CYLINDERS: A DATA STRUCTURE CONCEPT BASED ON RINGS

P. WESTON
S. M. TAYLOR

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS
This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, And U.S. Air Force) under Contract DAAB 07-67-C-0199; and in part by the OE C-1-7-071213-4557.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

Distribution of this report is unlimited.
# Table of Contents

Preface

Part I: Background and General Description of the Cylinder Structure

(α) Introduction 1
(β) Ring Structures 4
(γ) Cylinders 7
(δ) Examples of Use 13

Part II: CYL6 Programming Guide

(α) General Characteristics 17
(β) The Ring Building Functions 18
(γ) Link Removal Functions 22
(δ) The Ring Search Functions 23

Bibliography 29

Appendix: Table of Subroutine Characteristics 30
Preface

This report is designed to be of use as both a reference manual for Cylinder programming and as a more general introduction to the linked-list notions upon which CYL6, our Cylinder system, is based. Accordingly, it is divided into two major sections, the second being a detailed manual for the CYL6 routines, the first containing only more general description. A reader unfamiliar with list languages should start at the beginning of Part I, while beginning at section (γ) of Part I is appropriate otherwise. For routine reference use, only the Appendix, which is a condensed table of subroutine data, should ordinarily be needed.

Finally, because the rate of change of CYL6 is not yet absolutely zero, this report must be considered as a provisional description of it.
Introduction. Structuring of data within the computer store is a factor, implicit or explicit, in the design of any computer program. The point of tangency between data structure and program design lies in the necessity of imposing relationships upon data independently of their values, in order simply to express an algorithm or computing procedure upon them. This is particularly clearly seen in, for example, the case of matrix multiplication, where row and column relationships must be imposed in an appropriate way upon a set of stored numbers in order for the required procedure to be definable as a program. As a general principle, it may be said that, in the context of computation, data structure is dependent upon the processes to be performed, and is less an intrinsic quality of the data themselves.

In numerical applications it is ordinarily sufficient to structure data in matrix form provided that arrays with various numbers of dimensions can be handled, and compilers oriented toward numerical problems, such as FORTRAN and its successors, contain simple and direct means, including subscripting of variables and dimension declarations of arrays, for expressing the necessary relationships. What sets this type of structuring apart from linked structures is the fact that it is possible in an array to compute, in a reasonably simple manner, the location in the computer store—or address—which holds a given datum, on the basis of its description as an array element, i.e., the values of its subscripts and a "base address" corresponding to the variable itself. Indeed, even at the hardware level, current computers
universally include means for dealing directly with one-dimensional structures through the use of address-modification registers.

It will be convenient to refer to all structures based solely upon computation of "addresses" as array structures. Linked structures on the other hand are characterized by stored rather than computed addresses, various means being used to explicitly store with the data the locations of related items. These stored "pointers" to other items of data are often also referred to as links, and structures built from them are often called lists or linked lists. While the linking method is theoretically completely general, in practice it involves an overhead cost in terms of storage space and running time and is used only when necessary.

When an algorithm effectively precludes the use of arrays, some form of linked structure is the only presently available alternative, since content-addressed stores—which are the other presently interesting possibility—are not now included in ordinary equipment and may never be. The difficulties which lead to abandonment of arrays are encountered when the details of the required structure are either; (a) highly irregular, leading to great quantities of wasted storage due to empty positions in an array, or (b) highly variable, leading to massive shuffling around of storage contents when array parameters are redefined. An important subcase of the latter is that in which structural requirements are substantially dependent upon data values, as happens in most instances of artificial intelligence problems. The automatic parsing of sentences can be taken as an illustration. The entire object of such an algorithm is the generation of structure upon the incoming data. In this case, the structure represents the syntactic
relationships to be found in the input sentences, which are initially given simply as strings of symbols.

The Cylinder system to be described was developed in an effort to circumvent the major difficulties associated with the older list systems while maintaining flexibility of use and compactness of storage requirement. Two of the older and well established linked-list languages, IPL-V(2) and LISP(3), while they differ vastly as languages nonetheless share a common limitation, due to the restricted class of linkage structures realizable, i.e., in both cases only unidirectional linking is possible and no reentrant linkages are allowed, with the result that from an arbitrary internal element in a linked structure only a portion of the remaining structure is accessible. While for many purposes this objection is not actually as serious as it sounds, there do arise many cases in which complete availability of a structure from any element in it is a decided advantage. There have been attempts to provide more or less direct reversibility of linkage paths, e.g., Threaded Lists(4), and the SLIP system(5). The latter tends to impose heavy burdens upon memory space and the former involve some potentially severe problems due to untraceable connections when storage cells are "erased."

More recently, the usefulness of ring structures has been demonstrated(6), and they offer several advantages over the earlier approaches. The principal ones are (a) as the name suggests, the basic structures are closed upon themselves, so that complete availability of structure is implicit in the design without a great increase in storage space, and (b) as they have been implemented, ring structures usefully combine linking and array
structuring and in so doing make possible a fundamentally more efficient use of the computer. A further, most significant advance has been the invention of general-purpose low-level linked-list programming languages, the most widely distributed one being "L6" (7). In sharp contrast with earlier approaches, these are not tied to any particular linkage scheme, but are sufficiently general in concept and economical in the programming means they provide, to enable the user to realize whatever form of linked structure he may desire, using a fairly natural and compact programming notation.

(β) Ring Structures. In the terminology of linked lists computer storage is dividable into blocks and fields, in addition to the common word unit. A block is a group of consecutive computer words, i.e., having consecutive addresses, while a field comprises an arbitrarily defined region internal to a block, having limits fixed with respect to the block boundaries but not necessarily coinciding with any word boundaries. A common schematic representation for these relationships is shown in Figure 1. A linked structure may be created by using some fields in each block to hold the addresses of other blocks, i.e., to contain pointers. This is often represented in the way shown in Figure 2.

As a list structure a simple ring, which is the basic linked structure in the Cylinder system, is nothing more than a chain of pointers which closes upon itself. This can be represented as in Figure 3. Although the circularity of this type of structure could apparently lead a program to step endlessly from link to link in the course of a data search, (and for reasons of this type all reentrant structures were avoided in early linked-list systems), it is in fact a simple matter to prevent such a
Figure 1. A block is a group of consecutive computer words, within which areas called fields may be arbitrarily defined and labeled.

Figure 2. A schematic convention for linked structures, using arrows to indicate the blocks which are pointed to by stored links.

Figure 3. A simple ring is a closed chain of pointers.
difficulty, when the ring is the only closed linkage pattern allowed.

This is typically done by incorporating a specially designated and identifiable cell in each ring to serve as a reference point in searching and building functions. A similar device was used in our earlier Halo design (1), but in the Cylinder system emphasis is placed on eliminating all specially reserved storage for system functions—and most other restrictive system conventions—and the Cylinder ring is an entirely homogeneous structure. To insure termination of all ring-searching operations, the subroutines which execute them have been designed to automatically stop with a "fail" indication in the event that the starting point of the search is reencountered. This approach reduces storage overhead requirements and simplifies system conventions, simultaneously giving the user maximum latitude.

Given the choice of rings as the basic linkage pattern, which insures complete availability of each point in a structure from any other point with a minimally complicated set of system conventions, there still remains a wide field of choice in the exact manner in which the rings are to be used in representing data structure. Common to all methods is the use of rings to "thread" blocks of storage which may contain data as well as ring pointers, linking the blocks together somewhat as if they were beads on a string. One direct application of this basic scheme is in representing a classification of data. See Figure 4. Using individual blocks to hold descriptive information for the data, the representation is constructed by linking together on rings those data blocks which belong to each subclass to be represented. Because room can be made for more than one ring to thread a block, thereby representing membership of a datum in several classes simultaneously, there is no difficulty in setting up arbitrary hierarchical or
overlapping classifications upon a set of data. Using ring structures in this way requires little or no more storage than conventional lists do, while in contrast with conventional lists, rings allow full flexibility in the manner in which data may be searched, i.e., with equal ease a class (ring) may be searched for the data (blocks) it contains, or a datum may be searched for the classes which contain it.

\textbf{(γ) Cylinders.} Further discussion of the use of rings in creating data structures will be limited to the context of the cylinder conventions. These are:

\begin{enumerate}
  \item The only allowed linked structure is the ring, i.e., a non-branching, self-closed chain of pointers.
  \item A fixed number of fields, eight in the case of CYL6, are defined upon the first words of all blocks, and all ring pointers within the system must be contained in just these fields. However, there are no system-imposed restrictions on the use of fields which are not needed in a given program for building rings, i.e., if only a few levels are needed short blocks may be used or data may be stored at the unused levels, wherever in the block they may occur. Program context is the determining factor. In CYL6 the eight ring-pointer fields are identified by the letters "I" through "P".
  \item All of the pointers constituting any one ring lie in the same field within their respective blocks, i.e., if one pointer in a ring is contained in a "K" field, all pointers in that ring necessarily lie in "K" fields. It is convenient to speak of the allowed pointer fields as "levels" at which rings may be constructed and to speak of a ring as being at a certain level, e.g., in CYL6 a particular ring might be at the "K" level or "I" level,
Figure 4. Threading of blocks by multiple rings to represent a classification upon a set of data.

Figure 5. Conceptual scheme of pointer levels as defined in the CYL6 system. This is not the actual arrangement in storage.

Figure 6. Actual storage arrangement in CYL6, using the first four words in a block. Field 0 is the tag field shared by levels I and J. Fields U through Z are the tag fields for levels K through P. The first six bits of the first word are reserved for the CSL7 storage allocator.
etc. See Figure 5.

(d) Associated with each pointer field there is also a data field of limited size, called a tag field. All data search functions in the CYL6 system are sensitive only to the contents of the tag field associated with the level being searched. All tag contents are completely under control of the user and are primarily intended to identify, for search purposes, functionally distinct types of storage blocks within the context of an individual program. In many applications tags can provide all of the identifying information needed to govern every desired mode of search in a structure. In CYL6, the tags have a capacity of nine bits, which may be divided into independent three- and six-bit subfields or may be taken altogether for search purposes. It is convenient to describe the contents of a CYL6 tag field with an octal digit followed by a BCD character, e.g., "5A", "1/".

A peculiarity of the CYL6 system is that while there are eight ring levels there are only seven tag fields, the I and J levels having a single tag field in common. This is due to interaction with space reserved in the first word of each block by the storage allocator of CSL7(8), the dialect of L6 in which CYL6 is written. See Figure 6.

The four conventions listed above are the only restrictions which are required by cylinders. They define a system broad enough in scope to include most other ring structures and, in the case of CYL6, they are able to coexist with any other type of structure possible in L6, provided that the CYL6 routines are used only on rings of the cylinder type.

To gain further understanding of the structures that are available through the cylinder system it will be helpful to use an appropriate notation,
such as the one which grew out of the CYL6 development effort. Figures 7 through 11 help to demonstrate this notation and its interpretation in terms of the block and pointer notation of the earlier figures. In this scheme the chain of pointers constituting a ring is represented simply by an arrow, the direction of which indicates the sequence of the indicated data elements along the ring. Since it is understood that all pointer chains are closed upon themselves, this fact may be kept in mind making it unnecessary to draw the rings as closed figures. When a single block is threaded by rings at several levels it is natural in the cylinder context to treat such a block as a cross-linking between rings and this is the way it is drawn, as in Figure 7.

Although it is in fact generally the case that a structure consisting simply of crosslinked rings will tend to be unwieldy in any notation, some order can be extracted from the chaos by limiting one's attention to a single pair of rings at a time. In this case the possibilities are quite limited, a block is threaded by none, one, or both of the rings, and in the latter two cases may or may not be threaded by other rings as well (Figure 8).

The basic cylinder is formed from a pair of rings and employs cross linking between them in a particular way. The framework of the scheme is pictured in Figure 9, where the triangles are used to indicate the positions on each ring of the special cross-links which we have called "seams". The only defining property of the seam substructure is that the blocks which form the seams lie either in exactly identical or more usually in exactly
reverse sequences on the two rings. In the figure the arrow directions shown indicate this ordering. The existence of such seams is not a mandatory feature in the CYL6 system. Their insertion, removal, and selective tagging is left entirely up to the user's discretion.

The usefulness of having a sequence of identifiable seam cross links with the indicated ordering lies in the fact that between each pair of seams a type of subring is created which has the important characteristic of self-closure as does the main ring but has several added features of some usefulness as well. As Figure 9 shows, each subring contains four segments, consisting of the two sequences of ring links lying between the two bounding seams at the lower and upper levels respectively, and of the two seams themselves, which constitute bilateral connections between the upper and lower ring segments. One natural way of using this ring and subring configuration to represent data structure is to assign a datum to each seam and to represent binary relations between data by other cross links which carry tags identifying the relations they represent and are threaded by the lower ring segment of one subring, whose seam represents the value of the first argument of the relation and by the upper ring segment of the subring for the datum at the second argument position. Such a representation is shown in Figure 9.

This representation for binary relations is highly compact as a linked structure, and is quite complete in representational power in view of the facts that both the forward and inverse relation are simultaneously represented, and that the main rings constitute a master file from which all data or all the links representing any particular relation may be selectively
Figure 7. A schematic convention for cylinder structures. Above is the block and pointer equivalent of the lower diagram. The long arrows are understood to signify closed rings of pointers running in the sequence indicated by the arrow direction.

Figure 8. The basic linkages possible when a particular pair of rings is taken separately. The particular choice of levels here is unimportant.
retrieved using a tag-controlled data search. The storage requirement in terms of number of necessary links compares quite favorably in this case with that for older list systems such as IPL-V provided that all the above information is to be represented.

(6) **Examples of Use.** An example of a binary relational structure of a rather simple kind arises in the machine representation of line drawings involving points and straight lines as elements. If points are taken as the data, then lines can be represented as binary relations between the data points, and a structure of the sort depicted in Figure 10 results. This structure represents only the connections of the points and can stand for any pattern consisting of a triangle with a fourth line ending on one of its vertices. Supplying coordinate values for the points realizes a particular example.

As an example of the flexibility of cylinders, Figure 11 shows a completely compatible extension of the scheme in Figure 10 which allows the simultaneous representation of line to line relations, such as intersection, and of point-to-point relations including the line definitions in terms of point pairs. At the K and L levels a second cylinder is formed, having as seams the lines which appear as relational links at the upper level. Any points which are defined by line intersections, such as point 5 in the diagram of Figure 10, can appear as relation links at the lower level but as seams at the upper level, thus becoming capable of serving to define further lines and so on.

Because points which arise from line intersections depend for their coordinates upon the other points which define the intersecting lines, this
Figure 9. The subring structure based on 'seams'. In the upper structure links 1 and 2 represent $R_1(a,c)$ and $R_2(b,a)$ respectively according to the binary relational scheme described in the text.
Figure 11. An extension of the scheme of Figure 10 which allows intersection points to be represented as binary relations between lines in a cylinder at one level while the same lines are simultaneously represented as relations between points in a cylinder at another level. P5 is given a distinct tag value, lp, because it must be distinguished as a dependent quantity, and is not a freely variable point.
dependence is important to any program which operates upon such data structures and must be somehow marked, without destroying the identifiability of the datum as a point. This is easily done, e.g., through the use of the upper subfield of the tag in CYL6, marking it distinctly from the corresponding subfield for other points but keeping the lower portion the same. In this way the tags themselves can be made to represent a classification upon the data with no further burden upon the structure or storage requirement.

This concludes the introductory section of this manual. In the following section all of the routines available through CYL6 are described in sufficient detail for their use in CSL7 programs. The Appendix contains a tabular summary of the most important details.
The CYL6 Subroutine System

(\(\phi\)) General Characteristics. The CYL6 system is coded for the CSL7 language, and the CSL6 - CSL7 manual(8) should be used as a reference, since there will be little duplication of its material here. Briefly, the choice of CSL7 was made because it allows a tag field to be used in the first word of a block, which would not be possible in CSL6 because of conflict with bits reserved by the storage allocator. These bits are free in CSL7 because it does not recombine free storage blocks into larger units, and experience with an earlier cylinder system having its own allocator has shown that recombination is not actually needed. The range of block sizes which are ordinarily used is restricted to the smaller values and a 'population' of the common sizes builds up which recycles through the allocator fairly efficiently.

Physically, CYL6 is a CSL7 Subprogram containing 'DO' type entry points for each of the function subroutines to be described below. A program which uses CYL6 must observe several conventions, namely:

(a) At the logical start of the program CYL6 itself must be executed to define the pertinent fields, which are I through P, \(\emptyset\), U through Z, 1 through 7, and T. Fields 1 through 7 are full word fields for the correspondingly numbered words in a block and T is the three-bit field which fits between the six reserved bits of the \(\emptyset\)'th word and the upper address of that word. All of these field definitions are declared global in CYL6 and may be used by other routines through a global declaration.

(b) Information is passed to and returned from CYL6 routines through various combinations of the six bugs U through Z. Bugs A through E
are employed in the execution of some of the routines and when used, their contents are not saved. The remaining bugs F through T are never modified by CYL6.

(c) All but two of the routines are executable through the standard DO operation block, the remaining ones require the DOARG variant. Those routines having fail exits are individually mentioned below.

The routines will be described below in three sections according to function, Ring Building, Link Removal, and Search. Within each of these categories the ring level, for functions which apply to a single level, is specified by a suffix on the routine name rather than by an argument. The suffix consists of a period followed by the field designator for the required level, for example 'PRED.I', or 'PRED.O'. In addition the link-modifying functions employ a naming convention for two types of routines which operate on a set of levels rather than a single one. The prefix 'X' is used to signify a function which obtains a list of levels from the field contents push down stack, and the prefix 'MULT' is used to signify a routine which operates on a number of adjacent levels, the number given as an argument, starting at the I level and proceeding downward. Within the category of search routines further naming conventions are used which will be described in that section.

(3) Ring Building Functions.

(1) General. These functions divide into the categories of initializing and link insertion routines, with two driver routines included which both create and initialize storage blocks. Initializing a level in a storage block consists of inserting a pointer at that level which points to the block itself, thus forming the minimum-size ring, consisting of one
element and referred to as a 'null' ring. The importance of null rings is that they are used as building blocks in the construction of all cylinder structures. The linking routines actually are designed to perform a merging of rings, at least one of which is a null ring, in order to insure in a simple way that in ordinary use only true rings will be built. For this reason an attempt to insert a non-initialized cell in a ring will produce an on-line message. The null condition is testable through the search functions or directly through appropriate test statements written in CSL7.

(2) **Initializing Routines IC.1 through IC.P.** These simply create a null ring at the specified level in the input block. Bugs A through E are not used.

Execution is through a DO operation with a pointer to the input block resting in bug U. There is no fail condition.

(3) **Initializing Routines MULTIC and XIC.** MULTIC drives the routines IC.1 through IC.P in sequence stopping when the number of fields initialized is equal to the number which is contained in bug Y. Bug A is used within this routine.

Execution is through a DO operation, with bug U containing the number of fields to initialize. There is no fail condition.

XIC also drives the IC.x routines. A list of octal digits in arbitrary order must be pushed down in the FC stack, each digit representing a field to be initialized. The digit '0' corresponds to field I and '7' to field P. As in MULTIC, bug Y governs the number of fields which are initialized. Bugs A and B are used.

Execution is through a DO operation, with bug U containing a pointer
to the block to be initialized, bug Y containing the number of fields to initialize, and the FC stack containing a list of octal digits designating those fields to be initialized. There is no fail condition.

(4) **Block creation routines GET and GETP.** GET executes a CSL7 block creation operation, drives MULTIC to create a set of null fields in the new block, and exits with a pointer to the created block in bug U. Bug A is used in the internal call to MULTIC.

Execution is through a DO operation, with bug X containing the length of the desired block, and bug Y containing the number of fields to be initialized, starting from field I. The created cell is output through bug U. There is no fail condition.

GETP is analogous to GET, but drives XIC rather than MULTIC. Both bugs A and B are used.

Execution is through a DO operation with bug X containing the length of the desired block, and with bug Y and the FC stack set up as for XIC. Output is with a pointer to the new block in U. There is no fail condition.

(5) **Ring Building Routines LINK.I through LINK.P.** A cell is input to these routines by a pointer in bug U. This cell is examined and if it does not contain a null ring at the specified level an error message occurs and a fail exit is executed. If U points to a properly initialized cell, then a tag and a pointer are popped up from the FC stack and the insertion is made so that the cell obtained from the stack becomes the immediate ring predecessor of the originally null cell obtained from U. The tag obtained from the stack is also inserted. Bugs A and B are used.
Execution is through a DO operation with a pointer to the cell to be inserted contained in bug U, and with the FC stack primed by pushing down a tag and a pointer to the new ring predecessor, in that order. Bug U is not modified upon exit. The routine fails if the cell input through U has not been initialized.

(6) **Multiple Ring Building Routines MULTLINK and XLINK.**

MULTLINK drives the LINK.x routines under control of bug X. That is, beginning with LINK.1, k of the LINK.x routines are driven, where k is the contents of bug X. To feed the link routines, all of the required tag and predecessor pairs must be pushed down in the FC stack before entering MULTLINK. Because this stack operates on the last in-first out principle, the data for the lowest level must be pushed down first. Bugs A, B, and C are used internally.

Execution is through a DO operation with bug X set equal to the number of levels to be linked, bug U containing a pointer to the cell which will be inserted by the LINK operations, and the FC stack primed by pushing down the required tag and predecessor pairs, lowest level first. The routine will return to the system, thus terminating program execution, after an online error message from CSL7 if the cell input in bug U is not properly initialized. MULTLINK has no fail return.

XLINK drives the LINK.x routines through data planted in the FC stack. As in most of the multiple field routines the number of fields is given in bug X, but since the FC stack must also be primed for the LINK routines the level indicators must be interleaved with this information. This is done by feeding the stack in the sequence TAG, PREDECESSOR, LEVEL DIGIT, so that the octal digit indicating the level will come off the stack.
Execution is through a DO operation with a pointer to the cell to be inserted contained in bug U, and with the FC stack primed by pushing down a tag and a pointer to the new ring predecessor, in that order. Bug U is not modified upon exit. The routine fails if the cell input through U has not been initialized.

(6) Multiple Ring Building Routines MULTLINK and XLINK.

MULTLINK drives the LINK.x routines under control of bug X. That is, beginning with LINK.1, k of the LINK.x routines are driven, where k is the contents of bug X. To feed the link routines, all of the required tag and predecessor pairs must be pushed down in the FC stack before entering MULTLINK. Because this stack operates on the last in-first out principle, the data for the lowest level must be pushed down first. Bugs A, B, and C are used internally.

Execution is through a DO operation with bug X set equal to the number of levels to be linked, bug U containing a pointer to the cell which will be inserted by the LINK operations, and the FC stack primed by pushing down the required tag and predecessor pairs, lowest level first. The routine will return to the system, thus terminating program execution, after an on-line error message from CSL7 if the cell input in bug U is not properly initialized. MULTLINK has no fail return.

XLINK drives the LINK.x routines through data planted in the FC stack. As in most of the multiple field routines the number of fields is given in bug X, but since the FC stack must also be primed for the LINK routines the level indicators must be interleaved with this information. This is done by feeding the stack in the sequence TAG, PREDECESSOR, LEVEL DIGIT, so that the octal digit indicating the level will come off the stack
just before the LINK information for that level. Bugs A, B, C, and D are used internally.

Execution is through a DO operation with bug X set to the number of levels to be linked, bug U containing a pointer to the cell to be inserted, and the FC stack primed with TAG, PREDECESSOR, LEVEL triples. XLINK has no fail exit, but will return to the system after an on-line message from CSL7 if the input cell is not properly initialized.

(γ) Link Removal Routines.

(1) Basic Cell Removal Routines NULIFY.I through NULIFY.P. These are the basic routines for link removal. The predecessor of the cell which is to be removed is input through bug V. NULIFY acts upon the cell pointed to by V, relinking it to its second successor. Its first successor is reinitialized. There is no net effect if V points to a null cell. Bug A is used and is left with a pointer to the cell which is removed, though this will not normally be a useful output in CYL6 programming.

Execution is through a DO operation with a pointer in bug V to the predecessor of the cell to be removed. There is no fail condition.

(2) Cell Removal Routines CUT.I through CUT.P. These are driver routines, which accept as input the cell which is actually to be removed. They drive a search function to obtain the predecessor and then drive the corresponding NULIFY to accomplish the cell removal. The internal call to NULIFY uses bug A.

Execution is through a DO operation with bug U pointing to the cell to be removed. Upon exit, bug V will point to the original predecessor of the input cell and the input cell will be reinitialized. There is no fail
exit, and as in NULIFY, no net action is taken if the input cell is null.

(3) **Multiple Level Cell Removal Routines MULTICUT, XCUT.**

MULTICUT drives \( k \) of the CUT routines in sequence starting with CUT.I, where \( k \) is the contents of bug X. Bugs A and C are used internally.

Execution is through a DO operation with bug U pointing to the cell to be removed from the number of rings specified in bug X. Upon exit, bug V will point to the original predecessor of the input cell at the lowest level reached by MULTICUT. There is no fail exit.

XCUT drives the CUT routines under FC stack control, using octal digits to specify ring levels as in the other X-type subroutines. Bugs A, C, and D are used.

Execution is through a DO operation with bug U pointing to the cell to be removed, bug X containing the number of levels to be cut, and the FC stack containing a list of octal digits specifying the levels. There is no fail exit.

(6) **The Ring Search Routines.**

(1) **General Properties.** This category includes tag controlled search and index controlled search as its two major divisions and contains in addition a backward stepping function used principally as an adjunct to ring building and cutting operations, and several routines for manipulating the tag comparison lists which are used by the major search routines.

Each major search function utilizes a list of five 9-bit numbers against which the tags in cells which are encountered during a search operation may be compared. In the case of tag-controlled search, a match with any element of the list stops the search and results in an output which indicates
which element of the list was matched, while index controlled search stops
only after a given number of steps, but makes the tag comparison and output
indication when the last cell is encountered.

To facilitate the use of tag comparison lists, means have been
provided for specifying several different lists prior to their use, and sub­
sequently referring to them simply by an index number. The subroutine DEFTCL
accomplishes the list-defining function, while reference to defined lists is
possible through a program argument and through the subroutine SETTAG2. As
the system is now programmed, the pre-defined lists specify only the last
four elements, because it is more often necessary to vary one element out of
a comparison list than two or more. The first member is specifiable only
for the list which at a particular moment is active. This is done through a
program argument or by the routine SETTAG1.

Because it is sometimes desirable to initiate a search with a com­
pletely new tag comparison list, sometimes with only one new tag, and some­
times with no changes in the tag list, three corresponding entry points are
provided for each of the standard tag-controlled search routines. It is also
possible to limit the tag comparison to only the first three or last six bits,
and this choice of tag subfields is also indicated by entry point. Another
set of entries is provided to allow the use of non-standard tag subfields.

To accommodate the variations listed above a prefix and an additional
suffix are used in writing the tag-controlled search routine names. The final
suffix is still a period followed by field symbol to indicate ring level, but
preceding this a '2', '1', or no suffix is written to indicate, respectively,
a completely new tag list, new first tag only, or no change in tag list. The
tag subfields are indicated by a prefix with a 'P', 'T' or no prefix indicating upper three bits, lower six bits, or full tag. Examples of these names are 'PSRCH.J', 'TSRCH1.I', 'SRCH2.P', indicating, respectively, a J-level search using the latest tag list and comparing only the upper three bits of the tag field, an I-level search with a new first comparison tag and comparing only the lower six bits in the tag field, and a P-level search using a completely new tag list and comparing the full tag fields. Variable mask search, index controlled search, and predecessor search routines do not use the above convention and employ only the level suffix.

(1) The Tag Comparison List Routines DEFTCL, SETTAG2, SETTAG1, SETMASK. DEFTCL defines a list of nine-bit quantities to be a tag comparison list and associates this list with an index value in the range 0 through 7 for future reference by search functions. This action overwrites any list previously defined for that index value.

Execution is through a DOARG operation using a list of five arguments. The first argument should have a value in the range 0 through 7, but will be taken modulo 8 otherwise. The last four arguments are ordinarily given as octal constants and specify in order the last four entries in the tag comparison list whose index value is the value of the first argument. There is no fail exit.

SETTAG2 executes the tag list reference function of the suffix-2 form of search routine without performing a data search. This is useful in setting up conditions for an index controlled search.

Execution is through a DO operation with bug Y holding the index value of the desired tag comparison list and bug Z holding the first tag.
value, which completes the active list. There is no fail exit.

SETTAG1 replaces the first tag comparison value on the active list as in suffix-1 data searches, but without performing the search.

Execution is through a DO operation with bug Z holding the desired tag value. There is no fail exit.

SETMASK defines a non-standard mask which will be used to define the tag bits to be compared by all following variable-mask or index-controlled searches. A standard value is restored upon execution of a SRCH, PSRCH, or TSRCH routine, which forces the mask according to the prefix type, or by execution of SETMASK with a standard mask value, i.e., first three, last six, or all nine bits.

Execution is through a DOARG operation with a single argument specifying a nine-bit pattern to be used as a mask. The argument is most conveniently written as an octal constant. There is no fail exit.

(2) Basic Tag Controlled Search Functions SRCH.x. These use the tag comparison list most recently established, and search a ring for a match with any tag in the comparison list. The cell at the starting point is not tested for a match in order that a search may be easily restarted from its last stopping point.

Execution is through a DO operation with bug U pointing to the cell which is to be the search starting point. If a cell containing a matching tag is encountered, a normal exit is executed with a pointer to the matching cell in bug U, and a pointer to its immediate predecessor in bug V. A number in the range 0 through 4 is also entered in bug W indicating the position in the comparison list of the tag yielding the match.
If no cell in the ring yields a match with the tag list, the routine performs a FAIL exit. In this case V will point to the predecessor of the starting point, but bugs U and W will not be modified. FAIL is, of course, automatic if the starting point is a null cell.

(3) Variations of the basic search functions, pSRCHs.x. These execute the basic tag search described above in (2), but modify the initial conditions of the search according to the prefix and suffix values.

When p has the value 'P', only the upper three bits of the tag field are compared in testing for a match.

When p has the value 'T', only the lower six bits of the tag field are compared.

When s has the value '2', the index value for a new tag comparison list is obtained from bug Y and the value of the new first member of the tag list is obtained from bug Z.

When s has the value '1', only bug Z is accessed to obtain a replacement for the first tag list element.

(4) Variable-Mask Tag-Search Routines VSRCH.x. These perform the basic tag search functions as in (2) but do not set the mask which governs the tag comparison, so that SETMASK may be used with VSRCH to accomplish tag searches upon non-standard tag subfields.

(5) Index-Controlled Search XSRCH.x. A ring is searched to obtain the cell which is the k-th successor of the input cell, where k is the contents of bug X. If the k-th successor exists, its tag is compared with the last established comparison list according to the last established mask. If a match is found, bug W indicates its position in the list. If no match is
found, bug W is set negative upon exit.

Execution is through a DO operation with bug U pointing to the search starting point, and bug X containing the number of steps to be taken. If the ring is large enough to contain the specified successor a normal exit is taken with bug U pointing to the cell found, bug V pointing to its immediate predecessor, and bug W holding a number which indicates the tag in the current comparison list which matches the tag in the output cell, or a negative number if there is no tag match. A FAIL exit is taken if the specified successor does not exist. In this case bug U and bug W will remain unmodified but bug V will be left with a pointer to the immediate predecessor of the starting point. FAIL is automatic if the starting point is null, or if X contains a number less than 1.

(6) Predecessor Search PRED.x. These routines take one step backward along a ring to locate the immediate predecessor of a cell. No tag comparison is made.

Execution is through a DO operation with bug U pointing to the initial cell. If the starting point is not null a normal exit is taken with bug U unmodified and with bug V pointing to the predecessor cell. FAIL exit is taken only if the input cell is null, and in this case the routine exits with bug V set equal to bug U. This fail condition makes the PRED function a useful test for the null condition.
REFERENCES


Appendix

The following pages contain in tabular form the basic data for all of the CYL6 subroutines. The tables are followed by an alphabetical index to the subroutines, listing references to the appropriate table and to the description of each subroutine in Part II.

In the tables most entries are transparent in terms of L6 conventions, but one form used in the Input column requires an explanation. When "FC" appears as an input it refers to the Field Contents pushdown stack and the entries shown beside it are the required stack contents listed from the top of the stack downward. To fill the stack with these entries a Pushdown operation should be executed, listing the entries in reverse order, i.e., from the bottom of the given list upward.
<table>
<thead>
<tr>
<th>Routine</th>
<th>Call</th>
<th>Input</th>
<th>Output</th>
<th>Drives</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC.x</td>
<td>DO</td>
<td>U=Block to be initialized</td>
<td>None</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MULTIC</td>
<td>DO</td>
<td>U=Block to be initialized</td>
<td>None</td>
<td>IC.x</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y=Nr. of fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XIC</td>
<td>DO</td>
<td>U=Block to be initialized</td>
<td>None</td>
<td>IC.x</td>
<td>A,B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y=Nr. of fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FC=list of level digits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>DO</td>
<td>X=Length of new block</td>
<td>U=created block</td>
<td>MULTIC</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y=Nr. of fields to initialize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GETP</td>
<td>DO</td>
<td>X=Length of new block</td>
<td>U=created block</td>
<td>XIC</td>
<td>A,B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y=Nr. of fields to initialize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FC=list of level digits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINK.x</td>
<td>(FAIL), DO</td>
<td>U=Null cell to be inserted</td>
<td>None</td>
<td>--</td>
<td>A,B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FC=Predecessor Tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAIL: No action is taken</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routine</td>
<td>Call</td>
<td>Input</td>
<td>Output</td>
<td>Drives</td>
<td>Uses</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| MULTLINK | DO   | U=Null cell to be inserted  
X=Nr. of rings  
FC= Predecessor  
Tag  
{Etc.} | None | LINK.x | A,B,C  |
| XLINK   | DO   | U=Null cell to be inserted  
X=Nr. of rings  
FC= Level digit  
Predecessor  
Tag  
{Etc.} | None | LINK.x | A,B,C,D|
<table>
<thead>
<tr>
<th>Routine</th>
<th>Call</th>
<th>Input</th>
<th>Output</th>
<th>Drives</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULIFY.x</td>
<td>DO</td>
<td>V=Predecessor of cells to be removed</td>
<td>A=cell removed</td>
<td>--</td>
<td>A</td>
</tr>
<tr>
<td>CUT.x</td>
<td>DO</td>
<td>U=Cell to be removed</td>
<td>V=original predecessor of input cell</td>
<td>NULIFY.x</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PRED.x</td>
<td></td>
</tr>
<tr>
<td>MULTICUT</td>
<td>DO</td>
<td>U=Cell to be removed</td>
<td>V=original predecessor at lowest level cut</td>
<td>CUT.x</td>
<td>A,C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X=Nr. of rings</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCUT</td>
<td>DO</td>
<td>U=Cell to be removed</td>
<td>V=original predecessor of last level cut</td>
<td>CUT.x</td>
<td>A,C,D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X=Nr. of rings</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FC=list of level digits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3 Ring Search Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Call</th>
<th>Input</th>
<th>Output</th>
<th>Drives</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFTCL</td>
<td>DOARG</td>
<td>Arg1=Index of new tag list&lt;br&gt;Arg2=Tag nr. 1&lt;br&gt;Arg3=Tag nr. 2&lt;br&gt;Arg4=Tag nr. 3&lt;br&gt;Arg5=Tag nr. 4</td>
<td>None</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SETTAG2</td>
<td>DO</td>
<td>Y=Index of new tag list&lt;br&gt;Z=New 0'th tag</td>
<td>None</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SETTAG1</td>
<td>DO</td>
<td>Z=New 0'th tag</td>
<td>None</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SETMASK</td>
<td>DOARG</td>
<td>Arg1=Octal 9-bit mask</td>
<td>None</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SRCH.x</td>
<td>(FAIL), DO</td>
<td>U=Starting point cell</td>
<td>NORMAL:&lt;br&gt;U=Stopping Point cell&lt;br&gt;V=Predecessor of above&lt;br&gt;W=Position in tag list of the matching tag (0 to 4)&lt;br&gt;FAIL:&lt;br&gt;V=Predecessor of starting point&lt;br&gt;U,W unchanged</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SRCH1.x</td>
<td>(FAIL), DO</td>
<td>U=Starting point cell&lt;br&gt;Z=New 0'th tag</td>
<td>As SRCH.x</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Routine</td>
<td>Call</td>
<td>Input</td>
<td>Output</td>
<td>Drives</td>
<td>Uses</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------------------------------</td>
<td>--------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>SRCH2.x</td>
<td>(FAIL),</td>
<td>U=Starting point cell</td>
<td>As SRCH.x</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>DO</td>
<td>Y=Index of new tag list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z=New Ø'th tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSRCH.x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSRCH1.x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSRCH2.x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSRCH.x</td>
<td></td>
<td>As above, but compare only upper</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSRCH1.x</td>
<td></td>
<td>three bits of the tag field.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSRCH2.x</td>
<td></td>
<td>As above, but compare only lower</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>six bits of the tag field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSRCH.x</td>
<td></td>
<td>As SRCH.x but does not set the tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>comparison mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XSRCH.x</td>
<td>(FAIL),</td>
<td>U=Starting point cell</td>
<td>NORMAL:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO</td>
<td>X=Nr. of steps to take</td>
<td>U=stopping</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>point</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>V=Predecessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of stopping</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>point</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W=Position in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>list of the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>matching</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>tag, if match</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Negative, if no</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>match</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FAIL:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>V=Predecessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of starting</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>point</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>U, W unmodified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRED.x</td>
<td>(FAIL),</td>
<td>U=Starting point cell</td>
<td>NORMAL:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO</td>
<td></td>
<td>V=Predecessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of starting</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>point cell</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>U unmodified</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FAIL:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>V=U=starting</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>point</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: XSRCH does not set the tag comparison mask.
<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page Nr. Part II</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUT.x</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>DEFTCL</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>GET</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>GETP</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>IC.x</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>LINK.x</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>MULTIC</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>MULTICUT</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>MULTILINK</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>NULIFY.x</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>PRED.x</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>PSRCH.x</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>PSRCH1.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSRCH2.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETMASK</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>SETTAG1</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>SETTAG2</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>SRCH.x</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>SRCH1.x</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>SRCH2.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSRC1.x</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>TSRC2.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSRCH.x</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>XCUT</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>XIC</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>XLINK</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>XSRCH.x</td>
<td>26</td>
<td>3</td>
</tr>
</tbody>
</table>
DISTRIBUTION AS OF 22 OCTOBER, 1968

1 Dr A.A. Dougal
Asst Director (Research)
Ofc of Defense Res & Eng
Department of Defense
Washington, D.C. 20301

1 Office of Deputy Director
(Research and Technology)
ODD R&E-OSD
The Pentagon, Room 3-E-144
Washington, D.C. 20301

1 Director
Advanced Research Projects Agency
Department of Defense
Washington, D.C. 20301

1 Director for Information Sciences
Advanced Research Projects Agency
Department of Defense
Washington, D.C. 20301

1 Director for Materials Sciences
Advanced Research Projects Agency
Department of Defense
Washington, D.C. 20301

1 Headquarters
Defense Communications Agency (333)
The Pentagon
Washington, D.C. 20305

20 Defense Documentation Center
Attn: TISIA
Cameroon Station, Bldg 5
Alexandria, Virginia 22314

1 Director
National Security Agency
Attn: Librarian C-332
Fort George G. Meade, Maryland 20755

1 Weapons Systems Evaluation Group
Attn: Col Daniel W. McElwee
Department of Defense
Washington, D.C. 20305

1 National Security Agency
Attn: R4-James Tippett
Office of Research
Fort George G. Meade, Maryland 20755

1 Central Intelligence Agency
Attn: OCR/DD Publications
Washington, D.C. 20505

1 Colonel Kee
AFRSTE
Hq, USAF
Room ID-429, The Pentagon
Washington, D.C. 20330

1 Aerospace Medical Division
AMD (AMRXI)
Brooks Air Force Base, Texas 78235

1 AUL3T-9663
Maxwell AFB, Alabama 36112

1 AFFTC (FTBPP-2)
Technical Library
Edwards AFB, Calif. 93523

1 Hq SASSO (SMTTA/Lt Nelson)
AF Unit Post Office
Los Angeles, California 90045

1 Lt Col Charles M. Waespy
Hq USAF (AFRDSD)
Pentagon
Washington, D.C. 20330

1 SSD (SSTRT/Lt Starbuck)
AFUPO
Los Angeles, California 90045

1 Det #6, OAR (LOOAR)
Air Force Post Office
Los Angeles, California 90045

1 ARL (ARIY)
Wright-Patterson AFB, Ohio 45433
1 Commanding Officer
Office of Naval Research Branch Office
219 South Dearborn Street
Chicago, Illinois 60604

1 Commanding Officer
Office of Naval Reserve Branch Office
207 West 24th Street
New York, New York 10011

1 Commanding Officer
Office of Naval Research Branch Office
1030 East Green Street
Pasadena, California 91101

1 Commanding Officer
Office of Naval Reserve Branch Office
495 Summer Street
Boston, Massachusetts 02210

8 Director, Naval Research Laboratory
Technical Information Officer
Washington, D.C. 20360
Attn: Code 2000

1 Commander
Naval Air Development and Material Center
Johnsville, Pennsylvania 18974

2 Librarian
U.S. Naval Electronics Laboratory
San Diego, California 95152

1 Commanding Officer & Director
U.S. Naval Underwater Sound Laboratory
Fort Trumbull
New London, Connecticut 06840

1 Librarian
U.S. Navy Post Graduate School
Monterey, California 93940

1 Commander
U.S. Naval Air Missile Test Center
Point Magu, California 93041

1 Director
U.S. Naval Observatory
Washington, D.C. 20390

1 Chief of Naval Operations
OP-07
Washington, D.C. 20350

1 Director, U.S. Naval Security Group
Attn: G43
3801 Nebraska Avenue
Washington, D.C. 20390

2 Commanding Officer
Naval Ordnance Laboratory
White Oak, Maryland 21502

1 Commanding Officer
Naval Ordnance Laboratory
Corona, California 91720

1 Commanding Officer
Naval Ordnance Test Station
China Lake, California 93555

1 Commanding Officer
Naval Training Device Center
Orlando, Florida 32811

1 Commanding Officer
Naval Avionics Facility
Indianapolis, Indiana 46241

1 U.S. Naval Weapons Laboratory
Dahlgren, Virginia 22448

1 Weapons Systems Test Division
Naval Air Test Center
Patuxent River, Maryland 20670
Attn: Library

1 Head, Technical Division
U.S. Naval Counter Intelligence Support Center
Fairmont Building
4420 North Fairfax Drive
Arlington, Virginia

1 Mr Charles Yost
Special Asst to the Director of Research
National Aeronautics & Space Admin.
Washington, D.C. 20546
<table>
<thead>
<tr>
<th>Institution/Company</th>
<th>Address</th>
<th>Attn:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The John Hopkins University Applied Physics Laboratory</td>
<td>8621 Georgia Avenue, Silver Spring, Maryland 20910</td>
<td>Boris W. Kuvshinoff Document Librarian</td>
</tr>
<tr>
<td>Dr. Leo Young</td>
<td>Stanford Research Institute, Menlo Park, California 94025</td>
<td></td>
</tr>
<tr>
<td>Mr. Henry L. Bachmann</td>
<td>Wheeler Laboratories, 122 Cutterhill Road, Great Neck, New York 11021</td>
<td></td>
</tr>
<tr>
<td>School of Engineering Sciences</td>
<td>Arizona State University, Tempe, Arizona 85281</td>
<td></td>
</tr>
<tr>
<td>Engineering and Math. Sciences Library</td>
<td>University of California at Los Angeles, 405 Hilgred Avenue, Los Angeles, California 90024</td>
<td></td>
</tr>
<tr>
<td>California Institute of Technology</td>
<td>Pasadena, California 91109</td>
<td>Documents Library</td>
</tr>
<tr>
<td>University of California</td>
<td>Santa Barbara, California 93106</td>
<td>Library</td>
</tr>
<tr>
<td>Carnegie Institute of Technology</td>
<td>Electrical Engineering Dept, Pittsburgh, Pa 15213</td>
<td></td>
</tr>
<tr>
<td>University of Michigan</td>
<td>Electrical Engineering Dept, Ann Arbor, Michigan 48104</td>
<td></td>
</tr>
<tr>
<td>New York University</td>
<td>College of Engineering, New York, N.Y. 10019</td>
<td></td>
</tr>
<tr>
<td>Syracuse University</td>
<td>Dept of Electrical Engineering, Syracuse, New York 13210</td>
<td></td>
</tr>
<tr>
<td>Yale University</td>
<td>Engineering Dept, New Haven, Connecticut 06520</td>
<td></td>
</tr>
<tr>
<td>Airborne Instruments Laboratory</td>
<td>Deerpark, New York 11729</td>
<td></td>
</tr>
<tr>
<td>Bendix Pacific Division</td>
<td>11600 Sherman Way, North Hollywood, California 91605</td>
<td></td>
</tr>
<tr>
<td>General Electric Co Research Laboratories</td>
<td>Schenectady, New York 12301</td>
<td></td>
</tr>
<tr>
<td>Lockheed Aircraft Corp</td>
<td>P.O. Box 504, Sunnyvale, California 94088</td>
<td>Librarian</td>
</tr>
<tr>
<td>Raytheon Co</td>
<td>Bedford, Mass 01730</td>
<td></td>
</tr>
<tr>
<td>Dr. G. J. Murphy</td>
<td>The Technological Institute, Northwestern University, Evanston, Illinois 60201</td>
<td></td>
</tr>
<tr>
<td>Dr. John C. Hancock, Director</td>
<td>Electronics Systems Research Laboratory, Purdue University, Lafayette, Indiana 47907</td>
<td></td>
</tr>
<tr>
<td>Director</td>
<td>Microwave Laboratory, Stanford University, Stanford, California 94305</td>
<td></td>
</tr>
<tr>
<td>Emil Schafer, Head</td>
<td>Electronics Properties Infor. Center, Hughes Aircraft Co, Culver, California 90230</td>
<td></td>
</tr>
</tbody>
</table>
1 Dept of Electrical Engineering  
Texas Technological College  
Lubbock, Texas 79409

1 IBM Technical Info. Retrieval Center  
International Business Machines Corp.  
Armonk, New York 10504

1 Commander  
Test Command (TCDT-E)  
Defense Atomic Support Agency  
Sandia Base  
Albuquerque, New Mexico 87115

1 Commanding General  
U.S. Army Weapons Command  
Rock Island, IL 61201  
Attn: ANSWE-RDR (Gerald Reinsmith)

1 Col E.P. Gaines, ACDA/FO  
1901 Pennsylvania Ave, N.W.  
Washington, D.C. 20451

1 Mr Billy Locke  
Plans Directorate  
USAF Security Service  
Kelly Air Force Base, Texas 78241

1 W.A. Eberspacher  
Technical Consultant  
Systems Integration  
Code 5340A, Box 15  
Naval Missile Center  
Point Magu, California 93041

1 Director of Faculty Research  
Department of the Air Force  
USAF Academy  
Colorado 80840

1 Lt Col Richard Bennett  
AFRDD  
The Pentagon  
Washington, D.C. 20301

1 Weapons Systems Evaluation Group  
Attn: Col John B. McKinney  
410 Army-Navy Drive  
Arlington, Virginia 22202
This report is designed to be of use as both a reference manual for Cylinder programming and as a more general introduction to the linked-list notions upon which CYL6, our Cylinder system, is based. Accordingly, it is divided into two major sections, the second being a detailed manual for the CYL6 routines, the first containing only more general description. A reader unfamiliar with list languages should start at the beginning of Part I, while beginning as section (γ) of Part I is appropriate otherwise. For routine reference use, only the Appendix, which is a condensed table of subroutine data, should ordinarily be needed.

Finally, because the rate of change of CYL6 is not yet absolutely zero, this report must be considered as a provisional description of it.
<table>
<thead>
<tr>
<th>Key Words</th>
<th>LINK A</th>
<th>LINK B</th>
<th>LINK C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rings</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked Lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computer Programs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subroutine Systems</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Security Classification