Chapter 3

The Definitive Assembly Maintainer (DAM) machine

Richard Cartwright's Definitive Assembly Maintainer Machine (DAM machine) is "a dependency maintenance tool written in assembly code that maintains indivisible relationships between words in an area of computer RAM store" [Car99, §5]. It is an example of a definitive system implementation for 1-agent modelling that uses evaluation/storage strategy 2, evaluate-at-redefinition (see §2.2.1). Partly as a result of this basis, its design gives much prominence to dependency and little to agency. It is based on Cartwright's "Dependency Maintainer Model" (DMM), [Car99, §4], which is the most explicitly detailed specification of an abstract definitive machine currently in existence. This chapter first briefly examines the DMM in order to get a top-down perspective of the DAM machine, similar to the approach taken by Cartwright, but placed in the context of this thesis with other definitive systems.

Taking the complementary bottom-up approach, the machine itself and a surrounding interactive application environment named !Donald are then studied. The environment is extended to allow experimentation with a bottom-up style of use, involving novel interactions with a form of "definitive assembler".

The bottom-up approach reveals two problems that can be considered at the abstract level. Firstly, a problem involving references encountered when attempting to use dependency in a symbol table gives a good example of higher-order dependency and the associated difficulties this poses for implementation. Secondly, Cartwright's

DMM and DAM machine design abstract away the *location* of maintained machine words. This the standard convention in high-level languages. It is also a convention adopted in all existing work on definitive notations and systems, but it is shown that this abstraction causes problems when using dependency on data items whose representation is larger than one word or of variable size.

Finally a case study (constructed bottom-up) demonstrates proof of concept for configuring the DAM machine as a definitive text editor, where the question "why is that pixel white?" can be answered with reference to indivisible relationships within the store, rather than by reference to a previous write action of some unconstrained agent.

The primary source on the DAM machine and the DMM is Cartwright's PhD thesis [Car99], which describes the application of the DAM machine to two areas. Firstly, [Car99, §5.5] describes application of the DAM machine to visualising of a geometrical shape from its function representation [PASS95], the state of each pixel being (indirectly) determined by a definition. Secondly, [Car99, §5.4] cites Allderidge [All97], an undergraduate final year project report that describes !Donald, an implementation of the DoNaLD notation that uses the DAM machine to perform definition maintenance. Cartwright [Car99, §6] goes on to describe a Java implementation of the ideas described here, called the Java Maintainer Machine Application Programming Interface (JaM Machine API). JaM has recently been developed into its second version as part of Cartwright's continuing work in the BBC's Research and Development department. JaM is not discussed in this thesis as the underlying operational model is very similar to the DAM machine.

The DAM machine has fewer associated secondary source publications than the ADM, being cited in just two papers. The 1996 paper by Gehring *et al* [GYC+96] was written as implementation was progressing, and contains some preliminary statements about the envisaged machine. A second, 1998, paper by Allderidge *et al* [ABCY98] contains two sections about the DAM machine that are derived from Cartwright, along with some discussion of strategies for translation of definitive scripts into procedural programs.

In the first section below, the mathematical model presented in [Car99, §4] is reviewed, then the 'BRA' algorithm that operates in the domain specified by the

mathematical model. Subsequent sections move on to the implementation of the DAM machine, the !Donald application and the extensions and insights described above. Sections 3.1, 3.2, 3.3 take the form of a review of the DAM machine, incorporating a critical evaluation of Cartwright's account [Car99] and a clarification and exposition of details omitted from [Car99] that are essential for understanding my technical extensions and experimental work, as discussed in sections 3.4 and 3.5. The script digraph concept used in this chapter and throughout the thesis is described in an appendix to this chapter, §3.A (p.178).

3.1 The DMM and the BRA

3.1.1 State and the DMM

In [Car99, §4], Cartwright refers to the mathematical model he presents as "The Dependency Maintainer Model". It is not the only possible model of dependency maintenance — this thesis presents several others, so in this thesis, we shall use the acronym DMM to denote the particular model of dependency maintenance described in [Car99].

The DMM is an attempt to formalise various properties of scripts written in a simple "low-level" definitive notation (denoted here by the acronym LLDN[1]). LLDN was constructed by "examining scripts of dependencies and extracting from these common abstractions that can be used to reason about them" [Car99, p.105]. LLDN is not explicitly defined by Cartwright. In my analysis, LLDN has the following characteristics:

- a script is a finite set of definitions;
- a definition is an association of a value with an identity[2];

[1] Neither DMM or LLDN are abbreviations used in [Car99]: they are introduced here in order to distinguish these particular concepts from models of dependency maintenance in general and definitive notations in general respectively.

[2] Here I use identity rather than identifier as this identity is a numeric address.

Identity	Function	Arguments	Value
α	*times*	$(\alpha + 5, \alpha + 3)$	48
$\alpha + 1$	*power*	$(\alpha + 4, \alpha + 6)$	8
$\alpha + 2$	$value_4$	()	4
$\alpha + 3$	*double*	$(\alpha + 4)$	4
$\alpha + 4$	$value_2$	()	2
$\alpha + 5$	*add*	$(\alpha + 1, \alpha + 2)$	12
$\alpha + 6$	$value_3$	()	3

Table 3.1: An example script in LLDN

- values are taken from the infinite set of integers, $\mathbb{Z}$, which forms the only data type;
- identities are integer values in the contiguous range $[\alpha, \beta - 1]$;
- the current value associated with a particular identity is defined by the application of a single function to a (potentially unbounded) sequence of arguments;
- function arguments are values associated with identities;
- many functions are potentially available from the set of functions $\mathcal{F}$;
- each function $f \in \mathcal{F}$ maps a sequence of integer values (the arguments) to a single integer value (the result).

An example of an LLDN "script" is given in Table 3.1 (which is derived from [Car99, Table 4.3]). The word script is often used to mean a *sequence* of lines in a text file: as in, for example, a shell script. The term "definitive script" in current usage usually implies the sequential meaning. It is sometimes drawn abstractly as a rectangle containing horizontal lines, implying a sequence of definitions. However, what I mean by an LLDN "script" is formed from a *set* of definitions. Hence, Table 3.1 is to be interpreted as a relation, like a database table, where each Identity (the primary key) is associated with a Function, Arguments and a Value.

Cartwright [Car99, §4.2.1] describes a four stage process which can transform "any script"[3] [Car99, p.105] into a form representable in LLDN:

1. Every function in the script is replaced by a function in $\mathcal{F}$. This implies mapping all types available in the definitive notation to $\mathbb{Z}$. For example, the boolean definitions:

   ```
   a is not(b); b is true;
   ```

   might be replaced by:

   ```
   a is boolnot(b); b is 1;
   ```

   where the function `boolnot` : $\mathbb{Z} \to \mathbb{Z}$ maps i to $1 - i$. The DMM assumes that the data type used to represent $\mathbb{Z}$ is of unbounded size, or is at least large enough to be able to represent the values required.

2. Function arguments (which may be sub expressions or literal values) are replaced by references to new definitions. For example,

   ```
   a is boolnot(booland(b,1));
   ```

   might be replaced by:

   ```
   a is boolnot(x); x is booland(b,y); y is 1;
   ```

3. Now the total number of definitions is as it will be in LLDN form, so in stage three, variable names are replaced by integer identities. In the previous example, `a`, `x`, `y` and `b` might be replaced by the identities α through $\alpha + 3$ respectively. Note that the integer mapping is arbitrary: the script is still a set of definitions and adjacency of identities has no meaning.

4. Literal values are replaced by special "value functions" that take no arguments and return the value required. For example,

   ```
   y is 23;
   ```

   might be replaced by:

   ```
   y is value23();
   ```

[3] But we shall see in §3.4.2, §3.4.3 and §3.5 that scripts using HOD or lists cannot be transformed into LLDN.

Cartwright [Car99, §4.2.2] explains that the state of an LLDN script is then characterised by a mapping of each of the identities in the current set to:

- its associated value in $\mathbb{Z}$;
- its defining function in $\mathcal{F}$;
- a sequence of arguments to that function in A^*.

Formally, a transformed script $\mathcal{S}$ can be represented by a DMM $\mathcal{M}_S$, given by a 4-tuple:

$$\mathcal{M}_S = (A, F, D, V)$$

where:

- A is the range $[\alpha, \beta - 1]$ of reference numbers for the script $\mathcal{S}$ (the total number of definitions being equal to $\beta - \alpha$);
- F is a mapping $F : A \rightarrow \mathcal{F}$, associating a function from $\mathcal{F}$ with every reference number in A;
- D is a mapping $D : A \rightarrow A^*$, associating a sequence of arguments A^* (each argument itself being a reference to an identity) with every reference number in A;
- V is a mapping $V : A \rightarrow \mathbb{Z}$, associating an integer value from $\mathbb{Z}$ with every reference number A.

Having established these conventions, Cartwright goes on to formally define the function *lookup* (i.e. the association between reference number and value), the concept of *up_to_date* (i.e. that values are consistent with their dependencies), the presence of *cyclic dependency* (i.e. when a reference in the model directly or indirectly depends on itself), *dependencies* and *dependents* (in this thesis named sources and targets respectively, following [Yun90]), and the *stable state* (i.e. the DMM is up_to_date and no cyclic dependency is present).

3.1.2 Transitions and the block redefinition algorithm (BRA)

Thus far we have described the aspects of state in the DMM. Cartwright [Car99, §4.2.4] explains that a *transition* from a stable state:

$$\mathcal{M}_S = (A, F, D, V)$$

to another stable state:

$$\mathcal{M}_{S'} = (A', F', D', V')$$

can be described by a mapping K, determined by a subset of the set of all possible redefinitions, each of which (re)associates a function and a sequence of arguments with a reference number:

$$K \subset \{f \mid f : A' \rightarrow \mathcal{F} \times A'^*\}$$

The new range of reference numbers $A' = [\alpha', \beta' - 1]$ may be extended from the original $A = [\alpha, \beta - 1]$ as new references introduced in K (on the left and/or right hand sides) may extend the range. It is not possible to remove definitions and hence reduce the range through a redefinition in K [Car99, p.115]: Cartwright does not consider undefined values or removal of definitions.

For a given block of redefinitions K, the set of references whose value is to be updated is $U(K)$ [Car99, p.118] — in the vocabulary of this thesis, the set of all recursive targets of K (see Appendix §3.A, p.178). Once K has been shown to meet two criteria described below, the values of all references in U should be updated in order to perform the state transition from $\mathcal{M}_S$ to $\mathcal{M}_{S'}$. This kind of transition, which is only allowable if the two criteria are first met, is termed a *protected_update* [Car99, p.118]:

$$protected\_update(\mathcal{M}_S, K) = \begin{cases} \mathcal{M}_S & \text{if } cyclic((A', F', D', V)) \\ & \text{or } \neg suitable(K) \\ (A', F', D', V') & \text{otherwise} \end{cases}$$

The two criteria are that the new state must not contain cyclic dependency and that K must be *suitable* (Cartwright's term), that is to say, that for every new

reference in K, there is a new definition, as undefined values are not allowable in the DMM. The *protected_update* operation:

$$\mathcal{M}_{S'} = protected\_update(\mathcal{M}_S, K)$$

therefore encompasses some, but not all features of Slade's characterisation of a valid transition, described in the earlier §2.3.2. In Cartwright's terms, a valid transition is one where given $\mathcal{M}_S$ and K, there is only one possible resulting $\mathcal{M}_{S'}$.

The amount of computation required to make the transition from $\mathcal{M}_S$ to $\mathcal{M}_{S'}$ is the sum of the computation required to update each element in $U(K)$. Predicting the total time taken to make the transition is possible, in principle. The major transition $\mathcal{M}_S$ to $\mathcal{M}_{S'}$ is formed from minor transitions. Each minor transition is formed by the update of the value of one definition, involving reading the arguments from store, executing the appropriate operator (a function in $\mathcal{F}$) and writing the result back into store. It should be possible to estimate the time required for each operator — this could be done by close examination of the machine code, summing the time required for each instruction. This measurement is complicated by the presence of caches on modern systems. The timing of some operators may also be dependent upon the data input. However, the effort of estimation would be repaid if the results were reused across many scripts. This is possible as it is the time required to evaluate each available operator that is being estimated, rather than the time required to evaluate each individual script[4]. Given the timings required for each minor transition, on a sequentially executing machine, the time for the major transition is then the sum of the individual minor transition times. On a parallel machine, some knowledge of the hardware and mapping of minor transitions to processors would be required in order to characterise the major transition. This problem is similar to that solved by the CHIP^{3}S toolset developed at Warwick [PKNA95], which is now named PACE and is being applied to Grid computing environments [CJS+02].

A long-term goal for research in this area would be to derive a real-time characterisation of the performance of a dependency maintainer so comprehensive that it could be used to calculate the time for a major transition[5] and compare this with

[4]This may be contrasted with the estimation of the performance of a conventional program (benchmarking), where e.g. the number of iterations of a loop may be data-dependent.

[5]Any transition or a particular restricted set of specific transitions.

real-time deadlines automatically. The technical issues to be addressed in such characterisation include estimation of evaluation times for the whole variety of operators on a specific platform. This is a major engineering task beyond the scope of this thesis. A simpler characterisation of performance, based on estimating the number of element updates, captures much that is useful.

Cartwright [Car99, p.125] showed that, if updates are appropriately scheduled, the number of updates associated with a single redefinition need not exceed the total number of definitions in the script.

In the case where the mapping K is determined by a set that contains more than one redefinition, there are in general elements that are direct or recursive targets of two or more elements that are updated by definitions in K. Some implementations might redundantly evaluate such elements twice. In theory, this should not occur since $U(K)$ is a set. However, if $U(K)$ is in practice implemented as a sequence, without collapsing of multiple redundant entries, the implementation could perform more updates than are theoretically necessary. Cartwright [Car99, p.124] states that EDEN is such an implementation:

> In the use of the existing `tkeden` interpreter, the size of the sets of redefinition is typically of the order of one or two. The interpreter itself can only handle one redefinition at a time ($\|K\| = 1$) and is optimised for this. It often performs a large number of unnecessary calculations. Each `tkeden` single redefinition initiates a state transition, even though several redefinitions (a block) presented to the interpreter at the same time may conceivably pertain to the same change of state in the observed model.

We shall see in §4.3 that this assertion is not correct, for two reasons. Firstly, EDEN has a complicated evaluation scheduler which makes use of a mix of evaluation/storage strategy 2 (evaluate-on-redefinition) and strategy 3 (evaluate-at-use-when-necessary), the strategy in use being dependent upon current machine operating context. This is described in Chapter 4. This reduces the number of unnecessary evaluations — although this is difficult to observe through black-box testing. Secondly, Eden (the language) contains an `autocalc` facility which can be used to demarcate blocks of redefinitions. When this is used, EDEN (the implementation) effectively avoids redundant evaluation.

As a preliminary to the description of the DAM machine, Cartwright [Car99, §4.3] presents an algorithm "for efficient update of the DMM", named "the block

redefinition algorithm":

> ... where the word *block* signifies that the algorithm considers several redefinitions simultaneously. This algorithm finds a sensible ordering of work to be done, in the context of both the current dependency structure for the script represented and all the redefinitions in the block of redefinitions.

The block redefinition algorithm (in this thesis abbreviated to BRA) incorporates Knuth's algorithm for topological sorting of the elements of a partially ordered set [Knu73, p.262]. Cartwright's BRA has three phases:

1. Changing D to D' and F to F'. The move from D to D' corresponds to making any necessary changes to the arguments associated with each location. Moving from F to F' corresponds to making any necessary changes to the function associated with each location. These two changes thus correspond to modification of dependencies in the script. As literal values are translated to value functions in LLDN, moving from F to F' corresponds to modification of literal values in the script.

2. Knuth's topological sort, which achieves two results:

   (a) Calculation of an optimal evaluation ordering in order to update V to V'. This calculation finds all targets of K in D' as a sequence sorted by level assignment[6] with no repeated elements.

   (b) Detection of cyclic dependencies in D — an erroneous condition.

3. Subject to validity, performing the updates, following the ordering determined in step 2, updating V to V'.

Note that even if a redefinition in K is redundant in the sense that it does not change the current definition of a variable, then the relevant updates will still be performed.

If no cyclic dependencies are detected in phase 2, then the state transition from $\mathcal{M}_{\mathcal{S}}$ to $\mathcal{M}_{\mathcal{S}'}$ will be successful. If cyclic dependencies are detected in phase 2, the changes made in phase 1 are reverted and the state remains at $\mathcal{M}_{\mathcal{S}}$, as described earlier for the protected_update operation.

[6] This terminology is from [HNC65, p.267] — more explanation in Appendix §3.A, p.178.

The BRA requires some state additional to the representation of the dependency graph to be stored. Two kinds of additional state are required. The first requirement derives from Knuth's use of counters in the topological sort algorithm to represent the current state of the algorithm's traversal of the graph. Cartwright incorporates these counters into his BRA, naming them Knuth Counters (KC). One Knuth Counter per definition is required. The second requirement arises from an assumption [Car99, p.126] that the calculation of the set of targets[7] for any particular reference is "trivial" (i.e. computationally of negligible cost). The efficiency of the BRA depends heavily upon this calculation. It is possible to speed the calculation by effectively precomputing the result, storing doubly-linked source and trigger pointers rather than singly linked source pointers. This is the approach taken in the DAM machine implementation [Car99, p.151].

In connection with Cartwright's BRA, it may be noted that — because of the characteristics of Knuth's topological sort — the order of update execution is dependent on the order of redefinitions in the input block K. This has implications where the potential use of the BRA as the basis for a hybrid definitive-procedural machine is concerned and in particular in respect of the extension of the DAM machine to deal with actions (*cf.* the breadth-first sort used in EDEN, described in §4.3.3).

3.1.3 Efficiency and generality of the DMM

The number of element updates (minor transitions) performed during an update operation (major transition) in the DMM block redefinition algorithm, then, "is at worst n and the actual number of updates is often lower, depending on the context of the current dependency structure and the block of redefinitions" [Car99, p.125]. Cartwright [Car99, §4.4] presents two case studies "in which well-known sequential algorithms are represented in a dependency structure". The first case study consists of a dependency structure that finds the minimum and maximum values in a set of data of size m. Cartwright tabulates the number of element updates required for different $\|K\|$, which range from $2\log_2 m+1$ (for $\|K\| = 1$) to $3m-2$ (for $\|K\| = m$) respectively. The second case study consists of a dependency structure that sorts a

[7] Direct dependents, or 'd_dependents', in Cartwright's terms.

set of data, in a version of the Batcher bitonic merge sort algorithm [GS93], and a similar analysis is given.

But although the BRA is proposed on grounds of efficiency (*cf.* the quote from [Car99] concerning EDEN in the previous section), it requires many calculations to be made to topologically sort the updates and also much additional state to be stored. This is justified (although not explicitly) by a major assumption relating to Cartwright's application — [Car99, p.122] states: "The arguments presented are based on the premise that each update is expensive in comparison with the calculation of the update ordering". This may be true if the application involves rendering of complex geometry (the subject of [Car99]), but may not be true in general (for example, for many of the `tkeden` models in the empublic archive [WRB]).

Another, broader, assumption made by Cartwright and other authors of similar definition maintainers[8] is implicit. The DMM places no restrictions on evaluation, other than it must not occur whilst an update is in progress. The framework therefore requires the updates to be fully eager: i.e. that the value of everything must be up to date after the update operation has terminated. The DMM therefore uses evaluation/storage strategy 2, evaluate-at-every-redefinition, storing formulae and values. Again, this choice may be appropriate for some applications (e.g. if it is not possible to predict what state will be observed), but strategy 1 or 3 may sometimes be more appropriate.

Two factors determine the efficiency of Cartwright's BRA: the time taken to perform topological sorting in phases 1 and 2, and the time required for actual evaluation in phase 3. The optimal algorithm for topological sorting is not known. Knuth [Knu73, p.265] however states that the execution time of his topological sort algorithm, on which the BRA is based:

> ... has the approximate form $c_1m + c_2n$, where m is the number of input relations [pointers from elements to targets], n is the number of objects [elements], and c_1 and c_2 are constants. It is hard to imagine a faster algorithm for this problem!

The aim of the BRA is to effect a block of redefinitions in one single, efficient state transition. In this aim, it is similar to the ADM's aim of processing multiple redefinitions in a single major transition, conceptually in parallel. However, the `am`

[8] Dominic Gehring's MoDD [Geh] makes the same assumption.

implementation of the ADM avoids the problem of scheduling the updates by using evaluation/storage strategy 1: evaluate at every use. `am` does not require change to be propagated in an efficient ordering, as the values of definitions are evaluated on demand.

Both the ADM and the DMM use the notion of a machine cycle, implying global synchronisation through some kind of clock: the ADM is explicitly based on machine cycles and the DMM is synchronised through calls to the `update` routine.

3.2 The DAM machine, from the bottom up

The DAM machine is an implementation of the DMM and the BRA, described in [Car99, §5]. Cartwright's descriptions of the implementation take a top-down perspective, treating LLDN and the DMM as something to be implemented, the implementation being guided by the BRA. The specifics of the implementation do not receive much treatment.

The following section takes the opposite approach, starting at the hardware level and working upwards towards the abstract ideals. This bottom-up approach leads to new insights about LLDN and the DMM, some of which will be explained here. The insights also motivate §5.3, which presents a new DMM-like model.

The material that follows is based largely on practical experience I have had with the DAM machine implementation, rather than [Car99, §5] (which gives little implementation detail) — I have had privileged access to the same machine used by Cartwright.

The bottom-up description in the remainder of this section has the following form. The hardware platform supplied by the specific machine used for implementation is first described. Subsequent subsections discuss the representation of values in the DAM machine, how operators are called by the BRA, what internal data structures the DAM machine requires beyond that described in the DMM and finally how some agent external to the DAM machine can read and change definitive values. The following section then describes !Donald, an application that uses the DAM machine as a basis.

3.2.1 The platform basis for the DAM machine

The DAM machine is an implementation of the DMM and the BRA on the Acorn Risc PC platform. The particular platform that the DAM machine resides on is a Risc PC 700, made in circa 1995. A small amount of technical detail about this particular computing system relevant to the DAM machine implementation is warranted here. Most of the material is derived from [aco92]. Although this detail is not initially required for a good understanding of the DAM machine, it becomes relevant later in this chapter in §3.4 and §3.5.

The Risc PC[9] 700 contains a single ARM710 CPU, clocked at 40MHz. The ARM processor uses 32 bit words. A word can hold a complete processor instruction, an address reference or a some other data value. The ARM processor has sixteen 32 bit registers accessible when in user mode[10], named R0 to R15. R15 is the program counter (PC). R14 is used as a subroutine link register by instructions such as BL (branch and link). R13 has a special interpretation which is not important here. R0–R12 are free for use as general purpose registers.

The processor has a 32 bit data bus, so an instruction or 4 bytes of data can be fetched in a single step. It has a 26 bit address bus, so 64 Mbytes of memory can be directly addressed. The particular machine in question has 18 Mbytes of physical memory. The logical 64 Mbyte address space is mapped to the physical memory by the custom MEMC ("Memory Controller") chip.

The computer has video output circuitry which can drive a standard CRT display. Internally, the custom VIDC ("Video Controller") chip reads data from the video buffer (which is located within conventional addressable RAM) using DMA under control of the MEMC chip. The data is serialised by the VIDC chip into pixels (various modes with varying bits per pixel are available) and then passed through a colour lookup palette before being provided to three digital to analogue converters (DACs) which drive the outputs for the CRT display.

The final custom chip, IOC (Input/Output Controller) manages interrupts and

[9]Expansion boards could be purchased to add another ARM or Intel x86 processor to run in parallel with the standard processor, hence the "PC" ingredient of the model name.

[10]There are also 11 other registers accessible in other processor modes such as SVC "supervisor" mode, but these are not relevant here as the DAM machine runs entirely in user mode.

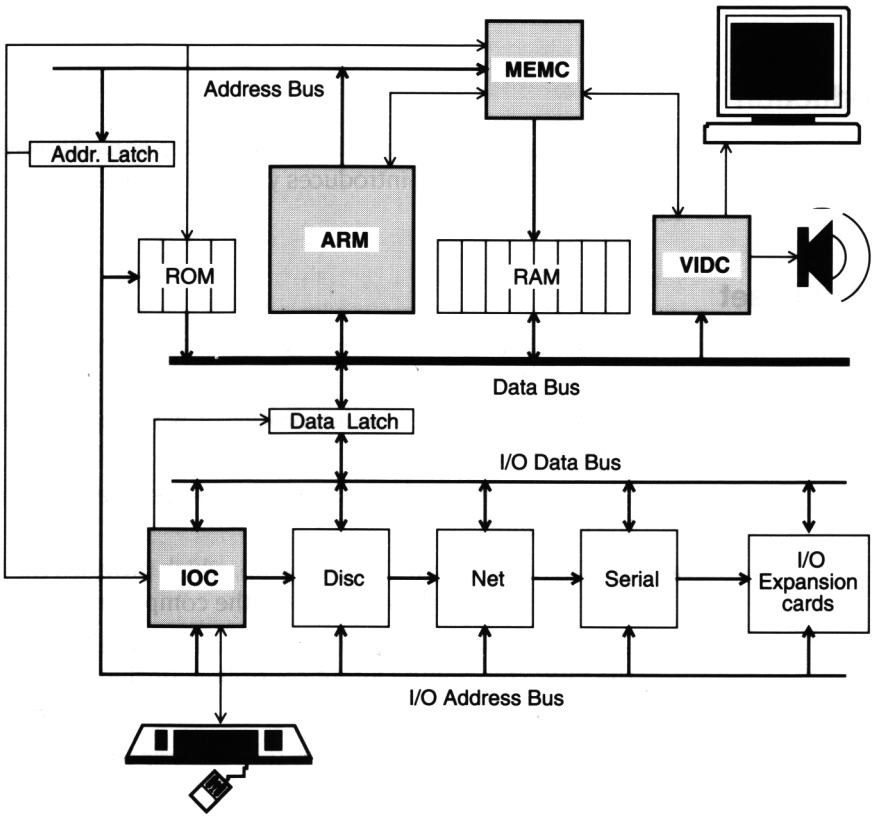


Figure 3.1: Architecture of an Acorn Archimedes 400 series computer, predecessor of the Risc PC (reproduced from [aco92, p.1-10])

peripherals. Figure 3.1 (reproduced from [aco92, p.1-10]) gives a schematic of the architecture.

The "RISC OS" operating system (on the machine in question, version 3.60) is modular and mainly located in ROM. The OS implements clipped graphics drawing routines, a memory heap manager and a cooperatively multitasking GUI among many other features. The main way of accessing OS routines from program code is through the use of an ARM SoftWare Interrupt (SWI) instruction, which changes the processor to SVC "supervisor" mode and branches code execution through a vector into the OS.

The computer system can be programmed directly in ARM assembler, or by using the provided BBC BASIC V interpreter.

3.2.2 Values

In the DAM machine, each definition value is stored in a single word of memory, and so, if considered as an unsigned integer, can have range 0 to $2^{32} - 1$. The values maintained by the DAM machine are stored in a contiguous area of memory of a fixed size known as the *definitive store*. We shall return to these limitations on the geometry of the definitive store in §3.5.

In the abstract DMM, all values are integers and they are assumed to have unbounded range (i.e. the representation problem is ignored at this level of abstraction). In the concrete DAM machine, values are essentially treated as a fixed size opaque type — i.e. each value is a sequence of 32 bits, the particular states of which hold no intrinsic significance to the DAM machine algorithm itself. Knowledge about the specifics of the type representation in use are encoded into the operators.

3.2.3 Operators

The programming user[11] of DAM must construct *operators* in ARM assembler, corresponding to *functions*[12] in $\mathcal{F}$ in the DMM. Operators are invoked in phase 3 of the BRA. When an operator is invoked, it is passed a sequence of 10 argument values[13] and when it terminates, it must return a single value. The problem of determining the necessary 32 bit representations for a particular domain is left to the programming user of the DAM machine who determines the representations as implicitly encoded in the way in which operators treat the values. This separation of concerns is similar to the way that assembly languages deal only with operations on basic types, such as arithmetic on integer and floating point representations and basic array indirection accesses, leaving higher level types to be translated into these atoms by the compiler.

Each register in the ARM processor is 32 bits wide and so can hold a DAM machine value or a machine address reference (recall that the Risc PC's address bus

[11] Meaning someone who writes a program incorporating the DAM machine, not the implementer of the DAM machine itself or the user of a program which incorporates the DAM machine.

[12] Note the change in terminology from functions to operators, which is intended to emphasise the operational aspect of the DAM machine: operators are *invoked* to bring state back into consistency with the relationship described by the function and arguments.

[13] Actually, references to values: see the next paragraph.

```
.add                ; On entry: R0 = address of value b
                    ;           R1 = address of value c
                    ; On exit:  R0 = b + c
   LDR R2, [R0]     ; Load R2 with value at address R0 (b)
   LDR R3, [R1]     ; Load R3 with value at address R1 (c)
   ADD R0, R2, R3   ; Set R0 equal to R2 + R3 (b + c)
   MOV PC, R14      ; Exit with a jump back to DAM
```

Listing 3.1: ARM code for a DAM machine `add` operator (from [Car99, p.153])

is 26 bits wide). The actual sequence of operations that constitutes a single element update in phase 3 of the BRA is shown below.

1. The BRA loads R0–R9 with the sequence of pointers to argument values within the definitive store region of memory. If there are less than 10 arguments, then the sequence is terminated by a pointer to address zero;

2. The BRA loads R14 with a pointer to the return point (the position in the BRA code where execution should continue after the operator has terminated);

3. The BRA jumps to the start of the operator code (as recorded when redefinitions are introduced to the data structure);

4. The operator code performs its task, using the values pointed to by R0–R9;

5. The operator code loads the single valued result into R0;

6. The operator code jumps to the address in R14, returning control to the BRA.

For example, the ARM code for a binary `add` operator, taken from [Car99, p.153] is shown in Listing 3.1.

3.2.4 Data structure

The DAM machine operates on a data structure, some of which is a direct implementation of that already described in the DMM and some of which is an extension.

Figure 3.2 illustrates the representation[14]. The script used in this example contains a variety of operators. The "top-most" definition, `i`, forms the power of `h` and `a`, so this example will be known as the "power" script.

As described earlier in §3.1.1, the DMM maps an identity to a function, a sequence of arguments and a current value. These mappings are accordingly represented in the DAM machine.

- Values are stored in a particular region of memory, ranging α to $\beta - 1$, known as the Definitive Store. Each Value is a 32 bit word.

- A Value stored at address $\alpha + i$ is associated with a Function Pointer, stored at $\alpha + \kappa + i$. Each Function Pointer is a 32 bit word intended to be interpreted as an address, which references the start of the sequence of operator code which is used in the calculation of the Value. Each sequence of operator code is terminated by the 32 bit opcode corresponding to `MOV PC, R14`, as shown for the `add` operator in Figure 3.2 and in the ARM code given earlier for `add`. The memory that holds the operator object code is known as the Function Store.

- A Value stored at address $\alpha + i$ is associated with a Sources List Pointer, stored at $\alpha + \lambda + i$. Each Sources List Pointer is a 32 bit word address which references the start of a sequence of Source Pointers (each being a pointer to a Value). Each sequence is terminated by a pointer to address zero (which is unused by the DAM machine). The memory that holds the sequences of Source Pointers is known as the Sources Store.

In addition to the mappings described in the DMM, to enable use of the BRA, identities are also mapped to Knuth Counter values and to improve efficiency, identities are also mapped to sequences of Target Pointers (as was briefly mentioned on p.107):

- A Value stored at address $\alpha + i$ is associated with a Knuth Counter value, stored at $\alpha + \varphi + i$.

[14] The figure is based on [Car99, p.149], but this version illustrates the redefinition interface, eliminates reference by identity within the data structure and uses terminology consistent with this thesis.

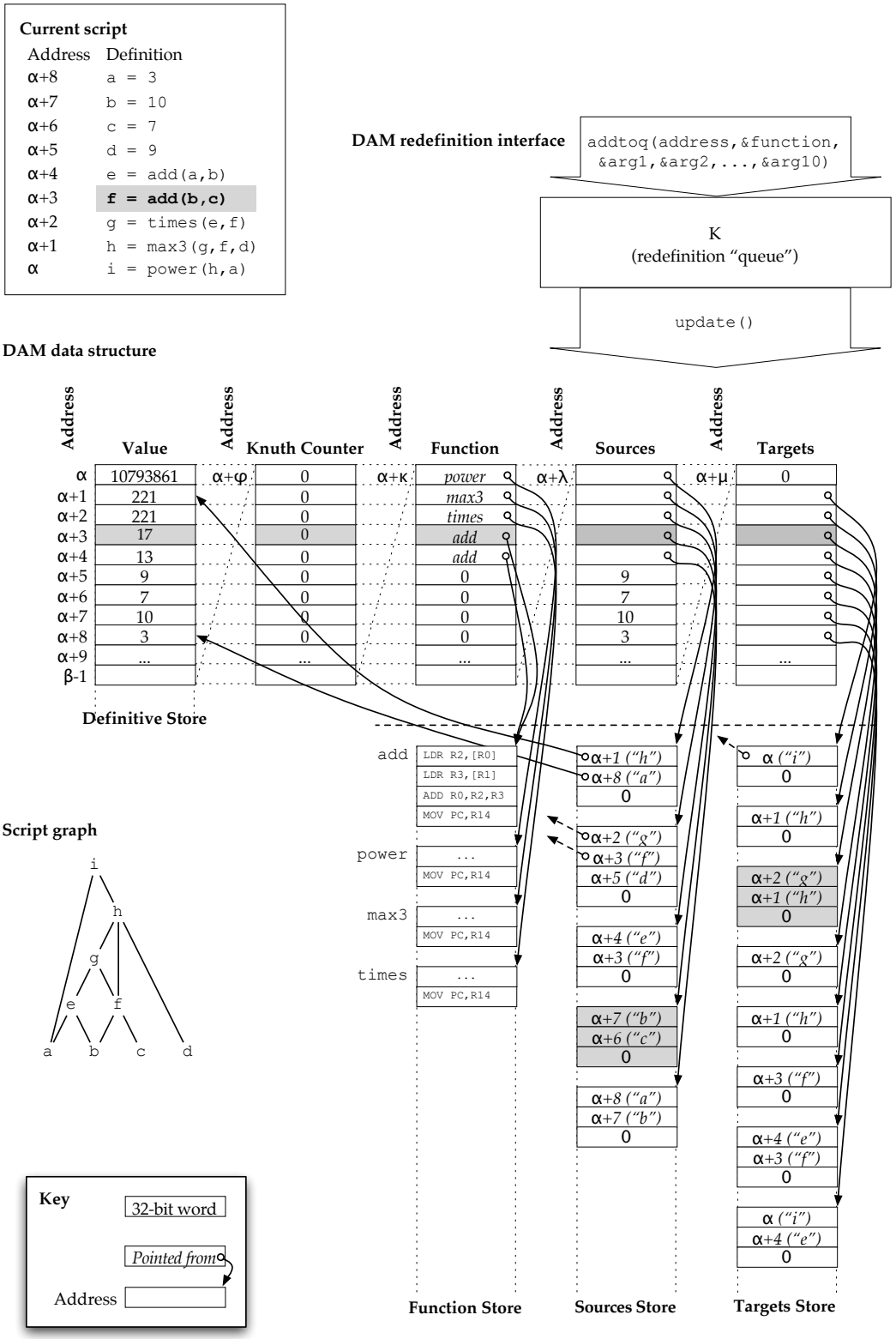


Figure 3.2: DAM machine data representation and interface routines

- A Value stored at address $\alpha + i$ is associated with a Targets List Pointer, stored at $\alpha + \mu + i$. Each Targets List Pointer is a 32 bit word address which references the start of a sequence of Target Pointers (each being a pointer to a Value). Each sequence is terminated by a zero pointer. The memory that holds the sequences of Target Pointers is known as the Targets Store.

For example, in Figure 3.2, the highlighted definition `f` has a value that is stored at address $\alpha + 3$. We should emphasise here that the DAM machine has no symbol table[15] and does not use symbolic references such as `f`: any occurrences of symbolic rather than numeric references in the figure are markings to aid human interpretation of the diagram. The identity-address mapping (shown alongside the symbolic script in the top left corner) has been deliberately reversed to emphasise this fact. The value is calculated by applying the operator `add` to the sources stored at $\alpha + 7$ and $\alpha + 6$ (`b` and `c`), as represented by the highlighted sequence of Source Pointers. The value is used by the definitions whose values are stored at $\alpha + 2$ and $\alpha + 1$ (`g` and `h`), as represented by the highlighted sequence of Target Pointers.

The data structure is spread across four areas of memory that are allocated in different ways.

- The values are stored in the statically allocated Definitive Store. The store is a fixed size.

- The Knuth Counter values and the Function, Sources and Targets Pointers are stored in statically allocated areas of memory at fixed locations offset from their corresponding Values by φ, κ, λ and μ respectively.

- The operator object code is stored in the Function Store. The location of the Function Store is not specified by the DAM machine, but in the current implementation is statically allocated and the operators are fixed.

- The Sources and Targets Stores are dynamically allocated, using the operating system's heap manager routines, as their size varies according to the number of references made by definitions in the current script.

[15] Although !Donald (see §3.3) does.

Each definition therefore requires five words of statically allocated storage (for the Value, Knuth Counter, Function Pointer, Sources List Pointer and Targets List Pointer) as a minimum, plus storage required for the sequences of Source and Target Pointers and the operator object code.

As described so far, the DAM machine can represent definitions that have the values of other definitions as source arguments, but there is no way of setting an explicit value. Literal values (for example, the definition `d = 9` in Figure 3.2) require a special case for representation in the DAM machine. The special representation corresponds to step 4 in the four stage process of translation to LLDN described earlier on p.101. Definitions whose values are defined literally use a special operator value_X [Car99, p.151], represented by a Function Pointer to address zero. The literal value is then stored in the Sources List Pointer. This is possible as literal values are source/leaf nodes (see Appendix §3.A, p.178) in the script graph and hence have no need for a list of argument references in the Sources Store. The literal value is copied across to the Value word when necessary by the BRA.

The other special case in the representation is used to describe sink/root definitions (again, see Appendix §3.A, p.178) which are not referenced by any other definitions (for example, the definition `i` in Figure 3.2). In this case, there is no need for a list of Target Pointers in the Targets Store and so the Targets List Pointer holds the special address zero.

3.2.5 DAM execution

The previous section described the static DAM machine state. This section considers dynamic DAM machine transitions.

Figure 3.3 summarises how a transition is made from a state S to the next state S' in the DAM machine. The example is the same "power" example from Figure 3.2. Redefinitions to state S are added to the redefinition queue K (see section §3.1.2) with the `addtoq` routine. When `update` is called, the BRA operates in three phases. In phase 1, the initial changes from K are made to the Operator, Sources and Targets Stores, changing definitions but not recomputing values, moving the state from the initial state S to the "most dirty" state, in a sense to be explained below. In phase 2, an evaluation ordering is calculated for the new script graph, and a check is made

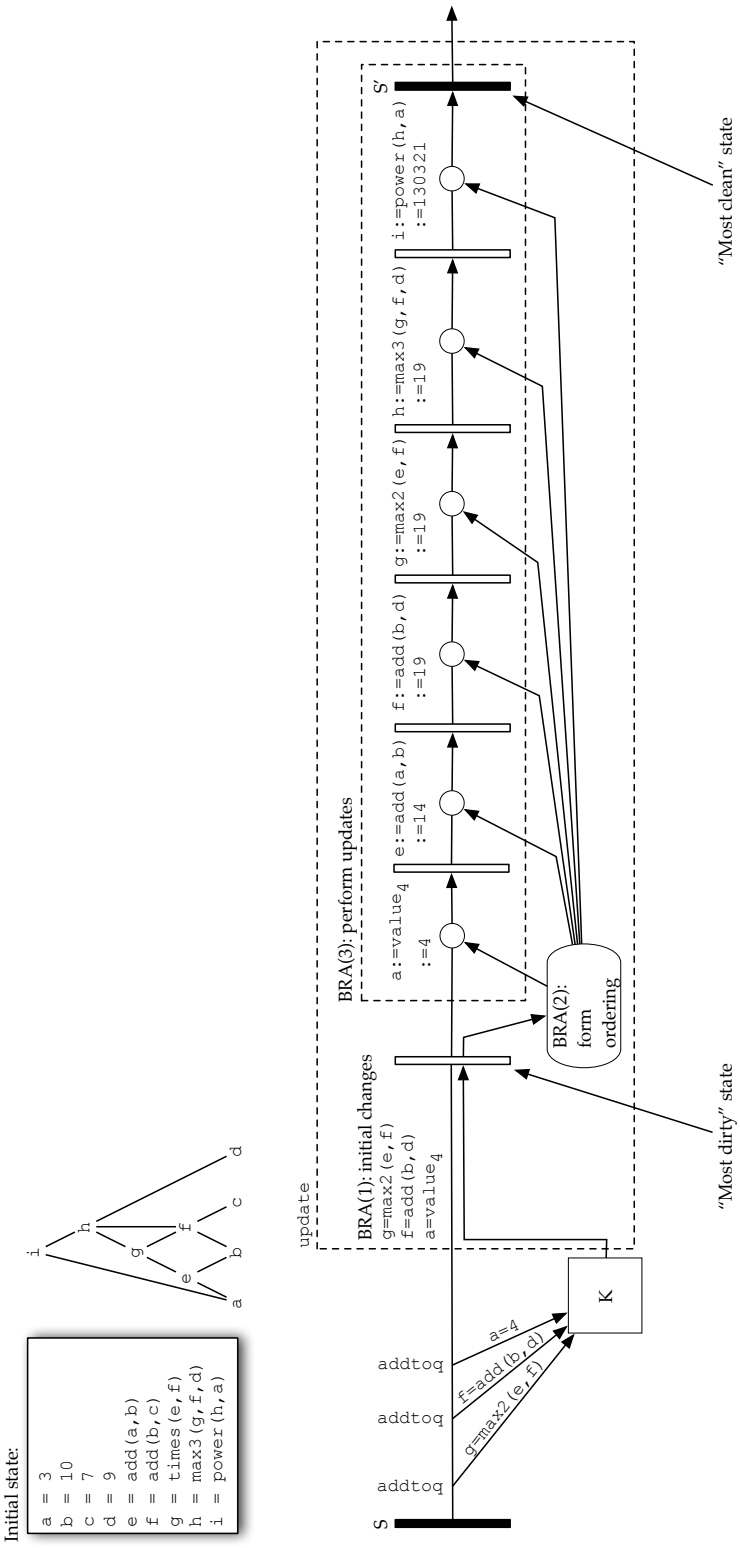


Figure 3.3: State transitions in the DAM machine

for cyclic dependency. In phase 3, the operators are executed sequentially in the ordering calculated by the topological sort in phase 2, finally leaving the values in the "most clean" final state S'.

The number of changes to state made in phase 1 of the BRA is equal to the number of items in K. $\|K\|$ is equal to the number of calls to `addtoq`. (An attempt to redefine the same location more than once results in an error.) The number of updates in phase 3 is $\|U(K)\|$, the value of which can range from $\|K\|$ to n, depending upon the contents of K and the current script. (See §3.1.2 for more explanation.)

The example shows a set of three redefinitions in K:

```
g=max2(e,f)
f=add(b,d)
a=4
```

The BRA calculates the update ordering `a`, `e`, `f`, `g`, `h`, `i`. Note that in this example, there is scope for parallel evaluation: the ordering could be ((`a`, `e`) // `f`), `g`, `h`, `i` (that is to say, the update to `f` could be performed concurrently with the updates to `a` and `e`). The DAM machine, however, is implemented on a sequential processor and does not exploit available parallelism.

In the ADM, values are always read from the previous state S during evaluation. S states result from major transitions, and so the values read are taken from a stable state, rather than an unstable $S*$ state. (See section §2.3.3 for more details.)

In the DMM, the situation is curiously different. We can consider the BRA in terms of a machine that moves from a "dirty" state to a "clean" state. The first phase in the BRA changes F to F' and D to D'. This causes inconsistency between the state as described by the definitions (in F and D) and the state as held by the values. The inconsistency is not limited to just the changed nodes: all nodes in a path above changed nodes are also inconsistent with their definition. (In the example, after the changes have been made to the definitions of `g`, `f` and `a`, then the values of all of `a`, `e`, `f`, `g`, `h` and `i` are inconsistent with their definition.) This state (effectively $S + K$) can be described as the "most dirty", since a large amount of inconsistency between definitions and values is present. The next two phases of the BRA schedule and then invoke the operators, which (through the BRA) read their sources as input and write their target as output. When phase three of the BRA

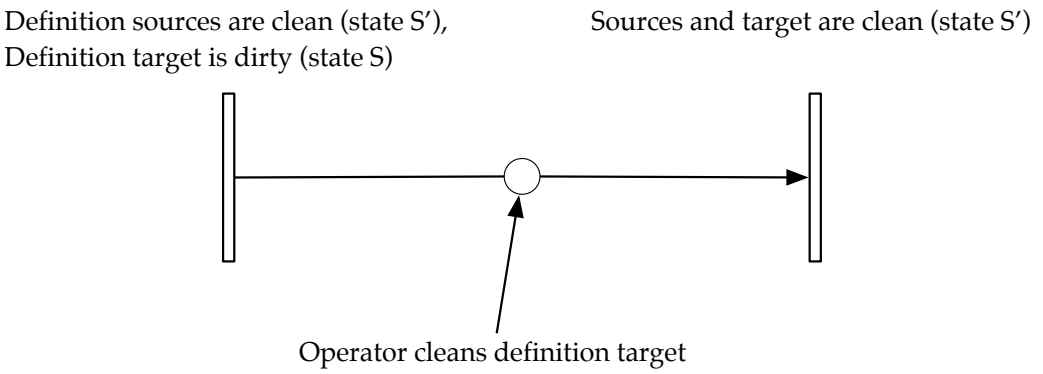


Figure 3.4: A DAM machine operator cleaning partial state

is complete, the state is the "most clean" state S', where all values are consistent with definition.

The topological sort in the BRA schedules operator invocation so that when an operator is invoked, its source values are in the "clean" state S' and its target value is in the "dirty" state S. The operator then "cleans" its target, moving it into the "clean" state S'. Figure 3.4 illustrates this concept. (Note that the figure is a slight simplification: operators in the DAM machine do not directly affect the state in memory as they interact with the BRA through processor registers, which is itself responsible for updating memory.)

3.2.6 External agency

The DAM machine is intended to be incorporated into other applications, as it is not useful on its own. This section examines the question of how other code can interact with the DAM machine.

The DAM machine redefinition interface is shown at the top of Figure 3.2. The DAM machine maintains a redefinition queue in another area of memory. Code external to the DAM machine can add redefinitions to the queue by calling the DAM machine '`addtoq`' interface routine. The routine takes several arguments, each being passed in an ARM register. Figure 3.2 uses C-style syntax, with the ampersand character `&` denoting a pointer (although the actual code is more likely to be directly coded in ARM assembler). The following arguments to the `addtoq` routine are summarised from [Car99, p.153]:

- R0: address of the Value to be redefined (in range $[\alpha, \beta - 1]$);
- R1: new Function Pointer;
- R2: new Source Pointer 1 (first argument);
- R3: new Source Pointer 2 (second argument);
- R4-R11: Source Pointers 3–10. If there are less than 10 arguments, then the sequence is terminated by a register holding address zero.

In the case of a redefinition to a literal value, the registers must be given the following values before `addtoq` is called:

- R0: address of the Value to be redefined;
- R1: zero (i.e. a value$_X$ operator);
- R2: literal 32 bit value.

Calling the `addtoq` routine has no immediate effect on the DAM machine data structure — the routine adds the specified redefinition to the queue. The "queue" is the equivalent of the set K in the DMM (see §3.1.2). The word "queue" is shown here (and in Figure 3.2) in quotes due to this inconsistency between the DMM and the DAM machine in this regard. In the abstract DMM, K is a set. In the DAM machine implementation, K is a queue — notice that the interface routine is named `addtoq`. It is important to keep in mind however that the BRA implements a transition as a set of redefinitions, not a sequence. This is similar to the discussion of command lists in the ADM, in §2.3.3.

The DAM machine `update` routine is the implementation of the BRA. It causes the block of redefinitions — built up incrementally in K through a sequence of calls to `addtoq` — to take effect on the DAM data structure atomically. This can be compared to the way in which a sequence of multiple database updates can be made to take effect on the database atomically through the use of the SQL `COMMIT` command. If any of the redefinitions in K would introduce cyclic dependency, the update is not performed, as the *protected_update* operation introduced in the DMM specifies. After the call to `update`, the redefinition queue is empty.

The two routines `addtoq` and `update` are the only DAM machine interface routines documented for use by external code. How then is external code to read DAM machine state? Cartwright states [Car99, p.155]:

> If a programmer wishes to make reference to a value in store, they simply need to load it into a register (using the "`LDR`" instruction). No DAM Machine mechanism stops them storing a value into this area of memory again (using the "`STR`" instruction). The definitive store only remains definitive if the programmer of an application that includes such a store only changes the store through calls to the "`addtoq`" and "`update`" subroutines of the DAM Machine.

External code should therefore read directly from the Definitive Store. The unstated assumption is that this only happens whilst an update is not in progress. As long as the external code is not running in an interrupt routine, on the test platform (an unexpanded uniprocessor Risc PC), this is a reasonable assumption. Interrupt routines aside, such a machine can be considered to be a single agent, playing a potential multiplicity of roles, but only one role at any given instant in time. When executing the `update` routine, such a machine is by definition not executing external code that could read from the Definitive Store.

Another way that external code can effectively "read" from DAM machine state is to include code in the operator routines that "writes" to state external to the DAM machine, in a type of operator "side effect". This technique is used in the !Donald application (see §3.3 below), but has some intrinsic problems as we describe in section §3.3.4.

Can and should external code read and write from other parts of the DAM machine data structure? The above quote implied that external code should not write to the Definitive Store, but when talking about extending the DAM machine for dynamic data structures, [Car99, p.159] continues:

> If arrays that change their size are required in an application, the programmer should implement code that is capable of shifting blocks of definitions around, either through reading the definitive store and all its associated data and repeatedly branching to the "`addtoq`" subroutine, or by moving the definitions around themselves ensuring they remain consistent to the internal representations of the DAM Machine.

A relevant concern is whether the DAM machine implementation is re-entrant: that is, whether the interface routines `addtoq` and `update` can be called from within an operator.

For most purposes, then, from the point of view of external code, the Definitive Store is considered read-only and the other parts of the DAM machine data structure are private to the DAM machine. Changes are made by using the two redefinition interface routines. In some circumstances, however, it may be necessary to directly read and write to "internal" DAM machine data structures.

3.3 !Donald

!Donald is an application written for the Risc PC architecture that implements the DoNaLD definitive notation for line drawing, using the DAM machine to perform dependency maintenance. It provides an interactive environment in which the DAM machine can be used.

This section is about an important technical point: should side-effect be allowed in definitive operators? Both the DAM machine and the more abstract DMM have a large emphasis on dependency — it is not possible for the user to program agency to the extent that this can be done in the ADM or EDEN. But if the operating system graphics drawing routines are to be used to create displays, some automated agency is required in order to call those routines. !Donald uses "graphical actions", which are DAM machine operators with side-effect to manifest this agency. This extension of DAM machine operators with side-effect causes various associated problems which are outlined in §3.3.4[16].

This section also relates to high level goals for this thesis. As !Donald includes a definition maintainer, it was hoped that it would enable interaction with meaningful state. This section shows that the interaction it does allow is limited. !Donald has been designed to allow a "top-down" style of use: the type of input is DoNaLD and the DAM machine is hidden from the user. Later sections in this chapter describe extensions to the !Donald application which reveal more of the DAM machine foundation, allowing the application to be used in a novel "bottom-up" manner. Before this can be explained, however, we must first discuss what already exists.

[16]Gehring's MoDD [Geh] definition maintainer Java API, which I used in 1998 [War98] to write an interactive definitive sheep simulator, uses a similar technique, and I encountered similar problems.

3.3.1 !Donald overview

!Donald is the work of James Allderidge, a final year undergraduate working on the project with the assistance of Richard Cartwright. There are only a few sources on !Donald. The primary sources are the author's final year project report [All97] and [Car99, §5.4]. "Enabling Technologies for Empirical Modelling in Graphics" [ABCY98] is a secondary source which mostly summarises the work in [Car99, §5.4]. The writing of this section has been informed by practical experience with the results as well as a reading of the above sources.

The !Donald application embeds the DAM machine into an application which itself is procedurally coded in BASIC and ARM assembler. The application provides several features that the DAM machine itself does not include.

- DAM machine operators which operate on graphical data types (e.g. `distance` finds the Euclidean distance between two Cartesian points);
- "graphical actions" — DAM machine operators which generate graphical output by side-effect;
- the DAMscript assembler-like language[17];
- a symbol table structure which associates variable identifiers with address locations. Each variable identifier is a string of characters. The association mapping is represented as a simple sequentially allocated array (not a more efficient hash table), accessed by some ARM assembler routines which find addresses from names and *vice versa* [All97, p.80];
- a DoNaLD to DAMscript compiler, which compiles DoNaLD script into the lower level DAMscript;
- a DAMscript parser, which interprets DAMscript and calls `addtoq`;
- a front end interface which allows DoNaLD scripts to be loaded and displayed.

[17] Cartwright [Car99, p.161] names the intermediate assembler-like language "pseudo-DAM code", the "pseudo" being included presumably to distinguish between the language and the information held in the resulting DAM machine data structure. Allderidge [All97, p.23], the original author of this part of the implementation, names the language DAMscript. This thesis uses DAMscript for brevity.

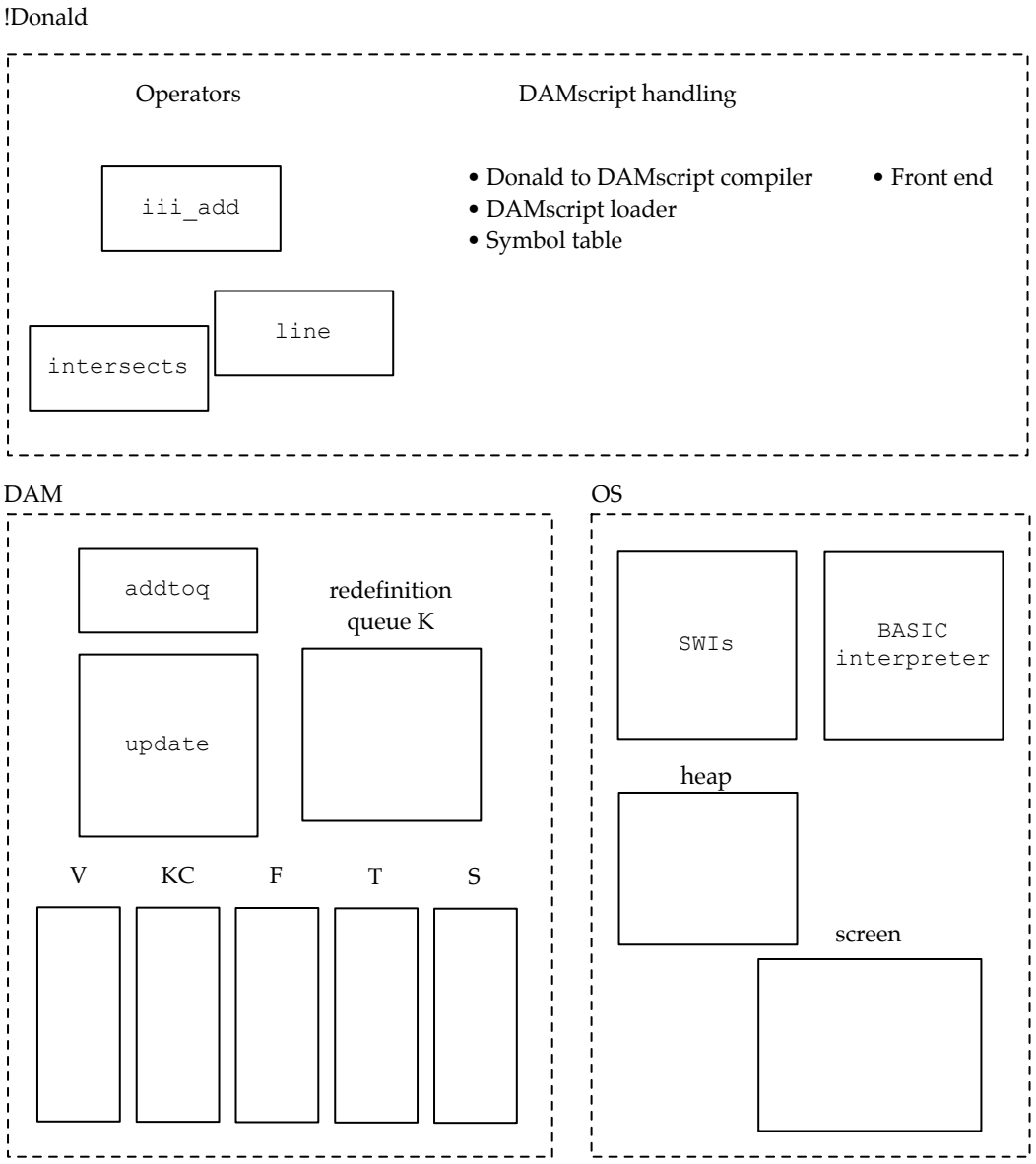


Figure 3.5: Overview of !Donald application components

Figure 3.5 shows an overview of the various components provided by the DAM machine, the operating system and the !Donald application.

The application is shown running in multitasking mode in Figure 3.6. The DoNaLD script file "engine" (1) has been loaded into the !Donald application (2). The graphical result is shown in the display window (3). The contextual menu (4) allows the DAMscript intermediate file to be saved. The "engineDAM" file (5) is

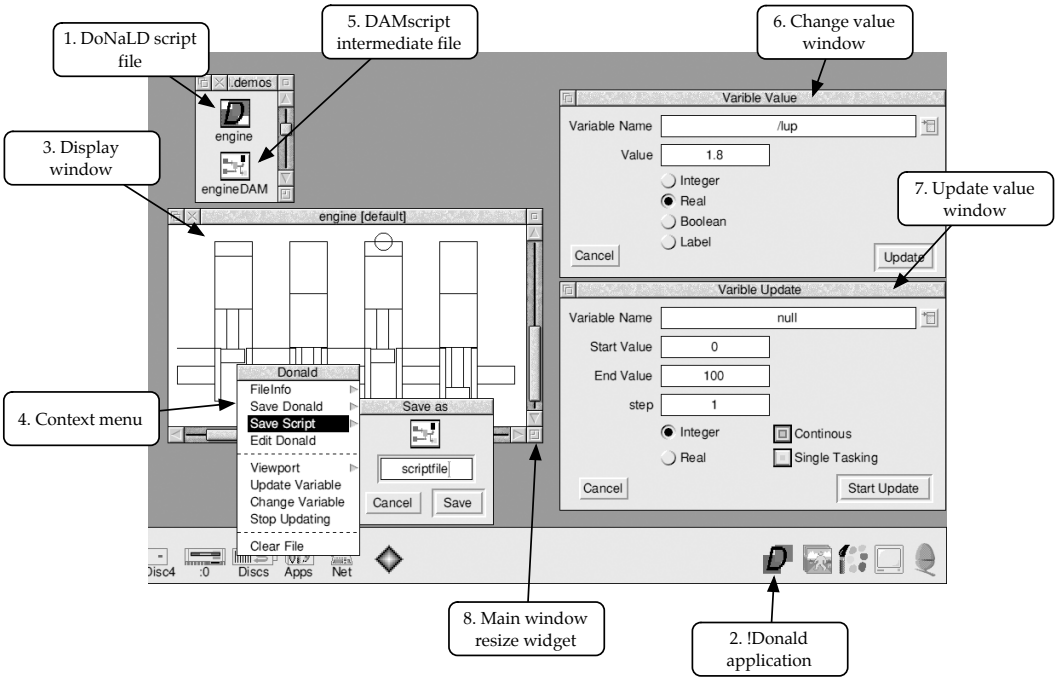


Figure 3.6: The !Donald application

the result of this operation.

The remaining three windows allow different types of redefinition to be made after the initial loading of the DoNaLD file. The "Change value" window (6) (opened from the contextual menu) allows a variable to be redefined to a literal value. In the example, the window has been used to change the `/lup` variable to the value `1.8`, which has caused the engine crankshaft to move to the configuration shown in the display window. The "Update value" window (7) allows a variable to be automatically set to a sequence of values, the user providing two limiting values and a step size. This allows animation of DoNaLD graphics to be produced. The animation can be drawn in the display window whilst cooperative multitasking with other applications continues, or the !Donald application can run in "single tasking mode", taking over the entire screen and temporarily preventing other applications from running.

Considered at a high level of abstraction, the !Donald application passes through two sequential stages of translation (using two separate parsers) in order to render a DoNaLD script on the screen. The first parser translates a DoNaLD script into

an intermediate assembler-like language named DAMscript, in a process which has a close correspondence with compilation of a high level language to assembly code. In the second stage of translation, the assembly-like DAMscript code is parsed and the `addtoq` routine is used to create definitions[18] in the redefinition queue K, in a "loading" process that corresponds roughly to conventional assemble and load operations, generating object code from assembler code and loading this into memory. In the third stage, the `update` routine is called. The definitions are created in the DAM data structure and !Donald's custom operators are invoked during the BRA, some of which generate output on the screen as a side-effect of their operation. Further calls to `addtoq` and `update` will then cause further state change which might be considered analogous to machine "execution", although the execution is entirely controlled by the particular definitions that are redefined.

The translation process is illustrated in Figure 3.7. The top of the figure shows a simple DoNaLD script (modified slightly[19] from [Car99, p.162]) that defines two points and a line. The DAMscript output of the compilation pass is shown in the middle of the figure. The bottom of the figure is a screen shot from the DAM machine running in "single tasking" mode, showing a hexadecimal dump of the DAM machine data structure and also the actual graphical line result, drawn between the coordinates {700, 100} and {800, 200} as a result of operator side-effect.

3.3.2 DoNaLD to DAMscript

The DoNaLD notation is defined by Beynon *et al* in [BABH86]. The process of compilation from DoNaLD to DAMscript is described in by Cartwright [Car99, §5.4]. Briefly, a conventional high level language (HLL) compiler reduces sequences of statements in the HLL to sequences of operations specified in assembler. The DoNaLD to DAMscript compiler performs a similar task, but the DoNaLD and DAMscript code are sets of definitions, not sequences of statements. (Some syntactical structures in both languages — for example, `openshape` containers in DoNaLD, or `viewport` statements in DAMscript — require a certain ordering of information,

[18] Technically redefinitions, but the DAM data structure is empty at this point.

[19] Although the example here is taken from Cartwright, he concentrates mostly on the mapping from DoNaLD to DAMscript. My contribution here is focussed on DAMscript and the levels below, both of which receive little treatment from Cartwright.

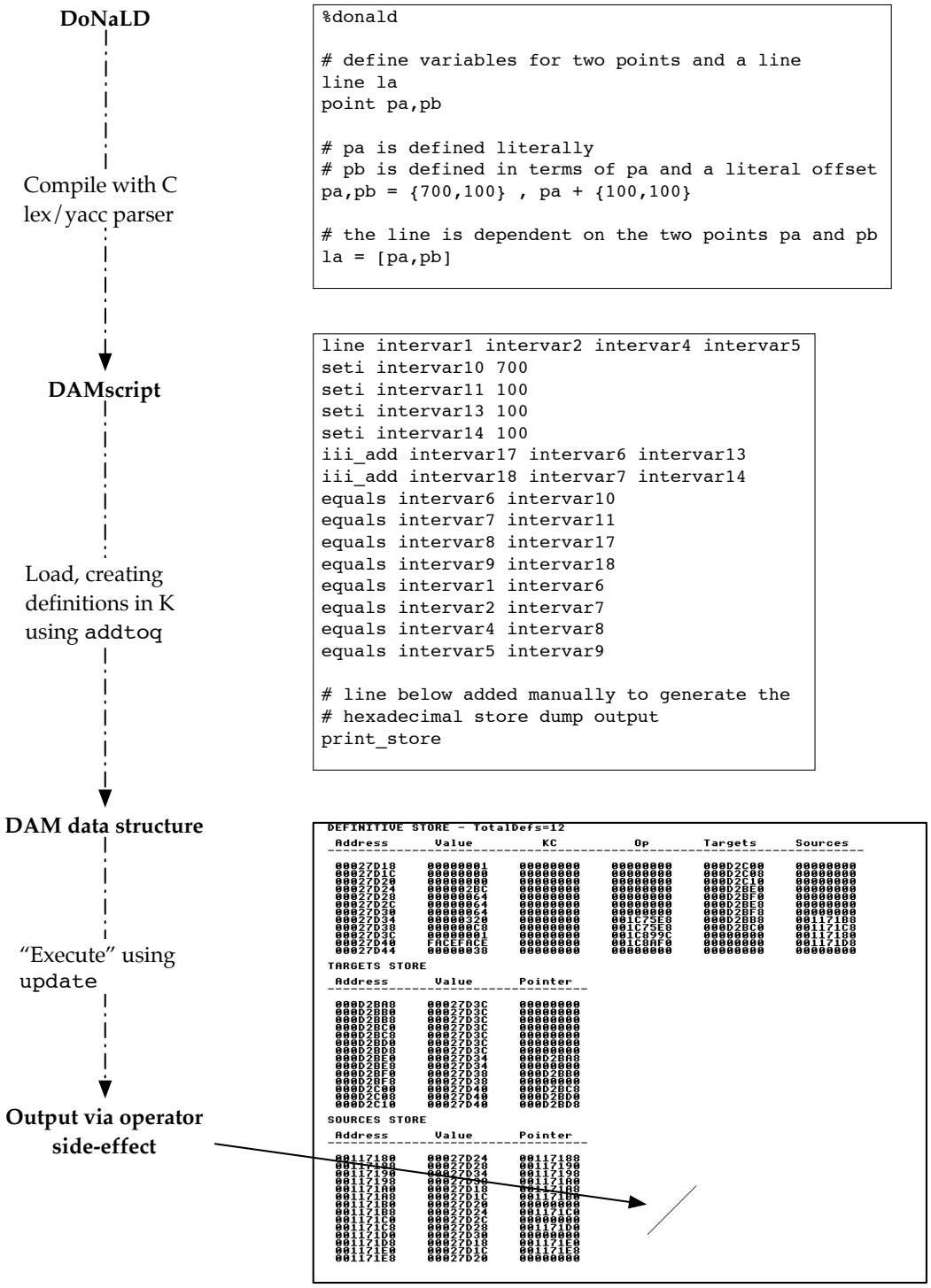


Figure 3.7: Compilation, Loading and "Execution" in !Donald

obscuring this notion.) The DoNaLD to DAMscript compiler in !Donald is derived from the DoNaLD parser used in `tkeden`, and is written in C, some of which is generated from yacc and lex files.

DAMscript is defined by Allderidge in [All97, p.24]. It provides a number of possible commands, allowing the description in human-readable textual code of:

- `set` commands, where a variable is associated with a literal value;
- DAM machine definitions, where a variable location is associated with an operator and a list of sources;
- `equals` commands, where an alias of a variable name can be constructed;
- `viewport` commands, where the graphical viewport used by drawing operators is set to a particular value until further notice;
- Graphical actions, where an operator with graphical side-effect is associated with a list of sources.

Some BNF for DAMscript is provided in [All97, p.24]. I have edited it for clarity and consistency and included it below.

file	::=	*file command* | *command*
command	::=	*set* | *definition* | *equals* | *viewport* | *graphical_action*
set	::=	*set_type variable constant*
set_type	::=	`seti | setf | setb | setc`
definition	::=	*operator_name variable dependent_list*
dependent_list	::=	*dependent_list variable* | *variable*
equals	::=	`equals` *variable variable*
viewport	::=	`viewport` *viewport_name*
graphical_action	::=	*operator_name dependent_list*

3.3.3 DAMscript to DAM data structure

Generally, a line of DAMscript results in the creation of one definition in the DAM machine data structure. The exceptions to this rule are the `equals` and `viewport` commands, neither of which cause any addition to the DAM data structure.

The DAMscript file is interpreted in two passes by the !Donald application in order to implement forward references: the first pass allocates addresses to symbolic names so that symbolic references encountered during the second pass are all defined.

The *set* and *definition* commands correspond to calls to `addtoq`, although DAMscript takes advantage of the symbol table provided in !Donald, allowing (in fact restricting) variable identities to be specified as symbolic names rather than addresses. The `equals` command does not directly affect the DAM machine data structure (although the notation appears to be introducing additional definitions, this is not the case): it manipulates the !Donald symbol table. The `viewport` command causes modification of some state internal to the assembler/loader stage, which is later used when graphical actions are created.

The DAMscript *graphical_action* command is similar to the DAMscript *definition* command. Definitions are depicted in DAMscript, as shown in the BNF above, using the form *operator_name variable dependent_list*. The definition 'a is b+c' would therefore be written in DAMscript as `add a b c`. Graphical actions are shown in DAMscript using the syntax *operator_name dependent_list*. An example action in DAMscript is `line x1 y1 x2 y2`.

There are two main differences between the DAMscript *definition* and *graphical_action* commands. Both commands create a DAM definition, which associates a sequence of input locations with an output location. When a *graphical_action* is created, three extra locations — representing viewport (`V`), x offset (`X`) and y offset (`Y`) respectively — are added to the input sequence. The actual location of `V` is related to the active viewport at time of creation of the line, and the values at the locations that `X` and `Y` reference describe the top left coordinate of the

DoNaLD graphical output window. These extra arguments are added automatically for operators that are known to be *graphical_actions* at the assemble/load stage. The extra arguments[20] are therefore not visible in the DAMscript code. The DAMscript command:

```
line x1 y1 x2 y2
```

would be created in the DAM machine data structure as:

```
line x1 y1 x2 y2 V X Y
```

The two types of commands also differ in terms of operator output. Although DAMscript *graphical_action* commands create a DAM machine definition which does have an output value in the Definitive Store, the output value is later ignored by !Donald. In addition, the location of output values from graphical actions are not stored in the !Donald symbol table and so cannot be referenced by other definitions.

Definition and graphical action operators are coded in ARM assembler. The !Donald application is provided with three pieces of information about each operator: the start address, the number of arguments required, and whether it is a graphical action[21].

3.3.4 Graphical action execution in !Donald

The BRA is designed to invoke operators in the correct sequence in order to produce a "clean" Definitive Store state, as detailed earlier in §3.2.5.

The introduction of graphical actions, however, has complicated the matter of scheduling. A need for redrawing the DoNaLD graphics can originate from two different kinds of stimulus: internally within the !Donald application or externally from the operating system.

[20]Note that the total number of arguments is limited to 10, including the hidden `V`, `X` and `Y` if appropriate.

[21]This last piece of information being encoded using the boolean variable '`has_depend%`'.

Internally to !Donald, the user might make a change to a DoNaLD variable using one of the !Donald Variable Update windows. If that change causes a subset of the geometry to change, then the graphical actions must be invoked to redraw the graphics. However they must be invoked at the correct time. The operating system graphical routines implement a global "clipping plane" setting. Calls to graphical routines will have no effect if they attempt to draw outside the current clipping planes (although if some portion of the result is within the current clipping planes, that portion will appear). If the clipping planes are not set correctly at the time that the graphical actions are invoked, at best the graphical actions will have no effect. At worst, they may draw over some part of the screen that is not currently within a !Donald window, corrupting the screen output.

External events may cause the operating system to request a partial screen redraw from !Donald. An example of an external event here would be if the user used the resize widget (marked (8) in Figure 3.6, p.126) to enlarge the !Donald window, revealing geometry that was not previously on the screen. Another example would be if the user dragged a window across the !Donald window, obscuring and then revealing portions of geometry. The operating system implements screen updates by first splitting the out-dated area of pixels into a set of rectangles, then taking each rectangle one at a time, sets the graphical routine's clipping planes to the rectangle, then sends a "redraw request" event to the application responsible for that area, continuing thus with the remaining rectangles until the screen is updated. !Donald may therefore receive a "redraw request" event resulting from some external causation.

!Donald must therefore ensure that graphical action operators are invoked only when a redraw request is made by the operating system. If there is an internal motivation for a redraw, !Donald must first ask the operating system to make a redraw request. Only when the operating system responds with the request may !Donald invoke the necessary operators.

To illustrate how the graphical actions are invoked at the correct time, the DoNaLD script shown in Listing 3.2 was created.

```
%donald
point bl
bl = {700, 100}
label lab1
lab1 = label("label one", bl)
viewport two
label lab2
lab2 = label("label two", bl)
```

Listing 3.2: A simple DoNaLD script with multiple viewports

```
seti intervar2 700
seti intervar3 100
equals intervar0 intervar2
equals intervar1 intervar3
label intervar6 intervar7 intervar8
setc intervar9 "label one"
equals intervar6 intervar0
equals intervar7 intervar1
equals intervar8 intervar9
viewport two
label intervar12 intervar13 intervar14
setc intervar15 "label two"
equals intervar12 intervar0
equals intervar13 intervar1
equals intervar14 intervar15

# this last line added manually to allow the store to be examined
print_store
```

Listing 3.3: DAMscript translation of the DoNaLD script shown in Listing 3.2

The DoNaLD script is translated by !Donald into the DAMscript code shown in Listing 3.3.

In !Donald, DoNaLD scripts have an implicit `viewport default` command at the top. The script therefore creates two textual labels, one in viewport `default`, and the other in viewport `two`. Only one viewport is displayed at a time in !Donald. The currently active viewport is selected using a contextual menu option, as shown in Figure 3.8.

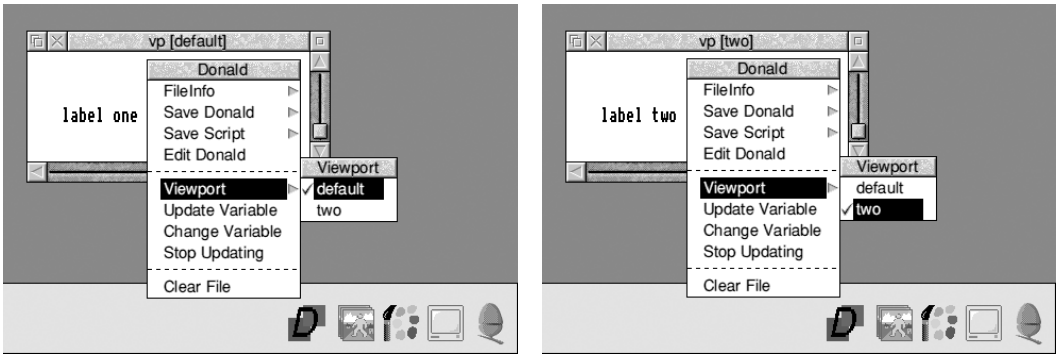


Figure 3.8: Selecting a !Donald viewport

```
DEFINITIVE STORE - TotalDefs=12
 Address       Value          KC           Op          Targets      Sources
-------------------------------------------------------------------------------
 00027D18     00000000     00000000     00000000     000D2BC0     00000000
 00027D1C     00000000     00000000     00000000     000D2C10     00000000
 00027D20     00000000     00000000     00000000     000D2C18     00000000
 00027D24     000002BC     00000000     00000000     000D2BD8     00000000
 00027D28     00000064     00000000     00000000     000D2BE0     00000000
 00027D2C     0024FFF0     00000000     00000000     000D2BB8     00000000
 00027D30     00250024     00000000     00000000     000D2BE8     00000000
 00027D34     000002BC     00000000     001C8A78     00000000     00117180
 00027D38     00000001     00000000     00000000     000D2C08     00000000
 00027D3C     00000001     00000000     001C8A78     00000000     001171B0
 00027D40     FACEFACE     00000000     001C8AF0     00000000     001171E0
 00027D44     0000003A     00000000     00000000     00000000     00000000
TARGETS STORE
 Address       Value       Pointer
-----------------------------------
 000D2BA8     00027D34     00000000
 000D2BB0     00027D34     00000000
 000D2BB8     00027D34     00000000
 000D2BC0     00027D34     00000000
 000D2BC8     00027D34     00000000
 000D2BD0     00027D34     00000000
 000D2BD8     00027D3C     000D2BA8
 000D2BE0     00027D3C     000D2BB0
 000D2BE8     00027D3C     00000000
 000D2BF0     00027D3C     00000000
 000D2BF8     00027D3C     000D2BC8
 000D2C00     00027D3C     000D2BD0
 000D2C08     00027D40     000D2BF0
 000D2C10     00027D40     000D2BF8
 000D2C18     00027D40     000D2C00
SOURCES STORE
 Address       Value       Pointer
-----------------------------------
 00117180     00027D24     00117188
 00117188     00027D28     00117190
 00117190     00027D2C     00117198
 00117198     00027D18     001171A0
 001171A0     00027D1C     001171A8
 001171A8     00027D20     00000000
 001171B0     00027D24     001171B8
 001171B8     00027D28     001171C0
 001171C0     00027D30     001171C8
 001171C8     00027D38     001171D0
 001171D0     00027D1C     001171D8
 001171D8     00027D20     00000000
 001171E0     00027D38     001171E8
 001171E8     00027D1C     001171F0
 001171F0     00027D20     00000000
```

label two

Figure 3.9: A screen shot of !Donald with Listing 3.3 loaded, showing the DAM store

Loading the DAMscript code into !Donald[22] and viewing the result in single-tasking mode, whilst continuously updating a new variable "null" in unit steps between 0 and 100, gives the screen shot shown in Figure 3.9. The constantly incrementing value for "null" is shown at address 0x27D44[23], and had the value 0x3A (61 in decimal) when the screen shot was taken[24]. Viewport two is being shown, as evidenced by the visibility of "label two".

The hexadecimal store dump can be interpreted back into the DAM data structure diagram shown in Figure 3.10. The three graphical actions are shown at the top. In the diagram, attributes of definitions are marked with the characters 'r', 'm', 'o', 'a' and 'v', representing reference, meaning, operator, address and value respectively[25]. The actions `lab1` and `lab2` at the top of the figure both take six arguments. From left to right, these are: the x and y positions of the label within the viewport, a pointer to the label text, a viewport variable and the x and y offsets representing the current viewport screen origin coordinates. The first three arguments are stated explicitly in the DAMscript code — the last three "hidden" arguments are added automatically during the !Donald "load" phase. The rightmost `print_store` graphical action (which I added manually to the file shown in Listing 3.3) takes only the three hidden arguments and requires no arguments at the level of DAMscript code.

Below the graphical actions are shown the dependencies, which in this case all happen to be literal values and hence use the "value operator" special case. The leftmost two definitions are the x and y values of the DoNaLD point `bl`. These definitions each have several reference aliases constructed by the DAMscript `equals` command, which are artefacts of the DoNaLD parsing process. Next on the left are two definitions whose values point to the label strings (which are not DAM machine definitions). Rightmost are four "hidden" definitions, which have been added automatically. From left to right, these are: the two viewport variables and the current viewport screen origin coordinates. Finally, the "null" variable

[22]Which is possible due to an extension I authored — see §3.4.1.

[23]The prefix 0x denotes hexadecimal, following the language C, as does the prefix &, used later, following the language BBC BASIC.

[24]Also made possible by another extension I authored.

[25]As noted previously, graphical actions in !Donald are DAM machine dependencies, but the address and the value are not stored in the symbol table and so are not accessible from other DAMscript commands, hence they are shown in parenthesis in the diagram.

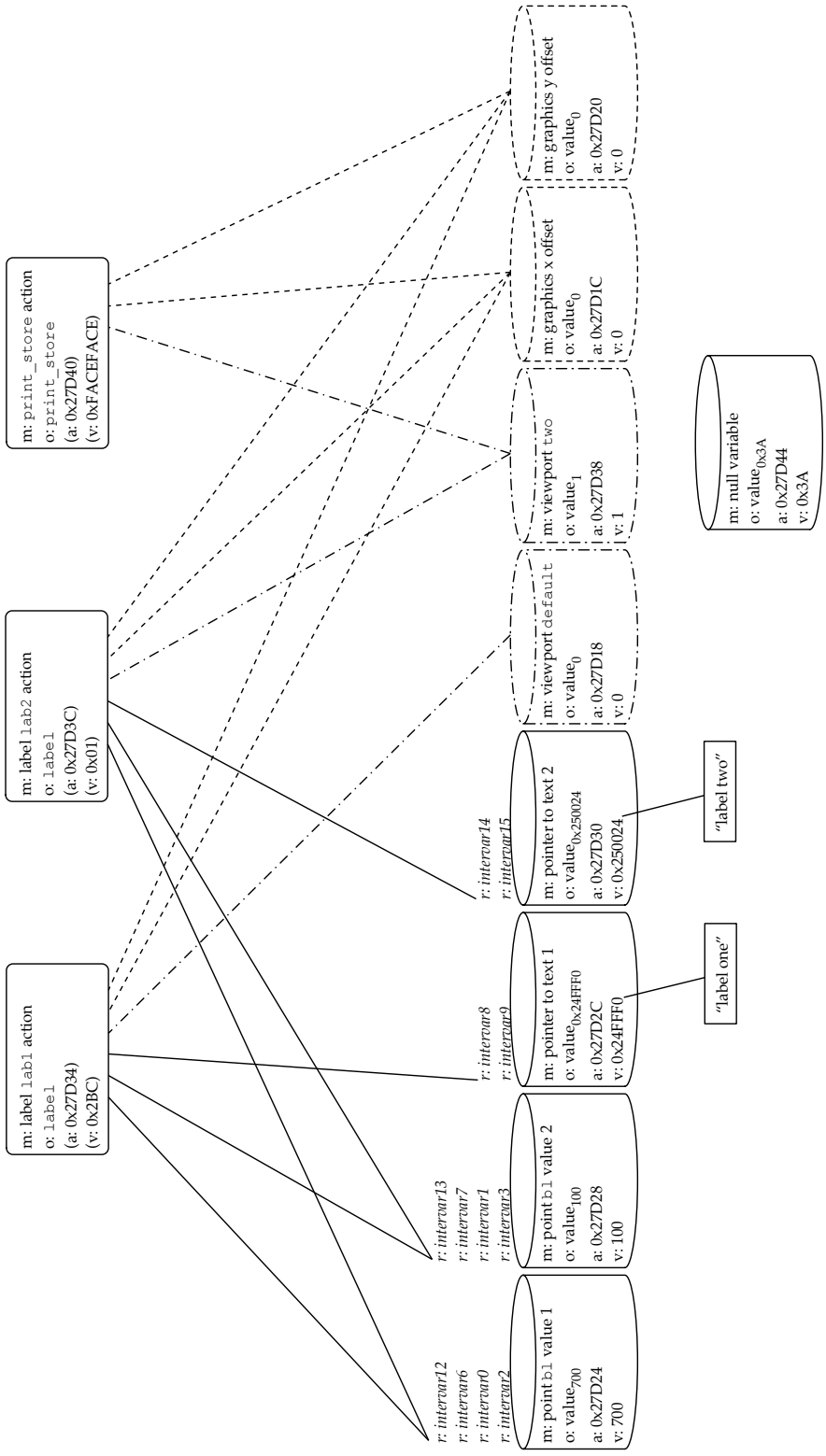


Figure 3.10: DAM machine data structure shown in Figure 3.9

introduced whilst viewing the graphical output in single-tasking mode is shown. It is an "orphan dependency" which is not used by any other dependencies.

The viewport variables are used to control when the graphical action operators are invoked, giving some control over their side-effect. Allderidge [All97, p.101] explains the process of re-displaying the DoNaLD viewport window (which I have rewritten for clarity and consistency):

1. The viewport variable for the active viewport (i.e. a single variable that is a source for all graphical actions in a particular viewport) is set to 1 using `addtoq`. All viewport variables for other viewports are set to 0.

2. The x and y viewport screen origin offsets (which are sources for all graphical actions in all viewports) are set to the correct values using `addtoq`.

3. The `update` routine is called.

4. All viewport and offset variables have been redefined. These variables have all the graphical actions as targets, and so the BRA invokes *all* the graphical action operators. (Recall that the BRA does not optimise in the $D' = D$ and $F' = F$ case.)

5. The first act of each graphical action operator is to check its viewport variable. If the value is 0, the operator exits without drawing. If the value is 1, the operator enacts its side effect, drawing the relevant graphics.

For example, the ARM code for the graphical action `line` operator, taken from [All97, p.137] is shown in Listing 3.4.

In summary, then, !Donald introduces graphical side-effect without fundamentally changing the DAM machine. This introduces three synchronisation problems that must be solved through the scheduling of operator action: 1) a graphical side-effect must not occur before the state it is graphically presenting is "clean" (in state S'); 2) it must be possible to invoke the graphical actions on demand from the operating system, and 3) it must be possible to inhibit graphical actions that exist in the dependency structure but do not correspond to an active viewport.

Allderidge solves these problems by introducing "graphical actions", which are DAM operators with graphical side-effect and extra "viewport" variables. Each

```
.line               ; On entry: R0 = line start X coordinate
                    ;           R1 = line start Y coordinate
                    ;           R2 = line end X coordinate
                    ;           R3 = line end Y coordinate
                    ;           R4 = viewport variable
                    ;           R5 = viewport X offset
                    ;           R6 = viewport Y offset
   CMP R4, #1
   MOVNE PC, R14    ; Exit if line not in the active viewport

   add R0, R0, R5
   add R1, R1, R6
   add R2, R2, R5
   add R3, R3, R6   ; Add offsets to start and end coordinates

   mov R5, R2
   mov R2, R1
   mov R1, R0
   mov R0, #&04
   swi OS_Plot      ; Call OS routine: move to start coordinate

   mov R0, #&05
   mov R1, R5
   mov R2, R3
   swi OS_Plot      ; Call OS routine: draw to end coordinate

   MOV PC, R14      ; Exit with a jump back to DAM
```

Listing 3.4: ARM code for the !Donald `line` graphical action operator

graphical action operator has as source variables the state it is graphically presenting and one of these viewport variables as a hidden source variable. The three problems are then solved: 1) because operators are invoked by the BRA only when their sources are in state S'; 2) because changing the viewport variable forces evaluation of the associated graphical actions, and 3) because every graphical action is given responsibility for first checking that their viewport is active.

There are many problems with this solution, however.

- Forced invocation is coarse-grained: Graphical action side effect occurs for *all* actions in the currently active viewport, and no others, when the above invocation process is followed.
- The scheme seems inefficient:
  - Extra variables are introduced, together with many references to them;
  - All graphical operators must first check that their result is within the active viewport.
- The scheme is sensitive to changes in the underlying BRA implementation. The purpose of the BRA implementation is to update the state to S' — it is a side effect that it achieves this by ordering the updates sequentially. Optimisation of the BRA (e.g. omitting updates where $F = F'$ and $D' = D$) in this scheme would then change the observed result.

The graphics implemented by the current !Donald are also rather simple — only 2D line plotting is implemented. More sophisticated graphics would in some cases require more sophisticated scheduling. For example, if DoNaLD object layering were to be specified and implemented (which is an important feature if solid polygons were made available), more finely grained control would be required in order to ensure that the objects are drawn in the correct order, furthest away first.

The scheduling implemented by the BRA is designed to produce "clean" definitive state, not invoke graphical actions. It is difficult to take a machine designed with a focus purely on dependency in this way and extend it to solve problems involving patterns of agency that were not preconceived during the design. A different type

of scheduling, producing a different pattern of agency, is also required when higher-order definitions are introduced — see §3.4.3. EDEN was specifically designed to separate dependency and agency concerns and is investigated in §4.3.

3.4 Definitive programming in !Donald2

3.4.1 Extending !Donald

In the original !Donald application, once a script is loaded, the only interaction possible is through the windows marked (6) and (7) in Figure 3.6 (p.126), which provide the ability to assign literal values to variables and to automatically set a variable to a sequence of values.

In order to experiment with the DAM machine at a lower level, I extended the application, renaming it !Donald2. The extensions include facilities to interact with !Donald2 using DAMscript, opening up possibilities for changing the definitive structure after the DoNaLD script has been loaded, going beyond the limited form of redefinition to literal values provided previously. The extensions include the following:

- The ability to load in DAMscript (rather than just DoNaLD) code;
- The ability to *redefine* a variable (rather than just set a new literal value) in a parameterised dialog box;
- The ability to interactively enter a line of DAMscript code, where any of the DAMscript commands can be used;
- DAMscript comments (on lines which start with #).

Facilities were also added to view the internal state and save this view. These facilities were used to generate the hexadecimal data structure dumps analysed in the previous section.

- A `print_store` graphical action (that exploits code written by Cartwright but which was not previously available in !Donald) to show the internal DAM machine data structure in a rudimentary way;
- The ability to take a screenshot whilst in single tasking mode.

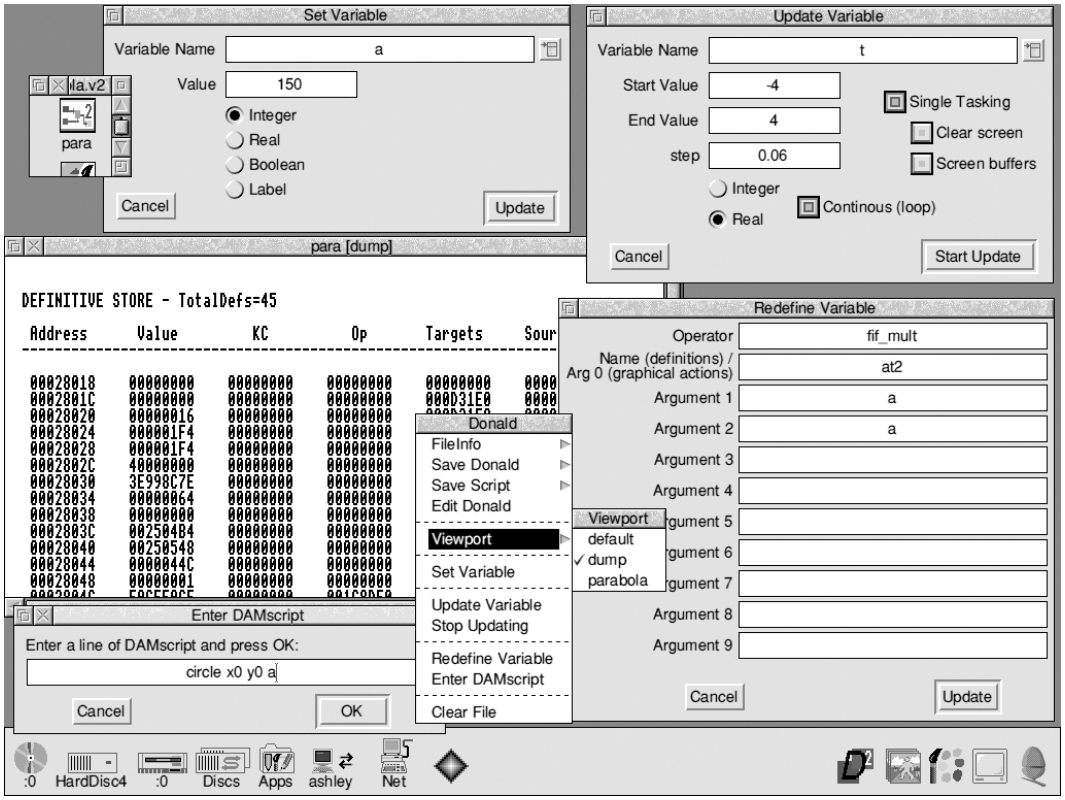


Figure 3.11: The !Donald2 user interface (compare Figure 3.6, p.126)

3.4.2 Using raw DAMscript: the parabola example

The screen shot in Figure 3.11 shows !Donald2, illustrating the new features in use. The figure can be compared with Figure 3.6 (p.126) which shows the older !Donald version.

The extensions allowing the use of DAMscript code permit !Donald2 to be used entirely at the DAMscript level, without using the DoNaLD notation at all. The DAM machine operators available in !Donald2 are described in Tables 3.2, 3.3 and 3.4 (which are derived from some joint work in April 2000 with Carlos Fischer). Some operator names describe their output and input types through the inclusion of the characters '`b`', '`i`' and '`f`' in the name (output type first), meaning boolean, integer and floating point respectively. Where names in the table use the characters '`X`', '`Y`' and '`Z`', meaning any of the above types, this abstraction is merely a device used to shorten the table — the actual operators do not use abstract types.

Boolean	Integer	Floating point	Integer and floating point
bXY_eq	iii_add	fff_add	XYZ_add
bXY_neq	iii_sub	fff_sub	XYZ_sub
bXY_gteq	iii_mult	fff_mult	XYZ_mult
bXY_gt	iii_div	fff_div	XYZ_div
bXY_lt	iii_mod	fff_mod	XYZ_mod
bXY_lteq	i_sqr	f_sqr	
bbb_and	i_ln	f_ln	
bbb_or	i_exp	f_exp	
bb_not	i_sub	f_sine	
		f_cos	
		f_tan	
		f_asine	
		f_acos	
		f_atan	
		f_int	
		i_flt	
		f_sub	

Table 3.2: Numerical !Donald2 operators

Boolean geometry	Polar	Geometry
pt_betwn_pts	pi_polar1	midpoint1
includes	pi_polar2	midpoint2
cinci	pf_polar1	intersect1
collinear	pf_polar2	intersect2
intersects		perpend1
separates		perpend2
distlargerpoint		perpend3
distlargerline		perpend4
distsmallerpoint		distance
		rot_pointssx
		rot_pointsy

Table 3.3: Geometrical !Donald2 operators

General	Graphical actions
itos	line
rtos	circle
if	ellipse
fn	label
gfn	print_store

Table 3.4: Miscellaneous !Donald2 operators

In order to demonstrate the way in which !Donald2 can be used at the DAMscript level, the "parabola" DAMscript shown in Listing 3.5 below was constructed using the operators described in the above tables.

```
# parabola in DAMscript
# Ashley Ward (ashley@dcs.warwick.ac.uk) December 2003
# with assistance from WMB and Russell Boyatt

# See the DAMscript BNF for a full explanation of syntax - but it may help
# understanding to note that definitions are 'operator target sources'

# do a hexadecimal store dump in a "dump" viewport to give a rudimentary
# display of the internal state
viewport dump
print_store

# do the rest of this script in another viewport
viewport parabola

# origin is (x0,y0), portion of screen used is (0,0) to (2x0,2y0)
seti x0 500
seti y0 500
iif_mult 2x0 x0 two
iif_mult 2y0 y0 two
setf two 2.0

# these are the variables that we /expect/ to be changed - although
# anything in this script can of course be redefined at any time
setf t 0.1
seti a 100

# x = x0 + a*(t^2)
fif_add x x0 at2
fff_mult t2 t t
fif_mult at2 a t2

# y = y0 + 2at
fif_add y y0 2at
fif_mult at a t
fff_mult 2at two at

# define a symbol named '0' that happens to have the value 0
seti 0 0

# the y axis
line x0 0 x0 2y0
# the x axis
line 0 y0 2x0 y0
# a vertical line at x0 - a
line x0minusa 0 x0minusa 2y0
iii_sub x0minusa x0 a
```

```
48
49  # form integer versions of x and y so they can be used as pixel positions
50  i_flt xi x
51  i_flt yi y
52
53  # the point p
54  setc ptext "p"
55  label xi yi ptext
56  # the horizontal line
57  line x0minusa yi xi yi
58  # the radial line
59  line aplusx0 y0 xi yi
60  iii_add aplusx0 a x0
61
62  # if p is not between (0,0) and (2x0,2y0), it is off the screen
63  bii_lt xilt0 xi 0
64  bii_lt yilt0 yi 0
65  bbb_or plt0 xilt0 yilt0
66  bii_gt xigt2x0 xi 2x0
67  bii_gt yigt2y0 yi 2y0
68  bbb_or pgt2xy0 xigt2x0 yigt2y0
69  bbb_or poffscreen plt0 pgt2xy0
70
71  # warning label that appears if p is off the screen
72  setc warntext "p is off screen"
73  label warnx warny warntext
74
75  # could use 'equals warny y0', but that creates a symbol table alias, not
76  # a new definition
77  iii_add warny y0 0
78
79  # "hide" the warning label if necessary by positioning it off the screen
80  seti largex 1100
81  # in C, the following would be 'warnx = poffscreen ? 0 : largex'
82  if warnx poffscreen 0 largex
```

Listing 3.5: The "parabola" DAMscript

The "parabola" script defines lines describing x and y axes (lines 44 and 42), a vertical line `a` units to the left of the origin (line 46), and two lines (lines 57 and 59) to the point `p`, where a textual label is located (line 55). The point `p` takes the coordinates {`x`, `y`}, where $\texttt{x} = \texttt{x0} + \texttt{at}^2$ and $\texttt{y} = \texttt{y0} + 2\texttt{at}$.

The script demonstrates the use of floating point, integer and boolean values. The above formulae are calculated using floating point values, but the results must be first cast into versions that use the integer type (lines 50 and 51) as graphical actions require pixel coordinates to be of integer type.

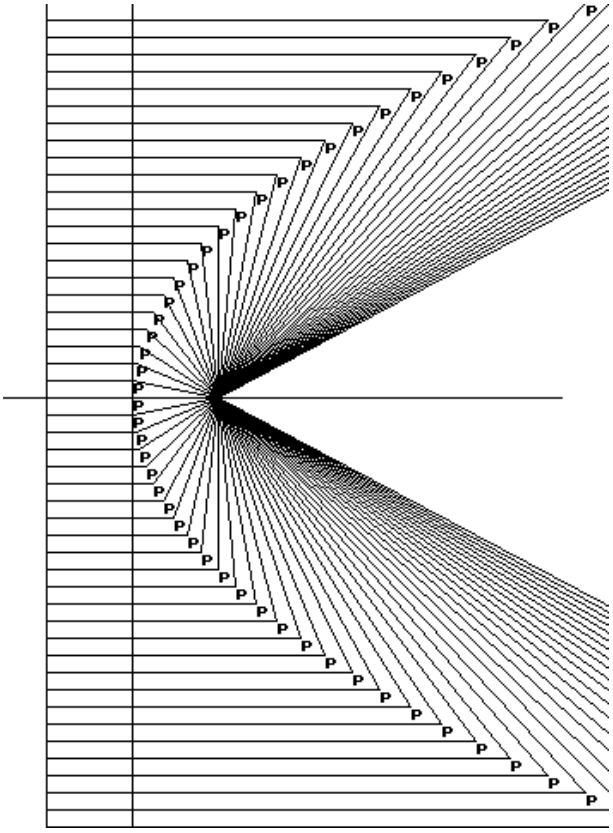


Figure 3.12: A screenshot of the parabola

Changing the value of `t` causes the point `p` to follow the parabola described by the above equations, as shown in screenshot Figure 3.12. The screenshot was taken by varying the value of `t` between -4 and $+4$ with a discrete step of 0.06, using the Update Variable window shown at the top right of Figure 3.11. Additionally, the ability to disable clearing of the screen before each update in single-tasking mode was added to !Donald2, controllable with a tick box in the Update Variable window. This enabled the resulting image for each different value of `t` to persist in the display and the parabola to be viewed.

The value `t` is not the only item in the script that can be changed: the user can redefine any item (except graphical actions, as these have no symbolic name and cannot therefore be redefined at this level). The interface presents several different ways in which changes can be made.

The Set Variable window allows a variable to be assigned a new literal value. Figure 3.11 shows the window set to redefine the variable `a` to the integer value

150. The type of a variable can also be changed using the Set Variable window. However if the type of a value is changed, the operators of the targets of that value will probably also need to be redefined to reflect their new input. For example, the `fif_mult` operator used to calculate `at` (line 35 of Listing 3.5) interprets the first input, `a`, to be a 32-bit integer value. If the value `a` was changed to a floating point value, `fif_mult` would still interpret the floating point 32 bit representation as an integer value, causing `at` to have an unexpected value.

The Redefine Variable window can be used to give a new definition to an existing variable, create a new variable with a definition or create a new graphical action. It cannot be used to give a literal value to a variable as literal values require the special case value_X form of DAM machine operator. Figure 3.11 shows the window set up to redefine the variable `at2` from $\texttt{at2} = \texttt{a} * \texttt{t2}$ to $\texttt{at2} = \texttt{a} * \texttt{a}$, causing the definition `x` to become $\texttt{x} = \texttt{x0} + \texttt{a}^3$.

Finally, the Enter DAMscript window subsumes both of the above windows, as it can be used to give any DAMscript command: `set`, (re)definition, `equals`, `viewport` or new graphical action. Figure 3.11 shows the window set to create a new `circle` graphical action centred on the origin with radius `a`. The "Enter DAMscript" dialog can be compared to the `tkeden` Input Window: it allows incremental changes to be made to the state. However !Donald2 presently lacks an easy way to view or save the state in DAMscript form, so the interaction possible is rather limited: the current state must be visualised mentally or on paper by the human user.

Notice that the line ordering of the script has no importance. (The line numbers in Listing 3.5 have been added for these printed pages for ease of reference.) For example, the value of `two` is referenced (line 19) before it is defined (line 21). The `viewport` command, however, breaks this rule. It sets the sequentially interpreting machine into a particular viewport context, and all definitions that follow it are implicitly in that context. Notice from the contextual menu in Figure 3.11 that three viewports have been defined: the `default`, which is empty; `dump`, which holds the `print_store` action defined at the head of the script and `parabola`, which holds the results of the graphical actions defined in most of the script. If it were desired to redesign the DAMscript notation so as to emphasise the unordered set nature of the input text, a more verbose form, stating the context explicitly for each definition

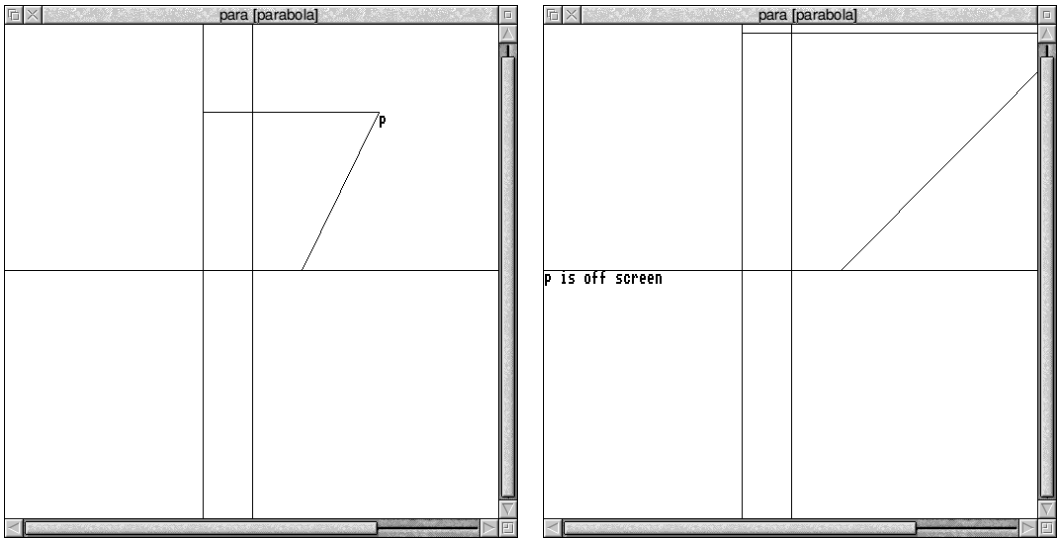


Figure 3.13: The "off screen" label in the parabola DAMscript

could be used. The following would be an excerpt from one possible solution.

```
parabola:      seti x0 500
dump:          print_store
parabola:      seti y0 500
parabola:      fif_add x x0 at2
...
```

Literal constants must be created in DAMscript. For example, line 39 of Listing 3.5 defines a symbol whose name is '`0`' that has the literal integer value zero. This is similar to how literal bit patterns are encoded into instruction data streams in von Neumann processors: literal constants must be represented at some addressable location somewhere. The `seti` command on line 39 allows the remaining script to refer to the symbol `0` as the literal value zero. (Symbols following the named variable in a DAMscript *definition* are all interpreted as variable references, so there is no uncertainty whether in that context, `0` is a symbolic variable reference or a literal constant.) Unusually, the symbol `0` can be redefined at any time to take the value of any other integer (or indeed any 32 bit pattern — see the discussion of the type of variable `a` above), and all uses of the symbol `0` will then use the new value.

The final portion of the script (lines 62–82) demonstrate the use of boolean values and the `if` operator to create a warning label that is visible if the point `p` is off the screen, as shown in Figure 3.13.

```
warny = y0
smallwarnx = 0
bigwarnx = 1100

label   {   text = "p is off screen"
            y    = warny
            x    = if p < (0,0) or p > (2x0, 2y0)
                        then smallwarnx
                        else bigwarnx
        }
```

Listing 3.6: Definitions relating to the "off screen" label, written in a fictional language

The warning label is hidden when necessary by moving it to a coordinate that is beyond the far right of the visible screen. It is not possible to remove it completely in the current application for two reasons: 1) it is not possible to remove definitions in the DAM machine (although they can given some innocuous definition such as the literal value zero) and 2) as the location of graphical actions is not stored in the symbol table (see §3.3.3), no reference can be made to the relevant graphical action in order to remove it. The definitive construct in lines 62–82 that causes the label to be moved off-screen when appropriate could be represented in some fictional higher-level definitive language as shown in Listing 3.6.

The `if` construct is implemented here as a standard functional dependency using a conventional DAM machine operator. This implementation could be written out (in some other fictional, slightly lower-level definitive language) as:

```
x = if(cond, smallwarnx, bigwarnx)
```

The `if` construct has an unusual property that makes it a form of higher-order definition[26]. In the example, the sources of `x` are the operator `if`, the boolean `cond` and *both* of the output cases `smallwarnx` and `bigwarnx`. Ideally, `x` should include *either* `smallwarnx` or `bigwarnx` in its sources, dependent upon the current value of `cond`. This would ensure that changes to the currently unselected value (`smallwarnx` or `bigwarnx`) would not cause the `if` definition (`x`) to be re-evaluated. The DAM machine does not implement higher-order dependency.

[26]This appears to be the first time that the higher-order definition properties of the `if` construct have been explained in print.

3.4.3 DAM machine operators in BASIC and the dereference problem

In !Donald, new DAM machine operators must be written in ARM assembler, somewhat limiting the accessibility of the system. Also, some of the information about the script — notably the symbol table — is not easily accessible from ARM assembler as the information is maintained by code written in BASIC. These limitations are addressed below and another form of higher-order dependency is explained.

If a section of ARM assembler has been invoked using the BASIC `CALL` statement, it is possible for that assembler to call back into the BASIC interpreter, using the `EXPR` routine[27]. In an attempt to overcome the limitations identified above, I implemented the new DAM machine operators `fn` and `gfn`, which take advantage of the `EXPR` routine, calling into the BASIC interpreter and allowing DAM machine operators and graphical actions to be coded in BASIC. Listing 3.7 below gives an example of their use in a DAMscript named "dereference", and Listing 3.8 gives the text of the BASIC functions used.

```
1  # dereference DAMscript BASIC demo
2  # Ashley Ward (ashley@dcs.warwick.ac.uk) August 2002, December 2003
3
4  # standard definitions and literal values
5  seti a 1
6  seti b 2
7  iii_add c a b
8  iii_add d b b
9
10 # names of BASIC functions used by 'fn' and 'gfn' operators
11 setc varaddress "varaddress"
12 setc deref "deref"
13 setc plothex "plothex"
14
15 # symbolic names of definitions we are addressing
16 setc name1 "a"
17 setc name2 "b"
18 setc name3 "c"
19 setc name4 "d"
20
21 # addresses of values associated with each name
22 fn addr1 varaddress name1
23 fn addr2 varaddress name2
```

[27] The `EXPR` routine is documented in [aco88, p.322]. It takes a tokenised BASIC expression string as input, evaluates it and returns the value. The location of the `EXPR` routine can be found through indirection of a value set in R14 by the BASIC CALL routine which invokes ARM assembler code. The precise implementation details are not relevant here — instead, this section focuses on the *use* of the facility.

```
fn addr3 varaddress name3
fn addr4 varaddress name4

# find values held at locations by dereferencing the addresses
# (should result in the value of each definition)
fn deref1 deref addr1
fn deref2 deref addr2
fn deref3 deref addr3
fn deref4 deref addr4

# describe geometry of table
seti xspace 200
seti yspace 40
seti x1 40
iii_add x2 x1 xspace
iii_add x3 x2 xspace
iii_add x4 x3 xspace
seti y1 980
iii_sub y2 y1 yspace
iii_sub y3 y2 yspace
iii_sub y4 y3 yspace
iii_sub y5 y4 yspace

# table column 1: symbolic names
setc nhead "NAME"
label x1 y1 nhead
label x1 y2 name1
label x1 y3 name2
label x1 y4 name3
label x1 y5 name4

# table column 2: values, referenced conventionally
setc vhead "VALUE"
label x2 y1 vhead
gfn plothex a x2 y2
gfn plothex b x2 y3
gfn plothex c x2 y4
gfn plothex d x2 y5

# table column 3: addresses
setc ahead "ADDR(NAME)"
label x3 y1 ahead
gfn plothex addr1 x3 y2
gfn plothex addr2 x3 y3
gfn plothex addr3 x3 y4
gfn plothex addr4 x3 y5

# table column 4: dereferenced values
setc dhead "DEREF(ADDR)"
label x4 y1 dhead
gfn plothex deref1 x4 y2
gfn plothex deref2 x4 y3
gfn plothex deref3 x4 y4
gfn plothex deref4 x4 y5
```

Listing 3.7: The “dereference” DAMscript

```
REM gfn "plothex" v x y
REM Plot a value v in hexadecimal at x, y
DEF FNplothex(v%,x%,y%,updatev%,xoff%,yoff%)
  IF updatev% = 1 THEN
    MOVE xoff% + x%, yoff% + y%
    PRINT ~v%
  ENDIF
= 0

REM fn resvar "deref" addr
REM Dereference a word of store at the given address
DEF FNderef(addr%, dummy1%, dummy2%, dummy3%, dummy4%, dummy5%)
= !(addr%)

REM Return the value of the null-terminated string at strptr%
DEF FNdamstring(strptr%)
  LOCAL r$, n%
  r$ = "": n% = 0
  WHILE (?(strptr% + n%) <> 0 AND n% <= 255)
    r$ += CHR$(?(strptr% + n%))
    n% += 1
  ENDWHILE
= r$

REM fn resvar "varaddress" varnamestr
REM Find the address of a named variable
DEF FNvaraddress(varnamestr%, dum1%, dum2%, dum3%, dum4%, dum5%)
= FNvariable_name_to_addr(FNdamstring(varnamestr%))
REM FNvariable_name_to_addr is a function defined in !Donald
```

Listing 3.8: The BASIC functions used by the Listing 3.7 DAMscript

Lines 34 onwards of the "dereference" DAMscript in Listing 3.7 define a graphical display in the form of a table. The numerical values in the table are defined using the `gfn` graphical action BASIC operator, using the BASIC function `plothex`. For example, the graphical action defined on line 58:

```
gfn plothex a x2 y2
```

results in a call to the BASIC function `plothex`, passing six arguments with the values of `a`, `x2`, `y2` and the "hidden" graphical action viewport variable and screen origin offsets. The BASIC function first checks that the viewport variable denotes

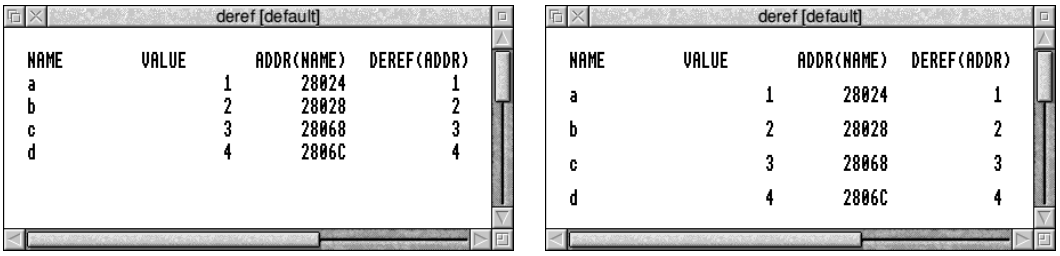


Figure 3.14: Graphical display from "Dereference" DAMscript, before and after a change to `yspace`

an active viewport (as this is a responsibility of all graphical actions — see §3.3.4); then moves the cursor to the position, correcting for the offset; and finally prints the passed value in hexadecimal.

The geometry of the table is defined in lines 34–45. When in its initial state, the graphical result is as shown on the left of Figure 3.14. If the value of `yspace` is changed from the initial 40 to 60, this change propagates to the target definitions in lines 42–45 and the table geometry is reconfigured. The result of this change is shown on the right of Figure 3.14.

As well as a demonstration of graphical table layout and use of BASIC operators, the "dereference" script is an attempt to use the symbol table (which is maintained by BASIC code) within the DAM machine. The initial part of the script (lines 5–8) defines a small set of values and definitions `a`, `b`, `c` and `d`. `a` and `b` are assigned literal values; `c` is defined to be the sum of `a` and `b`; `d` is defined to be `2*b`[28]. The values of these definitions are shown in the second column of the table, by explicit reference to their symbolic names. The leftmost column of the table shows the values of `name1...name4`, which are initially set to `a...d`. The rightmost two columns of the table are defined in terms of the `name` values shown in the leftmost column. Specifically, the `ADDR(NAME)` column shows the addresses of the locations of the values associated with the names shown in the `NAME` column. The addresses are determined by using the `fn` operator to evaluate the BASIC `varaddress` function, which looks up the name using a BASIC function from the original !Donald. These addresses are then dereferenced to find the values at each address in the final `DEREF(ADDR)` column, using the `fn` operator on the BASIC `deref` function.

[28] It could be redefined to be not `2*b` — but that isn't the question [Sha03].

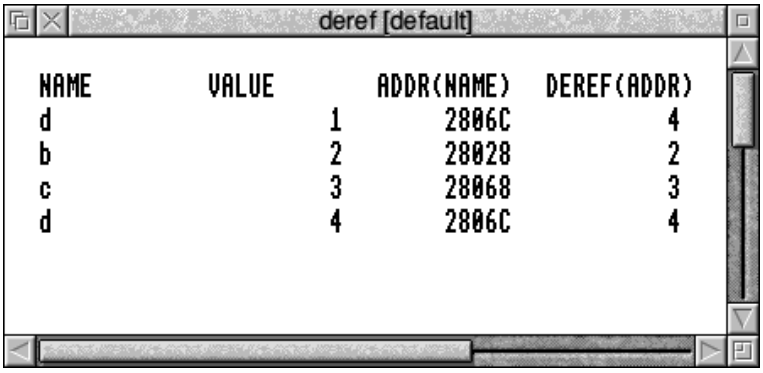


Figure 3.15: "Dereference" after the change to `name1`

If we compose the definitions used, we can think of the values shown in the final column of Figure 3.14 as being calculated by the function:

```
DEREF(ADDR(NAME))
```

If we change one of the `name` variables, the resulting looked up value changes as expected. Figure 3.15 above shows the resulting display after the DAMscript command:

```
setc name1 "d"
```

has been entered.

Notice that the top row of the last column correctly shows the current value of `d` (and also the third column shows the correct address). The second column still shows the value of `a`, as the graphical actions in the second column make explicit reference to symbolic names. The propagation of change invoked by changing a `name` variable is therefore correct.

However, the implementation of the indirect referencing means that some necessary propagation of change is omitted. If the value of `a` is redefined to 5, using the following DAMscript command:

```
seti a 5
```

then the resulting graphical display is as shown in Figure 3.16.

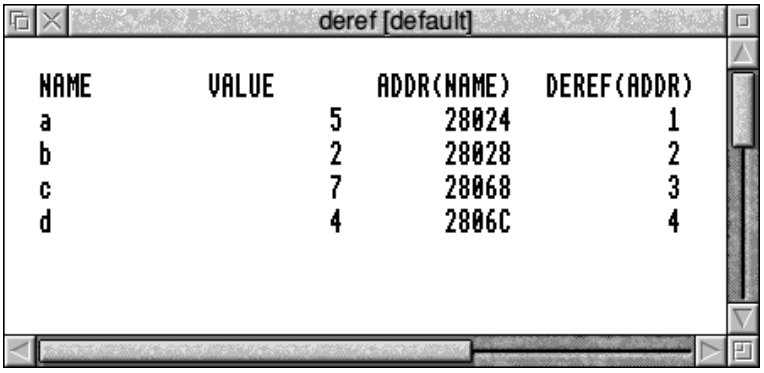


Figure 3.16: "Dereference" after the change to `a`

The values shown in the second column of Figure 3.16 (via an explicit reference) are all consistent with their definition:

```
c = 5+2 = 7
```

for example. However, the values in the last column have not changed from the pre-change configuration (which is shown in Figure 3.14). This is because the definition values visible in the last column only *indirectly* include `a` as a source. When the value of `a` changes, the value of:

```
DEREF(ADDR("a"))
```

is not updated. This flaw in the implementation leaves values inconsistent with their definition in this circumstance and exposes values from the previous state, before the transition was made.

Stated more abstractly, the sources of the composed function:

```
DEREF(ADDR(s))
```

should include the symbol whose name is referenced by the string `s`. If the string `s` changes, the sources of the function should change[29]. This causes a problem as it violates a fundamental assumption made in most definition maintainers: that the script graph only changes when a definition is changed, not when a literal value is changed.

[29]The same issue is encountered in spreadsheet tools that include the `INDIRECT` function — see p.33.

The symbol table is thus a kind of Higher Order Definition. It is comprised of symbolic references to data. When a definition uses a symbolic reference, the definition should be updated whenever the data changes as well as when the reference changes. This is not easily possible within the current DAM machine.

This problem can be related to the problem with the `if` operator described at the end of the previous section §3.4.2. There, the sources of:

```
if(cond, a, b)
```

changed according to the value of `cond`. Again, the script graph is changing when a literal value is changed.

The two examples show the need for a HOD facility in the definition maintainer, where definitions can modify the script graph. It is possible to envisage a DAM machine operator (perhaps implemented in BASIC in !Donald2) which calls `addtoq` in order to change a dependency. This might be termed a dependency-modifying action as, similar to a graphical action, it performs some action as a side effect of its invocation. The scheduling problems discussed at the end of section §3.3.4 in the context of graphical actions are then relevant to this problem. The dependency-modifying actions must be performed first in the update algorithm, as their action affects the script graph and hence the topological sort and ordering of operator invocation in phase 2 of the BRA.

3.4.4 !Donald performance

Cartwright's thesis presents some timings that compare !Donald and `tkeden`[30] and makes some preliminary claims about the advantageous performance of !Donald. This thesis explores both !Donald and EDEN in some depth, so a comparison would seem welcome. The purpose of this section is to revisit Cartwright's performance evaluation and explore the issues further. Does a definition maintainer using the BRA and coded in low-level assembler perform better than EDEN, coded in C and using a complex scheduling algorithm described in §4.3?

Cartwright evaluates the performance of two DoNaLD scripts in both !Donald and `tkeden`. The two scripts are the engine script (pictured in Figure 3.6, p.126) and

[30] Around 1999: before I introduced a full version numbering scheme.

a "shapes" test script. Both scripts are DoNaLD, not DAMscript. Measurements were taken by repeatedly increasing a parameter chosen to cause much screen geometry to be modified on each change, using the Update Variable facility in !Donald and a small script in `tkeden`, and recording the total time taken to move through all the states. !Donald was run on the Acorn platform described in §3.2.1 and `tkeden` on a SPARCstation 2. The following is an excerpt from [Car99, §5.4.2].

> The results presented here are not intended as a direct comparison of performance between systems because of their physical differences, but they do demonstrate the potential for fast dependency maintenance on a stand alone computer system running the DAM Machine. When the same DoNaLD script is executed on hardware of similar speed, the DAM Machine can produce a smooth animation where `tkeden` models appear jerky and slow.
>
> ... Timings were carried out for the DAM Machine on an Acorn Risc PC 700 with an ARM710 processor, clocked at 40MHz and capable of 36 MIPS. A similar specification SUN Microsystems SPARCstation 2 with a CY76601 processor clocked at 40MHz and capable of 28.5 MIPS was used to run `tkeden`...
>
> The table [Table 3.5] shows how the multi-tasking versions of the animation of the engine script produced a factor of 10 improvement[31] for the DAM Machine over `tkeden` and a factor of 20 to the single tasking version. For the shapes script, which contains more trigonometry and less line drawing, the speed increase is not quite so impressive with a factor of 8 speed increase for the multi-tasking SUN over the Acorn, and a factor of 11.5 for the multi-tasking SUN and the single-tasking Acorn. The visual difference between the two systems is marked. The single-tasking Acorn produces a smoother animation than the slower `tkeden` model.

Cartwright's table is reproduced as Table 3.5[32]. Timings are totals shown in seconds — lower figures are better.

The first sentence of the quoted text does acknowledge that the performance results are a demonstration of potential only. The argument presented there starts from the assumption that the two machines used have similar performance, as evidenced by similar MIPS ratings, and so timings can be compared. Unfortunately MIPS is not a good basis on which to make this comparison, as it assumes a comparable amount of work is done in each processor instruction. Patterson and Hennessy [PH98, p.76] state:

> There are three problems with using MIPS as a measure for comparing machines. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers

[31] The "factor of ten improvement" claim also appears in [ABCY98].

[32] The table headings have been changed, the column ordering organised and multipliers added to indicate performance relative to preceding columns, in the same manner as the quoted text.

Script	`tkeden` on a SPARCstation 2	!Donald on the Acorn, multi-tasking GUI mode	!Donald on the Acorn, single-tasking GUI mode
Engine	262	26 (x 10 over `tkeden`)	13 (x 20 over `tkeden`, x 2.0 over !Donald GUI)
Shapes	172	21 (x 8 over `tkeden`)	15 (x 11.5 over `tkeden`, x 1.4 over !Donald GUI)

Table 3.5: `tkeden` and !Donald performance compared when running on different machines (from [Car99, p.169])

Script	`tkeden` on the Acorn under ARM Linux	!Donald on the Acorn under RISC OS, multi-tasking GUI mode	!Donald on the Acorn under RISC OS, single-tasking mode
Engine	530	30 (x 18 over `tkeden`)	13 (x 41 over `tkeden`, x 2.3 over !Donald GUI)
Shapes	650	53 (x 12 over `tkeden`)	35 (x 19 over `tkeden`, x 1.5 over !Donald GUI)

Table 3.6: `tkeden` and !Donald performance compared when running on the *same* machine

> with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus a machine cannot have a single MIPS rating for all programs. Finally and most importantly, MIPS can vary inversely with performance!

Running all tests on the same machine would remove the hardware variable from the comparison. Accordingly, this author installed ARM Linux (version 2.0.36) on the Risc PC and compiled `tkeden` (version 1.17). Cartwright's tests were recreated as far as possible[33], all running on the Acorn machine. The test results are shown in Table 3.6.

Although the absolute timings are different, the ratios of improvement that

[33]Some information about the original tests is no longer available — particularly, the range over which the variable was changed and the step size. In the engine script, the `/lup` variable, and in the shapes script, the `/update` variable were changed from 0 to 255 in integer steps (although in the case of the engine script, the variable is actually of floating type type).

single-tasking mode has over multi-tasking mode are the same as in Cartwright's results, giving confidence that this is a similar experiment. However, the performance of `tkeden` appears significantly worse on the Acorn as compared to the SPARCstation 2, despite the Acorn having a higher MIPS rating than the SPARC, again indicating that MIPS is not a useful basis for comparison here. Is the DAM machine at least a 12 fold improvement over `tkeden`, then, or is `tkeden` just performing badly under the ARM Linux OS?

Much of the difference in observed total time may be due to the differing graphical hardware employed on each platform. Running `tkeden` on the same hardware as !Donald eliminates that source of variation, but there are major differences in the graphical subsystems *software* architecture between RISC OS and ARM Linux and in the ways that !Donald and `tkeden` use these. For example:

- The OS graphical routines called by !Donald on the Acorn write directly to screen memory, whereas the Tcl/Tk graphical routines used by `tkeden` on UNIX platforms communicate with an X Windows server process which writes to screen memory, using the X Windows protocol, facilitating graphical display from remotely executing programs.

- Qualitatively, Cartwright points out that the Acorn seems to produce smoother animation than `tkeden`. This is true, and is due to the use of a double-buffering technique employed by Allderidge in !Donald. The complete new state is drawn into an off-screen buffer which is then made visible during the CRT flyback period (thus limiting the frame rate to 60Hz in the single-tasking screen mode). The drawing process is thus not visible to the external viewer. I added an option in !Donald2 to disable the use of double-buffering, and the engine script running in this mode does not update smoothly. However it completes the above experiment in the even faster 8 seconds (*cf.* 13 seconds in Table 3.6), as in this mode it no longer needs to wait for the CRT flyback period. This is an example where the prediction of major transition time discussed on p.104 would prove useful — if this were possible, the ability of the machine to meet the real-time deadlines imposed by the CRT flyback period could be guaranteed for a particular definitive script.

Eden	DAMscript
`i=1;`	`seti i 1`
`i_sqr is i*i;`	`i_sqr i_sqr i`
`iii_add is i+i_sqr;`	`iii_add iii_add i i_sqr`
`iii_sub is i_sqr-iii_add;`	`iii_sub iii_sub i_sqr iii_add`
`iii_mult is iii_add*iii_sub;`	`iii_mult iii_mult iii_add iii_sub`
`iii_div is iii_sub/iii_mult;`	`iii_div iii_div iii_sub iii_mult`
`f_int is float(i);`	`f_int f_int i`
`f_sqr is pow(f_int,2.0);`	`f_sqr f_sqr f_int`
`fff_add is f_int+f_sqr;`	`fff_add fff_add f_int f_sqr`
`fff_sub is f_sqr-fff_add;`	`fff_sub fff_sub f_sqr fff_add`
`fff_mult is fff_add*fff_sub;`	`fff_mult fff_mult fff_add fff_sub`
`fff_div is fff_sub/fff_mult;`	`fff_div fff_div fff_sub fff_mult`
`i_flt is int(fff_div);`	`i_flt i_flt fff_div`
`bii_eq is iii_div==i_flt;`	`bii_eq bii_eq iii_div i_flt`

Listing 3.9: "Numeric" Eden script and DAMscript

- !Donald draws the complete new state after every transition, as all the graphical actions in the currently active viewport are invoked. In comparison, `tkeden` communicates only the changed geometry to Tcl/Tk, which is then responsible for moving the current display into the new state by un-drawing changed old state and drawing the new state.

Cartwright points out that the "shapes" script is likely to have a higher trigonometry calculation : line drawing ratio than the "engine" script. The lower improvement ratio that the DAM machine shows for the "shapes" script, both over `tkeden` and in single-tasking mode improvement over GUI mode, may be due to the larger amount of basic value calculation during update. To test this, this author constructed the "numeric" Eden script and DAMscript shown in Listing 3.9.

The two equivalent scripts calculate numeric values only: there are no graphical actions. Each script combines some operators into a graph structure. The same structure is built using integer operators and also using floating point operators, the two structures being linked by the single input and single output which gives a boolean result. The structure of the script is illustrated in Figure 3.17.

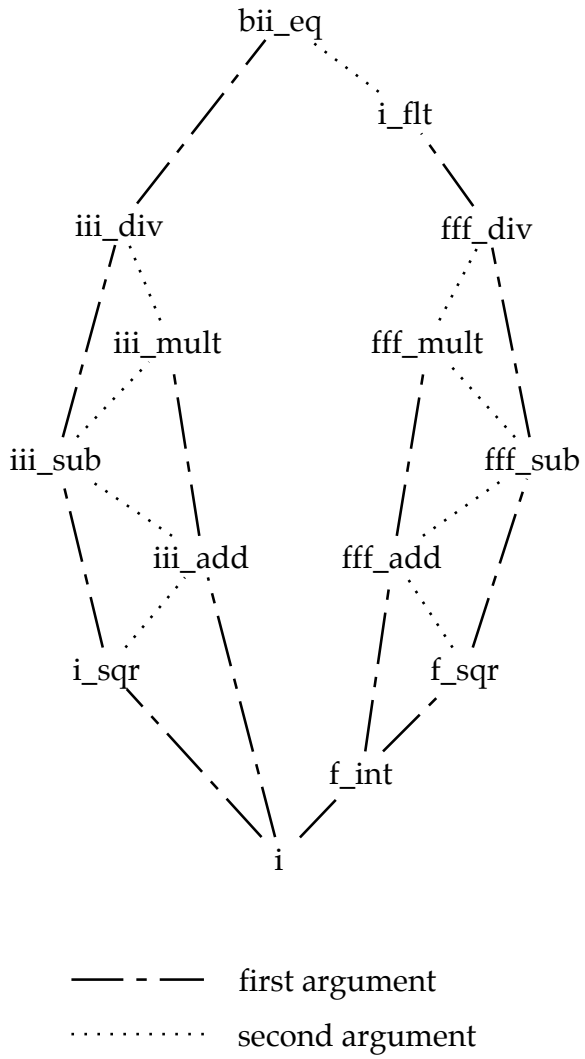


Figure 3.17: Structure of the "numeric" script

Script	`tkeden` on the Acorn under ARM Linux	!Donald2 on the Acorn under RISC OS, single-tasking mode
numeric	29 secs	26 secs (x 1.1 over `tkeden`)

Table 3.7: `tkeden` and !Donald2 performance when running the "numeric" non-graphical script (Listing 3.9)

The variable `i` was changed from 1 to 10240 and the total time taken to complete the move through all the intermediate states was recorded. The performance of the two definition maintainers is then very similar, and is shown in Table 3.7.

The performance advantage that the DAM machine appears to give may therefore be almost entirely due to efficient graphical routine implementations in the RISC OS operating system. (The slightly degraded performance of `tkeden` here may be due to kernel overheads imposed in a pre-emptive UNIX process scheduler, compared to the RISC OS cooperative scheduling system, where there are no other tasks running simultaneously with !Donald2 running in single-tasking mode.)

The major performance feature of the DAM machine — the BRA facility to optimise the update resulting from a block of multiple redefinitions — is not measured in the above tests, as only one variable was repeatedly changed. It is in fact rather difficult to test this facility in the current versions of !Donald as the user interfaces do not permit more than one redefinition to be made simultaneously. Chapter 4 contains an evaluation of EDEN in the light of the theoretical advantages of the BRA. It is found that when Eden's `autocalc` facility is used to demarcate blocks of redefinitions, the total number of evaluations performed is the same as that that the BRA would produce.

In definitive systems, as in many other areas of computing, then, performance comparisons are difficult to make. Performance comparisons in non-definitive systems are often made with respect to a "benchmark", which contains a particular mix of instructions relevant to some domain. Similarly, the evaluations above show that the performance of a definitive system is largely dependent upon the mix of operators used in the script. !Donald performs well if the script has a large proportion of graphical actions (in keeping with its principal function) but it gives no significant advantage over `tkeden` when the script involves only floating point and integer values with no visual representation. As there is no such thing as the average program, there is no such thing as the average script, making any attempt at a general performance comparison impossible.

3.5 Geometry of definitions in the DAM machine store

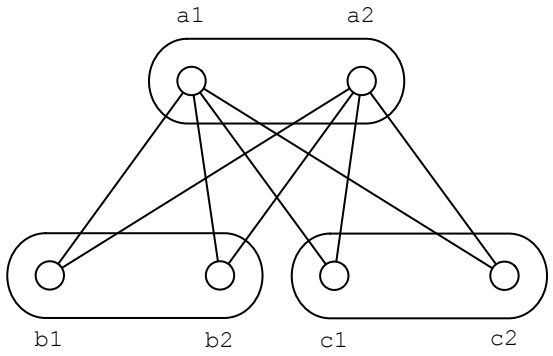
Cartwright's DMM and DAM machine design both abstract away the actual *location* of maintained machine words. In !Donald, this issue is addressed in the symbol table implementation, which is implemented as a part of !Donald using a mix of BASIC and ARM assembler. We have seen in §3.4.3 that the indirection introduced by a symbol table introduces problems of higher-order dependency. This section shows:

- that the abstraction causes problems when using dependency on data items whose representation is larger than one word or of variable size;
- how the geometry of definitions in store can be important when dependency is applied at a finely granular level for visualisation;
- how the abstraction moves definitive scripts away from the spatial aspects of spreadsheets, highlighting respects in which it is inappropriate to view definitive scripts as "generalised spreadsheets" (a description given, for example, in [Bey90]);
- and how the video hardware subsystem in the Acorn platform can be considered to be performing definition maintenance as a part of the DAM machine.

3.5.1 Dependency between multi-word data types

The DAM machine maintains dependencies between single words in store only. Provided that every data type used can be completely represented in a 32-bit word, this at first seems adequate. The conventional thinking would be that any problems that arise from this limitation can be dealt with at higher levels in the architecture. For example, the DAM machine provides no facilities for identifying the data type of a word, so careful use of operators must be made to ensure that the correct operators are used on the correct data type. It seems that this can be done adequately well at a higher level in most applications.

When a multi-word data representation is to be used, Cartwright suggests creating an operator for each 32-bit word component of the result. Figure 3.18 illustrates




```
a is b+c

a1 is double_addL(b1,b2,c1,c2)
a2 is double_addR(b1,b2,c1,c2)
```

Figure 3.18: DAM machine operator configuration for summation with double length data types

the idea. The figure is based on [Car99, p.158][34]. In Figure 3.18, `a`, `b` and `c` are 64-bit values, which are each split into two 32-bit word components. Splitting multi-word operators into multiple 32-bit operators increases the number of operators required and the number of nodes and arcs in the script graph. Also notice that each `double_add` operator requires all four arguments, due to the potential of a carry bit from the least significant word during the addition. Much of the addition operation will thus have to be performed twice, reducing performance.

3.5.2 Dependency on variable length data

Notice that the !Donald string data type is not represented within the current DAM machine store: the store contains only a 32-bit pointer to the string, which is stored elsewhere, as illustrated by '*intervar9*' and '*intervar15*' in the previous Figure 3.10 (p.136 — a graphical representation of the Listing 3.3 DAMscript on p.133, itself a result of translation from the Listing 3.2 DoNaLD script).

The splitting-operator solution can only be applied to fixed length data types. If the variables `b` and `c` in Figure 3.18 were variable length data types, the dependency links between them and the variable `a` would need to be modified when their length changed. The operators involved would also need to be changed to cope with the varied number of arguments.

[34]But with identifiers changed to make clearer that `a1` and `a2` are components of `a` and with the corresponding script included.

```
.lookup                 ; On entry: R0 = base address of value (p)
                        ;           R1 = offset of value (q)
                        ; On exit:  R0 = value at p+q
    LDR R3, [R1]        ; Load array offset value (q) into R3
    MUL R3, R3, #4      ; Calculate word address offset (4q)
    LDR R2, [R0]        ; Load array base value (p) into R2
    LDR R0, [R2, R3]    ; Load R0 with value stored at p+q
    MOV PC, R14         ; Return control to the DAM machine
```

Listing 3.10: A DAM machine 'lookup' operator (from [Car99, p.159])

For arrays, Cartwright suggests the use of a `lookup` operator in order to find the `q`th element in an array whose base address is `p`, giving the code shown in Listing 3.10 (pre- and post-requisites added by this author).

Unfortunately this solution has the same problem discussed in the context of the `DEREF()` and `if` operators at the end of section §3.4.3. Suppose that a definition `val` is created which uses Cartwright's `lookup` operator in the following hypothetical DAMscript:

```
seti arraybase &27D34[35]
seti offset 3
lookup val arraybase offset
```

Initially, `val` will have the expected value, taken from the memory location 27D34 (hex) + 4*3, but when the data at that location changes, the value of `val` will not be automatically updated. This operator and its use within the DAM machine does not capture all the dependency present in this situation.

One solution to this problem would be to explicitly create all the necessary dependency. With an imagined extension to DAMscript, using the `!` character to denote explicit reference to memory locations rather than the symbol table, an

[35]The & prefix denotes a hexadecimal value in BBC BASIC. The example address used here is hypothetical — it would be difficult to determine the necessary addresses in the current !Donald2 application.

example might look like the following:

```
seti arraybase &27D34
seti offset 3
lookup val arraybase offset !&27D34 !&27D38 !&27D3C !&27D40
```

Here, the elements of a four word array have been added as explicit arguments to the `lookup` operator, ensuring that changes to data will cause `val` to be recalculated. However, the total number of arguments is limited to 10 in the DAM machine (since operator arguments are passed using processor registers), so this is not a general solution. Even if more arguments were possible, this solution creates a large amount of data in the Sources Store for `val`, and the same amount of data (but spread across the entire list) in the Targets Store.

Also, the solution is now over-prescribing the dependency present in the situation. If the second element in the array (at location &27D38) is changed, `val` will be recalculated, although it is currently referring to the third element. The value of `val` is actually dependent only on the value at location (`arraybase + offset`). We have once again encountered the Higher Order Definition problem of the script graph changing when literal data changes.

One of the causes of this problem is the fact that the DAM machine was designed to maintain dependencies within a *set* of values. The set is ordered only by the script graph, which represents the dependency relationships over a set of value identities. The value identities themselves are not themselves comparable. This corresponds to the mathematical graph abstraction: only the set of nodes and the set of edges between nodes is relevant — the geometrical positions of a node (and hence the graphical layout of the graph) is irrelevant. In the DAM machine, the choice of where in store to place a word has no discernible effect on the working of the machine. Similarly, in a definitive script, changing the symbolic name of a definition and all references to that definition has no effect.

Within present EM tools, the problem of dependency maintenance has been abstracted away from the geometrical layout of the store. This abstraction can be considered a positive feature, as it represents a separation of concerns [Dij76, p.211], but with the abstraction come limitations of the sort described with the `lookup` operator. Arrays and lists imply geometrical relationships between data elements — if there is no geometry, then arrays and lists are difficult to implement

(e.g. linked lists must be employed). The abstraction has therefore impoverished the kinds of reference that can be employed. The contrast with dependency maintenance in spreadsheet tools is instructive in this context. The grid layout employed in spreadsheet tools does not just enable a spatially grounded user interface — the tools also exploit the geometrical relationships between cells to provide various forms of local referencing. A cell can reference "the cell above this one", and the cell can then be replicated (copied with formulae substitution) to another location, preserving the local reference.

3.5.3 Visualising the store

Despite the abstraction, the store in !Donald2 of course exists in a particular region of memory and is ordered (as is all other data in the machine) by address. As an experiment, I have added a facility for visualising the store directly[36] to !Donald2. This was done by allocating an extra RAM screen buffer and locating the DAM machine store within it. When running in single-tasking mode, a key can be pressed, causing the video hardware to be reconfigured to display the DAM store screen buffer. The results for the parabola script (Listing 3.5, p.144), which contains 38 definitions and the larger engine DoNaLD script (shown running in Figure 3.6, p.126), which contains approximately 800 definitions, are shown in Figures 3.19 and 3.20 respectively. (Note that the screen shots are colour inverted for this thesis as black-on-white prints more acceptably.)

In the video mode used for the visualisation, the screen is 640x480 pixels with a bit depth of 1 (i.e. each pixel can be either black or white). The screen is addressed starting at the top and moving along each row from left to right (corresponding to the raster scan of the hardware). Each DAM machine word is 32 bits and corresponds to 32 laterally adjacent pixels, least significant bit left-most. Each screen row shows 20 words. The screen has 480 rows, showing 9600 words in total. The top 96 rows show the 1920 maintained DAM word values. The remaining 384 rows (4/5ths of the total) show the Knuth Counter, Function, Target and Source Pointers respectively.

[36] [Car99, §5.5] claims to have achieved something similar, but the effect is actually achieved indirectly: after the BRA update has completed, some code (written in BASIC) copies a portion of the DAM store state onto the screen, making the appropriate geometrical mapping to pixel data as it goes.

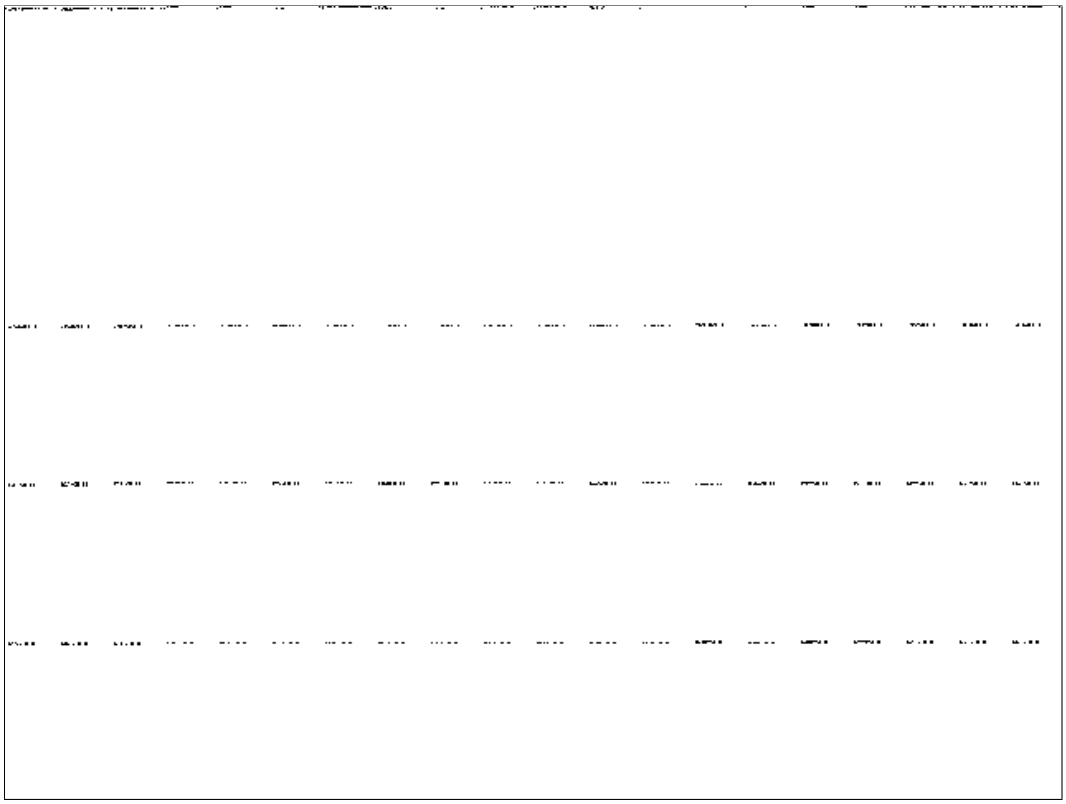


Figure 3.19: Visualisation of the “parabola” DAMscript DAM store

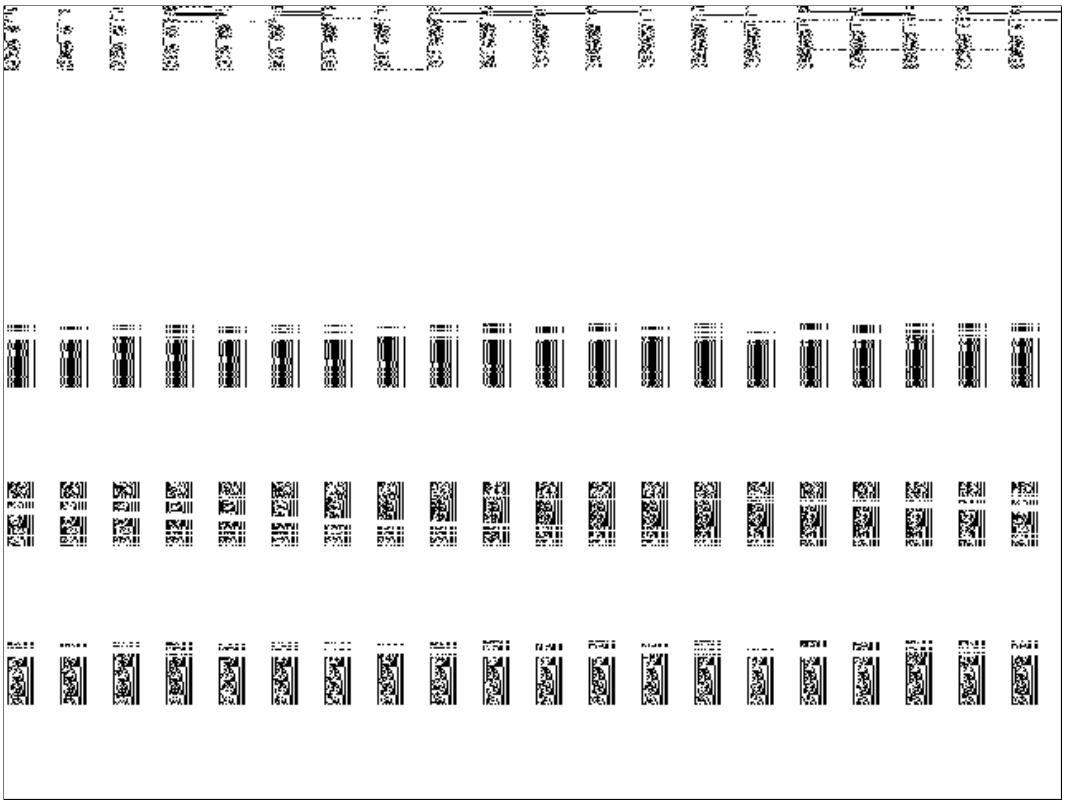


Figure 3.20: Visualisation of the “engine” DoNaLD script DAM store

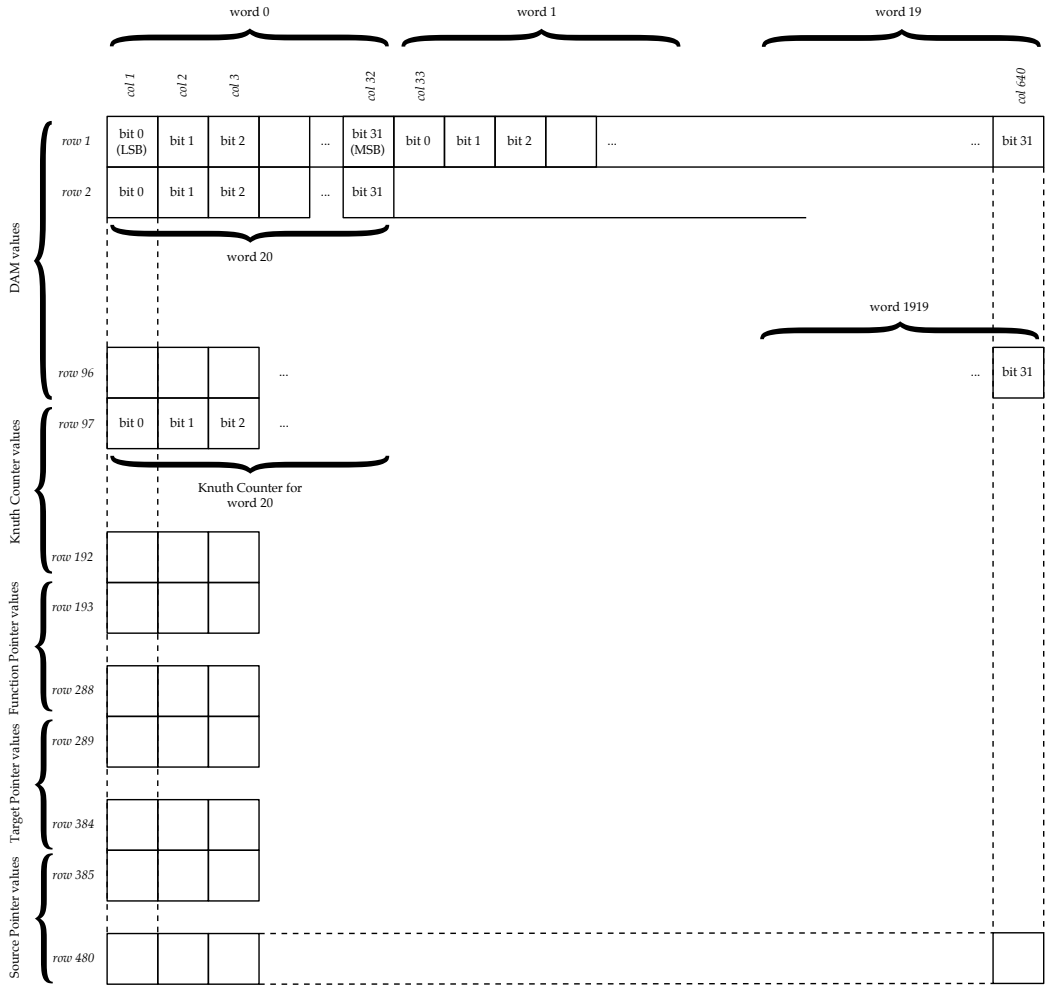


Figure 3.21: DAM store mapping to pixels

The pixel display therefore corresponds to a geometrical representation of some of the DAM machine data structure (which was shown in Figure 3.2 on p.115). The geometrical layout of the pixels is shown in Figure 3.21.

The “parabola” and “engine” examples shown use mostly byte size types, so only the 8 low order bits in each 32-bit word are used. This explains the large amounts of white space between each word in the Figures 3.19 and 3.20.

As the values in the script change in response to a single change entered in DAMscript or a sequence of automated changes from the Update Variable window, the pixels on the screen that show targets of the change can be observed to change. The Knuth Counter values are not visible in the screenshots, as the counters take

non-zero values only when the `update` routine is running, and whilst the routine is running it is not possible to capture a screenshot. However, the values are visible on screen whilst a `update` is in progress.

We can therefore consider the video hardware to be establishing a visual dependency for us. The brightness of the top left pixel in the display *is* the status of the least significant bit in the first DAM machine word. Due to this combination of video hardware and software definition maintainer, it is possible to answer the question "why is that pixel white?" without reference to past history of sequential code execution — it is white because it is defined to be that way, and the dependency structure can be traced to determine the values that affect the pixel brightness.

3.5.4 Exploiting the visualisation

This final section shows how control over the geometrical position of a given definition was obtained in !Donald2 and how this was then exploited to produce a pixelated display with meaningful state.

!Donald2 does (of course) deterministically map words to particular locations. The order that commands are given in a DAMscript file has an effect on the mapping to locations. As the DAMscript file is interpreted in two passes, the mapping is rather complex. DAMscript definitions and values are mapped to locations starting from the smallest numbered location in the DAM machine store, in the following order.

1. The default viewport variable, the x offset and the y offset (3 words in total) are created.

2. 'set' values, in the order presented, are mapped.

3. Dependencies, and a variable for each `viewport` command, are mapped, in the order presented. For a definition *operator_name variable dependent_list*:

   (a) First the references in *dependent_list* (left to right) are mapped, unless already defined,

   (b) Then the *variable* is mapped.

After having analysed this mapping, I exploited it in a rudimentary way. The "text" DAMscript shown in Listing 3.11 is ordered such that the loading process maps the values to DAM machine store in the same order that they are shown in the script — i.e. the mapping transformation described above has no effect. Listing 3.11 is an excerpt from the full listing, showing the first three rows only.

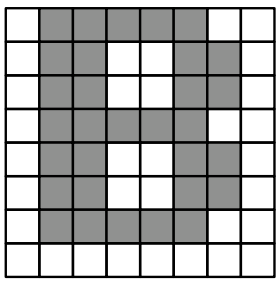
```
# text DAMscript example
# Ashley Ward (ashley@dcs.warwick.ac.uk) December 2003

# This script is ordered in the same way that it will be loaded
# and hence geometrically row 1 first, L to R, then row 2...

# -- ROW 1 follows ------------------------------------------

# col 01: taken by default viewport variable
# col 02: taken by viewport x offset
# col 03: taken by viewport y offset

# col 04: character code 1
seti cc1 65
# col 05: character code 2
seti cc2 115
# col 06: character code 3
seti cc3 104

# col 07: literal
seti zero 0
# col 08: literal
seti one 1
# col 09: literal
seti two 2
# col 10: literal
seti three 3
# col 11: literal
seti four 4
# col 12: literal
seti five 5
# col 13: literal
seti six 6
# col 14: literal
seti seven 7
# col 15: literal
seti eight 8
# col 16: pointer to name of BASIC function used
setc chargraphic "chargraphic"
# col 17: value shown in unused locations
seti blank 255
# col 18: fill to end of row
seti r1c18 0
# col 19: fill to end of row
seti r1c19 0
# col 20: fill to end of row
seti r1c20 0
```

```
48
49  # -- ROW 2 follows ------------------------------------------
50
51  # col 01: dummy blank
52  iii_add r2c01 zero blank
53  # col 02: dummy blank
54  iii_add r2c02 zero blank
55
56  # col 03: row number
57  iii_add rn1 zero one
58  # col 04: character graphic 1
59  fn r2cg1 chargraphic cc1 rn1
60  # col 05: character graphic 2
61  fn r2cg2 chargraphic cc2 rn1
62  # col 06: character graphic 3
63  fn r2cg3 chargraphic cc3 rn1
64
65  # col 07..20: dummy blank
66  iii_add r2c07 zero blank
67  iii_add r2c08 zero blank
68  iii_add r2c09 zero blank
69  iii_add r2c10 zero blank
70  iii_add r2c11 zero blank
71  iii_add r2c12 zero blank
72  iii_add r2c13 zero blank
73  iii_add r2c14 zero blank
74  iii_add r2c15 zero blank
75  iii_add r2c16 zero blank
76  iii_add r2c17 zero blank
77  iii_add r2c18 zero blank
78  iii_add r2c19 zero blank
79  iii_add r2c20 zero blank
80
81  # -- ROW 3 follows ------------------------------------------
82
83  # col 01: dummy blank
84  iii_add r3c01 zero blank
85  # col 02: dummy blank
86  iii_add r3c02 zero blank
87
88  # col 03: row number
89  iii_add rn2 zero two
90  # col 04: character graphic 1
91  fn r3cg1 chargraphic cc1 rn2
92  # col 05: character graphic 2
93  fn r3cg2 chargraphic cc2 rn2
94  # col 06: character graphic 3
95  fn r3cg3 chargraphic cc3 rn2
96
97  # col 07..20: dummy blank
98  iii_add r3c07 zero blank
99  iii_add r3c08 zero blank
100 iii_add r3c09 zero blank
101 # (etc) ...
```

Listing 3.11: “text” DAMscript

```
REM fn resvar "chargraphic" charcode rowno
REM Find the character graphic (one row only)
REM for a particular character code
DEF FNchargraphic(charcode%, rowno%, d3%, d4%, d5%, d6%)
  DIM chargraphicblock% 8
  chargraphicblock%?0 = charcode%
  SYS "OS_Word", 10, chargraphicblock%
= chargraphicblock%?rowno%
```

Listing 3.12: `chargraphic` BASIC function used by "text" DAMscript (Listing 3.11)



chargraphic(66,1) = %01111100 = 124 decimal
chargraphic(66,2) = %01100110 = 102 decimal
chargraphic(66,3) = %01100110 = 102 decimal
chargraphic(66,4) = %01111100 = 124 decimal
chargraphic(66,5) = %01100100 = 102 decimal
chargraphic(66,6) = %01100100 = 102 decimal
chargraphic(66,7) = %01111100 = 124 decimal
chargraphic(66,8) = %00000000 = 0 decimal

(ASCII character code 66 = character 'B')

Figure 3.22: Lookup of an ASCII character code glyph

The DAMscript creates three character patterns on the screen. The pixels showing the character patterns are dependencies of the form:

```
fn r2cg1 chargraphic cc1 rn1
```

i.e. a variable named `r2cg1` which uses the BASIC function `chargraphic` on the arguments `cc1` and `rn1`. The `chargraphic` BASIC function is shown in Listing 3.12. The function takes an ASCII character code and a row number (from 1 to 8) as input. It queries the 8x8 pixel character glyph graphic maps maintained by the operating system and returns the row of pixels for the given character and row as a byte-wide result, as illustrated in Figure 3.22.

The character pattern dependencies are shown in Listing 3.11 on lines 59–63 and 91–95. There are six other similar regions of definitions (making up the six rows of pixels not described in Listing 3.11) in the full listing. The character pattern dependencies have as sources the word at top row of their column and the word in col 3 of their row, holding the ASCII value and the row number (lines 14–18 and

lines 57, 89 and six other similar definitions respectively).

The screen pixels are laid out on the screen in a grid form, and so the dependency present in the script resembles a spreadsheet. Correspondingly, Figure 3.23 shows a spreadsheet with cell formulae set to similar dependencies as created in the "text" DAMscript.

The location transformation during the loading process forces definitions representing literal values to the earlier words in the DAM store. In this script, the first row is formed entirely from literal values. Hence, the row numbers (which are dispersed among the later dependencies) are formed by simple dependencies which add zero to the desired value. This is illustrated in the spreadsheet shown in Figure 3.23. Words that are unused, but which must be defined in order to coerce the loading process into creating the character patterns in the correct positions, are defined as equal to the "blank" value at row 1, col 17 (line 41 in Listing 3.11). The spreadsheet shown in Figure 3.23 does not show these definitions.

When the DAMscript is loaded into !Donald2 and the store visualisation is turned on, the resulting display is as shown in the lower part of Figure 3.24. The upper part of the figure shows a magnified portion of the upper left corner of the screen. The three character patterns for the letters `A`, `s` and `h` (ASCII 65, 115 and 104) are clearly visible at the top left. The patterns are laterally reversed, due to a mismatch between the output of the `chargraphic` function and the memory to pixel mapping described in Figure 3.21. The words showing the ASCII codes are also discernible at the top of each character pattern column, as are the row numbers to the left of the `A` pattern. The "blank" value is set to 255, creating the solid eight-pixel wide blocks at unused positions.

"Why is that pixel white?" is now a question whose answer can be traced from pixel location through to definition and determining sources. Changing the ASCII values or the row numbering changes the patterns shown on the screen by dependency.

The "text" DAMscript and !Donald2 configuration of video hardware is the first step towards an entirely definitive text editor (compare Y.W. Yung's first Eden text editor model described in §4.1.3), representing the output side of such an implementation. !Donald2 does not yet provide definitive *input* facilities, but this is

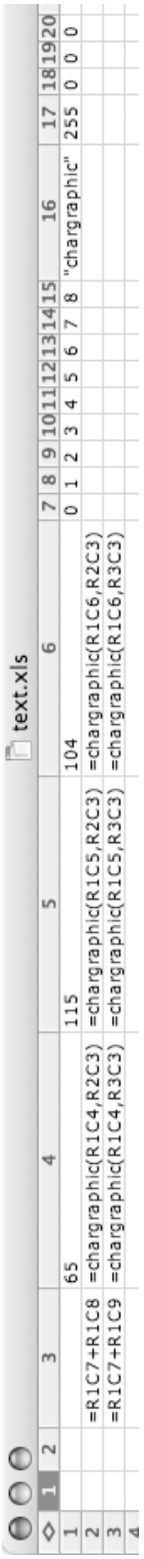


Figure 3.23: Spreadsheet form of “text” DAMscript

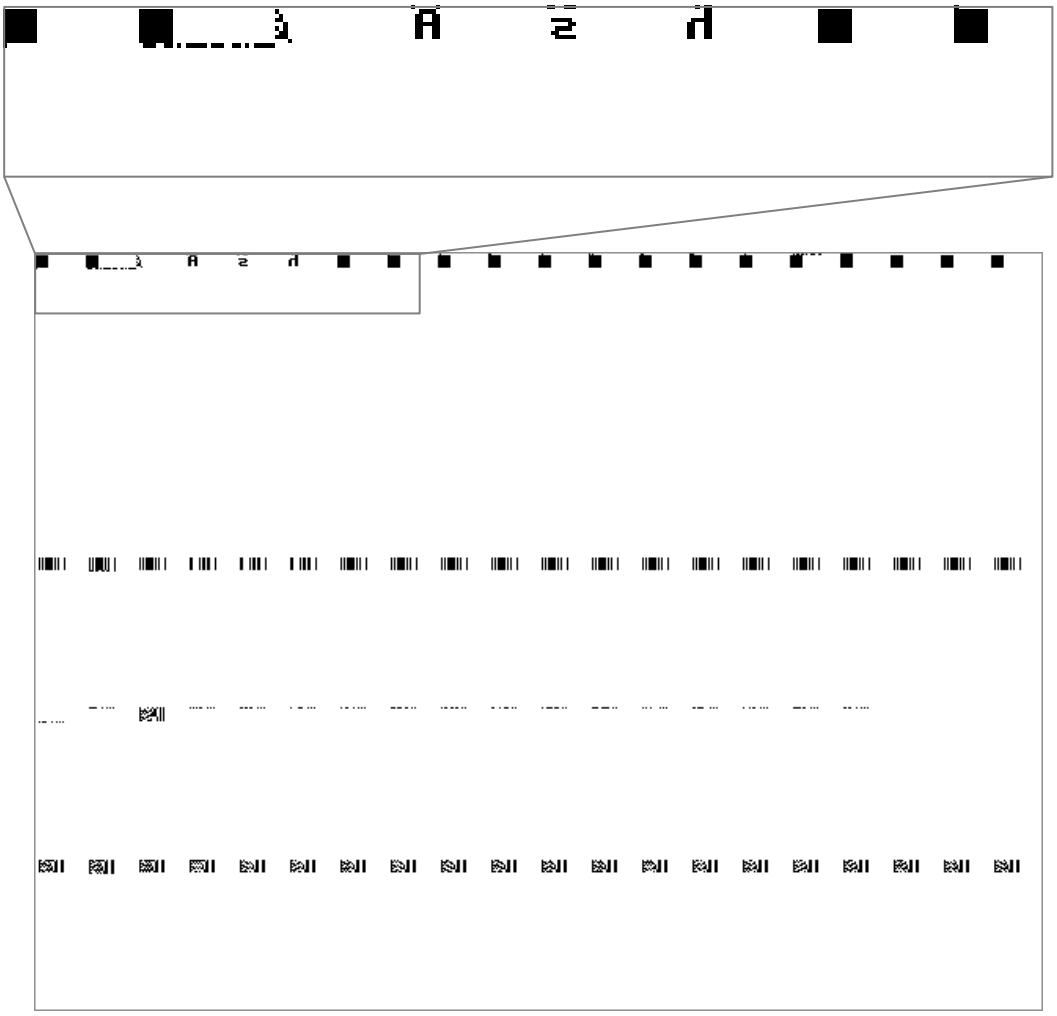


Figure 3.24: Visualisation of the “text” DoNaLD script DAM store

possible. If this were implemented, the determining sources of a pixel at the current cursor position might include the keyboard input. The operation of the machine could then be construed in a similar way to a mechanical typewriter and be analysed along similar lines, with definitions providing hard guarantees of cause and effect, like levers in the mechanical counterpart.

A full screen representation based on the principles illustrated would require several thousand[37] dependencies operating at a finely granular level. Exploiting dependency in this way is made more difficult by abstracting away the geometrical layout of store, so that identities cannot be compared with one another. The two-dimensional geometrical representation is appropriate for the screen display, but other geometrical representations might be more appropriate for other applications. In principle, different geometric representations might be used to organise different regions within the same machine store.

[37] 9600 to be precise — see p.166.

3.A The Script Digraph

Dependency describes a relationship between the variables in a definitive script. This section links this notion to basic graph theory, describing the structure I call a *script digraph*, or more informally, a *script graph*.

We can draw a diagram representing the dependency in a definitive script. Conventionally, we draw a graph structure. First, we represent each variable (definition or literal value) by a diagram point, or *node*. Secondly, we draw a line to show each use of a variable value. A line is drawn between a definition (say `d`) and a variable (say `v`) whose value is used by that definition. Because a change to the value of `v` implies that the value of `d` will change, we indicate a direction on the line, making it an arrow, or *arc*. This makes the whole structure a *directed graph*. Conventionally, we draw the direction of the arrow to show the *effect* of change: i.e. from `v` to `d`. Interpreted in the reverse direction, the arc shows a possible *cause* of change.

A directed line is required as there is no symmetry in the relationship between `d` and `v`: although a change to `v` implies a change to `d`, a change to `d` does not imply a change to `v`. This distinguishes dependency from traditional constraints.

An arc is drawn from every use of a variable to the definition that uses it. If a definition uses the same value twice, we only draw one arc — for example, in the case:

```
d is v+v
```

This is because conventionally we are only interested in the patterns of cause and effect and multiple uses of a value by one definition are not significant.

The directions of the arcs in the graph unambiguously describe cause and effect. If however the graph contains a loop, then the unambiguous nature is lost. We do not allow definition structures with cycles, the simplest of which is a directly self-referential definition, as shown in the upper part of Figure 3.25. Indirect self-reference is also prohibited. An example of indirect self-reference is illustrated in the lower part of Figure 3.25, where `a` is defined in terms of `b` and `b` is defined in terms of `a`.

Detection of graph cycles can be done in two different ways in an implementation. Either graph cycles can be prevented from occurring by making checks at

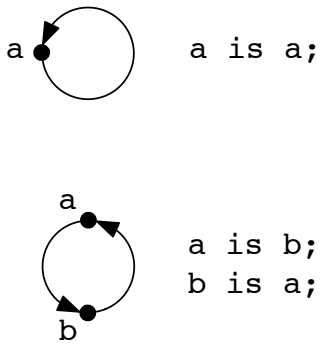


Figure 3.25: Simple graph cycles

redefinition time, or graph cycles can be allowed to occur but detected at evaluation time. The best choice depends, as does the choice of evaluation/storage strategy, upon the balance of redefinition and evaluation occurring in the application. In addition, HODs may complicate detection of graph cycles at redefinition time, since a redefinition to a literal value may cause a HOD-created graph cycle to appear.

Formally, then, the dependency within a definitive script can be described in this way using a *directed acyclic graph*. This term is sometimes shortened to *acyclic digraph* or the even more succinct *dag*.

Graph theory is a large topic which has application to many domains. Consequently there are many textbooks and a large amount of terminology. However, most texts talk of graphs generally and so are of somewhat limited use for information about directed acyclic script graphs. One notable exception is a book by Harary *et al* [HNC65], which is entirely devoted to the subject of directed graphs and has a chapter exclusively about acyclic digraphs. The remainder of this section attempts to give the minimum necessary information useful for an understanding of script graphs.

Harary *et al* [HNC65, p.267] define the ascending level assignment, which is a necessary concept for scheduling the evaluation of nodes and also for drawing a script graph diagram in a particular orientation.

> For each point [node] v_i let us denote by n_i the integer assigned to it. A digraph D has an *ascending level assignment*, and the integers are *levels*, if for each line [arc] v_iv_j of D the corresponding integers satisfy $n_i < n_j$. Thus the lines [arcs] of D are directed from lower to higher levels. Of course, if a digraph has a cycle, it has no ascending level assignment.

We shall consider only ascending level assignments and refer to them more briefly as *level assignments* in what follows. In general, there is not a unique level assignment for a given digraph, as the edges in the graph specify only a *partial ordering* on the level assignment. For most purposes we would consider level assignments with fewer levels to be "better" than those with more. The smallest number of levels possible in any level assignment is equal to the number of points on the longest directed path [HNC65, p.270].

The nodes in an acyclic digraph can be ordered by level assignment using an algorithm that performs a topological sort — Knuth's algorithm mentioned in §3.1.2 is one. However, topological sort algorithms produce only a sequential node ordering, which involves a loss of information relating to nodes that have the same level in a level assignment. On a sequentially evaluating machine this is of little significance, but a concurrent machine can potentially concurrently evaluate nodes at the same level.

Once levels are assigned, it is possible to draw the script digraph in a diagram form where the vertical geometrical relationships of the nodes are meaningful. Conventionally, a *script digraph diagram* is drawn with arcs pointing upwards, and we speak of change propagating *upwards* through the graph and demand-driven evaluation evaluating *downwards*. If the graph is drawn in such a way, then the directions of the arcs need not be shown as it is known that they point upwards.

In 1999, I wrote an Eden script (*vcgWard1999*) to interface the principal EM tool EDEN (see Chapter 4) to a "Visualizer of Compiler Graphs" tool named `xvcg` [San, San96]. Given a list of nodes and edges, `xvcg` can display a graph diagram laid out in a large number of possible ways, including a "fish-eye" layout, which compresses the layout so that a particular node and its immediate surroundings can be seen clearly, but nodes that are further away are reduced in size.

Figure 3.26 shows `xvcg` applied to the Eden *roomviewerYung1991* model. The graph contains 553 nodes and 797 arcs (which are directed in the reverse way to the normal convention). Top-most are the various Eden procedures that propagate internal changes of DoNaLD state to the interface. Near the top is the definition for the screen. At the bottom happens to be the `cart` cartesian coordinate function, which is used by many definitions.

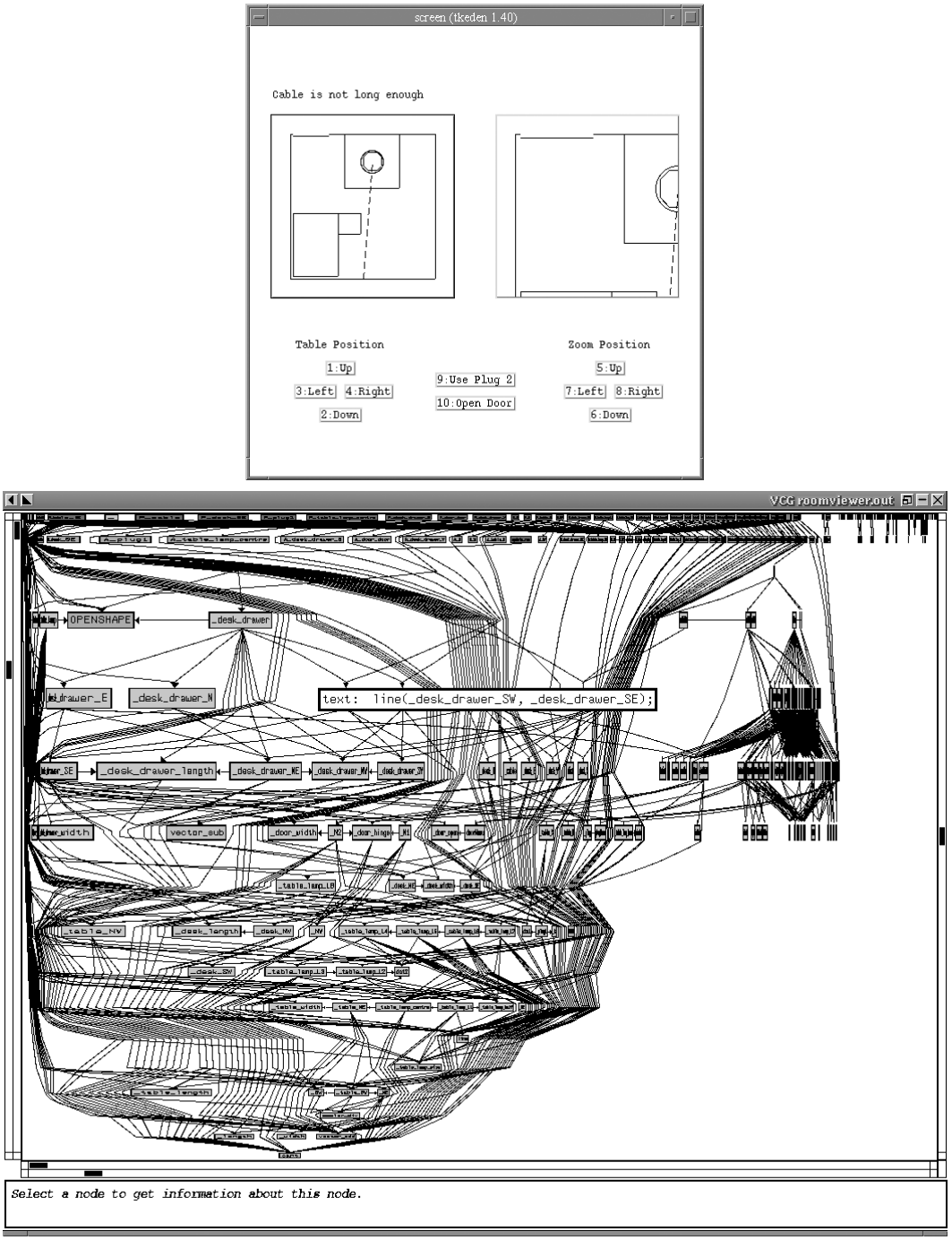


Figure 3.26: `xvcg` visualisation of *roomviewerYung1991*

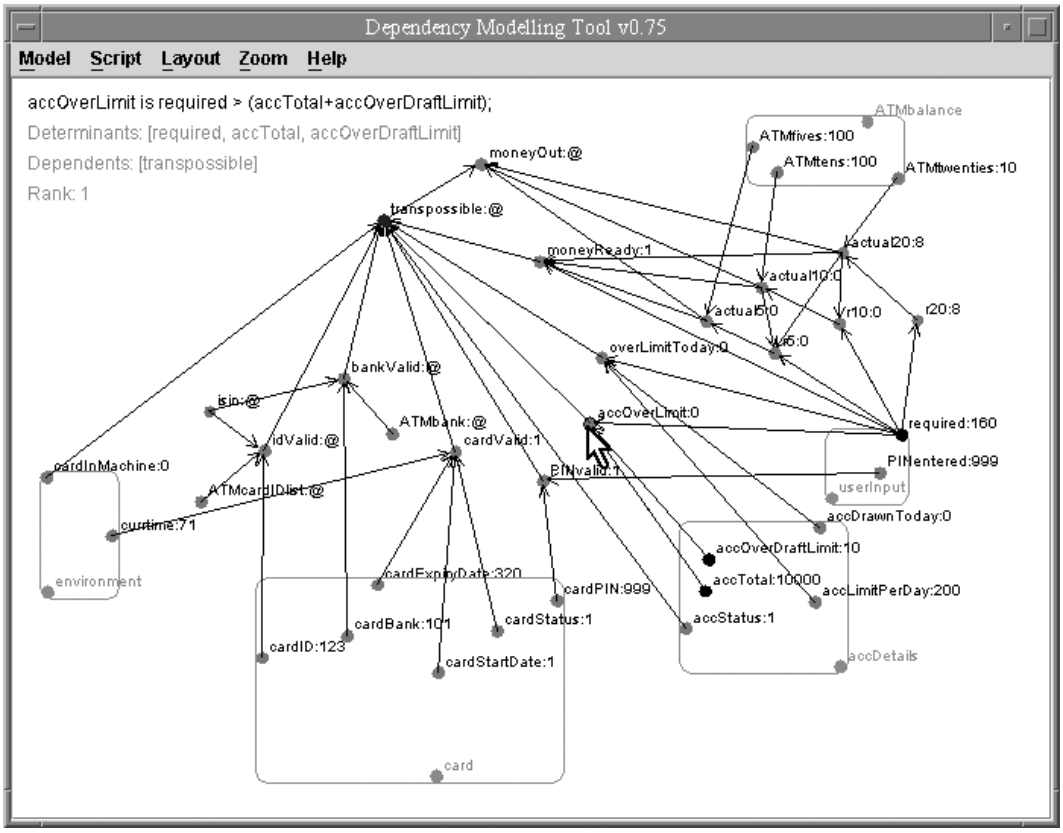


Figure 3.27: Wong's Dependency Modelling Tool, showing the ATM script

Allan Wong recently implemented a tool named "Dependency Modelling Tool" (DMT) that is based on similar principles — see [Won03, §8]. The DMT reads Eden scripts directly. Three layout algorithms are provided to assist with positioning nodes, which can also be done manually (this is not possible in `xvcg`). The tool implements many forms of interaction with the graph, indicating sources and targets (determinants and dependents in Wong's terms) of the pointed-to node with colour highlighting. It also allows edges directed toward a node to be hidden and replaced with a round-cornered rectangle in a form of "abstraction". Figure 3.27 shows the tool applied to a model of an Automated Teller Machine system. Note the "abstractions" and the use of directed edges to denote dependency.

Nodes at the bottom of a script digraph are known as *leaves*. Their values are defined entirely by references to literal values and have no dependencies on other

nodes. Leaves therefore have no incoming arcs:

$$\text{indegree(leaf)} = 0$$

Leaf nodes can be defined by using only literal values within a definition:

```
a is 42;
```

or through use of an assignment operator, or through the combination of a redefinition with an evaluate-immediately operator (if any of these facilities are provided in the notation used):

```
a := b+c;
a = | b+c |;
```

Whether these different methods of input give distinct results depends upon the implementation. There is a subtle difference in EDEN, which stores the exact text used on the RHS of a definition, but only the evaluated value from the RHS of an assignment.

Nodes at the top of the digraph are known as *roots* or *sink nodes*. They are not referenced by other nodes. Roots therefore have no outgoing arcs:

$$\text{outdegree(root)} = 0$$

Contrary to what the name may suggest, there can be more than one root in a script graph. Roots are conventionally drawn at the top of the diagram.

Nodes "inside" the digraph are *internal nodes*. They have both incoming and outgoing arcs:

$$\text{indegree(internal)} \geq 1 \wedge \text{outdegree(internal)} \geq 1$$

When a value attached to a node in the script graph or the script graph itself is changed, an internal node can be at the initial or at an intermediate point of a chain of change propagation. If an internal node is itself changed, it will be at the start of the chain. If a node downwards in the script graph is changed, then the internal nodes in the path above the change will be intermediate in the chain, in which case the nodes will be both affected by change and the cause of further change.

Nodes which are not related by dependency to any other node are said to be *isolated* [Wil96, p.12]. An isolated node constitutes a small unconnected subgraph "island" which will not participate in any change propagation:

$$\text{indegree(isolated)} = \text{outdegree(isolated)} = 0$$

We often talk of the sources, targets and triggers for a node[38].

The *adjacent sources* of a dependency are the nodes to which the dependency directly refers. The *recursive sources* of a dependency are all the nodes in the directed path below the dependency, to which the dependency directly or indirectly refers. The short-hand term *sources* refers to adjacent sources.

The *adjacent targets* of a node are the dependencies that directly refer to the node. The *recursive targets* of a node are all the dependencies in the directed path above the node, which refer directly or indirectly to the node. The short-hand term *targets* refers to adjacent targets.

The set of *triggers* of d is the set of recursive sources of d. A node is said *to trigger* d if it is one of the recursive sources of d.

Two definitive scripts have the same *dependency structure* if their script digraphs are *isomorphic* (simplistically[39], if the script digraph diagrams for each could be drawn identically).

The number of possible dependency structures grows large very quickly as the number of nodes increases. Sloane's On-Line Encyclopedia of Integer Sequences [Slo] has the necessary information as sequence A003087, citing Harary, Palmer and Robinson. Table 3.8 shows the sequence. With even just five nodes, there are 302 possible dependency structures. Beyond five nodes, we quickly obtain extremely large numbers of possibilities.

It is difficult to think about the coordination required in a concurrent definition maintainer without a pool of small example script configurations to draw on. One would sometimes like to find the smallest possible case that exhibits a particular

[38]Some authors use the terms *dependees* and *dependents/ants*, referring to sources and targets respectively, but these terms are very similar both in text and when spoken — I prefer these terms as laid out originally in [Yun90].

[39]Formally, two digraphs C and D are isomorphic if D can be obtained from C by re-labelling the nodes — that is, if there is a one-to-one correspondence between the nodes of C and those of D, such that the number of arcs joining any pair of nodes in C is equal to the number of arcs joining the corresponding pair of nodes (in the same direction) in D [WW90, p.82].

N	Acyclic digraphs
0	1
1	1
2	2
3	6
4	31
5	302
6	5984
7	243668
8	20286025
9	3424938010
10	1165948612902
11	797561675349580
12	1094026876269892596
13	3005847365735456265830

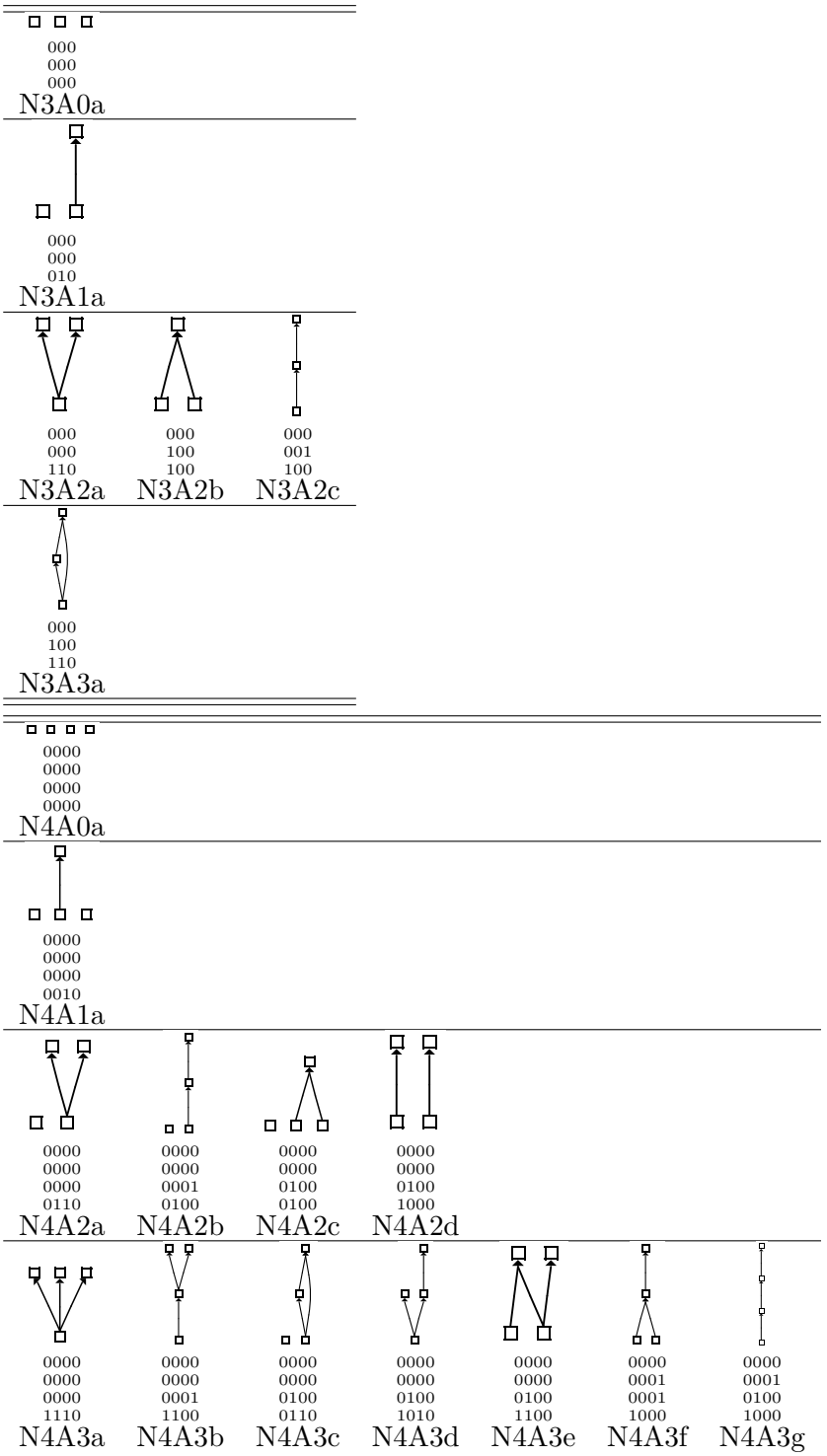
Table 3.8: Number of acyclic digraphs with N unlabelled nodes

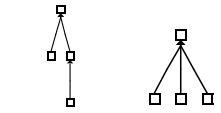
property, for example when constructing the small tests for systematic observation of EDEN presented in §4.3.3. I could not locate a source with a graphical enumeration of acyclic digraphs: [WW90, pp.19–24] contains an enumeration of the 208 unlabelled simple graphs for $N \leq 6$, but Table 3.8 shows there are many more possible acyclic directed graphs; [Car99, Figure 4.8, p.139] purports to show all possible patterns of dependencies for scripts with four definitions, but it omits many possible cases, including all cases with more than three arcs.

I abandoned a process of obtaining even the results for $N = 4$ by hand, as it was a very error-prone process, and wrote an Eden procedural program to enumerate the acyclic digraphs. The code enumerates all possible $N \times N$ adjacency matrices, treating the values in the adjacency matrix as a 1-dimensional bit vector which is assigned binary values from $00\cdots00$ to $11\cdots11$ (the bit sequence being $N \times N$ bits long). Each matrix was automatically evaluated to detect cycles and isomorphism with graphs already generated. Cycles were detected using a recursive function to follow arcs, detecting a cycle if a node was visited twice. Isomorphisms were detected by permuting the nodes of the current digraph in all possible ways, using

an algorithm based on [Sed90, p.628] and then comparing the permuted versions of the current digraph against the ones already stored. The code does not scale well: it took less than a second to run for $N = 2$ and $N = 3$, about one minute for $N = 4$ and over 14 hours for $N = 5$. Input suitable for the `xvcg` tool was then automatically generated (by Eden code), which was used to lay out and render each graph. Finally the graphs were ordered (again by Eden code) into classes by number of arcs and combined into a table which was rendered using LaTeX. The results are shown in Figure 3.28.

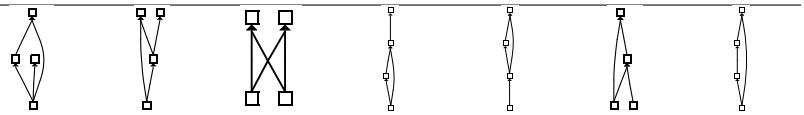
Figure 3.28: Acyclic digraphs graphically enumerated, $3 \leq N \leq 5$





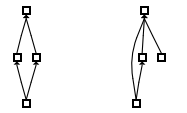
0000 0000
0001 1000
1000 1000
1000 1000

N4A3h N4A3i



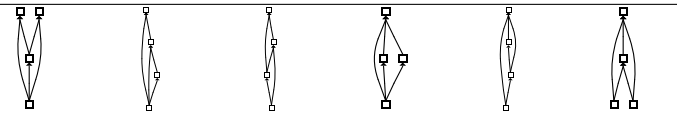
0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0001 0001 0001 0001
0100 0101 1100 0101 1000 1001 1100
1110 1100 1100 1000 1010 1000 1000

N4A4a N4A4b N4A4c N4A4d N4A4e N4A4f N4A4g



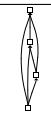
0000 0000
0011 1000
1000 1000
1000 1010

N4A4h N4A4i



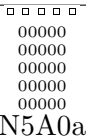
0000 0000 0000 0000 0000 0000
0000 0001 0011 1000 1000 1000
1100 1101 1010 1110 1001 1100
1110 1000 1010 1100 1100 1100

N4A5a N4A5b N4A5c N4A5d N4A5e N4A5f



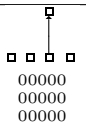
0000
1000
1100
1110

N4A6a



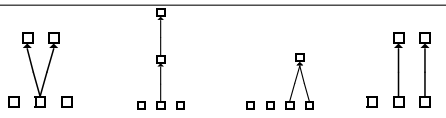
00000
00000
00000
00000
00000

N5A0a



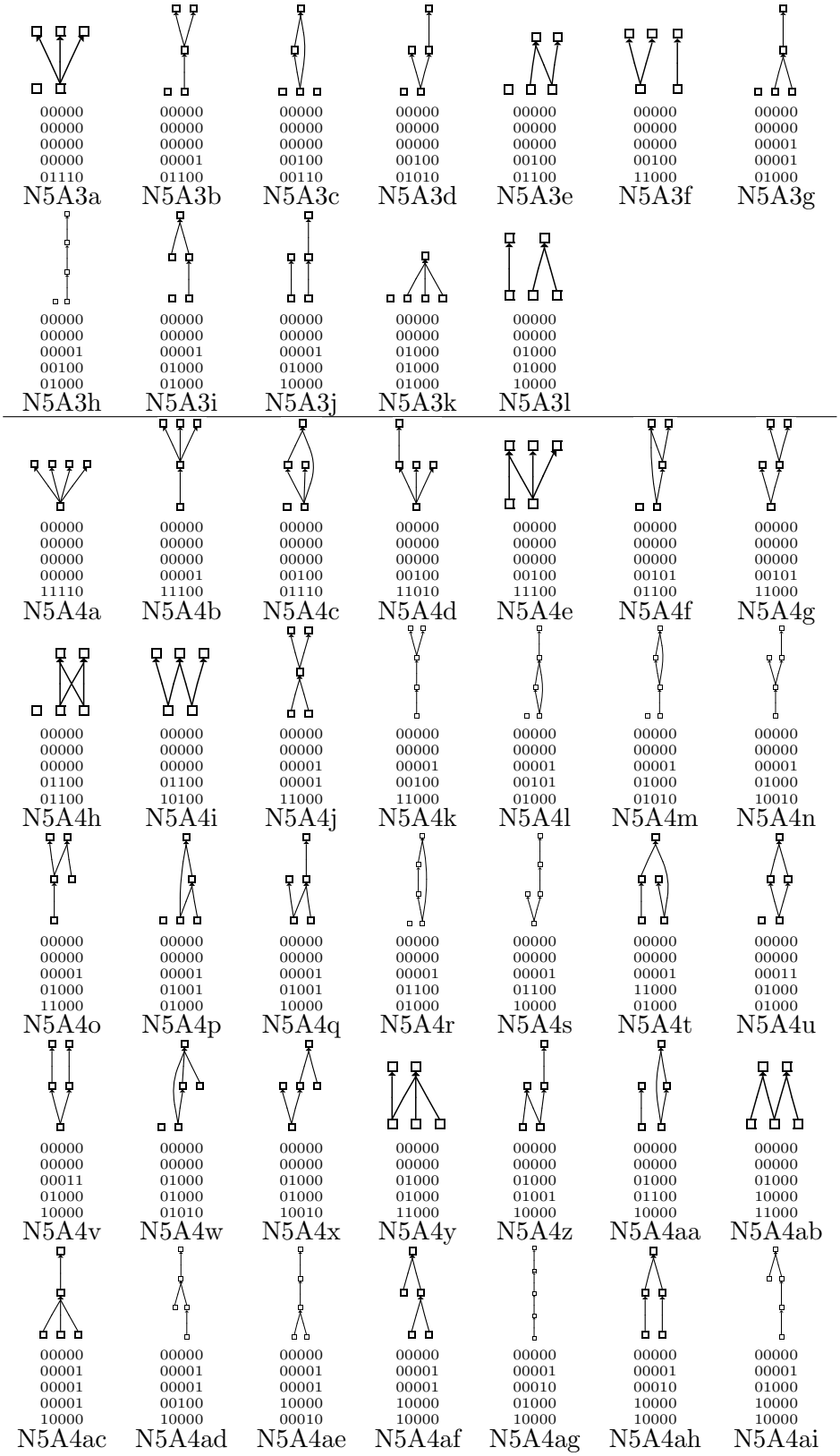
00000
00000
00000
00000
00010

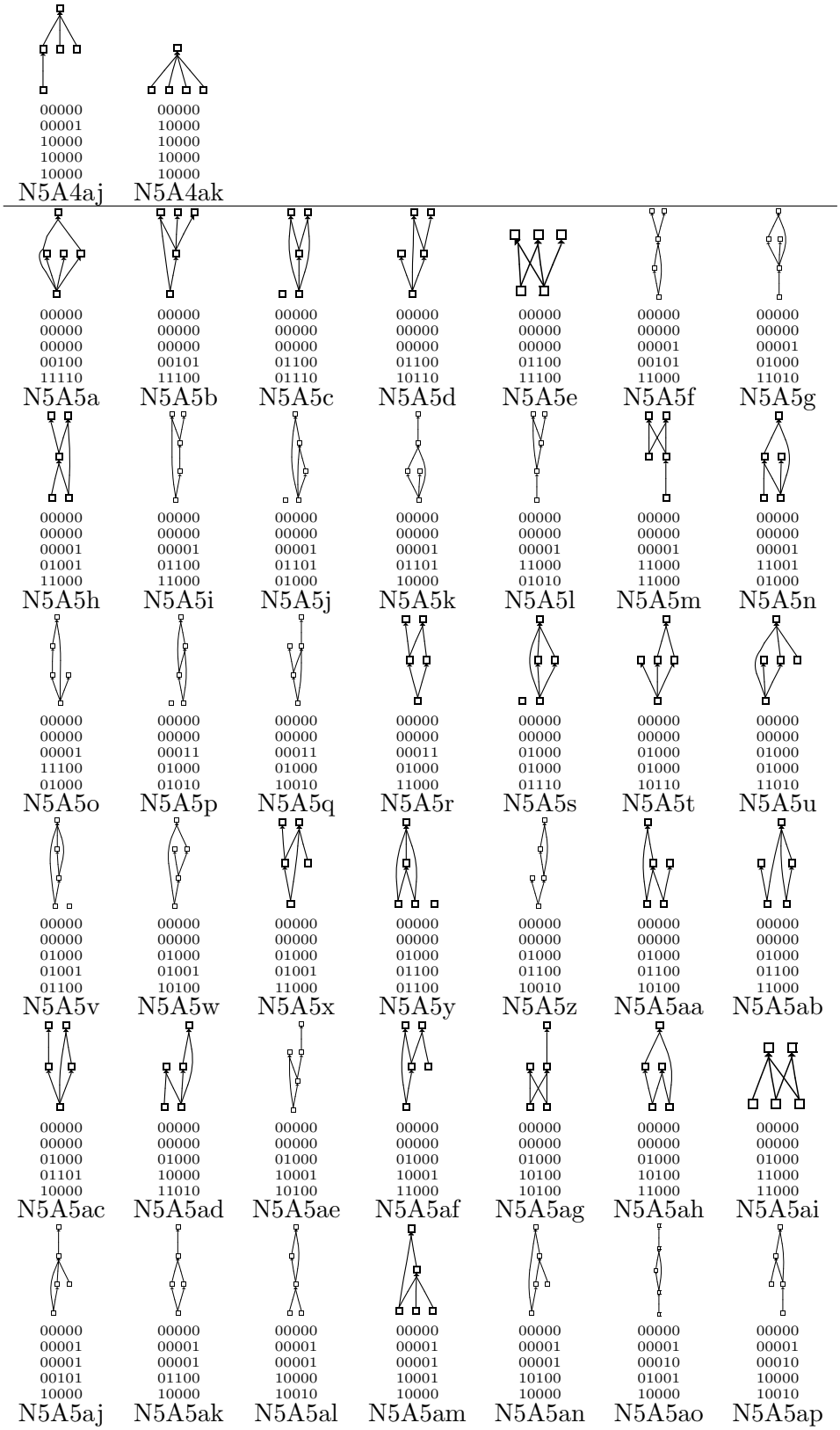
N5A1a

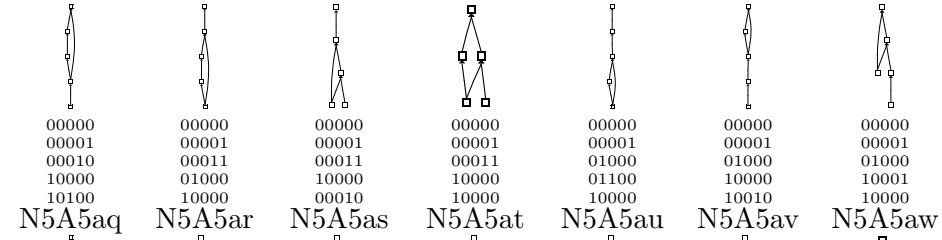


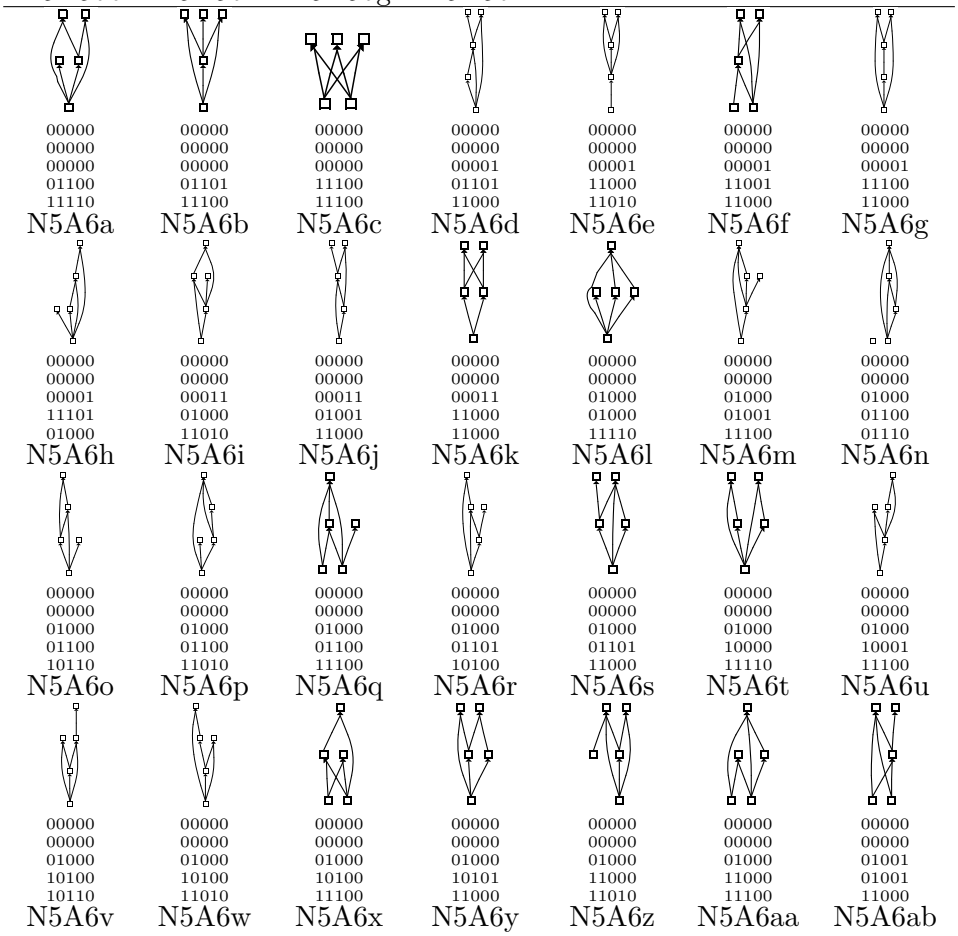
00000 00000 00000 00000
00000 00000 00000 00000
00000 00000 00000 00000
00000 00001 00100 00100
00110 00100 00100 01000

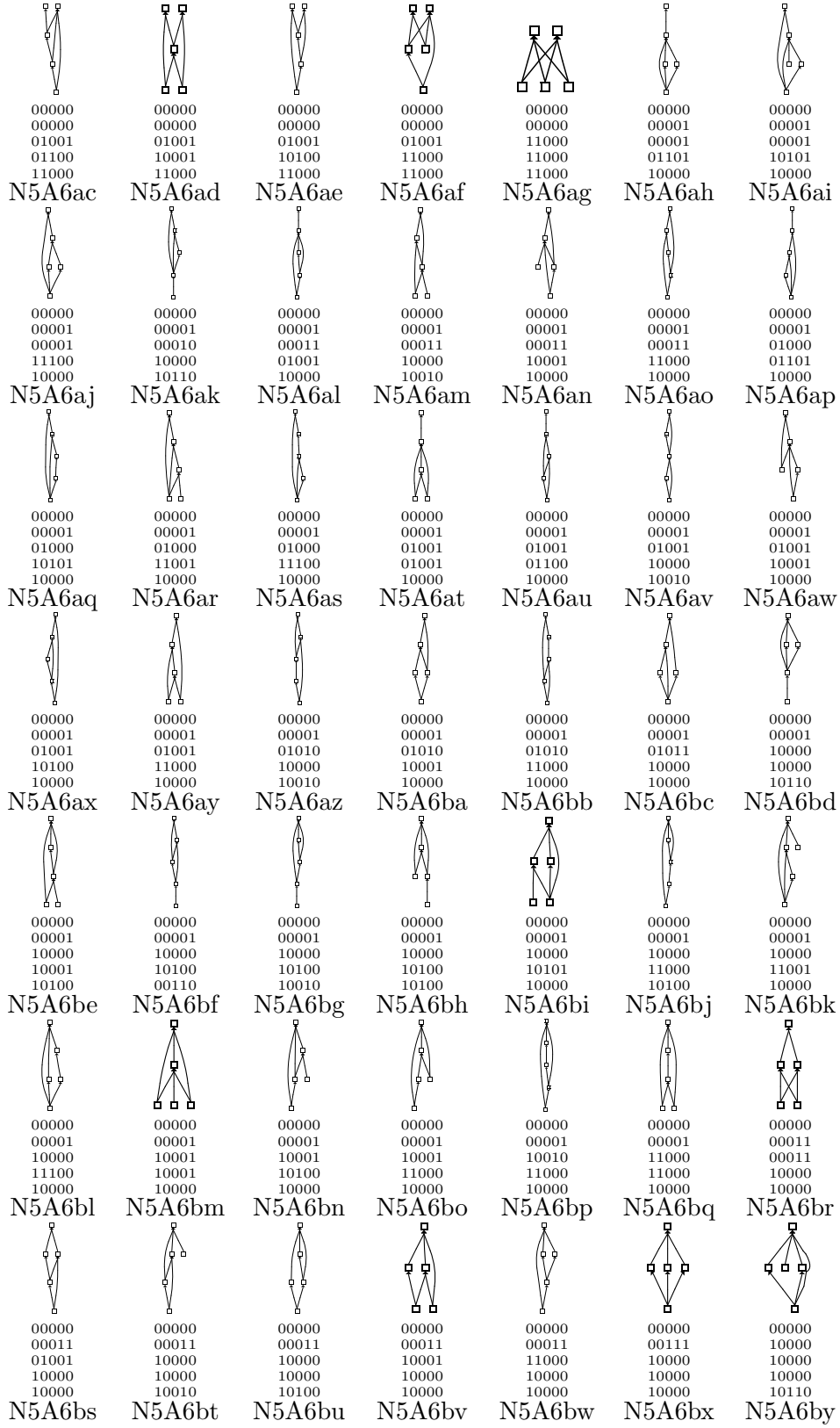
N5A2a N5A2b N5A2c N5A2d

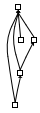

00000 00000 00000 00000 01110 N5A3a
00000 00000 00000 00001 01100 N5A3b
00000 00000 00000 00100 00110 N5A3c
00000 00000 00000 00100 01010 N5A3d
00000 00000 00000 00100 01100 N5A3e
00000 00000 00000 00100 11000 N5A3f
00000 00000 00001 00001 01000 N5A3g
00000 00000 00001 00100 01000 N5A3h
00000 00000 00001 01000 01000 N5A3i
00000 00000 00001 01000 10000 N5A3j
00000 00000 01000 01000 01000 N5A3k
00000 00000 01000 01000 10000 N5A3l
00000 00000 00000 00000 11110 N5A4a
00000 00000 00000 00001 11100 N5A4b
00000 00000 00000 00100 01110 N5A4c
00000 00000 00000 00100 11010 N5A4d
00000 00000 00000 00100 11100 N5A4e
00000 00000 00000 00101 01100 N5A4f
00000 00000 00000 00101 11000 N5A4g
00000 00000 00000 01100 01100 N5A4h
00000 00000 00000 01100 10100 N5A4i
00000 00000 00001 00001 11000 N5A4j
00000 00000 00001 00100 11000 N5A4k
00000 00000 00001 00101 01000 N5A4l
00000 00000 00001 01000 01010 N5A4m
00000 00000 00001 01000 10010 N5A4n
00000 00000 00001 01000 11000 N5A4o
00000 00000 00001 01001 01000 N5A4p
00000 00000 00001 01001 10000 N5A4q
00000 00000 00001 01100 01000 N5A4r
00000 00000 00001 01100 10000 N5A4s
00000 00000 00001 11000 01000 N5A4t
00000 00000 00011 01000 01000 N5A4u
00000 00000 00011 01000 10000 N5A4v
00000 00000 01000 01000 01010 N5A4w
00000 00000 01000 01000 10010 N5A4x
00000 00000 01000 01000 11000 N5A4y
00000 00000 01000 01001 10000 N5A4z
00000 00000 01000 01100 10000 N5A4aa
00000 00000 01000 10000 11000 N5A4ab
00000 00001 00001 00001 10000 N5A4ac
00000 00001 00001 00001 10000 N5A4ad
00000 00001 00001 00010 00010 N5A4ae
00000 00001 00001 00001 10000 N5A4af
00000 00001 00001 00010 01000 N5A4ag
00000 00001 00001 00010 10000 N5A4ah
00000 00001 00001 01000 10000 N5A4ai


00000 00001 10000 10000 10000
N5A4aj
00000 10000 10000 10000 10000
N5A4ak
00000 00000 00000 00100 11110
N5A5a
00000 00000 00000 00101 11100
N5A5b
00000 00000 00000 01100 01110
N5A5c
00000 00000 00000 01100 10110
N5A5d
00000 00000 00000 01100 11100
N5A5e
00000 00000 00001 00101 11000
N5A5f
00000 00000 00001 01000 11010
N5A5g
00000 00000 00001 01001 11000
N5A5h
00000 00000 00001 01100 11000
N5A5i
00000 00000 00001 01101 01000
N5A5j
00000 00000 00001 01101 10000
N5A5k
00000 00000 00001 11000 01010
N5A5l
00000 00000 00001 11000 11000
N5A5m
00000 00000 00001 11001 01000
N5A5n
00000 00000 00001 11100 01000
N5A5o
00000 00000 00011 01000 01010
N5A5p
00000 00000 00011 01000 10010
N5A5q
00000 00000 00011 01000 11000
N5A5r
00000 00000 01000 01000 01110
N5A5s
00000 00000 01000 01000 10110
N5A5t
00000 00000 01000 01000 11010
N5A5u
00000 00000 01000 01001 01100
N5A5v
00000 00000 01000 01001 10100
N5A5w
00000 00000 01000 01001 11000
N5A5x
00000 00000 01000 01100 01100
N5A5y
00000 00000 01000 01100 10010
N5A5z
00000 00000 01000 01100 10100
N5A5aa
00000 00000 01000 01100 11000
N5A5ab
00000 00000 01000 01101 10000
N5A5ac
00000 00000 01000 10000 11010
N5A5ad
00000 00000 01000 10001 10100
N5A5ae
00000 00000 01000 10001 11000
N5A5af
00000 00000 01000 10100 10100
N5A5ag
00000 00000 01000 10100 11000
N5A5ah
00000 00000 01000 11000 11000
N5A5ai
00000 00001 00001 00101 10000
N5A5aj
00000 00001 00001 01100 10000
N5A5ak
00000 00001 00001 10000 10010
N5A5al
00000 00001 00001 10001 10000
N5A5am
00000 00001 00001 10100 10000
N5A5an
00000 00001 00010 01001 10000
N5A5ao
00000 00001 00010 10000 10010
N5A5ap


00000 00001 00010 10000 10100
N5A5aq
00000 00001 00011 01000 10000
N5A5ar
00000 00001 00011 10000 00010
N5A5as
00000 00001 00011 10000 10000
N5A5at
00000 00001 01000 01100 10000
N5A5au
00000 00001 01000 10000 10010
N5A5av
00000 00001 01000 10001 10000
N5A5aw
00000 00001 01000 10100 10000
N5A5ax
00000 00001 01000 11000 10000
N5A5ay
00000 00001 01001 10000 10000
N5A5az
00000 00001 01010 10000 10000
N5A5ba
00000 00001 10000 10000 00110
N5A5bb
00000 00001 10000 10000 10010
N5A5bc
00000 00001 10000 10001 10000
N5A5bd
00000 00001 10000 10100 10000
N5A5be
00000 00001 10000 11000 10000
N5A5bf
00000 00011 10000 10000 10000
N5A5bg
00000 10000 10000 10000 10010
N5A5bh

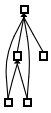

00000 00000 00000 01100 11110
N5A6a
00000 00000 00000 01101 11100
N5A6b
00000 00000 00000 11100 11100
N5A6c
00000 00000 00000 01101 11000
N5A6d
00000 00000 00001 11000 11010
N5A6e
00000 00000 00001 11001 11000
N5A6f
00000 00000 00001 11100 11000
N5A6g
00000 00000 00001 11101 01000
N5A6h
00000 00000 00011 01000 11010
N5A6i
00000 00000 00011 01001 11000
N5A6j
00000 00000 00011 11000 11000
N5A6k
00000 00000 01000 01000 11110
N5A6l
00000 00000 01000 01001 11100
N5A6m
00000 00000 01000 01100 01110
N5A6n
00000 00000 01000 01100 10110
N5A6o
00000 00000 01000 01100 11010
N5A6p
00000 00000 01000 01100 11100
N5A6q
00000 00000 01000 01101 10100
N5A6r
00000 00000 01000 01101 11000
N5A6s
00000 00000 01000 10000 11110
N5A6t
00000 00000 01000 10001 11100
N5A6u
00000 00000 01000 10100 10110
N5A6v
00000 00000 01000 10100 11010
N5A6w
00000 00000 01000 10100 11100
N5A6x
00000 00000 01000 10101 11000
N5A6y
00000 00000 01000 11000 11010
N5A6z
00000 00000 01000 11000 11100
N5A6aa
00000 00000 01000 11001 11000
N5A6ab


00000 00000 01001 01100 11000
N5A6ac
00000 00000 01001 10001 11000
N5A6ad
00000 00000 01001 10100 11000
N5A6ae
00000 00000 01001 11000 11000
N5A6af
00000 00000 11000 11000 11000
N5A6ag
00000 00001 00001 01101 10000
N5A6ah
00000 00001 00001 10101 10000
N5A6ai
00000 00001 00001 11100 10000
N5A6aj
00000 00001 00010 10000 10110
N5A6ak
00000 00001 00011 01001 10000
N5A6al
00000 00001 00011 10000 10010
N5A6am
00000 00001 00011 10001 10000
N5A6an
00000 00001 00011 11000 10000
N5A6ao
00000 00001 01000 01101 10000
N5A6ap
00000 00001 01000 10101 10000
N5A6aq
00000 00001 01000 11001 10000
N5A6ar
00000 00001 01000 11100 10000
N5A6as
00000 00001 01001 01001 10000
N5A6at
00000 00001 01001 01100 10000
N5A6au
00000 00001 01001 10000 10010
N5A6av
00000 00001 01001 10001 10000
N5A6aw
00000 00001 01001 10100 10000
N5A6ax
00000 00001 01001 11000 10000
N5A6ay
00000 00001 01010 10000 10010
N5A6az
00000 00001 01010 10001 10000
N5A6ba
00000 00001 01010 11000 10000
N5A6bb
00000 00001 01011 10000 10000
N5A6bc
00000 00001 01011 10000 10110
N5A6bd
00000 00001 10000 10001 10100
N5A6be
00000 00001 10000 10100 00110
N5A6bf
00000 00001 10000 10100 10010
N5A6bg
00000 00001 10000 10100 10100
N5A6bh
00000 00001 10000 10101 10000
N5A6bi
00000 00001 10000 11000 10100
N5A6bj
00000 00001 10000 11001 10000
N5A6bk
00000 00001 10000 11100 10000
N5A6bl
00000 00001 10001 10001 10000
N5A6bm
00000 00001 10001 10100 10000
N5A6bn
00000 00001 10001 11000 10000
N5A6bo
00000 00001 10010 11000 10000
N5A6bp
00000 00001 11000 11000 10000
N5A6bq
00000 00011 00011 10000 10000
N5A6br
00000 00011 01001 10000 10000
N5A6bs
00000 00011 10000 10000 10010
N5A6bt
00000 00011 10000 10000 10100
N5A6bu
00000 00011 10001 10000 10000
N5A6bv
00000 00011 11000 10000 10000
N5A6bw
00000 00111 10000 10000 10000
N5A6bx
00000 10000 10000 10000 10110
N5A6by



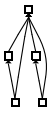
00000
10000
10000
10001
10100

N5A6bz



00000
10000
10000
10100
10100

N5A6ca



00000
10000
10000
10100
11000

N5A6cb