. The D code for operating these windows is contained in the file `startup_dvt.d`, so that it is loaded when the *dvt* process is started up. The three windows are:

"TheHorses" shows the list of D machines that currently form a cluster. The *dvt* is listed at the top, followed by the connected *dnodes* (described as hostname: port#). The background of each entry shows

the status of the node: white for 'ready', and green for 'busy'. The currently selected target(s) for console input is (are) are highlighted by showing their name(s) in blue and boldface. You can select a D machine as target by a mouse click. You can select a group of *dnodes* as targets by control/clicking on one of the nodes of the group. Only one *dnode* or group will be selected at a given time. You can force the *dvt* to set a *dnode* connection to the 'ready' status by shift/clicking on it. If you have a multi-button mouse: shift/click and control/click are equivalent to clicking on buttons 2 or 3.

From the keyboard, the emacs key sequence `meta-down` (usually Alt-Cursor_Down) and `meta-up` (Alt-Cursor_Up) switch between targets in "TheHorses". The emacs key sequence `meta-shift-down` (Alt-Shift-Cursor_Down) and `meta-shift-down` (Alt-Shift-Cursor_Up) selects the next *group* or last *group*.

"DVT Macros" displays a list of command mnemonics. When you click on a mnemonic, a string is placed into the console screen. You can execute this string as a console phrase by moving the cursor over it and pressing 'return' (function key 'F1' combines both actions). Typically, you will have to click two mnemonics to produce a complete command, one designating a source and the other a destination object. The source and destination objects, in turn, are derived from objects selected in the third window, called "TheEye".

For instance, you may:

1. select a .tex file using "TheEye" by clicking on it
2. click in "DVT Macros" on 'tex' on the 'PrintFrom' line
3. click in "DVT Macros" on 'pdf' on the 'PrintTo' line
4. press 'F1'.

This will compile the .tex file and submit it to a program for display and optional printing. The services of the "DVT Macros" are defined by shell scripts and by D code contained in `startup_dvt.d`.

Note that you can modify console phrases produced by "DVT Macros" by editing them prior to submitting them for execution: clicking mnemonics of "DVT Macros" simply provides templates for com-

mands that are hard to remember. Such a command phrase is placed into the console buffer so that you can execute it (press 'F1'), or execute it after some editing from your hand.

The bottom line is a gui interface to keyboard commands: press "f1-send" to send the last command, as if by pressing 'F1', "f2-continue" acts as if you pressed the 'F2' key, sending `! continue` and ending "scream mode", and so forth.

"TheEye" lets you view any object contained in the associated D machine, and any file system known to the host from which the "TheEye" originates. Thus "TheEye" is the universal browser of D machines. Both the *dvt* and *dnodes* connected to a *dvt* will create and maintain their own "TheEye" windows (with their owners identified in the window header). "TheEye" is operated through the mouse.

- The body of the window is divided into two pages. The left page shows permanent objects at the top followed by objects opened during use of "TheEye". The right side lists the objects in the currently open object.
- A field along the bottom of each page lets you scroll to the beginning or end of the page, up and down by one page, or up and down by one line. Click on the respective symbols.
- Composite D objects are highlighted in both pages by backgrounds in different shades of green; simple D objects are displayed on a white background. File directories are listed on a purple background, and simple files on white background.
- The left page shows in two columns the class of D object in an intuitive notation, or 'DIR' for file directories. The second column shows the name of the object.
- The right page shows in the left column the name of D objects and in the right column an intuitive description of the composite object or the value of a simple object. A filesystem is displayed in a single column of names, starting with directories (on a purple background) and followed by simple files. Each group of entries is alphabetically sorted.
- A field along the top of the two pages is used to display information on the object (of either page) that you click on.

- Control-clicking (or clicking the right mouse button) onto a composite D object or directory opens the contents of that object in the right page and appends the object to the left page (unless it already exists there). You can remove an item from the left page by shift-clicking on it.
- Clicking the middle mouse button (or pressing the 'superkey' and the left mouse button) selects an item. You can select only one item on the left page but many items on the right page. Selected items are highlighted in boldface and red (left page) or blue (right page).

Selected items can be inquired by your D code from "TheEye" running from the same D machine using procedures that are loaded with the 'startup' file into *userdict*:

stringbuf index	***faxLpage***	stringbuf new_index
stringbuf index	***faxRpage***	stringbuf new_index
—	***getLpage***	object true
		false
—	***getRpage***	[D_object ...]
		[(path) [(dir/filename) ...]]
		[ ]

faxLpage or ***faxRpage*** append to a byte array the string representation of a directory selected in the left page or of directories/files selected in the right page and update the buffer index to the position folowing the insertion. The string format is suitable for presenting arguments to shell commands. If nothing is selected or other types of object are selected, nothing is appended.

getLpage returns a D object selected in the left page, or a string containing the path of a selected directory, and true, or simply false if no object is selected. ***getRpage*** returns a list, which is empty if no object of the right page is selected. Selected D objects are returned as the value of the list. For selected directories or files a path string followed by a list of directory/filename strings is returned.

3.3 The D node (*dnode*)

The *dnode* is the specialization of D machine that provides you with computational workhorses. A *dnode* is brought up by a shell command addressed to its host. The command is 'dnode #' where '#' is a numeral that defines a communication port at which this 'dnode' will listen for connection requests made via the network. (The specified port number will be used as an offset into the range of user-reserved ports, with no absolute warranty that your chosen port is free – in rare instances you may have to choose a different port). The D machine of the *dnode* is assigned minimal memory resources at this time. More memory can be allocated when you take control of the *dnode* via the network.

A *dnode* is alerted (from a state of dormancy that uses no CPU cycles, or amidst the execution of D code) whenever a connection request is made to it, when an established connection is the receiving end of the **send** operator executed in another D machine, or when an established *X Window* connection requires attention. The code that schedules reactions to such events is buried in the mill of the *dnode* (whereas in the *dvt* these events are attended to by code written in D).

The *dnode* mill tests for external conditions at intervals no greater than the time needed to go round 100 turns, which typically is a very small amount of time on a human scale. When an event is detected it is attended to, interrupting and suspending ongoing execution of D code. You can protect a D context against such interruption by executing it using the **locked** operator. A *dvt* normally will pace the execution of your console phrases so that the *dnodes* it supervises will not be interrupted. X window events do not pace their demands made to the *dnode*, particularly when they arise from an impatient user. It is therefore a good idea to protect X window responses of a *dnode* by **lock**. The interruption of other D code by X window events is not a problem because X window response procedures leave behind the same D machine state that existed when the interrupt occured (please remember when you write your own).

The *dnode* mill schedules D code in response to X window events in the same way as the *dvt* operator **nextevent** (see 3.2.2). Thus the same D code can be used in *dvt* and *dnode* machines to service X windows. When a message is received from another D machine (that uses **send**), the re-

ceived object is made active and pushed on the execution stack. If this (composite) object is not a string, a **save** is executed before the object is unfolded into the VM space (see also 2.6.11).

A *dnode* is terminated by killing its process from the Linux shell that was used to bring it up. A *dnode*, however, will typically not be killed after a job: it rather is put into a dormant state (using **vmresize**) from which it can be recruited by a *dvt* for new work. Thus *dnodes* are used like servers.

Whenever the mill of the *dnode* detects an error condition, it initiates a consistent error response. Four objects are pushed on the operand stack (from bottom to top): the host name of the *dnode*, the port# of the *dnode*, a string (revealing the instance of discovery, like the operator name), and a numeral (coding for the type of the error). The active name 'error' is pushed on the execution stack. 'error' normally resolves to the operator **error**. **error** formulates an error message from the four objects pushed by the mill on the operand stack, sends the message to the current console, and invokes the operator **halt**. **halt** has two effects: (1) it pushes a copy of itself on the execution stack and (2) it directs the mill to submit phrases entered to the console to execution. Thus you can interact with the failed *dnode*, e.g., inspect stack and object contents related to the error while the context that produced the error is suspended. If you can fix the error from the console, you may choose to execute **continue**, which drops the execution stack below the topmost **halt** object and thus resumes execution of the context suspended by that **halt**. Alternatively, if you consider the problem fatal, you can terminate the flawed context by executing **stop** (if you have set up a capsule with **stopped**) or **abort**. **abort** drops all stacks to their floors (which on the dictionary stack leaves only **systemdict** and **userdict**). This cancels all activity in the *dnode*, and returns the machine to a state in which it will respond to *dnode* and X window connections.

3.3.1 Operators for administrating a *dnode*

long_array/null	**vmresize**	bool
—	**halt**	—
—	**continue**	—

—	**stop**	—
—	**abort**	—
active_obj	**lock**	??
active_obj	**unlock**	??
—	**locked**	bool
—	**console**	consolesocket/null
consolesocket/null	**setconsole**	—
string	**toconsole**	—
string	**tostderr**	—
hostname port# sourcestr numerr	**error**	—
hostname port# sourcestr numerr strbuf	**errormessage**	mesg_str
—	**getmyport**	port#
—	**Xwindows_**	bool
(hostname:screen#)	**Xconnect**	—
—	**Xdisconnect**	—
num	**makethreads**	—
—	**threads**	num
active_obj	**serialize**	...
active_obj	**inter_lock**	...
active_obj	**inter_unlock**	...
bool	**inter_lock_set**	—
—	**getlibdir**	string
(path) (filename)	**loadlib**	lib_dict
—	**hi**	(library description)
—	**libnum**	num
null/lib_dict	**nextlib**	dict true false

The *dnode* lets you resize the VM and stack dimensions (**vmresize**). When given the **null** object as operand, **vmresize** establishes a tiny VM, renders the *dnode* dormant, and returns **true**. When given the array of type long as operand, **vmresize** establishes stack, VM, and **userdict** dimensions as specified by the elements of the array, in the order: operand stack size (in objects), dictionary stack size, execution stack size, VM capacity (in MB), and userdict capacity (in associations). **vmresize** indicates sucess/failure of the attempted memory allocation in *bool*. When used with the array

operand, **vmresize** also sets the startup and working file directories back to the original working directory (from which the *dnode* has been started up); lastly, **vmresize** makes the *dnode* interpret the file `startup_dnode.d`. **vmresize** requires that *dnode* be in 'tiny' state if it gets passed an array, and that it *not* be in tiny state if it gets pass **null**.

halt suspends the execution of the object file currently held on the execution stack, while accepting and executing keyboard phrases from the console as well as any new objects placed on the execution stack via the console (e.g., to investigate a condition that produced execution of **halt**). The suspension is removed (and the halted activity continued) by **continue**. **stop** or **abort** also unblock the execution stack but drop the execution stack according to the latest context established by **stopped** or to the bottom of the stack (**abort** also clears operand and dictionary stacks, excluding the permanent dictionaries).

The *dnode* normally reacts to any message received from another D machine within 100 turns of the mill. If a context is not to be interrupted this way, execute this context by making it the operand of **lock**. Lock contexts can be nested, including unlocked contexts by calling **unlock** with an active object to be executed. Both operators return whatever their operand returns, unchanged. Both are also **stop**, **abort**, and **halt** safe (**halt** automatically creates an unlocked context for itself). **locked** reports whether the innermost lock context is **lock**'ed or **unlock**'ed. If the *dnode* hits the bottom of the execution stack (say via an **abort**), the lock context is reset to unlocked.

Several operators allow a *dnode* to organize its interactions with the current console. **console** inquires the socket currently subserving the console connection. It returns a socket-type **null** object when a console is connected. Otherwise, a plain **null** object is returned. **setconsole** when given the object returned by **console** directs **toconsole** to the connected console or to 'stderr' of the *dnode* host. **toconsole** sends the contents of its string operand to the target chosen by **setconsole**. **tostderr** sends its string operand always to 'stderr' of the host.

Upon detecting an error the *dnode* mill invokes the active name *error*, which it provides with four operands: hostname (string), port number (where the *dnode* is listening for connection requests, actually an offset used when the *dnode* is started from a shell), a string describing the source

of the error, and a numeral specifying the nature of the error (positive: D machine error; negative: extrinsic operator error) on the operand stack. *error* is by defaults associated with an operator that composes an error message from this information (in red) and sends it to the current console (or 'stderr' by default). *error* executes *halt* after the error information has been successfully decoded; otherwise it executes *abort*. *errormessage*, rather than sending the error message to the console and interrupting execution in the *dnode*, simply places the message into a string buffer and returns the message string (excluding color accents) for use by *dnode* code itself..

*Xwindows_* inquires whether X Window capabilities have been compiled into this *dnode* when the *dnode* was built (by contrast, the common operator *Xwindows* inquires whether a connection to an X Window system has been established so that you can make windows). An X Window connection is made by *Xconnect* and broken by *Xdisconnect*. See 2.6.13 for a description of the protocols by which the *dvt* and *dnode* cooperate to connect up to X windows.

The standard mathematical operators described in Chapter 2, and their 'old-style' extensions described in 3.3.2, can be executed using multiple threads (processors) available in the host Linux box. *makethreads* creates the number of threads specified by the numeral operand (1 through a maximal number, which currently is 8). When a *dnode* is started up or when it is reduced to the dormant state by *vmresize*, it operates with a single thread. It is possible, though unlikely to be helpful, to have more threads than physical processors in a Linux box. If you wish to execute a context in a single thread although multiple threads currently exist, submit the context to *serialize*. *threads* inquires the current number of created threads. The 'new-style' (BLAS- and LAPACK-based) extensions of mathematical operators use an intrinsically set number of threads (fixed when the *dnode* program is built for the particular host).

The *inter_lock* operators are for coordinating the actions of forked children, especially regarding the behavior of multi-threaded operators. If **inter_lock_set** is called with a value of 'true', then all threaded operators (such as BLAS and array operators) will block on a semaphore shared between all d-machines descended from the man *dnode* or *dpawn*. The operators **inter_lock** and **inter_unlock**apply the same inter-process semaphore lock

or unlock respectively while executing its operand, respecting **inter_lock_set**.

In addition to the intrinsic operators, *dnodes* can make use of libraries of extrinsic operators (which typically support specific projects or experimental extensions of the D machine). **getplugindir** inquires the directory path where libraries for the *dnode* are stored (this path is established when the *dnode* is started); **getexecdir** returns the path to internal executables (executables not intended for direct end-user use). **gethomedir** returns the home directory of the user who launched the D-machine. **getconfdir** returns the system configuration directory. *loadlib* loads a library of operators; libraries are stored from the top of the VM downwards (competing for VM space with D objects, which are stored from bottom up) and returns the dictionary of the extrinsic operators contained in that library. *loadlib* also merges the extrinsic operators of the library into *systemdict* (with the exceptions of *hi* and *libnum*). You cannot load multiple copies of the same library. A library dictionary must include two obligatory operators with library-specific effects, *hi* and *libnum*. *hi* returns a string specifying the library name and version. *libnum* returns a dynamic index for this library (1,..., in the order of loading). You can inquire which libraries are currently loaded by *nextlib*, which works similarly to *nextobj*. Extrinsic operator libraries are not individually unloaded; they rather are altogether removed by *vmresize*.

3.3.2 More mathematical operators (old style)

These operators have been added to the D machine as ad hoc extensions. They fill demands made by specific applications and alleviate computational bottlenecks. The old style operators should be considered obsolete for most new work – they have been or will be replaced by new-style versions that execute faster by better use of the hardware. These old-style operators describe two-dimensional arrays as lists of row arrays. New-style operators will consistently use different representations of multi-dimensional arrays.

c a b	**matmul**	c
b a	**mattranspose**	b
c a b	**matvecmul**	c
array	**integrateOH**	array
array	**integrateRS**	array
y_array dx_array	**integrateOHv**	
a b c r u	**solvetridiag**	u bool
bandlist bandrh main_idx	**solv_bandmat**	bool
a idx	**decompLU**	d true
		false
a idx b	**backsubLU**	b
a idx b	**invertLU**	b true
		false
bandlist llist main_idx idx	**bandLU**	bandlist llist main_idx idx bool
bandlist llist main_idx idx b	**bandbs**	b
array dir	**complexFFT**	array
array dir	**realFFT**	array
array dir	**sineFFT**	array

matmul forms the matrix product $c = a * b$. Each matrix is described as a list of row arrays. The number of columns in a must equal the number of rows in b, and in c the number of rows is the same as in a and the number of columns is the same as in b. All arrays are of type '/D'. *mattranspose* transposes matrix a and returns the result in matrix b (the number of colums in a equals the number of rows in b, and the number of rows in a equals the number of columns in b). *matvecmul* forms the product $c = a * b$ of the matrix a with the column vector b (given as an array) and returns the result in column vector c (an array). (If matrix a is $m \times n$ then b is of dimension n and c is of dimension n). These matrix operators are multi-threaded (like the common mathematical operators).

integrateOH forms a running sum of array elements in place using the one-half rule of integration (it takes arrays of all types and it returns twice the sum). *integrateRS* forms the simple running sum of a '/D' type array. *integrateOHv* takes any type of ordinate array and replaces its value by twice the running integral based on the one-half rule and the (generally non-uniform) abscissa intervals given in *dx_array* (which needs not match the type of *y_array*).

solvetridiag takes five array operands all matching in type (either '/S' or '/D'): *a* holds the subdiagonal matrix elements (the first element is ignored); *b*, the diagonal elements; *c*, the supradiagonal elements (the last element is ignored); *r*, the right-hand sides; *u* receives the solution of the tridiagonal linear equations system. The boolean indicates whether a valid solution was obtained.

solve_bandmat solves a linear equation system whose coefficients are a bandmatrix (based on *bandet1* and *bansol1* of Wilkinson/Reinsch I/6). The list *bandlist* contains the coefficients as band arrays running parallel to the main diagonal and starting with the bottom-most diagonal; the index of the main diagonal array in *bandlist* is given by the numeral *main_idx*. The first element of all band arrays corresponds to the first column of the matrix; positions falling outside the matrix must be filled by zeroes. *bandrh* provides the right-hand side vector and is overwritten by the solution, whose validity is indicated by *bool*. All arrays must have identical dimensions and type '/D'.

decompLU performs the LU-decomposition of the matrix *a* in place (given as a list of row arrays of type '/D'). The array *idx* is of type '/X' and receives row permutation information to be used by *backsubLU*. The returned numeral *D* signals an even/odd number of permutations, and *bool* signals a valid decomposition (or a singular matrix). *backsubLU* computes the solution of a linear equation system using the LU-decomposed coefficient matrix *a* with the permutation information *idx* and a column vector of right-hand values *b*, which is overwritten by the solution. *invertLU* inverts the matrix *a* and returns the inverse in matrix *b* (internally using LU-decomposition and repeated backsubstitution); upon return, *a* and array *idx* contain the LU-decomposed matrix and row permutation information. A **false** value of *bool* signals singularity of the matrix.

bandLU performs the LU-decomposition of a bandmatrix described by *bandlist* and *main_idx* (see *solve_bandmat*) and returns the results in *bandlist*, the bandmatrix buffer *llist*, and the '/X' array *idx* (see *decompLU*). *bool* is set false when singularity is discovered. For details see Press et al. 2.4. *bandbs* computes a solution of the banded linear equation system for the right-hand vector given in array *b*, which overwrites the vector.

complexFFT perform the FFT transform on an array of complex numbers (presented as alternating real and imaginary parts) in place, in the di-

rection specified by *dir* (1 - forward, -1 - inverse). The array is of type '/D' and its dimension is a power of two. *realFFT* performs the forward transform on an array of real numbers, returning in the modified array: real parts of first and last spectral point, then alternating real and imaginary parts up to the Nyquist frequency. *sineFFT* takes an array of real numbers and returns the real amplitudes of the sine spectrum and vice versa, as directed by *dir*. Sequential application of the forward and reverse transforms reproduces the original array data in all forms of these FFT transforms.

3.3.3 More mathematical operators ('blas' and 'lp' styles)

The operators of this group are based on matrix-algebra libraries that have been imported into the D machine (BLAS and LAPACK). These libraries accept matrices that are mapped in a variety of possible ways on memory for minimizing space demands while maintaining efficient access. We have implemented at this time only the general matrix. Maps for banded or other forms of sparse matrices could be implemented, but are not at this time.

Although these linear-algebra operators use only vectors and matrices, we base their usage on a general map that projects a n-dimensional array onto a one-dimensional D array. The D array is dimensioned to hold (at least) the entire value of the n-dimensional array. The projection is separately described by a *map*, which is in the form of a list of numerals

$$[NN_{-1}N_{-2}\ldots N_{-n}] \tag{3.1}$$

where N is the product of all dimensions, N_{-1} the product of the $n-1$ dimensions excluding that of the highest-ranked dimension, and so on, down to and including $N_{-n} = 1$ (the last element, although trivial, is necessary to consistently describe arrays of any number of dimensions including zero).

The form of the *map* facilitates the mapping of elements of the next-lower array dimension, an operation that can be repeatedly applied down to

the zero-order dimension if a simple element of the n-dimensional array is to be mapped. Since the data storage and the map describing it are separate, a D array can be used with multiple maps to create a variety of multi-dimensional arrays. Storage efficiency extends to the maps themselves because submaps created in the operation of accessing array elements are children objects of the full map rather than created new.

(dimensions)	**map**	map
array map i	**ss**	subarray submap

map converts a list of dimensions into a list holding the corresponding map (this is done in place). The dimensions are given in descending order; the lowest-ranked dimension is 1.

The 'subscript' operator has been given a short name, *ss*. It returns the subarray of *array* holding the highest-order *i*-th element of the multi-dimensional organization described by *map*; it also returns the map of the subarray (which is the sublist of *map* comprising the second and following elements of *map*). Repeated use of *ss* yields descriptions of lower- and lower-ranked elements of the multi-dimensional array.

```
( 15 20 100 1 ) map dup /Amap name 0 get /d array /A name
A Amap 2 ss 9 ss 12 ss /amap name /a name
```

This example defines A as a three-dimensional array and associates it with the map *Amap*. Then performs three rounds of indexing to extract element $a_{2,9,12}$. This returns the map of a zero-dimensional array (as the D list [1]), and the zero-dimensional array containing the element (as a D array of length 1).

The 'blas' and 'lp' operators are executed using the cache facilities and processors in the host of the *dnode*. They do not themselves distribute work among different hosts in a cluster.

matmul_blas	$C \Leftarrow \alpha A^{t?} B^{t?} + \beta C$
matvecmul_blas	$y \Leftarrow \alpha A^{t?} x + \beta y$
solvetriang_blas	$x \Leftarrow A^{t?,-1} x$

givens_blas	compute Givens transformation
rotate_blas	rotate
norm2_blas	compute $\|x\|_2$
decompLU_blas	compute LU decomposition of A
backsubLU_blas	compute LU backsubstitution with RHS
invertLU_blas	compute inverse matrix following LU decomposition

C Cmap beta A Amap Atrans B Bmap Btrans alpha	**matmul_blas**	C Cmap
y beta A Amap Atrans x alpha	**matvecmul_blas**	y
x12	**givens_blas**	c s
c s x y	**rotate_blas**	x y
x	**norm2_blas**	num
x A Amap Atrans upper unit	**solvetriang_blas**	x
A Amap pivot	**decompLU_lp**	A Amap pivot true false
rhs A Amap pivot	**backsubLU_lp**	rhs
A Amap pivot	**invertLU_lp**	A Amap

The operands specified as capitalized names (e.g., A) are /D arrays holding matrix data, described by a map (*Amap*). A boolean (*Atrans*) when true designates use of the transposed matrix rather than the matrix itself. Names in greek letters stand for /D scalars, and lower-case names are vectors (/D arrays), a /L array (*pivot*), or /D scalars (c, s).

matmul_blas and **matvecmul_blas** perform generalized matrix-matrix or matrix-vector multiplications.

givens_blas computes the cosine (c) and sine (s) of a Givens transformation for a two-element vector h, overwriting the vector. The plane Givens rotation on a pair of vectors x and y is effected in place by **rotate_blas**.

solvetriang_blas solves the linear system with the triangular coefficient matrix A where the boolean *upper* indicates in which half of A the triangular coefficients are located; the boolean *unit* forces unit diagonal elements regardless of their values in A. The vector x contains the right hand side on

input and the solution on output.

decompLU_lp performs the LU-decomposition of matrix A in place and inserts pivoting information into the vector *pivot*. The decomposed matrix and pivoting are used by **backsubLU_lp** and **invertLU_lp** to generate a solution given for a right-hand-side vector (returning it in the vector), or to compute the inverse of the decomposed matrix in place.

Some of these operators return a boolean indicating success/failure of the attempted operation. Others (e.g., **solvetriang_blas**) can also fail, but, for reasons due to error-reporting quirks of the BLAS/LAPACK libraries, report an operator error via the **error** mechanism of the D *dnode*. An additional message detailing the error reported by the library is written to 'stderr' (which you may wish to redirect to a log file for having an ear at the horse's mouth).

3.3.4 Communicating with a cluster of *dpawns*

[n1 dict1 n2 dict2 ...]	**makerthreads**	socket
null	**makerthreads**	—
—	**rthreads**	n
—	**checkrthreads**	bool
idx active_obj / string	**rsend**	—
socket comp_obj	**send**	—
sig	**rsendsig**	—

A cluster of *dpawns* is created on the network by **makerthreads** (make remote threads). The list argument of the operator generally has several to many paired entries. The first item of a pair gives the number of *dpawns* to be brought up on the Linux box described by the second item, a dictionary. The dictionary is specific for the *mpi* environment and provides, under *mpi*-specific names a set of associated strings. **makethreads** starts a child process of the *dnode* called the 'rook'. The rook mediates communications between the *dnode* and *dpawns* via *mpi*. **makerthreads** returns a socket (i.e. socket-type null object) for communication with the rook via the *dnode* operator **send** (see below). Messages sent by *dpawns* to the

dnode will be addressed (via *mpi*) to the rook, who communicates them to the *dnode* through the socket established by **makethreads**. A second usage of **makerthreads**, taking a **null** argument, kills the rook and the current *dpawns* (it returns nothing). The rook is used to couple *dnode* and *dpawns* in this way in order to facilitate cleanup of failed *dpawn* activities without compromising the continuity of the *dnode* itself.

rthreads inquires the current number of *dpawns*, independent of their functional states. **checkrthreads** returns **true** if the rook and all current *dpawns* are responsive. If the rook is unresponsive, **checkrthreads** eliminates all remote threads from the *dnode* and returns **false**. If one or more *dpawns* are unresponsive, **checkrthreads** simply returns **false**.

You can send messages to the rook using **send** with the socket number obtained from **makerthreads**. The rook will broadcast such a message to all current *dpawns*. The standard method for transmitting messages to *dpawns*, however, is **rsend**. To identify a particular *dpawn* as target, specify its index *idx* (automatically assigned by **makerthreads** in the order of its argument list elements, starting from 0). To send to all current *dpawns*, specify *idx* as 'undefined' (*). The active object or string will be submitted to the target *dpawn(s)* for execution (see also **send**).

rsendsig sends a signal to all pawns, where the signal is an integer that is mapped in the source file `dm{signals.c` as `sigmap` to a signal number for the current platform. However, these mappings are identified by name in the dictionary ***SIGNALS***, which has such members as ***QUIT*** and ***KILL***, which are the proper enumeration to send a `SIGQUIT` or `SIGKILL` to all pawns. See the **sendsig** operator which functions similarly for *dvt* and *dnode* machines.

3.3.5 The PETSc procedure library of the *dnode*

/A ... /type rows columns	***mat_create***	A
/X length	***vec_create***	X
/Y X	***vec_dup***	Y
X data	***vec_fill***	—
A ∼row–maker	***mat_fill***	—
X data	***get_vector***	data
A data	***get_matrix***	data
/B A	***mat_dup***	B
... A	***mat_transpose***	—
Y beta A Atrans X alpha	***pmatvecmul***	—
data Y beta A Atrans X alpha	***get_matvecmul***	data
/name	***ksp_create***	K
X	***vec_destroy***	—
A	***mat_destroy***	—
K	***ksp_destroy***	—
K A X Y	***ksp_solve***	—
K X Y	***ksp_resolve***	—
K A X Y data	***get_ksp_solve***	data
K X Y data	***get_ksp_resolve***	data
bool	***report***	—
length ∼active	***execrange***	—

These procedures internally communicate with a cluster of *dpawns* where they initiate operations involving PETSc operators (see section 3.4.3). The *dnode* keeps records of the PETSc objects that are created in the form of dictionaries (referenced as A or B for matrices in the table, X or Y for vectors, and K for solvers). Currently three varieties of PETSc object are represented this way:

Table 3.2: Vector dictionary

id	name
N	length

The ***id*** is the name used when creating the PETSc object. ***N*** is the length of the vector.

Table 3.3: Matrix dictionary

id	name
m	global rows
n	global columns
mtype	matrix type
params	dictionary of type–specific parameters
mmax	the maximum number of local rows on any matrix

mtype is a name defining the format of the matrix: ***/sparse***, ***/dense*** or ***blockdense***. ***mmax*** is the number of rows on the last pawn: if there are 9 rows and 4 pawns, ***mmax*** would be 3. ***params*** is a dictionary whose format is associate with ***mtype***.

Table 3.4: Sparse Matrix Parameter dictionary

irows	row offsets
icols	column associated with each non–zero datum
icols_off	list of global row offsets

irows and ***icols*** are active names defining the CSR format for the matrix. On each pawn, ***irows*** has a length equal to the number of rows on that pawn plus one; each element defines the offset into the data (and ***icols***) of the start of a row, in order, and the last element gives the offset of the the last element of the data plus one. On each pawn, ***icols*** is the same size as the number of non-zero elements of the data, and holds the column number for the associated datum. ***icols_off*** is a list of the offsets of the first row on each pawn into the global matrix. The names pointed to by ***irows*** and ***icols*** are defined by the user on each pawn, but ***icols_off*** is transparently created by the *petsc D* procedures.

For ***/dense*** and ***/blockdense*** types, the param dict is empty.

Table 3.5: Krylov Space Solver dictionary

id	name
rtol	relative tolerance (1e–12)
atol	absolute tolerance (*)
dtol	divergence tolerance (1/rtol)
maxits	maximal number of iterations
pctype	type of preconditioner (*, i.e. JACOBI)
ksptype	type of ksp solver (*, i.e. GMRES)
kspparam	parameters for the ksp type (null)
pcparam	parameters for preconditioner type (null)
monitortype	type of the monitor for ksp solves (*, i.e. petsc default)

rtol and ***atol*** are the tolerances for convergence tests. Convergence is detected if $||r_k||_2 < max(rtol * ||b||_2,\ atol)$, where $r_k = b - Ax_k$. On the other hand, ***dtol*** is the tolerance for detecting divergence: $||r_k||_2 > dtol * ||b||_2$. ***maxits*** is the number of iterations before divergence is assumed. If any of these are set to $*$, the *petsc* defaults are used. See Balay et al. [2004, Section 4.3.2] for a discussion of these defaults.

ksptype, ***pctype***, and ***monitortype*** are integers that map to a solver, a preconditioner, and a monitoring output for the latter. The integers are defined in `dmpetsc.c` in the structures `ksptypes`, `pctypes`, and `monitors`. The names for use from *D* are defined in the dictionaries ***ksptypes***, ***pctypes***, ***monitorytypes*** defined by `petsc.d` in the ***PETSC*** module.

Associated with ***ksptype*** is ***kspparam***, and with ***pctype*** is ***pcparam***. The '*param' names map to *D* objects that are necessary for the solver (and its preconditioner). Currently, none of the defined types use them - both should be **null**.

mat_create creates a matrix distributed over the *dpawns*. The ***rows*** and ***columns*** operands are the global size of the matrix. The ***/A*** operand is the name of the matrix, from the point of view of the pawns; usually, this will also be what you call the output reference dictionary. The ***/type*** operand is a name: ***/sparse***, ***/dense*** or ***/blockdense***. That type determines the parameters representde by '...'. This procedure returns a dictionary as in table 3.3.

For sparse, the parameters are: ***~irows*** ***~icols***. ***~irows*** is a name that has

been defined, on the pawns, to return the offsets into the local data for the beginning of each row, while ***~icols*** returns on the pawns the column number for location in the matrix. This a local CSR format. For dense and blockdense, there are no extra operands.

vec_create creates a vector distributed over the pawns. The ***/X*** parameter is the name of the vector on the pawns, which should usually be the same as the name of the dictionary referencing the vecotor, and ***length*** is the global length of the vector. This procedure return a dictionary referencing the vector, as defined in table 3.2. ***vec_dup*** replicates the vector ***X*** as ***/Y*** on the pawns, including data. It returns a new vector dictionary for the *dnode*.

vec_fill copies the d-array data, referenced by the name ***~data*** on each pawn for the local portion, into the vector replacing current values. ***mat_fill*** does the same for matrices, but instead of referencing a d-array, the last parameter references a procedure, the ***row_maker***:

A row ***row_maker*** data icols

where ***A*** is the matrix (see the pawn petsc operators, section 3.4.3), ***row*** is the local row number to be filled, ***data*** is the d-array to copy into that row, and ***icols*** are the column numbers of each of the elements of that row.

get_vector collects all the data for vector ***X*** into ***data***, which is just a D array of doubles. The call returns the sub-array into which ***X*** fits. ***get_matrix*** does the same for a matrix: the elements (non-zero for a sparse matrix) of the matrix are inserted into a D array of doubles.

pmatvecmul	$y \Leftarrow \beta y + \alpha A^{t?} x$
get_matvecmul	$y \Leftarrow \beta y + \alpha A^{t?} x$
ksp_solve	$x \Leftarrow A^{-1} y$
ksp_resolve	$x \Leftarrow A^{-1} y$
get_ksp_solve	$x \Leftarrow A^{-1} y$
get_ksp_resolve	$x \Leftarrow A^{-1} y$

pmatvecmul and ***get_matvecmul*** multiply the (possibly transposed) matrix by a vector, and add it to a scale vector, returning the value in that latter vector. The 'get' version returns the resultantant vector to the dnode.

mat_dup creates a new matrix B from a matrix A, and then copies the non-zero elements from A to B. ***mat_transpose*** transposes a matrix A, keeping the same identifier. The latter requires varying parameters according to type: For ***/sparse***, the parameters are ***~irows ~icols***, where ***~irows*** is the name to use on the *dpawn* for the row data, and likewise for the ***~icols***. See ***mat_create*** for further explanation: the only difference here is that they aren't predefined on the *dpawn*, but are created by this procedure. For ***/dense*** type matrices, the optional parameters are empty.

ksp_solve, ***ksp_resolve***, ***get_ksp_solve*** and ***get_ksp_resolve*** all use an iterative solver to calculate $A^{-1}y$. The 're' versions solve for a new x from y, given that the matrix A is unchanged for the solver. The 'get' versions return the resultant x to the dnode. So, ***ksp_solve*** must be called once with a matrix A and solver K, and then K may be used with ***ksp_resolve*** to avoid pre-conditioning A again.

An important element of ksp reuse is that any new matrix A_n must have an identical structure to the previous matrix A_o. This is guaranteed by testing that the A_n was created by some sequence of calls to ***mat_dup*** on A_o.

The solver is created by calling ***ksp_create***, with a *dnode* name. The currently defined ***kspsettings*** dictionary is used for the solver parameters, as defined in table 3.5 (except for ***id***, which is the parameter passed on the stack). It returns that a copy of that dictionary, with the ***id*** filled in. In ***PETSC***, dictionaries have predefined names for the numeric parameters: ***ksptypes*** for ***ksptype***, ***pctypes*** for ***pctype*** and ***monitortypes*** for ***monitortype***. As of this writing, none of the defined ***ksptypes*** or ***pctypes*** use ***kspparam*** or ***pcparam***, respectively.

The procedures ***ksp_destroy***, ***mat_destroy***, ***vec_destroy*** all eliminate the parameter (of the matching type). They **restore** the box associated with the *dnode* and *dpawn* dictionaries, and un-allocate any external memory used by the PETSc libraries (and are therefore preferred to simply **restore**'ing a box that contains the dictionaries). PETSc external memory allocation only occurs on the *dpawn*, and not on the *dnode*.

report simply turns on or off verbose reporting (**true** to turn verbose reporting on).

execrange runs an active object on all *dpawns*, passing in the range for that pawn in particular. It should be used for initializing per-pawn data and parameters for vectors and matrices. The parameter ***length*** is chopped up into (offset, length) pairs for each node, which is passed to ***~active*** on the pawn. So, with 5 pawns and a length of 11, the first four pawns run ***~active*** with 0 2, 2 2, 4 2, and 6 2 on the stack, while the last pawn runs ***~active*** with 8 3 on the stack.

3.4 The D pawn (*dpawn*)

The *pawn* variety of D machine is created and killed by operators of a supervising *dnode*. The *dpawn* includes the general-purpose operators also available to a *dnode*, uses a different set of communication operators based on the *mpi* protocol, and taps the computational facilities of the PETSc library through D operators that map to PETSc library functions.

The operator set of a *dpawn* differs from that of a *dnode* in the following ways:

- The operators
  console
  setconsole
  getmyport
  Xconnect
  Xdisconnect

3.4.1 Operators for administrating a *dpawn*

Most of these operators are synomymous with *dnode* operators but have slightly different behaviors.

long_array/null	**vmresize**	bool
—	**halt**	—
—	**continue**	—

—	**stop**	—
—	**abort**	—
active_obj	**lock**	—
—	**console**	consolesocket/null
consolesocket/null	**setconsole**	—
string	**toconsole**	—
string	**tostderr**	—
hostname port# sourcestr numerr	**error**	—
hostname port# sourcestr numerr strbuf	**errormessage**	messstr
—	**getmyport**	port#
—	**Xwindows_**	bool
(hostname:screen#)	**Xconnect**	—
—	**Xdisconnect**	—
num	**makethreads**	—
—	**threads**	num
active_obj	**serialize**	—
—	**getlibdir**	string
(path) (filename)	**loadlib**	lib_dict
—	**hi**	(library description)
—	**libnum**	num
null/lib_dict	**nextlib**	dict true false

3.4.2 Operators for communicating via *mpi*

NB: Not tested yet. These operators are for inter-pawn communications. They can be individual, collective or source-destination.

Individual

Individual operators are run on one pawn to produce global or local data about the network of mpi processes.

— **mpirank** rank
— **mpisize** size

These two operators define the relationship between the current process and all other *dpawns*. **mpirank** returns the rank id of the current process. This is an integer from the range $[0 \ldots n)$, where n is the **mpisize** of the system. It is an arbitrary id number, but by convention in the mpi world rank 0 does any sequential or global activity.

mpisize returns the total number of mpi processes.

Collective

Collective operators must be run on *all* mpi processes. Processes will block until all other processes have at least entered the operator.

—	**mpibarrier**	—
root object	**mpibroadcast**	object

mpibarrier is a synchronization operator. All processes block until all other processes have entered the barrier. In other words, no operations after **mpibarrier** on any process begin until all operations before **mpibarrier** on all processes have ended.

mpibroadcast sends an object to all processes. The "root" process is the sender of the object; all other processes are the receivers. For the receivers, the input object may be null (it is discarded). For the sender, the output object is the same as the original. The root integer is the **mpirank** of the sender.

Source-destination

Source-destination operators work as pairs, where one process initiates the operation with an operator, and another process completes the operation with a different operator.

dest object	**mpisend**	—
src	**mpirecv**	object
src	**mpiprobe**	src
*		src
src	**mpiiprobe**	src true
src		false
*		src true
*		false

mpisend sends the object to the destination, as identified by its **mpirank**. It may or may not block while sending, so you must guarantee that the receiver will eventually call **mpirecv**, and that no deadlocks will occur.

mpirecv is the matching operator to **mpisend**, returning the object sent from the source. It will block until an object is sent by the source process (as identified by it's **mpirank**).

mpiprobe and **mpiiprobe** check for objects being sent by mpi processes to the current process. They can either check for a message from a specific process when called with a process's **mpirank**, or if they are called with '*' (undefined) they will probe for messages from any process, returning the rank of the sender to the caller. The difference between the two is in blocking behavior: **mpiprobe** blocks until at least one message is queued to be received, while **mpiiprobe** returns false if no message is pending and true if a message is pending (in addition to the source id). Broadcast messages from **mpibroadcast** can not be probed for.

3.4.3 The PETSc operator library of the *dpawn*

size	**petsc_vec_create**	X
X	**petsc_vec_dup**	Y
X Y	**petsc_vec_copy**	Y
X	**petsc_vec_syncto**	X
data offset X	**petsc_vec_copyto**	X
X	**petsc_vec_syncfrom**	X

X offset data	**petsc_vec_copyfrom**	data
X	**petsc_vec_max**	max
X	**petsc_vec_min**	min
X	**petsc_vec_destroy**	—
X	**petsc_vec_norm**	norm
rows columns	**petsc_mat_dense_create**	A
<l irows> <l icols> columns	**petsc_mat_sparse_create**	A
rows–per–process columns–per–process rows columns	**petsc_mat_blockdense_create**	A
A	**petsc_mat_destroy**	—
A B	**pesc_mat_copy**	B
A	**petsc_mat_dup**	B
A	**petsc_mat_syncto**	A
A	**petsc_mat_syncfill**	A
A	**petsc_mat_endfill**	A
<d data> <l cols> row A	**petsc_mat_fill**	A
<d data> row start–column A	**petsc_mat_copyto**	A
A	**petsc_mat_syncfrom**	A
A row start–column <d data>	**petsc_mat_copyfrom**	<d data>
A	**petsc_mat_transpose**	A^t
A	**petsc_mat_getnz**	num–nonzeros
<l rows> <l cols> A	**petsc_mat_getcsr**	<l rows> <l cols>
Y beta A transA X alpha	**petsc_mat_vecmul**	Y
ksptype kspparam pctype pcparam monitortype	**petsc_ksp_create**	K
K	**petsc_ksp_destroy**	—
K rtol atol dtol maxits	**petsc_ksp_tol**	—
K A/null X B	**petsc_ksp_solve**	X
—	**petsc_log_begin**	—
(dir) (filename)	**petsc_log_summary**	—

These are the primitive operators called on a *dpawn* to create and manipulate petsc objects. They are primarily collective in nature, in the *mpi* sense: they must be called on each pawn independently but synchronistically. Therefore, these are primarily intended to be internal operators, called indirectly via procedures on the *dnode* (see section 3.3.5).

They work on three different objects produced by the `dmpetsc.c` plugin. Each is a D dictionary that encapsulates petsc data structures and memory allocations:

Table 3.6: dm_petsc_vector

VECTOR_VECTOR	integerized pointer locating the petsc vector
VECTOR_N	the local number of elements
VECTOR_GN	the global number of elements
VECTOR_ASS	boolean that signals whether the vector has been assembled by the petsc libraries

Table 3.7: dm_petsc_matrix

MATRIX_MATRIX	integerized pointer locating the petsc matrix
MATRIX_M	local number of rows
MATRIX_N	local number of columns
MATRIX_GM	global number of rows
MATRIX_ASS	boolean that signals whether the matrix has been assembled by the petsc libraries
MATRIX_DUPID	integer identifying the structure a duplicate has the same dupid, and can be used with the same ksp
MATRIX_MTYPE	an integer identifying the type of the matrix: sparse, or dense

Table 3.8: dm_petsc_ksp

KSP_KSP	integerized pointer locating the petsc ksp
KSP_N	The number of columns in the matrix and vectors associated with this ksp
KSP_KSPTYPE	An integer identifying the solver type, as mapped to from ***ksptypes*** in ***PETSC***
KSP_PCTYPE	An integer identifyint the preconditioner type, as mapped to from ***pctypes*** in ***PETSC***
KSP_DUPID	An integer identifying the matrix 'structure': once initialized, it will be identical to the ***MATRIX_DUPID*** of matrices that may be used with this ksp
KSP_PCSETUPFD	an integerized pointer to internal data for initializing the preconditions
KSP_KSPSETUPFD	an integerized pointer to internal data for initializing the solver

petsc_vec_create creates a vector of a given *local length*. **petsc_vec_dup** duplicates a vector into a new vector of the same length, but without the elements being initialized. **petsc_vec_copy** copies the data from a vector X into a vector Y of the same length.

petsc_vec_copyto copies data elements from a D double array into a vector, starting at element n. Of course, the length of the array + n must be less than or equal to the length of the vector. **petsc_vec_syncto** must be called on idling pawns when **petsc_vec_copyto** is called: in otherwords, the combination of the two operators is collective.

petsc_vec_copyfrom is the inverse operation: it copies the elements of a vector, starting at n, into an a D array. The number of elements copied is the length of the array or the length of the vector - n (which ever is smaller), and the sub-array of that length is returned. **petsc_vec_syncfrom** is used for idling pawns during this operation: the union of both operators is collective.

petsc_vec_max and **petsc_vec_min** are collective operator that return on every pawn the *global* max and min, respectively, of a vector. **petsc_vec_norm** returns on every pawn the *global* length of the vector ($||X||=\sqrt{\sum_i X_i^2}$).

Finally, **petsc_vec_destroy** should be called to deallocate a vector: it destroys the dictionary and any internal petsc data structures associated with the vector. Use solely of **save-restore** pairs will leak through the PETSc libraries.

petsc_mat_dense_create creates an empty matrix of a given numer of rows and columns. **petsc_mat_sparse_create** creates a sparse matrix whose structure is defined by a maximum number of columns and CSR pair of long D arrays: the offset into the column array for the beginning of each row, and the filled columns; of course, the last element of the row array is one past the last column. **petsc_mat_blockdense_create** creates a block-dense matrix, which has blocks defined by the number of columns per pawn, and rows per pawn.

All these matrix types are cleaned-up by the single **petsc_mat_destroy** operator, which both eliminates the dictionary and cleans up any PETSc data structures associated with it.

petsc_mat_dup duplicates a matrix, creating a new matrix with the same structure as the original, both type and layout. **petsc_mat_copy** copies the data from a matrix to a matrix with the same non-zero pattern (but it needs not be a duplicate).

petsc_mat_fill copies data from a D double array into elements of a row of a matrix. The mapping is defined by a long array of the same length of the data array which identifies the column number for each data element. The row number *m* is the *local* row number. On idling pawns, **petsc_mat_syncfill** must be called: the combination of these two calls are collective. When done filling the rows of a matrix, **petsc_mat_endfill** must be called to assemble the matrix: this actually handles beginning the communications between different pawns about their updated elements. No other PETSc calls can be made on the matrix between the initial **petsc_mat_fill**/**petsc_mat_syncfill** and the final **petsc_mat_endfill** on all pawns.

Similarly, **petsc_mat_copyto** copies the elements of a D array into a *local* row of the matrix, starting at a given column number. It fills all the 'non-zero' elements of the rows until the input array is exhausted. All idling pawns must call **petsc_syncto**. On the other hand, **petsc_mat_copyfrom** copies from a *local* row of the matrix starting at a given column number into a D array, until it runs out of columns or exhausts the D array.

petsc_mat_syncfrom must be called on idling pawns.

petsc_mat_transpose transpose the data of a matrix. After this operation, the matrix A will no longer be a duplicate of it's earlier state (i.e., can not be used with an ksp).

petsc_mat_getnz returns the total number *local* non-zero elements. This can be used to construct the arrays for **petsc_mat_getcsr**, which fills an array with the CSR structure of the matrix: an array of offsets for the beginning of each row, into an array of column numbers.

petsc_vec_matmul evaluates $y \leftarrow \beta y + \alpha Ax$, where y and x are vectors and A is a matrix. If β is 0 (the integer), the shortcut $y \leftarrow \alpha Ax$ is applied, and if β is 1 (the integer), the shortcut $y \leftarrow y + \alpha Ax$ is applied.

petsc_ksp_create creates a solver object ('ksp'). The parameter 'ksptype' is an integer, as enumerated in ***ksptypes*** of ***PETSC***, defining the solver type (GMRES, ...); if **null**, then the PETSc default is used. The 'kspparam' is a solver type specific parameter, currently **null** for all. Likewise, 'pctype' defines the precondition; the possible values are defined in ***pctypes*** of ***PETSC***. As well, 'pcparam' is a preconditioner specific parameter (currently **null** for all). This operator is matched by **petsc_ksp_destroy**, which discards the 'ksp' dictionary and any associated PETSc memory.

petsc_ksp_tol sets the divergence/convergence tolerances for the solver. The parameters are:

K solver
rtol relative tolerance for convergence.
atol absolute tolerance for convergence.
dtol relative tolerance for divergence.
maxits maximum iterations before divergence is assumed.

Convergence is detected if $||r_k||_2 < max(rtol * ||b||_2,\ atol)$, where $r_k = b - Ax_k$. Divergence is detected if $||r_k||_2 > dtol * ||b||_2$, or if $maxits$ is reached. An error is signaled if divergence is detected (see `dmpetsc.c` for possible divergence errors, and their mappings from PETSc error codes).

petsc_ksp_solve actually solves for x in $Ax = b$. On the first use, A must be given; for later solutions with the same left-hand side matrix, **null** can be used instead (which will optimize the solver). If a new A is used, it *must* be a 'duplicate' of the original matrix, as created by **petsc_mat_dup**. This

guarantees that the 'zero' elements in the same column/rows; the values, of course, of the non-zero elements need not be the same.

petsc_log_begin activates PETSc internal logging (see 'PetscLogBegin' in Balay et al. [2004]). To save the current log, use **petsc_log_summary** which takes a directory and a filename to save the entire log to.

References

Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

Index

Procedures