Storage Organization in Programming Systems

Jane G. Jodeit, Rice University, Houston, Texas

This paper describes the system of program and data representation that has been in use on the Rice University Computer for five years. Each logical entity in storage occupies a block of consecutive memory locations. Each block is labelled by a codeword and may contain a program, a data vector, or codewords which in turn label blocks to form arrays. This storage arrangement is discussed with its realized advantages in programming systems: simplicity of programmed addressing, flexibility of data structures, efficiency of memory utilization, variability of system composition during execution, means of linkage between programs and from programs to data, and basis for storage protection. The application of labelled blocks may be extended to areas of time-sharing and multi-media storage control. On the basis of experience at Rice, some ideas on such extensions are presented.

Jane G. Jodeit, Rice University, Houston, Texas

Introduction

This paper describes a representation of data and programs in storage that contributes organizational simplicity, coding convenience, and functional versatility in programming systems. Here programming system means the realization of a problem solution on a computer, anything from mathematical analysis to language translation.

A problem solution is defined by a collection of entities, programs and data items specifically. The generic term for such an entity is an array. Each array is named and contains as elements data (which may be the instructions of a program) or sub-arrays. In a programming system for the Rice Computer the elements of an array form a block, a set of consecutive memory locations which has been called a "segment" [3]. Each block is labelled by a codeword, a word which corresponds to the name of the array whose elements occupy the block. If A is an array, the i^{th} element of A is designated (A,i). If the elements of A are sub-arrays, the i^{th} word in the block for A is a codeword which labels the array (A,i).

Thus, an array is a tree structure. The name is the source from which the elements branch. If the elements are arrays, they in turn branch; if the elements are data, they are terminal. A source of branches is represented by a codeword; the set of

branches from a single source is represented by a block containing codewords or data as appropriate.

A codeword which corresponds to a simple name, as A above, but not (A,i), is called a primary codeword. All sub-arrays and data elements of an array are addressed "relative" to the simple name. This just means that the m^{th} element of the n^{th} sub-array in the array DATA is named (DATA,n,m); it has no other designation. The set of primary codewords then completely catalogs the entities of a programming system and all addressing is done through these codewords. The operating system provides dynamic allocation of blocks and maintenance of codewords. Primary codewords never move, and the addressing is independent of system composition and storage allocation.

Jane G. Jodeit, Rice University, Houston, Texas

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 corresponds to the name of the block; it 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:

(Figure 1. here)

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

I is the relative address of the first word in the block labelled by C

X is on if the block labelled by C contains codewords

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

K is present if the block labelled by C is indexed, i.e., if individual words in the block are to be addressed from outside the block; K then specifies the index register used to give the relative address of a

word in the block (Data vectors are indexed; programs are not.)

F is an address associated with the block labelled by C so that the address of the first word in the block is $F' = F + I$

The portion of a codeword used in indirect addressing is 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 eight registers may be specified for each step. 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 proceeds 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 on, return to step (1) for codeword C^{i+1} at level i+1.

If $*^i$ is not on, 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 portion

$*^{O}$, 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.

The full generality of codewords can be implemented with maximal efficiency with such hardware. It is surprising that more computers do not employ such an indirect addressing definition or some equivalent addressing mechanism. With more restrictive hardware capabilities 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 labelled blocks are used to build the elements of a programming system.

In general, any "named" item in the codeword system is called an array. On the highest level, that addressed in code, is a single codeword which corresponds to the name of the array and labels a block which may contain codewords. On the lowest level is the data of the array. The intermediate levels are formed by blocks of codewords, the structure of the array.

The array forms most frequently encountered will be discussed in detail.

A program P may be considered as a set of words to be executed as instructions and should, for efficient control hardware utilization, occupy consecutive storage locations. Thus, a program P occupies a memory block. Assume a single entry point to P; then the block for P need not be indexed because only one word need be addressed from other programs. If P is of length k words with p words of linkage information, the program and its codeword appear as

(Figure 2. here)

Control is transferred to program P by the single operation:

transfer control to $*C_P$

where * specifies indirect addressing through C_P, the codeword for P. A single step of indirect addressing is performed:

$$*C_P \xrightarrow[\text{addressing}]{\text{indirect}} F+p$$

and the final address obtained is F+p, the address of the first word of code for program P. The address F+p never appears in code, only in the codeword for P. The address C_P which appears in all coded references to the program is invariant while the address 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 addressed by their relative position in the set and should, for efficient index hardware utilization, occupy consecutive storage locations. Thus, the vector V occupies 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

(Figure 3. here)

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

(1) set index register i to p

(2) access $*C_V$

where * specifies indirect addressing through C_V, the codeword for V. A single step of indirect addressing is performed in step (2):

$$*C_V \xrightarrow[\text{addressing}]{\text{indirect}} p+F-1$$

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-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, may be realized as 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 occupies m indexed memory blocks (one per row) of n data words each, and the codewords for the rows occupy an indexed memory block of m codewords. 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

(Figure 4. here)

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 $*C_M$

where * specifies indirect addressing through C_M, the codeword for M. Two steps of indirect addressing are performed in step (3):

$$*C_M \xrightarrow[\text{addressing}]{\text{indirect}} *(p+F-1) \xrightarrow[\text{addressing}]{\text{indirect}} q+G_p-1$$

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 elements of M is dependent only on the invariant highest level

(primary) 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, array definition is extremely flexible in the codeword system, so this organizational form lends itself naturally to a very large variety of computer problem descriptions.

If A is an array, the elements A_i are all data or all arrays. If A_i are data, A is one-dimensional (as programs and vectors described earlier). If A_i are arrays, they are just sub-arrays of A; any array A_i may be defined or undefined at any time. The dimension and size of any A_i is independent of all others. The same rules of definition apply for arrays A_i as for A.

Thus, a matrix may have rows of unequal length, as in the case of a triangular matrix, or only a subset of its rows defined at any time. An array of programs may be defined. This has been useful in compilation at Rice where on the basis of three integer values a code-generating routine is selected; not all triads are meaningful, so the array of code-generating programs is sparse. Programs may be inserted when new triads are defined, and any program may be modified and replaced without effect on others.

Jane G. Jodeit, Rice University, Houston, Texas

Codeword Location and Reference by Programs

The organization of the codeword system provides parallel tables with one entry per named item:

- symbol table (ST) containing names of items, and
- value table (VT) containing values of scalars and codewords for non-scalars (arrays).

The address of the VT entry for an item named A will be denoted VT_A. If A is a scalar, it is addressed at VT_A during execution. If A is a non-scalar, it is addressed indirectly through its codeword at VT_A. The location of VT and the order of VT entries is a function of system composition. So a coded reference to an item named A is made indirectly through a linkage word L_A in the program:

transfer control to $*L_A$

or access $*L_A$

Loading of the program provides the address VT_A in the linkage word L_A; * is on in L_A only if A is a non-scalar. The first indirect addressing operation then provides:

$$*L_A \xrightarrow{\text{indirect addressing}} \begin{cases} VT_A, \text{ for scalar A} \\ *VT_A, \text{ for non-scalar A} \end{cases}$$

Subsequent addressing is just as if VT_A had been addressed initially.

These linkages are illustrated in the following diagram by the program P which references scalar SCALR and non-scalar ARRAY.

(Figure 5. here)

The linkages discussed thus far have been for fixed references, the name for an item being known at coding time. Programs may also reference parameters which take on value assignments at each execution. Linkages to parameters are for variable references.

Parameters are provided to a program on a push-down list W. The first free space in W is maintained during execution as a pointer WP. A parameter reference is coded indirectly through a linkage word located in W at a fixed position relative to the value of WP upon entry to the program. Index register P is set to the value of WP initially in each execution, and the linkage word for parameter A is located in W at $W_A=(P)+P_A$ where P_A is constant for all executions. Program reference to parameter A is accomplished by indirect addressing through W_A:

transfer control to $*((P)+P_A)$

or access $*((P)+P_A)$

which may be written

transfer control to $*W_A$

or access $*W_A$

Linkage for parameters and content of linkage words in W are illustrated in the following diagram by program R which provides scalar SCALR and non-scalar ARRAY as parameters to program Q.

(Figure 6. here)

One further case must be considered. If PAR is a parameter in program R and R must execute Q with PAR as a parameter, R very simply copies its linkage word W_{PAR} into W in the list of parameter linkages prepared for Q. Then, parameters may be passed to any level of program nesting during execution.

Jane G. Jodeit, Rice University, Houston, Texas

Dynamic Storage Allocation

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

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

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

activation, or creation, of a memory block labelled by a codeword, and

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 will be sequentially located in the storage domain. As a run progresses blocks may be inactivated and new ones activated, 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 of codewords. Each active block is headed by a back-reference word which contains the codeword address for the block.

Each inactive block in the storage domain contains in its first word its length. One inactive block is used as the source of space for activations. This source is initially the whole domain. When the source is exhausted, active blocks are packed at one end of the domain, and the resulting single inactive block is designated as the source. This packing procedure is called reorganization. The memory of the Rice Computer is not paged; if it were, reorganization would be effected by packing the page table [1,4]. Notice that with paging some storage economy is sacrificed since two blocks should not occupy the same page.

Each dynamic allocation request is specified 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 of all blocks in the array labelled. Each block inactivated has its codeword cleared to signify that it no longer labels an active block.

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.

Operations on Arrays

There are many useful operations on arrays that are easily implemented in the codeword system. Already the storage control operations of block creation to form arrays and freeing of arrays have been mentioned.

Mathematical operations on data arrays are familiar, such as transposition of a matrix or cross-correlation of two vectors. A routine to perform such an operation receives the name of an operand, i.e., its codeword address, as an argument. The routine then has access to the codeword for the array as well as the array elements. Information such as dimension, size, and natural array indices are available without being given explicitly as arguments.

The codeword system provides a file structure very much like that described for secondary storage organization in the MULTICS system [2]. The implementation on the Rice Computer provides a representation in primary storage which is immediately applicable through a hierarchy of storage devices. The same information which facilitates addressing and system component linkage is used by the operating system for file handling functions such as input-output, execution, insertion and deletion, and establishment of paths to file elements. The same notations, or naming conventions, are used in designating file manipulations and in the description

of data processing. Also, the file-level operations may be carried out from the console, as an operation quite independent of program execution, or from a program as it runs.

Jane G. Jodeit, Rice University, Houston, Texas

Memory Protection

Interest in multiprogramming and time-sharing computer applications has focused considerable attention on the problem of memory protection [3,4,9]. The objective has been mainly to prevent each memory resident from interfering with all others. Codewords provide the basis for a logical protection scheme, one that insures that no memory references violate the block definitions of the running system. This scheme differs from those which provide protection per page of memory. If vector V is defined to contain elements $V_1, V_2, \ldots, V_5$, logical protection will prevent reference to V_6; physical, or page, protection will prevent this reference only if the word after V_5 lies in a different page and that page is unallowed to the program generating the reference.

The first premise for logical protection with codewords is that all memory references from a program to blocks outside itself are through primary codewords. This is not an unreasonable requirement; it is not different from the requirement that separate entities be given distinct segment numbers [3,4]. Then, for each codeword in the indirect addressing chain which labels an indexed block, the index value k is checked to see that

$$I \leq k < I - L.$$

I and L are given in the codeword and are the relative address

of the first word in the block and the length of the block respectively. This checking can be implemented in the hardware and is planned for the Rice Computer at no loss in memory speed. Logical protection is now implemented in software at Rice*; because it is slow, it is used only for debugging.

To prevent a user from using a codeword which labels an array which is not his requires a notation in (or on) the codeword which has not been included in the earlier definitions. One bit would suffice; it would be maintained as execution switches from user to user because it would appear only on the small set of primary codewords for the user in control. Alternatively, a field could contain a user number which would not change while his system was resident in memory. Shared data would have a codeword for each user allowed access.

*The Rice Computer hardware provides two tag bits per word [8] which are not part of the data content but are used for control. Codewords are "marked" with a tag value which cause a trap out of the indirect addressing chain to a service routine. The service routine checks the validity of index values on the basis of the content of codewords in the chain.

Extensions

The codeword system for the Rice Computer provides organization and control of primary storage for a single user. The restrictions of this particular implementation are not imposed by inadequacies of the theory. The descriptive properties of codewords, the modularity of array storage, and the protection potential in the system allow the codeword storage organization to be applied in a multiprogramming environment. Interrupt logic in the hardware and adequate secondary storage media would be essential. Hardware features for codeword recognition and special actions due to particular codeword content are suggested.

The design of a codeword system to serve more than one memory user at a time would require that each user have his own table of codewords (Value Table described earlier). Each table would be an element of an array which would catalog the systems of all users. Shared entities would have codewords in several tables, or a collection of users would be permitted access through some tables.

Immediately, the allocation of primary storage involves overlay and automatic retrieval from secondary storage. As in the B8500 system [9], codewords may be marked when the block labelled is not available; an interrupt would allow intervention

for retrieval. When a block is not in memory its codeword may be used to designate where it is. Codewords may be used for the collection of usage statistics [5] to aid in the decision about what to overlay. Dynamic demand would determine which blocks were in memory at any time; not all arrays or all of any array for a running system need be present.

Structured arrays have been designed for secondary storage files [2]. This has been done at Rice with no representational difficulties, but only on magnetic tape, which is a poor medium for the application. It has been proposed at Rice that the device which controls transmission between primary and secondary storage would recognize codewords; it would set codeword and back-reference content to properly define an array in the storage to which it is being transmitted. Thus, a single command would suffice to move an entire array to or from memory; buffering and processor control would be minimized.

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.

1. Comfort, Webb T. A computer system design for user service. AFIPS Conference Proceedings, Vol. 27, 1965 F.J.C.C., Spartan Books, Washington D.C., 1965, p. 619.

2. Daley, R.C. and Neumann, P.G. A general-purpose file structure for secondary storage. AFIPS Conference Proceedings, Vol. 27, 1965 F.J.C.C., Spartan Books, Washington D.C., 1965, p. 213.

3. Dennis, Jack B. Segmentation and the design of multi-programmed computer systems. J. ACM 12 (October 1965), 589.

4. Glaser, E.L., Couleur, J.F., and Oliver, G.A. System design of a computer for time sharing applications. AFIPS Proceedings, Vol. 27, 1965 F.J.C.C., Spartan Books, Washington D.C., 1965, p. 197.

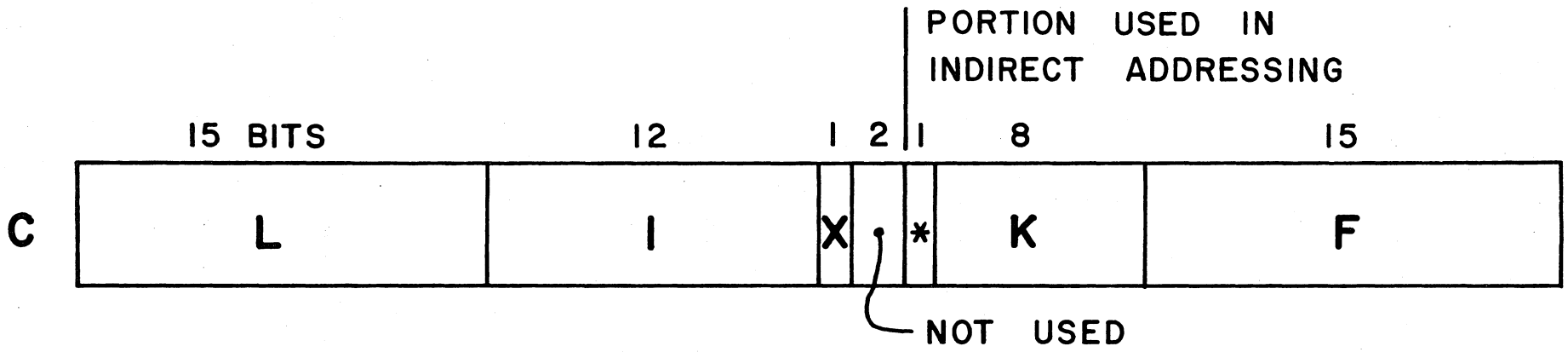
5. Graham, Martin. Memory hierarchy study. University of California, Computation Center, Berkeley, Technical Report, 1965.

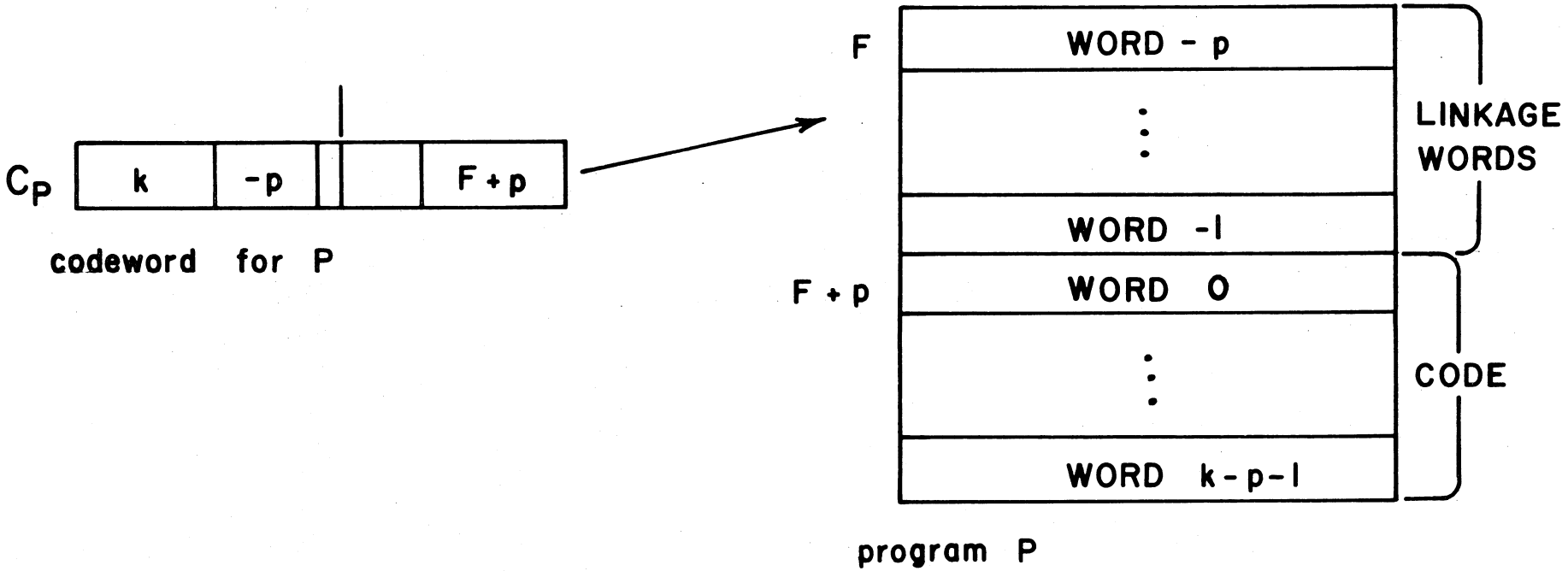
6. Iliffe, J.K. The use of the Genie system in numerical calculations. Annual Review in Automatic Programming, Vol. II, Pergamon Press, New York, 1961, p. 1.

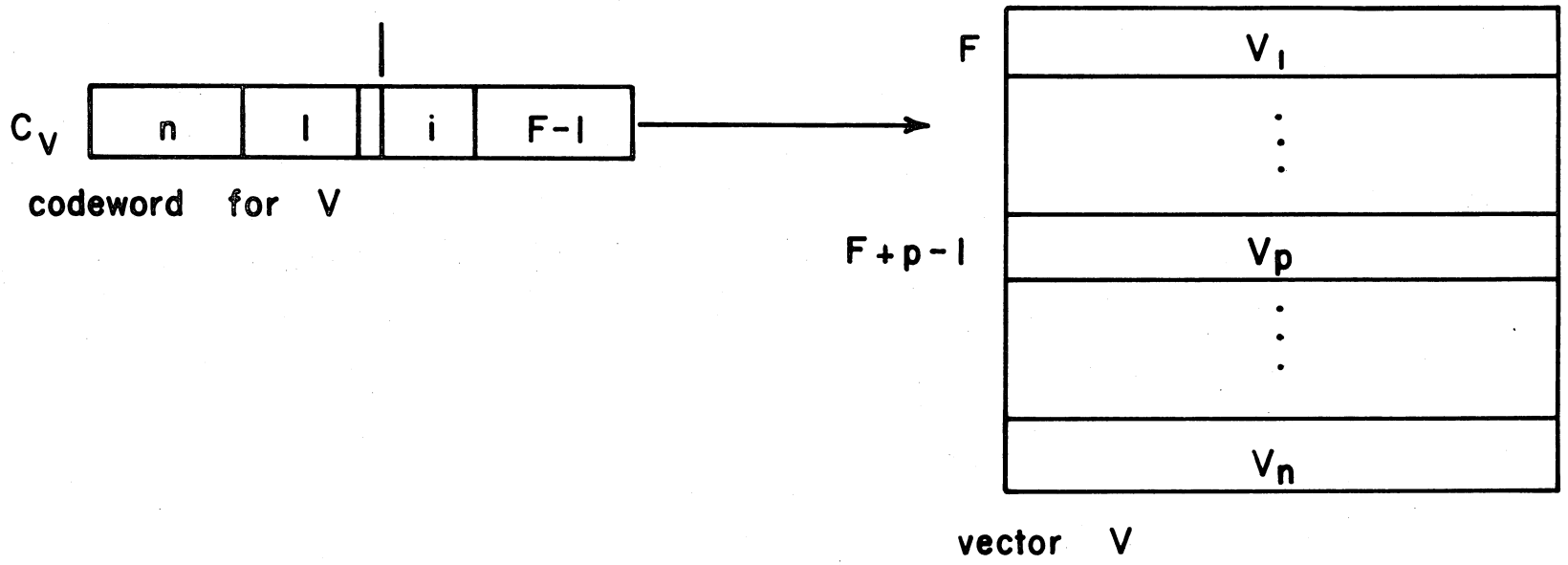
7. Iliffe, J.K. and Jodeit, Jane G. A dynamic storage allocation scheme. Comput. J. 5 (October 1962), 200.

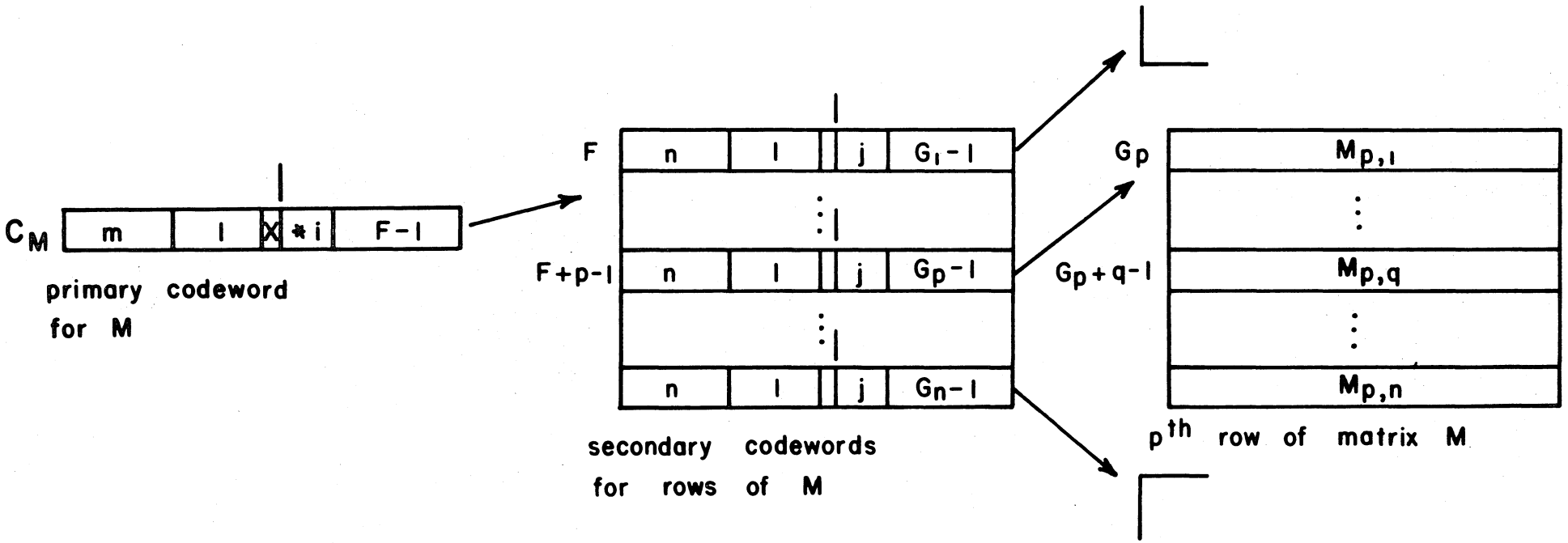
8. Jodeit, Jane G. and Sitton, Gary A. Tags for description and control. Rice University Computer Project, Report ORO-2572-9, Houston, February, 1967.

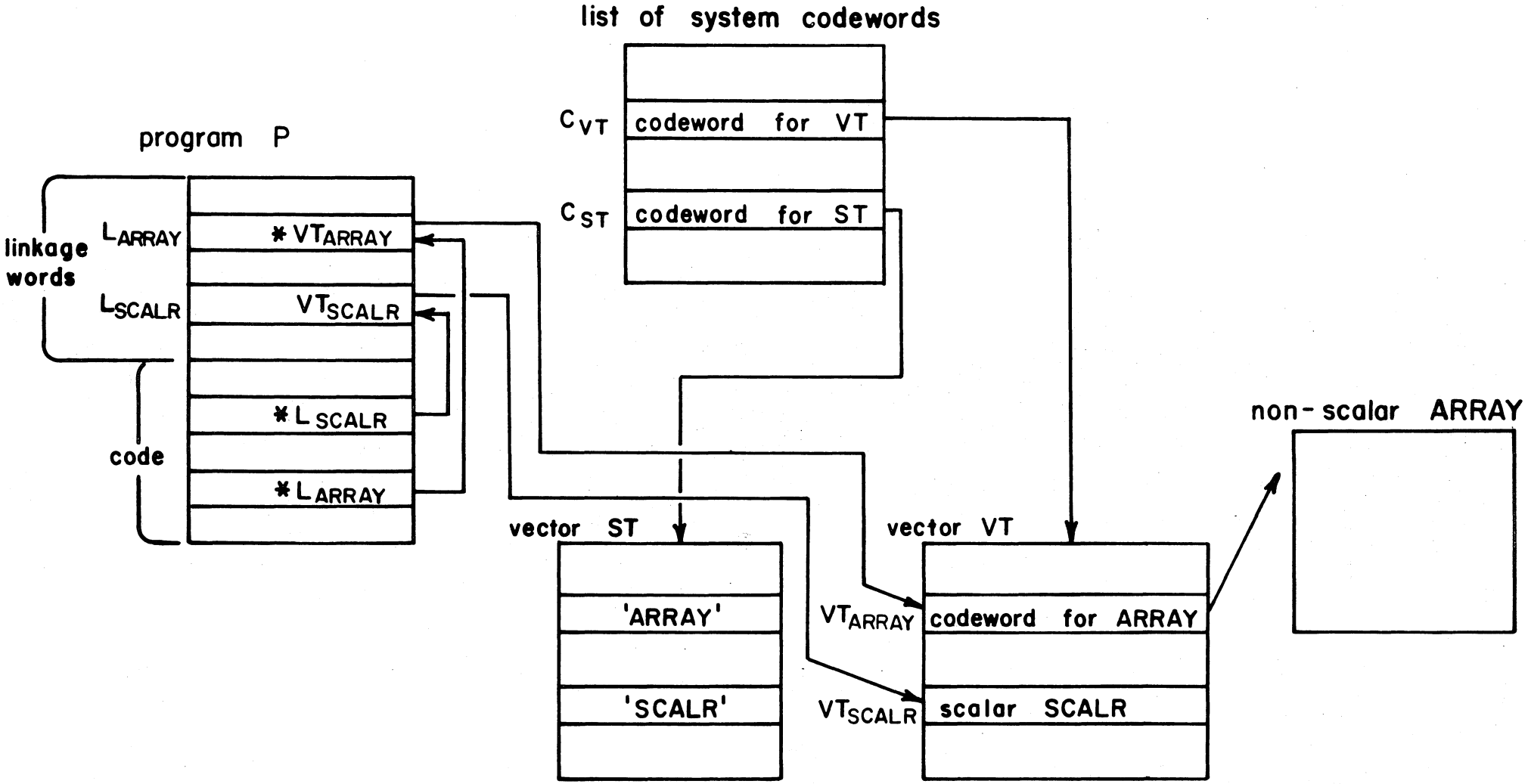
9. McCullough, James D., Speierman, Kermith H., and Zurcher, Frank W. A design for a multiple user multiprocessing system. AFIPS Conference Proceedings, Vol. 27, 1965 F.J.C.C., Spartan Books, Washington D.C., 1965, p. 611.

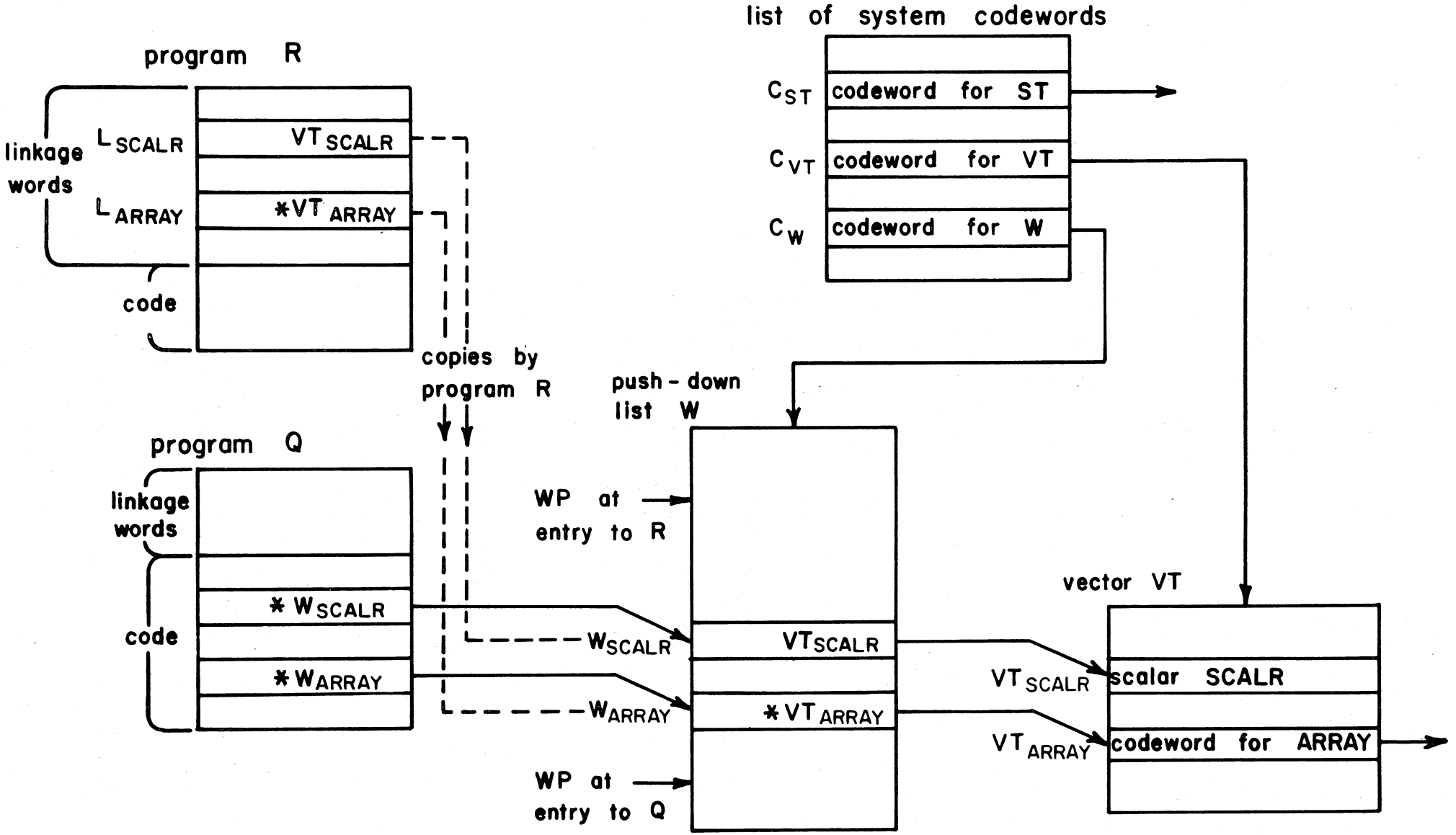

PORTION USED IN
INDIRECT ADDRESSING
15 BITS
12
1
2
1
8
15
C
L
I
X
*
K
F
NOT USED


C_P
k
-p
F + p
codeword for P
F
WORD - p
WORD -1
F + p
WORD 0
WORD k-p-1
LINKAGE WORDS
CODE
program P


C_V
n
l
i
F-l
codeword for V
F
V_1
F+p-l
V_p
V_n
vector V


C_M
m
1
X
*i
F-1
primary codeword
for M
F
n
1
j
G_1-1
F+p-1
n
1
j
G_p-1
n
1
j
G_n-1
secondary codewords
for rows of M
Gp
Gp+q-1
$M_{p,1}$
$M_{p,q}$
$M_{p,n}$
p^{th} row of matrix M


list of system codewords
C_{VT}
codeword for VT
C_{ST}
codeword for ST
program P
linkage words
L_{ARRAY}
* VT_{ARRAY}
L_{SCALR}
VT_{SCALR}
* L_{SCALR}
code
* L_{ARRAY}
vector ST
'ARRAY'
'SCALR'
vector VT
VT_{ARRAY}
codeword for ARRAY
VT_{SCALR}
scalar SCALR
non-scalar ARRAY


list of system codewords
program R
linkage words
L SCALR
VT SCALR
L ARRAY
*VT ARRAY
code
C ST
codeword for ST
C VT
codeword for VT
C W
codeword for W
copies by program R
push-down list W
program Q
linkage words
code
* W SCALR
* W ARRAY
WP at entry to R
W SCALR
W ARRAY
WP at entry to Q
VT SCALR
*VT ARRAY
vector VT
VT SCALR
VT ARRAY
scalar SCALR
codeword for ARRAY