A dynamic storage allocation scheme

By J. K. Iliffe and Jane G. Jodeit

This paper describes a system of semi-automatic storage control which has now been in operation on the Rice University computer for a period of 18 months. Experience in using the system reveals the possibility of a high level of automation in solving storage allocation problems with comparatively simple provisions in machine hardware. Further areas in which machine design may assist in solving these problems are suggested.

1. Introduction

The technique to be described in this paper is based on the use of *codewords*, which are essentially pieces of descriptive information held in the machine during the execution of a program. These provide the basis of communication between programs, and between programs and arrays of data. The use of similar control devices has increased significantly in recent years, notably with reference to commercially oriented systems, and that presented here is an integral part of an operating and compiling system (Iliffe, 1961). It is particularly convenient when it can be arranged that the information about arrays which is acquired during the compilation phase of a program can be carried over, essentially unchanged, to the execution stage in the form of codewords. This tendency towards blurring the hitherto sharp distinction between compilation and execution processes seems to be one of the most important aspects of the codeword technique, and it is capable of much further development than we shall be concerned with here.

Codewords are not, however, merely an adjunct of a compiling system. They have been used extensively with both "machine language" and "symbolic assembly" systems, whenever it was found convenient to leave storage allocation to the machine. Such a situation arises at first out of necessity, as when writing a program to operate on arrays of unknown length, and subsequently out of choice, since one quickly comes to realize that by leaving storage allocation to the machine the programmer wins a new degree of freedom in the mechanics of putting together separately prepared programs and data, as a means of applying a computer to a particular problem. Out of such mechanics have arisen also the "relocatable" and "relativized" forms of programs, the "common" region and "communication" list as means of passing information from one program to another, and the *loading routine*, which does immediately prior to execution what could not be done before in the way of storage allocation.

Until recently, "leaving it to the machine" has meant simply either stating maximum array sizes at compilation time or specifying array sizes by giving the values of certain parameters along with all the other information which is given to the machine at load time. The second method avoids wasting space, but makes for a more complicated loading routine. Such techniques deal inelegantly, if at all, however, with situations in which either (i) the total problem size exceeds the capacity of the available "main store," or (ii) the actual storage requirements depend on values derived during the course of the calculation, or (iii) two or more problems are to be scheduled to share the store at a given time. Such situations are important: a compiler is a set of programs to which all three conditions may pertain. A solution to these difficulties is to allow both the extent and position of elements of a program to vary, in the course of problem solution, at the discretion of the programmer, or the operating system, or of both: it is this process which we term "dynamic storage allocation."

We shall demonstrate that the codeword system combines the normal functions of a loading routine with the ability to allocate storage dynamically. It also provides the basis for a powerful operating system; it simplifies the indexing of arrays; and it provides for some elegant extensions of problem-oriented languages.

In the Rice University system, the technique is applied to scientific problems, in which fairly elaborate data structures are handled, but simple input-output controls are sufficient. The following three Sections describe the basic system of codewords, the allocation of storage space in a single-level store, and the method of operating on arrays. Section 5 introduces a technique, still in the experimental stage, for extending the apparent size of the main store by using a backing store with some automatic control over block transfers. Finally, the significance of the codeword technique in both hand coding and compiling is discussed, with particular reference to the page-address scheme of the Ferranti Atlas.

2. The codeword representation

Consider a store S of consecutively numbered locations in the range (L, H). A sequence of consecutive locations within S will be termed a *block* and identified by the pair $[F, N]$ giving $N(>0)$, the number of words or elements in the block, and F, the address of the first element.

A block may be *indexed* in which case the content (consisting of a signed integer or zero) of a particular register I is associated with the block during the addressing process which is described below. Given an indexed block, a *base b* may be associated with it, with the significance that in some external referencing system the first word of the block has *index number b*.

The block description is completed by specifying a *type t*, which is used by output procedures in determining the format of the block elements, and four numbers p, q, r, s, whose use will be given in detail below. In general terms, p is used to indicate whether or not the block is in main memory at a given instant, q indicates whether or not the block has a reserved (fixed) position in the backing store, r indicates whether or not the elements of the block are codewords referring to other blocks, and s is used in determining the activity of the block. We shall denote p, q, r, s, t collectively by the symbol θ.

The specification of (F, N, I, b, θ) in coded form requires something of the order of 48 bits, or one machine word on many contemporary machines. F and N are of the order of a full address length (15 bits) each; I is one of a small group of special-purpose index registers (modifiers); b is normally a small positive or negative integer or zero; t can occupy 2 or more bits, depending on local requirements; p, q, r and s require only 1 bit each. Table 1 shows the coding of the block specification on the Rice University computer, of which the 56-bit word length is more than sufficient, particularly when allowance is made for the extravagant coding of I and b.

A block specification thus represented in a machine location is termed a *codeword*; if this is stored at location B then it is possible without ambiguity to refer to the "block B," meaning that whose codeword is found in location B.

Assuming that the length of a single machine word is sufficient to contain a codeword, the possibility arises that the elements within a block B may themselves be codewords referring to a set of N "lower level" blocks $\{B_i\}$, $i = b, b+1, \ldots, b+N-1$. This situation is dealt with by setting $r = 1$ in any codeword referring to a block of codewords. In other cases $r = 0$, indicating that the block elements are single-word data representations. We shall refer to the structure consisting of a block B together with (possibly) a descending tree of sets of sub-blocks as the *array B*. The content of location B is the *principal* codeword of the array; any other codewords which may occur will be termed *auxiliary*. In practice, arrays with up to four distinct levels of codewords have been commonly used. Mixed blocks with both codeword and numerical elements are not normally permitted.

In detail, operations on arrays are defined in terms of operations on their individual elements. A definition of an array operation consists of a program which obtains access to array elements by indirect referencing through the codewords of each array. Let A be an array codeword. In symbolic coding practice the appearance of "A" as an address field of an order indicates that the contents of location "A" is to be used as an operand. In the present case we want to use A to specify a codeword, and then to use the information in the codeword to advance to an element in the array A. This is done in the following way: the given first word address F of the block A is modified, using the content (I) of the given index register I; the address so obtained is either that of an operand or of a codeword. In the former case the process terminates, and in the latter case it is applied again to the new codeword, as it was to A. One way of looking at this is to imagine stepping automatically through a list structure with N branches at each node instead of two, the actual branch taken depending on the value of a parameter I associated with the node and set prior to entering the addressing sequence. From this point of view, the sequence terminates when an atomic element of the list has been selected. In order to denote addressing of this type we shall use the notation "$*A$" and associate with it the application of the following Addressing Rule:

Rule 1: Let $A = (F, N, I, b, p, q, r, s, t)$ where the content of I is i.
Assume that $p = 0$ and $0 \leqslant i - b < N$. Then if $r = 0$ the operand selected by "$*A$" is in location $F - b + i$; if $r = 1$, the operand selected by "$*A$" is the same as that obtained by applying the Addressing Rule to the address "$*F - b + i$."

In more usual terminology, addressing is performed by iterative replacement through a sequence of codewords, with B-modification at each stage of the iteration. The iteration is terminated $(r = 0)$ when the address of an elementary operand has been found. The condition $p = 0$ assures that the block is in storage when Rule 1 is being applied; if $p = 1$, Rule 2 is applied, as described in Section 5. If the condition $0 \leqslant i - b < N$ is not satisfied; the operand of $*A$ is undefined: this means, in effect, that the location number of the selected block element is outside the permitted range $(F, F + N - 1)$. Because of Rule 1, it is usual to store $F' = F - b$ in the codeword A in place of F. If A is not an indexed

Table 1

Coding of the Block description on the Rice University computer

F, N	15 bits each
I	6 bits (selecting one index register out of six)
b	12 bits
t	2 bits, decoded as follows: 00: Octal Data, format # 1 01: 6-bit character strings 10: Octal Data format # 2 11: Decimal Data
p, q, r, s	1 bit each
(Spare)	2 bits

block, then by definition $b = i = 0$ and the effective address obtained at each stage of the iteration is F, so that it is not normally meaningful to have unindexed blocks appearing within an array structure, except as terminal elements.

It is important to emphasize the dynamic nature of the addressing process: in any application of Rule 1 the current contents of a series of index registers $I_1, I_2, \ldots, I_k$, determined by the codeword hierarchy of the array, are used. Programming through codewords involves presetting the contents of these registers to make the correct selection of array elements.

Example 1(a). An important example of an array structure is afforded by the representation of a rectangular matrix, whose elements, occupying single machine words, are stored by row in the machine. Let row i of the matrix be found in block $[F_i, N]$, with codeword $R_i = (F_i, N, J, 1, 0, q, 0, s, t)$ which indicates that the block is indexed by J, with external subscript range running from 1 to N. Let the row codewords be arranged in order in a block $[F, M]$ with codeword $C = (F, M, I, 1, 0, q, 1, s, t)$ indicating that the block is indexed by I, with external subscript range running from 1 to M. Consider an execution situation in which $(I) = i$, $(J) = j$ and the address "$*C$" is to be decoded by Rule 1. We then have "$*C$" replaced by "$*F - 1 + i$" (since $r = 1$). But $(F - 1 + i) = R_i$, by construction of the row codeword block, and "$*R_i$" gives the operand in address $F_i - 1 + j$ (since $r = 0$); the latter, however, is the address of the jth element of the ith row of the matrix, which we may denote by $C_{i,j}$ in accord with the conventional notation.

The importance of this result is in the fact that it provides a method of addressing a matrix element which is independent both of the actual position of the matrix in storage, and of its row and column sizes. All that is required during coding is a knowledge of the location of the principal codeword C, the determination of which is as simple a matter as locating single-word numerical operands. During execution, provided the codeword structure is maintained, the matrix can be placed anywhere in storage, and its position, or the position of some of its rows, may be changed to meet other storage requirements if desired; the block of auxiliary codewords C also has this property of mobility during execution.

Example 1(b). In the previous Example, we may have constructed codewords $C^T = (F, M, J, 1, \theta)$, $R_i^T = (F_i, N, I, 1, \theta_i)$, i.e. interchanging the roles of I and J. With the same initial conditions, addressing "$*C^T$" would now give the element $C_{j,i}$. In other words, constructing C^T and R_i^T "virtually" transposes the matrix, without altering the position of any of its elements in storage.

Example 1(c). Again referring to Example 1(a), it is fairly easy to see that if we apply a permutation operator $\mathscr{P}(i \rightarrow \pi(i))$ to the block C, then addressing $*C$ will

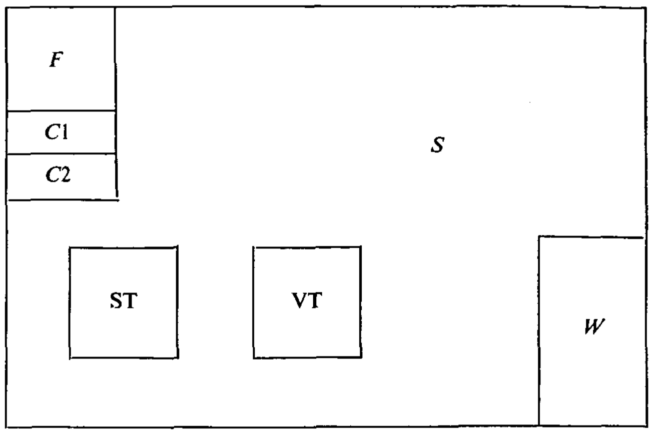


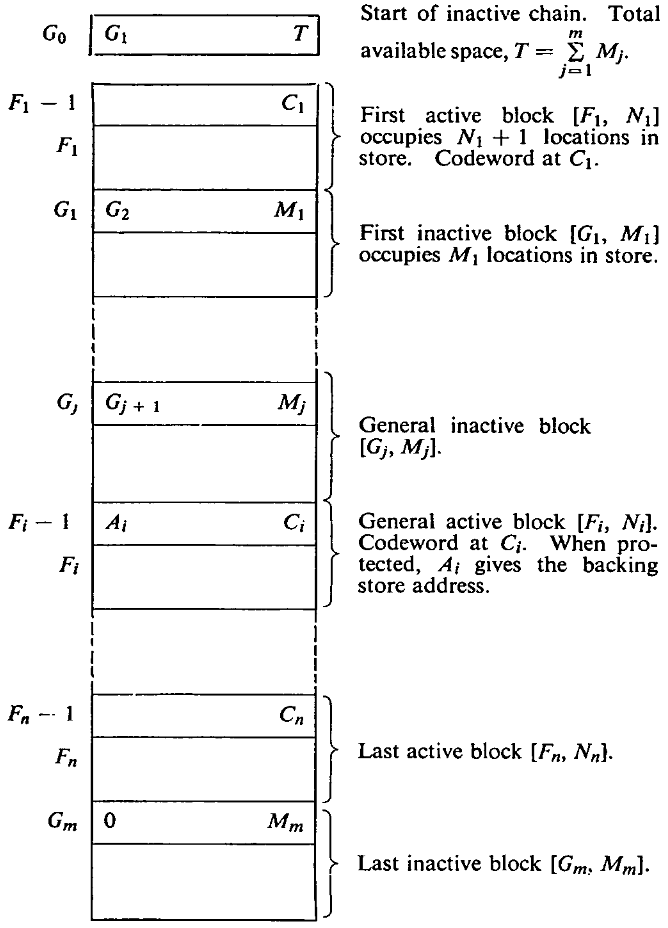
Fig. 1.—Main storage areas

give the element $C_{\pi(i),j}$ from the same initial conditions. A row permutation has been effected by rearranging the auxiliary codewords. It is not so easy to effect a column permutation. What might be done, given a permutation $\mathscr{R}(j \rightarrow \rho(j))$, is to set up a new vector Q, where $Q_k = \rho(k)$, $k = 1, 2, \ldots, N$. Let Q be indexed by the content of register K. Then $C_{i,\rho(j)}$ will be obtained by first setting $(K) = j$ and addressing $*Q$, thus obtaining $\rho(j)$, then setting $(J) = \rho(j)$ and addressing $*C$. This illustrates the general method of coding subscripted subscripts.

3. Memory organization

Fig. 1 shows the main divisions of the addressable (random-access) store in the codeword system for the Rice University computer. The region F denotes a set of fixed stores associated with the accumulators, trap addresses, and manual operating system. Region C is a basic "directory" of codewords referring to arrays in the remaining region S. C is divided into $C1$ which refers to the "system" arrays, and $C2$ which refers to arrays defined by the programmer and the more common library routines for elementary functions, matrix operations, etc. Thus the simplest way for a programmer to assign addresses to his array codewords is to place them in $C2$; once chosen they normally remain unchanged for the working life of a problem. The final region W is used as a common working storage push-down list by all routines: one index register is set aside to index the list. F, $C1$ and $C2$ contain 64 words each, W contains 128 words, and the remainder of the 8,192-word store is devoted to S. However, both $C2$ and W are readily extended in length if necessary.

Provision is also made for identifying programs and data by name by maintaining a table of symbols, ST, which identifies each named object in the machine at a given time. To each entry in ST corresponds an entry in the table of values, VT; for scalar quantities this entry will contain the value itself; for programs and arrays the corresponding principal codeword will be found in VT. At the start of a computing run, ST and



G_0
G_1
T
Start of inactive chain. Total available space, $T = \sum_{j=1}^{m} M_j$.
$F_1 - 1$
C_1
F_1
First active block $[F_1, N_1]$ occupies $N_1 + 1$ locations in store. Codeword at C_1.
G_1
G_2
M_1
First inactive block $[G_1, M_1]$ occupies M_1 locations in store.
G_j
G_{j+1}
M_j
General inactive block $[G_j, M_j]$.
$F_i - 1$
A_i
C_i
F_i
General active block $[F_i, N_i]$. Codeword at C_i. When protected, A_i gives the backing store address.
$F_n - 1$
C_n
F_n
Last active block $[F_n, N_n]$.
G_m
0
M_m
Last inactive block $[G_m, M_m]$.

Fig. 2.—Layout of available memory S

VT of suitable length are selected in S; they are themselves referenced through codewords in $C1$. At any time, the principal codeword for any array in the machine will be found in VT (if it is named), in C otherwise.

Consider a storage configuration requiring the use of n blocks in S, $[F_i, N_i]$, $i = 1, 2, \ldots n$, where the N_i are initially known, and F_i are to be determined. Assume that $\sum_{i=1}^{n} N_i \leqslant H - L + 1 - n$, so that the available storage $S(L, H)$ is adequate for the complete configuration. Then a method of selecting the F_i is simply given by the formula:

$$F_i = L + \sum_{j=1}^{i-1} N_j + i.$$

This is the rule adopted on initially loading S. It will be noted that blocks are loaded sequentially, and that an extra location is taken before each block, which is used to form a "back-reference" from each block pointing to its codeword C_i.

After the initial assignment, the remaining space in S is designated as an *inactive* block. As calculation proceeds, further blocks are taken from the inactive region, and in due course previously active blocks are returned to the inactive region. In a general situation, therefore, S is divided randomly into active and inactive areas as illustrated in Fig. 2. Let the inactive blocks be $[G_j, M_j]$, $j = 1, 2, \ldots, m$. Then to facilitate searching for space the first location of each inactive block is used to give its size M_j, and the location of the *next* inactive block in memory, G_{j+1}, thus forming a simple chain. The starting point of the chain is in a fixed location G_0 of the *C1* region, and it is terminated in G_m, for which the "next inactive block" is given address 0. G_0 also contains $T = \sum_{j=1}^{m} M_j$, the "total available" inactive store at any time.

Through the control program described in Section 4 requests are made for blocks of storage of arbitrary length, say E. Assuming $E + 1 \leqslant T$, i.e. that there is sufficient space to satisfy the request, three methods of taking space, of increasing complexity, are attempted.

Procedure S1. Starting at G_1, search is made through the G_j sequence to find the first M_j for which $E + 1 \leqslant M_j$. If some such region, say $[G_I, M_I]$ is found, the block $[G_I, E + 1]$ is activated, and a smaller inactive block $[G_I + E + 1, M_I - E - 1]$ is substituted for $[G_I, M_I]$ in the inactive chain.

Procedure S2. If *S1* fails, an attempt is made to find two or more adjacent inactive regions which can be combined to give the required block size. Thus, if there is a J for which $G_J + M_J = G_{J+1}$, then a new block $[G_J, M_J + M_{J+1}]$ can be formed and substituted in the chain for the previous $[G_J, M_J]$ and $[G_{J+1}, M_{J+1}]$. After combining inactive blocks, *S1* is reapplied.

Procedure S3. Finally, if no useful space is recovered by method *S2*, a complete reorganization of memory takes place, in which all active blocks are compressed to the low-address end, leaving a single inactive block of adequate size to meet the request. Let $[G_1, M_1]$ be the lowest-addressed inactive block in memory. Then it is either the last block (in which case the recovery process terminates), or it is followed by a block $[F_i, N_i]$. Assume $[F_i, N_i]$ is active (if it is inactive it can be combined with $[G_1, M_1]$ by the method in *S2*), and transfer the elements of $[F_i, N_i]$ to $[G_1, N_i]$, thus leaving the lowest-addressed inactive region at $[G_1 + N_i + 1, M_1]$. This process is repeated until there are no more active blocks to be moved.

In the process of moving an active block there are two adjustments which may have to be made in other parts of memory. Firstly the codeword referring to the block must be made to give the correct starting location. Secondly, if the block itself consists of a set of codewords, each back-reference from the lower-level block must be adjusted accordingly.

There are obviously certain restrictions on mobility which it is practical to make on a conventional machine, even though they could be removed in theory. In the first place, principal codewords in C are fixed, and referred to by fixed addresses in all programs. The codewords in VT are referenced by a type of transfer vector (see Fig. 3) in each program which is automatically set up by the loading routine. Hence it is not possible to allow VT to move once a set of programs has been loaded for execution.

Finally, there is the problem of maintaining correct link addresses through a series of subroutine transfers, in case one of the subroutines is moved before a return is made. To do this in complete generality requires the saving not of an absolute return address, but of sufficient indices to select the correct point of return to the higher-level program (which may itself be an element of an array). In effect, this means that subroutine entry and exit become small subroutines in those cases where a memory reorganization may take place in the lower-level routine. Since, in fact, only a minority of subroutines are of the latter type the general scheme of linkage has not been put into effect, and immobility is achieved by a simple device described in the next Section. In a machine making great demands on the store, facilities for this type of linkage would be a highly desirable feature of the hardware.

4. Operations on arrays

It is important conceptually to be able not only to address and manipulate individual elements of an array, but also to regard it as a whole, and perform what might be thought of as "macro-operations" on it. This facility is equally desirable with regard to programs which the programmer has written, and now wishes to regard as single operators in building up a larger system, as with the programs in the system library, or with the arrays on which they are intended to operate. Thus given a matrix inversion routine INV, and a matrix M, we wish to apply INV to M with the minimum of bother about where INV and M are, how big M is, and so on: these are details which the routines can find out for themselves, through the codewords. Under the generic title of "arrays" we include programs, vectors, matrices, and higher-order structures devised by the programmer to suit the ordering of his data, or the logical structure of his programs. The sort of operations which can be applied to arrays include purely numerical manipulations, transmission from one storage medium to another, and "growth" processes such as generation, extension, branching and decay: these operations may be applied either by the programmer in the course of his program, or by the operating system.

It is convenient to introduce a pseudo-code to describe operations on arrays or parts of arrays. This is essentially a generalized single-address code which is obeyed interpretively by part of the operating system. In the Rice University machine, each command in this pseudo-code is encoded in a single machine word which is termed a *control word*.

A "pseudo-address" is used to select an array, or a block, or a segment of a block. Thus a codeword address C is given, together with a pair of numbers (f, n), indicating that in the block C the block of elements with (external) indices $f, f+1, \ldots, f+n-1$, is to be the operand of the control word. It is convenient to set $(f, n) = (0, 0)$ if in fact the entire block C is the operand, for in this case the true values (b, N) can be obtained automatically from C. If, in C, $r = 1$ (indicating that C contains auxiliary codewords) then the operation can be applied either to the selected block, or it can be applied iteratively to each block referred to by the auxiliary codewords. Which alternative is taken is determined in the pseudo-code. For example, it is very useful to be able to print out codewords referring to an array, as well as the array itself, during testing procedures. In the "iterative" type of addressing, the same choice is made when any further arrays of codewords are encountered; thus it is possible to cause an operation to be performed on all elements of an array of arbitrary structure.

One of the principal operations on an array is the act of defining it. Pseudo-orders are provided to generate codewords and take space automatically for a vector or rectangular matrix or a program; this space may be left occupied by zero words, or having taken space, a READ operation may be used to fill it from paper tape punched in one of a variety of octal, hexad, or decimal formats. To define more elaborate structures, control words are provided which enable codewords to be addressed relative to the "zero external index" position of any block in the machine. This so-called "base selection" procedure works as follows.

(1) Under "base zero" conditions, any codeword address C which is used refers to the fixed machine address C.
(2) It is possible to select a block B, and indicate that thereafter any codeword address C will be taken to refer to that in machine location corresponding to $B_C = B_0 + C$. In these conditions, codeword referencing is relative to base B.
(3) Several successive block selections may be made, each one relative to the previously determined base, in order to reach a particular block in a complicated array. A control word is provided to "step back" to base zero conditions, or through a specified number of levels.

Example 2. Consider a matrix M whose elements are programs $M_{i,j}$. It is required to define program $M_{I,J}$ by reading it from paper tape. The necessary control words have the functions:

1. Select base M;
2. Select base I;
3. Read block J;
4. Return to base zero.

Apart from defining an array, operations are provided for erasing it or part of it (i.e. returning space to the inactive chain) and for dumping it on the backing store.

Given any block, it is also possible by means of control words to erase it in part, or to insert a segment of zero elements at either end or anywhere in the middle.

The remaining pseudo-operations are for the conventional functions of the operating system, namely making corrections, checking data, printing, punching, setting tags (or "flag bits"), and initiating the execution of programs. Control words are normally presented in their octal form, but provision is made for identifying arrays symbolically. No attempt has been made to put the complete pseudo-orders into mnemonic form, since they contain a large amount of information which would thereby become unnecessarily cumbersome, and the present format has proved quite adequate for both manual and tape-controlled operating procedures.

The "READ" operation is naturally the basis of the loading routine. Since all the system subroutines are themselves in array form, however, it is of interest to note the method of getting the system into operation. A short program is automatically read into the store, sufficient to reserve the regions F, C and W, and to be able to perform enough of the READ operation to load simple programs and vectors with codewords in C. The following programs define the basic parts of the interpretive scheme, and take space for ST and VT. At this point a control word is read which transfers the loading operation to the general part of the interpretive scheme, with the initial storage allocation scheme outlined in the last Section. This enables symbolically defined data to be loaded, and appropriate cross-references to symbolic data to be placed in each program, making further use of address replacement.

It should be remarked that each program is executed in a relativized form, so it is relocatable, without alteration, to any part of the store. References to a principal codeword in VT are made through a word at the end of the program into which the appropriate VT address is inserted by the loading routine. Thus the program remains relocatable; when it is moved, only its codeword in VT or C has to be changed (see Fig. 3).

In order to avoid the complications of subroutine linkage introduced by the reorganization scheme ($S3$) it is arranged on loading that all programs which may call, directly or indirectly, on $S3$, are placed at the low-address end of the available store. After these have been loaded, a control word is used to indicate that all blocks subsequently defined may be freely moved by the storage allocation program. This system works tolerably well, since many programs and all data arrays are not so restricted. As remarked earlier, the restriction can be removed at the cost of lengthening subroutine entry and exit procedures.

5. The use of the backing store

The system so far described works independently of any supplementary storage which may be available apart from the main memory. The existence of a backing store makes it possible to obtain an effective "single-level store" of much greater extent than the main memory region, through an extension of the use of codewords. Our work in this direction has been restricted by the use of magnetic tape as a backing storage medium, which has obvious disadvantages. It seems more useful to consider a tentative way in which a more suitable medium may be employed.

A backing store with the access time of magnetic drums is required in an efficient system. To match the main memory organization it must also be capable of accepting and transmitting a block $[F, N]$ with arbitrary F and N, identified by an address A. When a block is in the backing store, its address A is placed in its codeword in place of F. At the same time, the bit p is set equal to 1.

During the addressing process (Rule 1) it was assumed that $p = 0$. The condition $p = 1$ causes an interruption of the addressing sequence, and transfer of control to a routine which will obtain the block, if it exists in the backing store, and restart the main calculation from the point of interruption. If a complicated array is initially held entirely in the backing store, several interruptions may take place in a single addressing sequence in order to load blocks of auxiliary codewords to the main memory prior to obtaining the required array elements.

In the total extent of a calculation, a given block may be defined for all or part of it; when it is defined it will spend some time in the backing store and the remainder in main memory. It is assigned no fixed position in either store. In particular, it may vary in size dynamically, so that any position it occupies in the backing store must be relinquished when it is brought into main memory. Using magnetic tapes, this means that blocks are stored in an irregular sequence, with gaps which are difficult to use up; transfers from tape leave more gaps, and transfers to tape are made to the end of the sequence. A "recovery" procedure, involving writing out all arrays on to a new tape and then writing them back into the backing store, is possible but highly inefficient. Again the availability of drum storage with short addressable blocks would offer considerable gains in efficiency.

There is an important subset of arrays, notably most library and system programs and certain tables, which never change their content, and hence may be left in permanent positions in the backing store, without need to write them up from main memory after they have been used. These are the "protected" arrays in which the digit $q = 1$ in the codeword. For unprotected arrays, $q = 0$. If $[F, N]$ is a protected block in S with codeword C, then location $F - 1$ will contain in addition to C the address A of the block in backing store.

Given an unprotected block $[F, N]$ in S, it may be "erased" by referring to C and setting $F = N = 0$. At the same time, $[F, N]$ is added to the inactive chain of store. If $[F, N]$ is protected, it is "erased" by referring to C and setting $F = A$, $p = 1$, and inactivating $[F, N]$ in storage.

Given an unprotected block at address A of the backing store, it is erased by referring to the codeword and setting $F = N = 0$. It is impossible to operate on a block whose codeword is not in main memory. In particular, it is impossible to load it into main memory. It is possible, however, to erase a block while it is in backing store, since this involves an operation on the codeword only. In the magnetic-tape system, block A is not available for use subsequently (unless a recovery operation takes place); in a more practical backing-store arrangement, the storage used by A would immediately become available for re-use.

Thus the operations which are available to the coder through the use of control words applied to an array $C = (F, N, I, b, p, q, r, s, t)$ are:

ERASE (C) Clear C from the single-level store, making its space inactive, unless it is protected, in which case it is only erased from main memory. In detail:

If $q = r = p = 0$, erase C. If $q = r = 0$ and $p = 1$ erase C from backing store. If $q = 0$ and $r = 1$ and $p = 0$, apply ERASE to all elements of block C and then apply it to C. If $q = 0$ and $r = p = 1$, fetch C to main memory then apply ERASE. If $q = 1$ and $p = 0$, apply ERASE as for $q = 0$. If $q = 1$ and $p = 1$, no action is required.

DUMP (C): If C is in main memory and unprotected, write it in the backing store and erase from main memory. If C is in main memory and protected, erase it from main memory. Otherwise, no action is required. Details follow similar lines to ERASE.

Either of these commands requires as parameter the address of the codeword C, which may be principal or auxiliary.

It is also necessary to provide an explicit command for loading an array into main memory. By using this in an anticipatory fashion, the coder may obtain a faster-running program, although if he fails to do so the interrupt procedure will automatically load the required blocks.

FETCH (C): If $p = 0$, ignore. If $p = 1$, $r = 0$, load C into S from backing store. If $p = 1$, $r = 1$, load C into S from backing store and then apply FETCH to each element of the block C.

It should be noted that by explicitly using the FETCH and DUMP orders the coder initiates a complete transfer of the array C to or from the backing store, whereas the automatically generated transfers will be seen to concern only single blocks. The reason for this is that calculations often involve only one or two blocks at a time from a given array, and it is important to avoid loading the main memory with unrequired array elements. On the other hand, on the assumption that the coder's anticipation is accurate, there is no loss of flexibility in allowing FETCH and DUMP to work iteratively through an array.

Provision must be made for the eventual situation in which a block is requested by referring to a codeword with $p = 1$, or through a control word, but the total available inactive space T is insufficient to satisfy the demand. It must therefore be met by placing one or more presently active blocks in the backing store. In order to minimize the use of data transfers, two criteria must be established: (*a*) the method of selecting an active block for transfer to the backing store, and (*b*) the rules for combining (*a*) with the search procedures *S1*, *S2*, *S3* described in Section 4.

For example, given a technique for placing a "priority value" π_i on each active block $[F_i, N_i]$, $i = 1, 2, \ldots, n$, which in some way approximates to the chance of it being used in the immediately ensuing calculation, it does not follow that the block with least priority is the most suitable for disposal. The decision must be related also to the amount of space E which is requested, and the position and size of each block.

It must also be assumed that the use of backing storage in the form of tape or drums or both involves the existence of perhaps several autonomous transfers taking place between main memory blocks and peripheral devices at a given time. This possibility makes the application of *S3* impossible except in special circumstances. We are also faced with the possibility, in reconsidering subroutine linkage, that a program may not only move around in the main store, but also be dumped, so that on returning to it from a lower level subroutine it may have to be recovered from the backing store.

A recovery procedure U for obtaining an inactive block of length E in main memory will now be described. It involves a search through arrays, which is conducted by starting at a principal codeword in C or VT, and passing down the array hierarchy to the lowest level blocks ($r = 0$), thence working up to the principal codeword, examining each block which is encountered. C and VT are scanned cyclically in repeated applications of U, i.e. one search is satisfied by a block in array B, then the next search is started at the array following B in C (or VT). In each codeword which is examined in the search, the digit s is set equal to 1. If a block is unused between two searches, s will remain equal to 1; otherwise it will have been set to 0.

Procedure U1. For each $[F_i, N_i]$, if $p = 0$, $q = 1$ and $s = 1$ the block is erased. If, as a result of this, a block of sufficient length becomes available, the search is terminated immediately.

Procedure U2. If *U1* fails a second search is made, dumping each block for which $p = 0$, $q = 0$ and $s = 1$, again terminating the search if a block of sufficient size is inactivated.

Procedure U3. If *U2* fails, the blocks which remain ($p = 0$) are all those for which originally $s = 0$. In all these now $s = 1$. Hence *U1* and *U2* are repeated; this time the search is bound to terminate, provided E is within the capacity $H - L + 1$ of S.

It is now possible to give an improved version of the Addressing Rule.

Rule 2. Let $A = (F, N, I, b, p, q, r, s, t)$, where the content of I is i.

(i) If $s = 1$, set $s = 0$.

(ii) If not $0 \leqslant i - b < N$, exit to an error routine.

(iii) If $p = 0$, proceed as in Rule 1.

(iv) ($p = 1$). Request a block of N inactive words through the operating system (this will be satisfied by first searching the $[G, M]$ chain, by $S1$ and $S2$ and then applying U if this fails). F is now a backing-store address. Transfer the block from backing store to main memory setting $F =$ initial address, and $s = p = 0$. Now proceed as in Rule 1.

The use of magnetic-tape storage in a conventional sense (as opposed to employing it as backing storage) is also provided for through codeword control. Assume that facilities exist for variable-length working, and that any block of information on tape can be identified by an address $A^{(T)}$. Then the process of writing an unprotected block $[F, N]$ to tape is accompanied by replacing F by $A^{(T)}$ in the block codeword, and setting $p = 1$. (It is not normally meaningful to write protected blocks.) The recovery operation FETCH (C) applied to a codeword containing an address $A^{(T)}$ is sufficient to enable the block C to be located on tape and transferred to main memory. Unless it is protected, it will at that point become undefined on tape; a block may, however, be written to tape, and then protected (setting $q = 1$), and thereafter held permanently on tape as opposed to the backing store.

To enable buffered tape-processing operations to take place it is convenient to provide one or more blocks in the permanent part of main memory into which blocks from tape can be read. This is not quite the same as the FETCH operation of the previous paragraph, since (i) the blocks on tape are essentially anonymous, and (ii) the main memory block is reserved permanently, so that there is never any need to enter the Search procedure. The operator FILL (C, n) causes block C in main memory to be loaded from the next block on tape number n; EMPTY (C, n) writes the content of C on to tape number n. In each case the size of the block on tape is equal to the length of C.

In writing an array to tape the elements of the array must be accompanied by blocks of auxiliary codewords. Since auxiliary codewords cannot be written until all lower-level blocks have been written, the information placed on tape would be very awkwardly arranged for reading back unless special provisions were made. In fact, the block addresses to which lower-level blocks will be written are anticipated, and information is stored in the correct order for reading back.

Finally, regarding VT as an array, it is possible to write on to tape all symbolically defined information in the machine at a given time. This is done by a special operation, Write Definition Set, WDS (C, n), where C gives the name to the set of definitions which has been written on to tape number n; at the same time, all the corresponding ST entries are written as a block on tape. By means of the reverse operation, Read Definition Set, RDS (C), a previously existing machine state can be re-established, or a new state can be built up out of a combination of sets of definitions.

6. Programming techniques

There are evidently a number of consequences of the codeword addressing system which affect the human as much as the automatic coder. The most notable advantage offered is flexibility in handling arrays and making cross-references between them. Second to this is the basis of a dynamic storage allocation scheme, although it will not be forgotten that other solutions to this problem, in particular the page-address system (Fotheringham, 1961), are possible. It is interesting to contrast the relative merits of what may be termed the machine-oriented page-address system with the problem-oriented codeword system, to see what advantages emerge in any total design.

In the first place it should be said that there is no inherent preference on the part of a machine user between problem- and machine-oriented storage control schemes, since both are intended to remove the same problem from his sphere of influence without substituting anything for it. We would also regard the actions of the more sophisticated programmer who was willing to insert explicit commands in his code of the form "(Fetch/Dump) (Block/Page) X" as roughly equivalent in both cases. The main advantage of the page-address system is in the simplicity of uniform page storage in both main store and backing store, and in the absence of time-consuming search and recovery procedures. Against this can be put the suggestion that by close packing the codeword system allows fuller use to be made of the main memory; it also has some advantage in handling blocks exactly corresponding in size to the pieces of data and programs being used and which, moreover, can continually be adjusted in size to meet new problem requirements: exactly what weight can be attached to this advantage is rather a subjective matter.

It should be noted that reference to data through a codeword is only made when the data is part of an array; the existence of VT serves to distinguish all the common external names of a set of routines from the private internal quantities, whose values are stored within the separate programs, and referenced directly. This remark also applies to transfers of control within a program. An analysis of a selection of existing programs for the Rice University computer showed that references to data through codewords amounted to 10–15% of total data references. Thus, the address comparison which is an essential part of the page-address system is avoided in a high proportion of store references when codewords are used. There are no lock-outs on the

Rice computer, but errors arising from the omission of a check at this point are extremely rare, particularly as nearly all programs are assembled or compiled from a symbolic external language in which references outside the program area, other than through VT or C, are impossible. The most likely source of error in an address is through an index value of an array running wild, but since by construction of the codeword system all array references are through VT or C, it is feasible to make a check at this point to see that the bounds of the array have not been exceeded. (This is done in the B-5000 Descriptor System (Burroughs, 1961).) Evidently, such a check is even stronger than the page lock-out scheme, and one of a number of less rigorous and less complicated schemes could be accepted.

An advantage of the codeword scheme is that elements of arrays of two or more dimensions can be addressed with "true" index values in the index registers, rather than having to compute the value of an address function. Many benefits to automatic coding are thereby purchased. The price of this, of course, is that two or more memory cycles may be taken (applying Rule 1) where traditional methods would require one, providing the compiler were sufficiently adept at loading and incrementing modifiers. Little has been done so far to mitigate this effect in compiled programs, but a number of steps can be taken in hand-coded routines to regain lost efficiency. Consider, for example, a routine R operating on a block $[F, N]$ specified by a codeword C whose address is given to R as a parameter. Provided the main computation in R makes no demands for extra storage (which may involve re-allocation) it is permissible to extract F and N from C and use these as the true parameters, as in conventional coding. A second device is available on a machine with a set of stores with appreciably faster access time than the main memory: any codeword which is frequently being referenced is brought to one of these for the duration of that routine, or for parts of it.

Finally, given a well-defined set of arrays constituting a complete method of solution for a given problem, it is possible to fix at least a subset of these in main memory and replace cross-references through VT by direct references to blocks, thereby reducing by one the number of iterations in applying Rule 1 in many cases.

A useful side-effect of the automatic facilities for applying Rule 1 is the ability to set up communications between data and programs generated by a subroutine-type compiling system. Fig. 3 illustrates a memory scheme containing two programs P, Q, a vector R, and a symbolically-identified scalar value S. P makes reference to Q and R, and Q makes reference to S. The program P is followed by two words (a, $a + 1$) in which the addresses of the codewords of Q and R are entered on loading P. These addresses are qc, rc respectively. Since Q and R are non-scalar, replacement bits, indicated by *, are also inserted. No changes are made in the program. Similarly, when Q is loaded, the address sc is inserted in the word (b) which is at the end of Q,

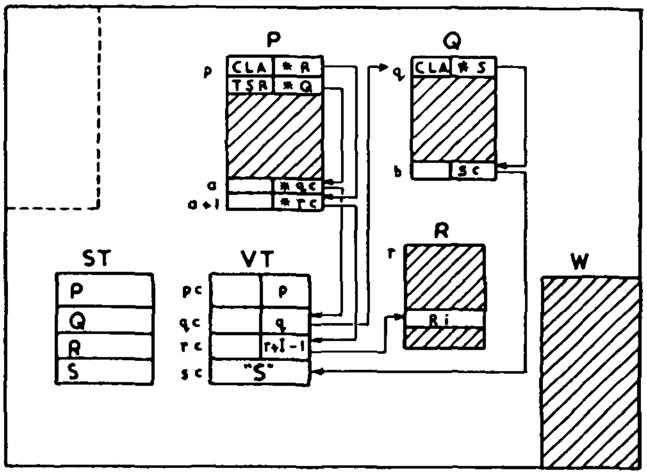


Fig. 3.—Communication through VT (External variables)
[Mnemonic operations: CLA = Clear and Add
TSR = Transfer to Subroutine]

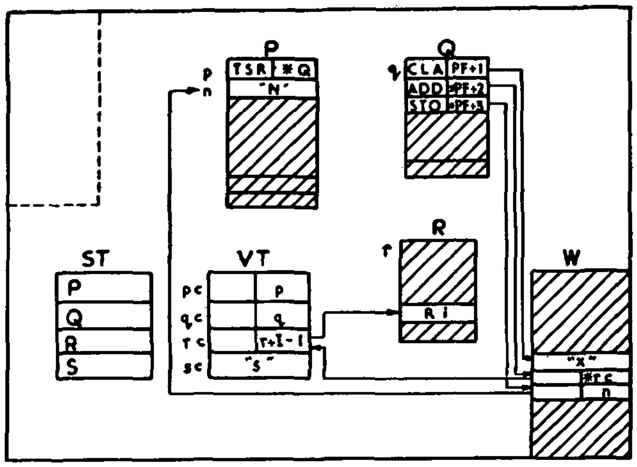


Fig. 4.—Communication through W (Parameters)

but since S is a scalar, no * bit is inserted. The words thus formed act as "partial" codewords in forming links in replacement chains through which correct data references may be established. The paths of the chains are indicated by the arrows in Fig. 3. Thus the reference at p in P passes through $a + 1$ (indirect relative address) to rc (indirect absolute address) to $r + (I) - 1$ (direct absolute indexed address). If it is necessary to move R, only (rc) needs to be changed. No problem arises if an array varies in size or structure in the course of a run, or from problem to problem.

By analogy with the above method of referencing the external variables of a program, those of its parameters which are called by name can be handled through a list of addresses which is built up in the W-region by the calling routine prior to entering a subroutine. For parameters called by value, which we restrict to being scalar quantities, the value is stored in the W-list and the * bit is omitted from the address in the program. Fig. 4 illustrates the use of program Q which has one parameter called by value, and two by name. In the particular configuration given, the value of a quantity x has been stored in the W-list, together with the address of a vector R whose name appears in ST, and the address of a scalar parameter N which is to be found internal to the program P. The quantity PF is a pointer set by the calling routine and fixed throughout the execution of Q. It is used by Q as a reference point in addressing parameters. Arrows point out the paths taken in the replacement chain.

Detection of the state $p = 1$ in a codeword can also be the means of locking out arrays during transfer operations, or of obtaining operands defined in a number of unconventional ways. For example, elements of a matrix may be defined by means of an algorithm rather than by an explicit array; or a variable may be defined formally in terms of others, and require to be evaluated interpretively.

It need hardly be added, finally, that the unique correspondence between codewords and the data they reference makes it possible to provide information to the coder which he would otherwise have to make special provision to obtain. For example, in the algebraic language such functions as ROW ($\# M$), COL ($\# M$), applied to a matrix M, and LENGTH ($\# V$) applied to a vector V, are used to obtain the sizes of any arrays. By executing the program VSPACE ($\# U$, a, b) a vector U of $b - a + 1$ elements is defined, with external subscripts ranging from a to b; any previous definition of U is erased, and a pre-assigned B-register is used to index U. The ability to include matrix and vector operations within the algebraic formula language is very simply gained.

It can be seen that a large number of attractive facilities are provided through the use of a codeword addressing scheme. Its disadvantages are encountered when heavy demands are made on the main memory space by blocks of varying size, as, for example, may be the case in a large-scale scheduling system. Thus a demand for a block of size E, currently in backing store, leads to the successive operations:

1. search the $[G_j, M_j]$ chain for a suitably-sized block E,
2. if no block can be found, apply the recovery procedure U,
3. transfer the required block to main memory.

In general, none of these three operations can be overlapped, in contrast to the page-address scheme in which a spare page is always kept on hand in main memory, and the transfer from backing store can take place while a search is being made for a new spare page. Hence the frequency with which the recovery procedure has to be applied is a critical parameter in assessing the performance of the codeword system on a particular machine. On a large machine, on which recovery time becomes vital, it might be feasible to include a scavenging program of low priority in the list of programs under execution, automatically increasing its priority when the available inactive space fell below a critical level.

7. Acknowledgement

The work described in this paper was supported in part by grants from the National Science Foundation, G-7648, and from the National Aeronautics and Space Administration, NsG-6-59. Construction of the Rice University Computer was sponsored by the Atomic Energy Commission under grant AT-(40-1)-1825.

References

Burroughs Corporation, "The Descriptor," 1961.

Fotheringham, J. A. (1961). "Dynamic Storage Allocation in the Atlas Computer Including an Automatic use of the Backing Store," *Communications of the A.C.M.*, Vol. 4, p. 435.

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