DRAFT

USE OF DYNAMICALLY ALLOCATABLE LABELLED MEMORY BLOCKS IN PROGRAMMING SYSTEMS

Jane G. Jodeit
Computer Project
Rice University
Houston, Texas

October, 1965

draft

Use of Dynamically Allocatable Labelled Memory Blocks in Programming Systems

ABSTRACT

This paper describes a system of program and data representation that has been in use on the Rice University Computer for four years. Each block in the storage domain is labelled by a codeword and may contain a program, a data vector, or codewords which in turn label blocks to form arrays. This internal storage configuration is discussed with its realized advantages in programming systems:

- ease of addressing
- flexibility of array structures
- efficient memory utilization
- dynamic allocation
- means of linkage between programs and from programs to data
- operations on blocks and arrays
- influence on formula language definition and compiler coding

The application of labelled blocks may be extended to areas of time sharing, multi-media storage control, and more complex data representations. On the basis of experience at Rice, some ideas on such extensions are presented.

Objectives

Memory content in any computer application consists of programs and data. The representation of these elements in storage contributes directly to the convenience of coding, the organization of programming systems, and the efficiency of computation.

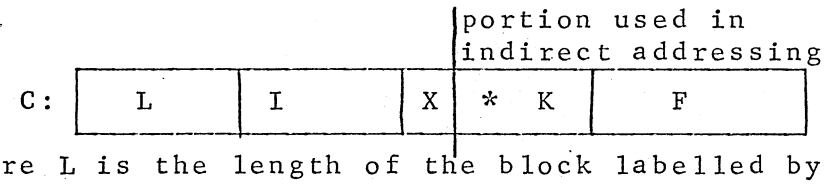
This paper shows how dynamically allocatable labelled memory blocks offer great advantages in all of these areas. A system of codewords, which are just the one-word labels on blocks of consecutive memory locations, has been in use on the Rice University Computer for four years. This system is described in detail, and it is shown to provide the following specific features:

- ease of addressing
- flexibility of data structures
- efficient memory utilization
- a powerful basis for dynamic storage allocation
- convenience in linkage of programs to programs and programs to data
- representation which encourages operations on blocks
- attractive influence on formula language definition and compiler coding

Codewords as Block Labels and Their Use in Addressing

A set of consecutive storage locations is called a memory block. Every such block is labelled by a single word called a codeword. The codeword for a block contains descriptive information about the block, and a portion of the codeword is used in indirectly addressing the block content.

As realized on the Rice University Computer the general codeword format is:



where L is the length of the block labelled by the codeword C

I is the relative address by which the first physical word of the block labelled by C is to be addressed

X is present if the block labelled by C contains codewords

* is present if indirect addressing is to be iterated into a word of the block labelled by C

K is present if the block labelled by C is indexed, i.e., if the words of the block are to be addressed randomly; K then specifies which index register is used to dynamically contain the relative address of the word within the block

F is an address associated with the block labelled by C so that the address of the first physical word of the block is

$F' = F + I$ if K is present

$= F + I - 1$ if K is not present

The portion of a codeword used in indirect addressing is, as described above, designed to be used with the hardware definition of the Rice Computer. Indirect addressing may be iterated any number of times and indexing by any of six registers may be specified for each iteration. If C^i is the codeword in use at the i^{th} level of indirect addressing, the hardware obtains $*^i$, K^i, and F^i from C^i and performs as follows:

(1) If K^i is present, use contents of register specified and add to obtain
$C^{i+1} = (K^i) + F^i$
If K^i is not present,
$C^{i+1} = F^i$

(2) If $*^i$ is present, return to step (1) for codeword C^{i+1} at level i + 1.
If $*^i$ is not present, use C^{i+1} as final address and do not iterate.

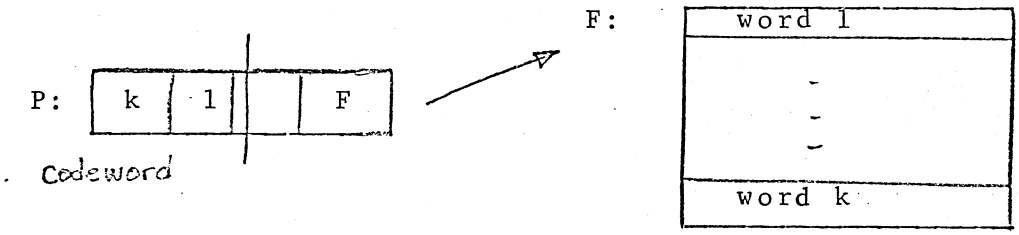
The initiation of indirect addressing is from an instruction, say at C^o, which contains in its indirect addressing protion $*^o$, F^o, and perhaps K^o. Thus, from a single instruction the codeword address C^1 is determined and the hardware iterates through the indirect addressing procedure to provide the final address for continued execution.

The full generality of codewords can be implemented with maximal efficiency only with such hardware. It is surprising that more computers do not employ this simple but extremely powerful indirect addressing definition. With more restrictive definitions of indirect addressing the full generality of a codeword system can be realized at the expense of some efficiency, or some generality can be sacrificed and the most common applications handled efficiently.

Block Content and Addressing

Given the codeword definition of the previous section, we now examine how it is used to build the elements of a programming system. The simple elements are programs and data lists, to be called vectors.

A program P may be considered as a set of words which are to be executed as instructions and should, for efficient control hardware utilization, occupy consecutive storage locations. Thus, the program P defines a memory block. Assume a single entry point at the first word of P so that only the first word meed be addressed from another program; so the block for P need not be indexed. If P is of length k words, the program and its codeword appear as



Control is then transferred to program P by the single operation:

transfer control to *P

where * specifies indirect addressing through the codeword P. A single indirect addressing level is decoded:

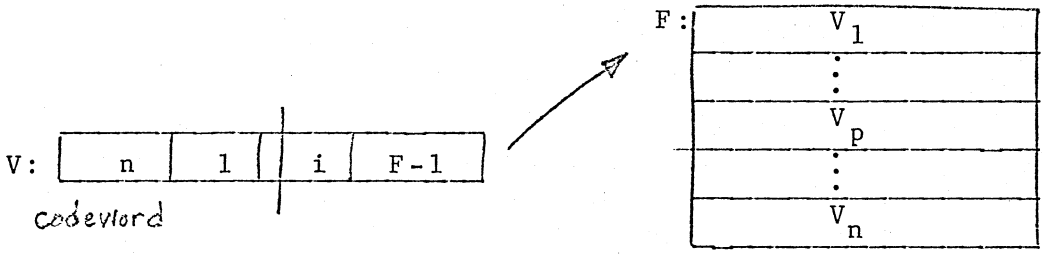
transfer control to *P

F

and the final address obtained is F, the address of the first word of program P. The address F never appears in code, only in the codeword for P. The address P which does appear in

all coded references to the program is invariant while F may vary from run to run or even within a run, as a function of total storage requirements.

A vector V may be considered as a set of words which may be addressed randomly by their relative position and should, for efficient index hardware utilization, occupy consecutive storage locations. Thus, the vector V defines an indexed memory block. If V is of length n words with the first word at relative position 1 and register i is to be used for indexing, the vector and its codeword appear as



Access to the p^{th} element of vector V is accomplished by the two operations:

(1) set index register i to p

(2) access *V

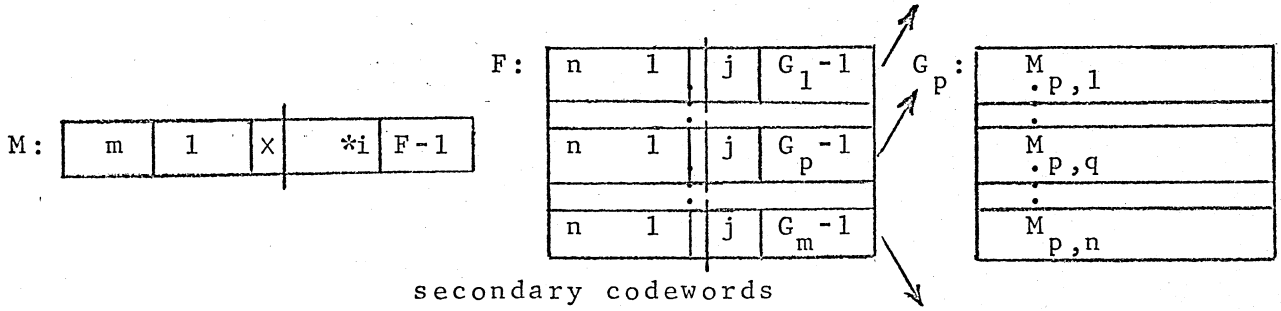
where * specifies indirect addressing through the codeword V. A single indirect addressing level is decoded in step (2):

(2) access *V

$p + F - 1$

and the final address obtained is that of the element V_p, the p^{th} word in the block beginning at location F. Again, the address F (or in this case F - 1) never appears in code, only in the codeword for V. Code which references V is dependent only on the invariant codeword address, never on the physical location of the memory block for the vector.

A two-dimensional data structure, matrix M, is simply a vector of vectors. If the matrix M is m rows by n columns in size, then it will be represented as a vector of m vectors each n words in length. Thus, the matrix M defines m indexed memory blocks (one per row) containing n data words each, and the codewords for the rows define an indexed memory block containing m codewords to complete the representation. If the "upper left" element of matrix M is to be element $M_{1,1}$ and row and column indices are to be specified in registers i and j respectively, the matrix structure appears as



Access to the q^{th} element of the p^{th} row of matrix M is accomplished by the three operations:

(1) set index register i to p

(2) set index register j to q

(3) access *M

where * specifies indirect addressing through the codeword M. Two indirect addressing levels are decoded in step (3):

(3) access *M

*p + F - 1

q + G_p - 1

and the final address obtained is that of the element $M_{p,q}$, the q^{th} word in the block beginning at location G_p, which is addressed from the p^{th} word in the block beginning at location F. The physical locations of the blocks which form the matrix

never appear in code, only in codewords. Code which references M is dependent only on the highest level (primary) invariant codeword address. Another very important point is that the code for access to matrix M in no way depends on the lengths m and n, only on the fact that M is two dimensional. Hence, while the location of blocks which comprise M may vary as a function of total storage requirements, the size parameters m and n may as easily vary as a function of dynamic problem definition.

In general, a storage configuration in the codeword system is called an array. On the highest level, that addressed in code, is a single codeword which labels a block which may contain codewords. On the lowest level of what is a tree structure is the data of the array. The intermediate levels are formed by blocks of codewords, the structure or control for the array.

Some Interesting Arrays

The array forms described in the last section satisfy a large share of programming needs. Because they are encountered most frequently, they are termed standard forms.

- A standard program is a one-level array, all of whose words comprise a memory block for computer execution.
- A standard vector has initial index of 1, and even the indexing register is fixed in practice.
- A standard matrix is rectangular, has initial row and column indices of 1, and even the indexing registers are fixed in practice.

The full generalities of codeword definition and array structure are not particularly well illustrated by the standard forms. The logic and organization of a programming system can often be supported in more unusual array structures, and a few of these will be examined here.

The logical or physical situation represented in an indexed data array may be best satisfied by the use of negative as well as positive indicies. A plot of y values for a grid in x across x = 0 is a good example. The left x end point will be ≤ 0 and determine the initial index on the vector of y values to be plotted. Similarly, a two dimensional array may contain z values for a grid in x and y over the origin x = 0, y = 0. Initial row and column indices will be ≤ 0 and they may not be equal to each other. Also, there is no reason to adhere to a rectangular array, for the grid may not be rectangular. Each row may contain data elements for a given y value as x varies, and only the x range of definition for that y value need be stored for that row. So each row may have different

initial index and length. In all cases, the indices used in code are "natural" to the application, and only the storage for the defined grid area need be represented at any time.

The content of a program does not always contain just instructions to be executed. Constants and temporary storage locations are conveniently assigned by a translator to words which follow the code; these need be referenced only from within the program, and there is no need to maintain a descriptor of this area external to the program. Another requirement for programs may be that each be linked to other arrays at the time they are loaded. The burden of this function cannot be placed on the program itself, rather, it is rightly a supervisory programming function. Therefore, a descriptor of the collection of linkage words need be maintained external to the program. The words themselves are to be referenced during program execution and are best kept in the block for the program. It has worked out very well to collect linkage words prior to executed code, but code always begins at relative location 1 in the block. If in the codeword for a program the initial relative address is $\ell < 1$, then the $1 - \ell$ words which precede the code are maintained by supervisory routines as necessary. The indirect addressing portion of the codeword is maintained so that a transfer of control through the codeword produces access to relative address 1 in all cases.

In mathematical programming it is not uncommon to encounter matrices with many zero elements and "bunched" non-zero elements. Triangular and diagonal matrices are good examples. With a codeword system these configurations are easily and economically represented in storage. The upper right triangle of an $n \times n$ matrix is represented as n row vectors with initial indices

varying from 1 to n and lengths from n to 1. The lower left triangle requires variation of only row lengths, from 1 to n, while all initial indices are 1. A diagonal matrix might most economically be represented as a one-dimensional array, a vector. So consider an n × n matrix with only a "strip" three elements wide down the diagonal. The n row blocks would have initial indices 1,1,2,3,...,n-2,n-1 and lengths 2,3,3,...,3,3,2. These condifurations illustrate some important advantages of codewords:

- Only the range of elements to be addressed are stored.
- The economy of data storage is not traded for coding complexity.
- The code utilizes "natural" indices in all cases.

A very useful concept is that of an array of programs. If a different program is to be executed for each parameter value i = 1,2,...,n, these may most conveniently be stored as a vector of programs. This structure is a two-dimensional array with unindexed data blocks on the lowest level, certainly not of uniform length. The programs may be developed independently. They will certainly be of more manageable length than a single composite routine. Only those to be executed need be present in storage at any time. A good example of a sparse four-dimensional array of program arises in compilation: code generators with array access on the basis of the three indices of left operand type, right operand type, and rank of connecting operand. The array is sparse because not all triplets may occur in practice. With a minimum of effort, programs may be individually modified and new programs may be added as new triplets become meaningful.

Codeword Location and Reference by Programs

Program references may be to internal or external quantities. Internal variables are located within a program and are referenced only from that program; with a codeword system these are only scalar quantities since all programs and data arrays are in independent memory blocks. External variables are located outside all programs and may be referenced from any program; scalar quantities to be referenced by more than one program, all programs, and all data arrays fall into this category.

The memory layout in the codeword system provides a fixed codeword region, C, which contains variables to be addressed by number. The region C is a range of addresses of locations which contain values of external scalars and codewords for external non-scalars (arrays). If c is an address in C which is the codeword address of an external non-scalar, a program reference is accomplished by indirect addressing through c:

```
        transfer control to *c
or      access *c
```

Numeric assignment of external quantities to locations in C is limiting and inconvenient in many applications, in particular where names are used for all references in the coding language. Parallel tables are provided:

- symbol table (ST) which contains names of external variables, and
- value table (VT) which contains values of external scalars and codewords for external non-scalars (arrays) which are to be addressed by name.

The value table then is of the same form as the fixed codeword region C. But the VT addresses are not known for coded

references. Named references to external variables are made indirectly through a linkage word in the program (located prior to code) which is filled with the appropriate VT address as a supervisory function during loading of the program. If A is the name of an external non-scalar, the name A will appear in ST, and the codeword for A will occupy the corresponding VT entry (say located at VT_A). Program reference to A is accomplished by indirect addressing through the linkage word named A in the program:

transfer control to *A

or access *A

But loading of the program provides the address VT_A in the linkage word A. So the first indirect addressing operation provides

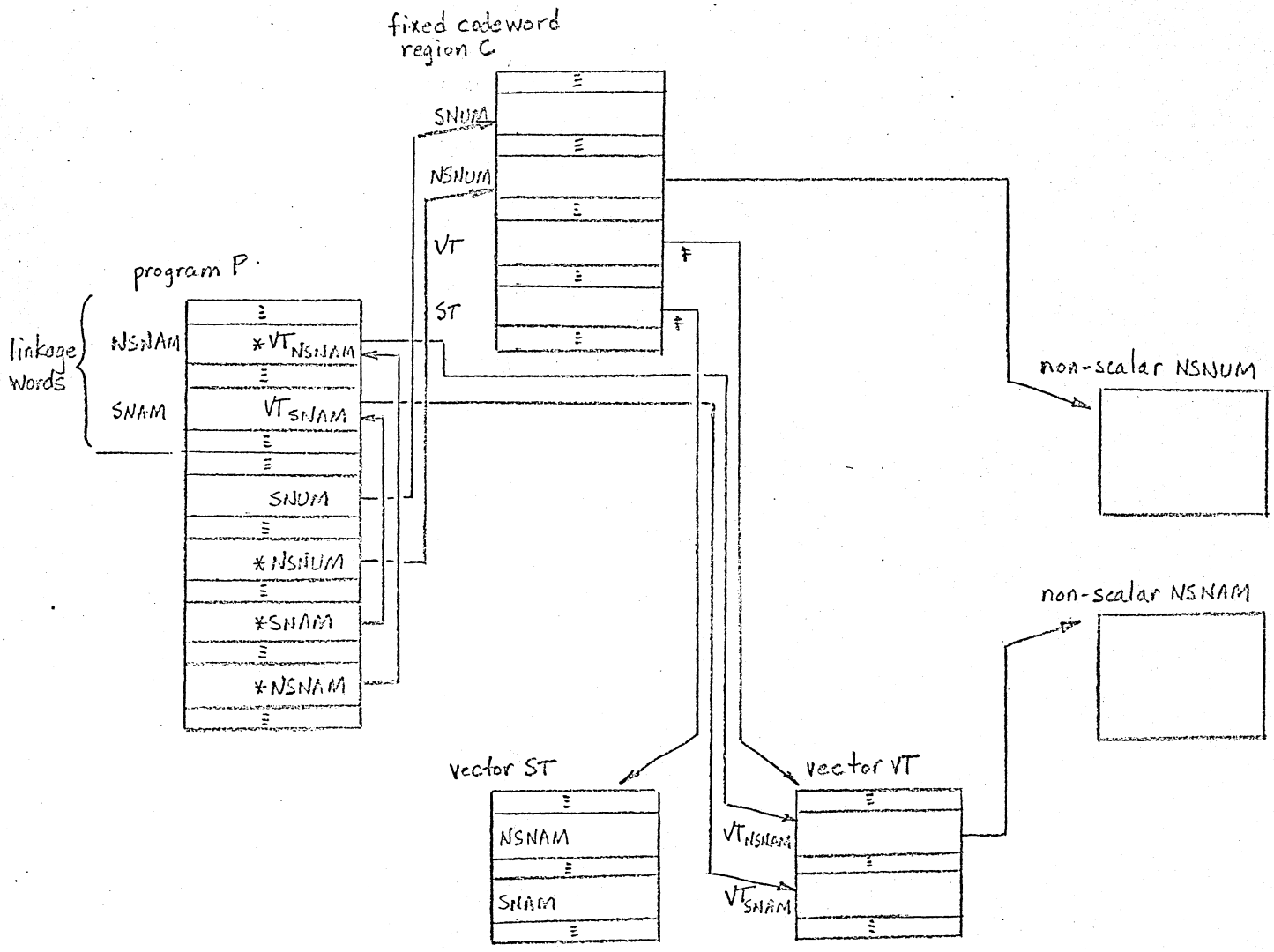
transfer control to $*VT_A$

or access $*VT_A$

Subsequent addressing is just as if a location in C had been initially addressed.

These linkages are illustrated by the program P which references

- scalar SNUM by number
- non-scalar NSNUM by number
- scalar SNAM by name
- non-scalar NSNAM by name



‡Note that the vectors ST and VT are memory blocks with codewords in C.

The linkages discussed thusfar have been for static program references, where the variable is known to be internal or external at coding time and the number or name for an external variable is known. Programs may also reference parameters which take on argument assignments at each execution. Linkages to arguments are for dynamic program references.

Arguments are provided to a program on a push-down working storage list W. Parameter references are coded indirectly through a dynamic linkage word in W located at a fixed position relative to the value of the pointer into W upon entry to the program. Then a parameter address takes on static relative value, say $\dot{w}$, in W, and a program reference is accomplished by indirect addressing through $\dot{w}$:

transfer control to *$\dot{w}$

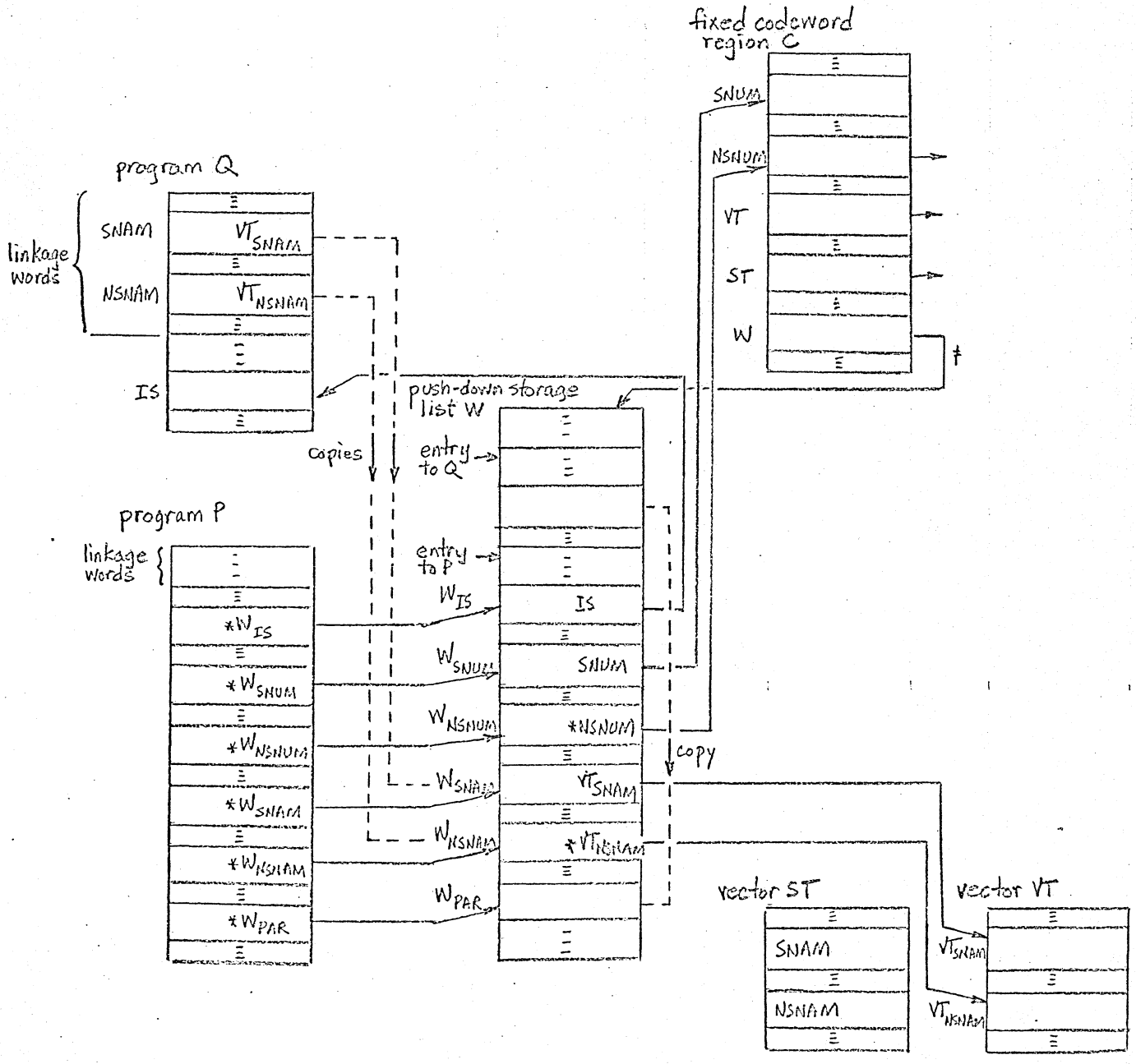
or access *$\dot{w}$

The word at $\dot{w}$ is generated dynamically (for each execution of the program), and the argument address obtained varies appropriately.

It should be pointed out that at time of execution arguments are either internal to some program or external, and if external are either named or numbered. This may not even be known to the calling program for an argument may be passed through several dynamic levels of programs. These considerations in no way affect parameter references but must be considered for dynamic set-up of argument linkage words in W. Dynamic linkage and content of linkage words in W is illustrated by program P which provides arguments to program Q as follows:

- scalar IS, internal to P, linkage in W by the direct address IS

- external scalar SNUM, addressed by number, linkage in W by the direct address SNUM in region C
- external non-scalar NSNUM, addressed by number, linkage in W by the indirect address *NSNUM in region C
- external scalar SNAM or non-scalar NSNAM addressed by name, linkage in W for Q as static linkage word for P
- scalar or non-scalar PAR, parameter in P, linkage in W for Q as linkage in W for P



‡Note that the push-down list W is a memory block with codeword in C.

Storage Domain for Dynamic Allocation

The memory configuration for dynamic allocation in the codeword system consists of

- o first, the control area which contains special machine registers, manual communication region, and the fixed codeword region C;
- o second, any memory blocks which are not to be dynamically allocated -- as the elements of the operating system;
- o third, the dynamic storage allocation domain which occupies the remainder of the memory.

Dynamic allocation in memory is defined by the two basic procedures:

- o activation or creation, of a memory block labelled by a codeword, and
- o inactivation of a memory block labelled by a codeword so that the space may be subsequently used in allocation for other blocks.

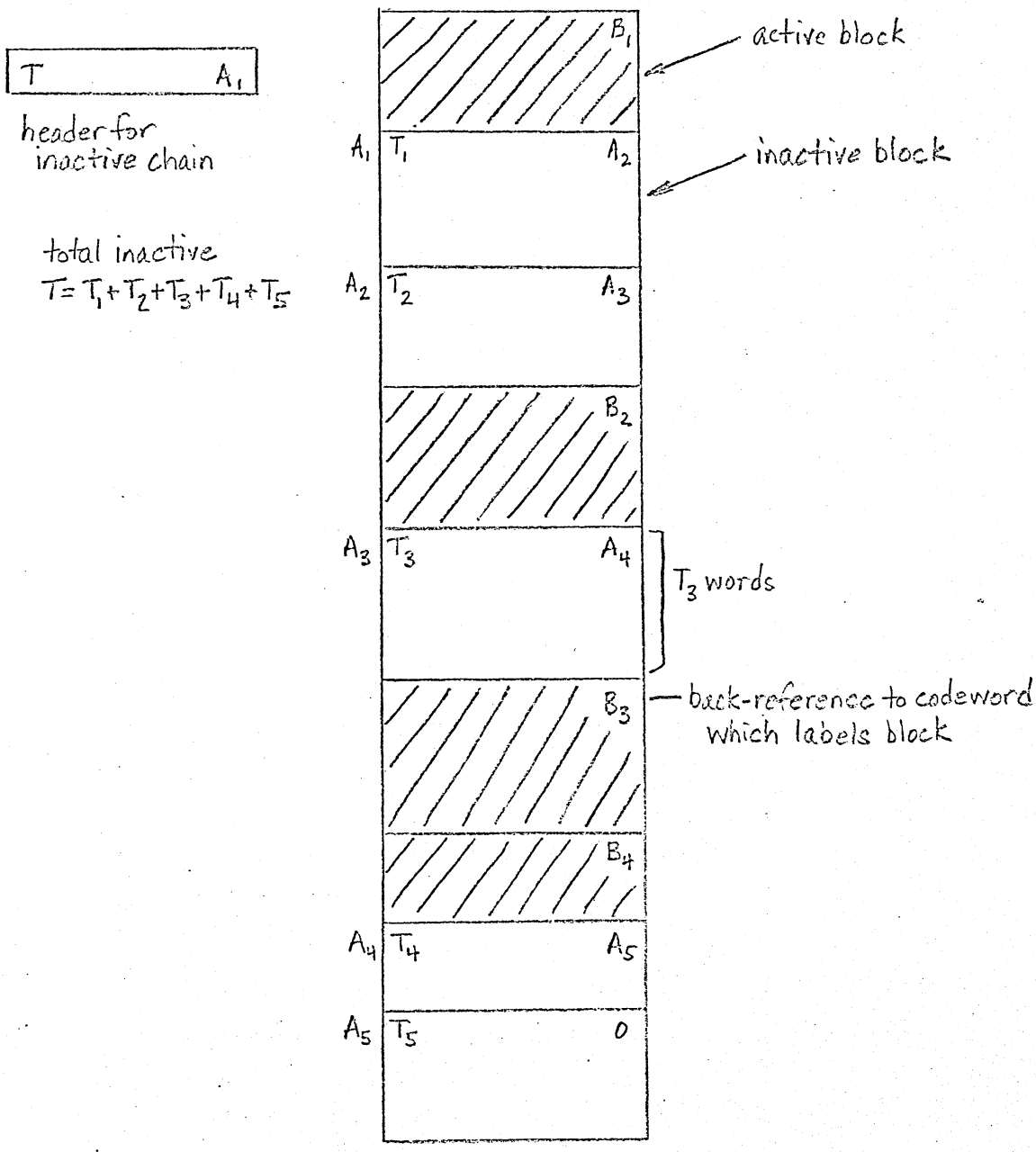
Initial loading of programs and data is just a sequence of activations, and the blocks loaded will be sequentially located in the storage domain. But as a run progresses blocks may be inactivated and new one octivated, so the general state of the storage domain is a mixture of active and inactive blocks.

Each active block in the storage domain is labelled by a codeword (which may itself be a word in an active block). Each active block is immediately preceded by a back-reference word which contains the codeword address for the block. So, the activation of block to contain n data words requires n+1 words of storage.

Each inactive block in the storage domain contains in its first word its length and the address of the next inactive

block. The inactive blocks then form a chain. The header for the inactive chain is located at a fixed address; this contains the total number of inactive words in the domain and the address of the first inactive block in the chain. The last inactive block is distinguished by a null next block address.

The general layout of the dynamic storage allocation domain is illustrated in the following diagram:



Dynamic Allocation Procedures and Operations

The basic dynamic storage allocation procedures are inactivation and activation of blocks in the domain.

Inactivation of a block in the dynamic storage allocation domain is accomplished simply by linking it into the chain of inactive blocks. The codeword is cleared to signify that it no longer labels an active block. The total inactive space, kept in the header for the inactive chain, must be updated.

Activation is attempted only if the total active space is adequate for the requested allocation. Then three means of obtaining the space may be utilized:

- o First, a simple scan of the chain of inactive blocks is made, and the first which is large enough to provide the requested allocation is used; any inactive space remaining in the block is re-chained.
- o Second, if the scan fails, combination of adjacent inactive blocks is performed and the scan is made again.
- o Third, if the scan fails after combination, reorganization of the storage domain is performed. All active blocks are packed to the low-address end of memory. A single inactive block defines the inactive chain, and this will provide the requested allocation. The packing of active blocks requires copying memory blocks and for each changing the addressing portion of the labelling codeword to appropriately reflect the new block location. If the block contains codewords, the back-reference word for the block labelled by each must be changed to reflect the new codeword location.

Each dynamic allocation request is defined by a codeword address and the allocation operation to be performed on the

block labelled by the codeword: to free the block or to take a space of length n words.

The freeing of space labelled by a given codeword is performed by recursive inactivation so that the array labelled is completely freed.

The taking of a block of n words to be labelled by a given codeword is performed by first freeing the array labelled (if any) and then obtaining an active block n+1 words long (including the back-reference word) in the domain. So, a new block definition automatically replaces an old definition.

Implications of Block Mobility

If all addressing of memory blocks and the words in them was done relatively, any memory block could be moved (in the reorganization procedure described in the preceding section) at any time. In practice, it is possible with the hardware and often efficient in code to obtain absolute memory addresses. If the words addressed are subject to being moved and the addresses are retained through a reorganization, subsequent use is meaningless.

By relative addressing is meant one of the following:

- All addressing of arrays is done through primary (highest level) codewords in the fixed codeword region, C, or the value table, VT. This means that VT must be outside the dynamic storage allocation domain, not an impractical restriction in many cases.
- Or, a "trace" may be kept of the path taken into an array, and this will be appropriately updated in the reorganization procedure. Such a trace consits of the primary codeword address and relative addresses on lower levels to the element whose absolute address is to be retained. This absolute address is stored for reference by programs and for system maintenance in reorganization.

Some cases in which these considerations have proved important in programming for the Rice Computer will be discussed.

For mobility of programs all internal references must be relative. This is facilitated by the hardward of the Rice Computer because the instruction counter is an index register. But since the value of the instruction counter is an absolute address, a hardware feature of base address plus offset would be preferred for program addressing. Similarly, program return addresses must not be saved as absolute addresses, but as addresses relative to the base

of the calling program. Thus, a trace of program execution must be maintained, this to include not just base and offset at each call but also relative codeword address. Such a control trace is not implemented in the Rice operating system since a restriction on program mobility is considered tolerable: If a program causes reorganization of the dynamic storage allocation domain (itself or in any program called), it may not be located in the domain. Hardware to prevent the necessity of absolute control address retention within programs and to efficiently trace control would be very desirable.

The array structure is recursively defined so that sub-arrays are arrays. Operations on arrays should likewise be well defined at any level. If an array operand is a sub-array in the dynamic storage allocation domain, its primary codeword address is subject to change in the re-organization procedure. Therefore operand traces must be maintained for complete generality of operations on arrays. This is done in the operating system on the Rice Computer; only unary array manipulations are available, so a single trace suffices. The cost of trace maintenance for multiple operands prohibits the fully recursive implementation. Again, hardware to facilitate such a feature would be of great value.

In the codeword system in use on the Rice Computer the codeword storage areas, C and VT, are not movable once loaded. In applications where a single processor and its extension in supervisory system is to operate on a number of problems in short time intervals, it is not feasible to reload for each problem. It should be possible to dynamically allocate C and VT as they pertain to each problem, and then addressing into these blocks must also be relativized.

Supervisory Operations on Arrays

Many mathematical operations on data arrays are familiar, such as the transpose of a matrix or the addition of two vectors which represent polynomials. The example of plotting vector element values versus the vector indices has been mentioned earlier. These operations are performed by subroutines which are coded for general application, and particular parameters may be obtained from the codewords for the operands at each execution.

The operating system for the Rice Computer is defined as a set of operations on arrays. Communication is through a "macro-instruction" called a control word which is interpretively executed. A control word prescribes

- the codeword address or name of the array to be operated upon,
- the operation to be performed,
- where in the array (relative address) the operation is to begin, and
- the extent (number of elements) of the operation.

A supervisory operation may be applied to the memory block labelled by the specified codeword; this is simple application. And more generally, a supervisory operation may be applied to all data of an array; this is recursive application. For example, the operation of printing may be applied to the two-dimensional matrix M to provide output of the vector of row codewords as a simple operation or all matrix elements as a recursive operation.

Also, any supervisory operation may be applied to an array which is a sub-array. The data trace of the operating system (described earlier) is set to the level of the memory block of which the array to be operated upon is an element. The codeword address in the control word is a relative address. The operation

may be specified for simple or recursive application. Suppose the array L is a vector of two-dimensional matrices. Then setting the trace to L and doing a recursive print with codeword address n would provide output of the n^{th} matrix in array L. Setting the trace to L, then n and doing a print with codeword address k would provide output of the k^{th} row of the n^{th} matrix in array L.

The supervisory function of loading has been mentioned. This is done with the read operation in the Rice Computer system. Read is used to take space for a memory block, set up the codeword, and fill the space with either zeros or data obtained from an input medium. The correct operation overlays all or a portion of an existing memory block with either zeros or data obtained from an input medium.

An array may be inactivated (in the sense of dynamic storage allocation) by the free operation. Block lengths may be changed by the operations to insert zeros or delete elements at a specified relative location. The operation to set initial index to a specified value will cause the relative addressing within a block to be changed.

Control for computer execution may be given to any block element by the execute operation, intended only for application to programs. Of interesting consequence is the execution of a block or an array of control words by the operating system; this is accomplished by the operation to execute a control word sequence.

Output operations are provided which are appropriate to the output media of the computer.

Consequences of Codewords

Program addressing of array elements, for data access or program execution, has been described in detail. The simplicity of the use of a single indirect address and natural relative indices in code is an immediate advantage of the codeword system. The addressing format is independent of

- the type of array operand being addressed, named or numbered, a parameter or known external;
- the dimension, index range, and irregularity of the array;
- the location of the array in storage.

The hardware execution of index setting and indirect addressing is most often less expensive in time than direct computation of addresses as a function of array location and structure. In the case of constant indices and a fixed array, a single direct reference could be made with optimum efficiency. Experience with the Rice Computer system has shown that it is not always convenient to have the indexing register specified in the data structure as it is with codewords; rather, it would often be useful to allow the register used to vary from instruction to instruction.

The independence of memory blocks afforded by codewords is an asset in programming systems. This would not be possible without the linkages described in earlier sections, of programs to programs, programs to data, and within arrays from level to level. Codewords facilitate the supervisory function of maintaining linkages, removing all such concern from individual programs.

Dynamically, the set of codewords afford a convenient catalog of essential information about the state of the running system. And any specific codeword is available to any program

for a detailed description of the block it labels. This description need not be passed from program to program; the codeword address (with trace) suffices.

In turn, the advantages of block independence and labels on blocks greatly facilitate dynamic storage allocation. No program depends in any way on the location of any array.

Dynamic storage allocation in the codeword system then provides the full flexibility of array definition. It is extremely important that arrays may not only be created and destroyed, but may also be altered in size, "shape", and even purpose, with the dynamic variability of program demands. The allocation scheme with codewords does not depend on fixed block length, so the blocks in use may be packed with no "waste" space.

There are many influences of the codeword system in formula language definition. The uniformity of addressing into arrays is an advantage in code generation. Operations with arrays as entities are very easily provided because a single address suffices for addressing. Examples of explicit specification of array manipulations, on the matrix M, would be

- to create M
- to erase M
- to print M

An example of implicit array manipulations, on matrices, is seen by the expression

C=A×B

to execute the matrix multiplication routine with A and B as arguments, creating the resultant matrix, and label the result with codeword named C. The mechanisms for labelling of arrays by named codewords is, of course, essential to the utilization of a formula language in programming systems. No array

allocation need be done during compilation since all allocation functions are provided by the operating system at time of execution.

Extensions of the Codeword System

It is interesting to consider hardware designed for a codeword system. Instructions should contain no absolute memory addresses, only relative addresses which may be positive or negative. A relative address should be the increment on a base address which is a trace content, the result of addressing into an array. Trace maintenance for control and data references should be provided. A protect mechanism for each block would easily be implemented as a test of index or relative address against the range of

initial index

through

initial index + block length - 1

as specified in the codeword for the block. Variability of index registers for addressing of any given array should be easily controlled from code, not necessarily a parameter of array definition. The level of indirect addressing for any instruction should be a function of array structure but also easily specified in the code, to facilitate access to the structural portions of an array.

In a time-sharing application protection on memory blocks would be essential. Mobility of all blocks would also be necessary. The dynamic context for the application of the computer to any problem would be found in just the codeword vectors for that problem, with interrupt register contents. Since all defined blocks would not be required for execution in the time interval of computer execution, allocation could be done when a codeword is utilized and labels a block not in memory. Block usage could be observed as codewords are utilized in execution and this information used in the decision of what to overlay in memory. Storage into a block could be

dynamically indicated in its codeword so that unnecessary writes out of memory would not be done in overlay. Organization of supervisory control might be facilitated by considering all problems on hand as a vector with correspondence of priority and indices.

A dynamic storage allocation scheme with codewords can be extended to more than one storage medium. As suggested for a time-sharing application, not all of an array need be in program-addressable memory at all times. Trapping should be provided that is activated by an indication in a codeword when it is indirectly addressed at any level of address computation. The indicator would signal that the block labelled is not in memory but must be obtained from auxiliary storage and allocated to memory. The codeword can be used in this case to contain information about auxiliary residence because the memory addressing portion is not functional. The problems of buffering onto block-oriented stores (as disc) and linear stores (as magnetic tape) will not be addressed here. But catalogs of content of auxiliary stores may be allocated to memory only when needed.

There are some useful array operations and data structures which are not permitted by the codeword system for the Rice Computer. With the array definition in use each memory block belongs to only one array, in fact has only one label. If two memory blocks require the same content, there should be no need to have both in memory. Thus, one block could have more than one label, one block belonging to more than one array. If an array has indices $k_i, i=1,\ldots,n$ on levels into the array, then the logical structure with indices $k_i, i=1,\ldots,j-1,j+1,\ldots,n$ is an array only if $j=1$. There are cases where advantages would be gained if j could take on

any value. For example, the rows of a two-dimensional array are vectors; the columns are not. Chain storage could occupy allocatable memory blocks if relative chaining is used. Combinations of linear and chain storage have application and present interesting problems: chains of vectors and vectors of chains.

Acknowledgment

The work described in this paper was supported in part by The United States Atomic Energy Commission, Contract Number AT-(40-1)-2572 to the Rice Computer Project, Rice University, Houston, Texas.

References

Iliffe, J. K., "The Use of the Genie System in Numerical Calculations," Annual Review in Automatic Programming, Vol. II, Pergamon Press, 1961.

Iliffe, J. K. and Jodeit, Jane G., "A dynamic Storage Allocation Scheme," The Computer Journal, Vol. 5, No. 3, October, 1962.

Jodeit, Jane G., Letter to the Editor, Communications of the ACM, Vol. 5, No. 9, September, 1962.