A FRESH APPROACH TO DATA BASE MANAGEMENT SYSTEMS
PART II: EXTERNAL SPECIFICATIONS FOR THE TEXTAG DBMS

Wm. Orchard-Hays

December 1978 WP-78-65

2361
Laxenburg
Austria

International Institute for Applied Systems Analysis

Contents

A FRESH APPROACH TO DATA BASE MANAGEMENT SYSTEMS

Wm. Orchard-Hays

PART II: External Specifications for the TEXTAG DBMS

FOREWORD

This is the second part of a two-part paper. In Part I, more general considerations for a data base management system were discussed. In this part, detailed external specifications for a particular DBMS are presented. A few additional observations of a conceptual nature are also made.

A few definitions given in Part I are used here without restatement. The reader may find it necessary to refer to the earlier part for the definitions of datum (plural "datums") and their distinction from data, attribute, item, peer groups and inferior sets.

1.0 INTRODUCTION

The DBMS presented here is a system for creating, maintaining and using data banks of wide diversity, but primarily for highly-structured, "refined" data. The capacity depends, of course, on the underlying (host) computer system on which it is implemented. The design depends on an interactive host which is also assumed to have various peripheral devices in addition to a reasonably high-speed central processor with good character handling capability and large, fast storage devices such as disks or drums The system itself is intended to be permanently available on an attached storage device together with users' files. Although the system has its own specialized filing subsystem, this in turn is dependent on good file-handling capabilities in the basic hardware and software. Provision is made for use of magnetic tapes for back-up storage and mass transfers but normal operation requires random-access devices.

The system is not merely an unsubstantiated proposal. An existing system for a specialized area may be regarded as the prototype of TEXTAG. All the techniques implied by the external specifications given in the sequel have existing, working counterparts in the prototype system. The question is not whether the TEXTAG system can be effectively and rather inexpensively implemented -- it can be on at least one host system -- but whether it will provide a useful general capability for data management. The prototype system has proven extremely effective in one application area and TEXTAG will have modified or additional features of greater generality, without some of the narrow specializations of the prototype. This does not prove TEXTAG's overall applicability and convenience, of course. This document should provide the means of desk-checking, so to speak, possible applications. However, no such system can be fully appreciated without trying it with a live implementation.

The prototype system was implemented on an IBM 370 under VM/CMS. To avoid vagueness and hedging on some operational matters, TEXTAG is presented as though implemented on the same

host. (Undoubtedly, the first implementation should be to take advantage of a large body of existing computer code which is applicable and well-tested.) Clearly, other suitable host systems exist and, for the most part, implementation on one of them would not affect TEXTAG capabilities or language. The main exception to this might be in the declaration and accessing of files, and the related matter of loading user-provided subroutines from a subroutine library. The latter is an important capability which host systems do not make too easy to effect. The existing prototype system has this feature but certain practical limitations exist. The prototype system is not portable in general though adaptation to a couple of other makes of computers would be relatively easy.

The name TEXTAG could probably be given several different acronymic interpretations. In fact, however, it is formed from two proto-Indo-European roots: teks meaning to weave or fabricate, and tag meaning to arrange or set in order. These two concepts reflect the central purpose of the system and, roughly, the kind of resulting construct. Data must be arranged and it must be threaded into a pattern that renders it useful, that is, a data bank must be fabricated, both from actual data and from the relationships perceived in and imputed to it. The analogy to weaving is thus not so far-fetched.

The presentation is conversational in style. Concise summaries for reference are given in appendices. Section 9 gives the complete set of commands but grouped functionally. Insofar as possible, displays such as prompting or instructional text produced by the system will be identical to those used herein. After reading this document, a prospective user should already have a feel for the system's style and be somewhat at home on first use of it. Explaining a DBMS in some language completely different from that used by the system itself would be a foolish impediment to the purpose. The conventions used should be readily understandable to anyone with some experience in common computing practice.

External representations of numbers are essentially those of

FORTRAN but used more or less indifferently. For nonnumeric datums, single quotes enclosing a datum always mark it as nonnumeric, i.e., a character string. An unenclosed string which looks like a number is always taken as a number.

1.1 Typing Conventions in This Document

Certain conventions over and above a language are necessary for discussing the language. (These are sometimes called a meta-language.) The following are used in the sequel.

| A vertical bar denotes

(1) OR in the sense of union. Thus if a and b are sets, a|b denotes their union.

(2) OR in the sense of either. Thus if a,b,c denote optional values or sets, a|b|c means any one of or any member of any one of a or b or c may be used.

The sense (1) is used only in Figure 1 in the next section which defines primary datum forms. Thereafter, a form set is referred to only as a form and denoted by it letter-name.

[ ] denotes optional.

... denotes "more of the same" as what preceded.

{ } denotes a set (except in Figure 1).

a,...,b denotes "from a up to b" where the sequence is obvious or explained on the side.

upper-case is used for explicit command names, form designators, key words and the like. Upper-case is also used for an explicit example. Lower-case is used for a general example. Thus "A" is an example of "letter". Use of upper and lower case herein should not be construed as indicating actual typing of statements. Typing conventions are largely functions of the terminal and host.

2.Ø FORM AND STRUCTURE OF THE DATA BASE AND BASIC COMMANDS

We begin this section by defining the complete set of all possible datum forms and build up structures from there. The term root refers here to the main node of the data bank, which is defined automatically when a data bank is initiated.

B	bit	(Ø,1)	
X	hex byte,RJ	(Ø,1,...,255)	
I	integer, RJ	(-32767,...,32767)	2s-complement
E	short floating-point	Ø.Ø|±(16^-15,..,16^3Ø)	precision 16^-6
F	full floating-point	Ø.Ø|±(16^-15,..,16^3Ø)	precision 16^-14
L	letter char	($,a,b,...,z)	
D	digit char	(Ø,1,...,9)	
A	alphanumeric char	.|L|D	
C	any printable char		
N	symbol (name)	L[A...]	max of 8, LJ
W	word (no internal ')	'C[C...]'	max of 8, LJ
S	string (no intrnl ')	'C[C...]'	max of 255, as is
P	pointer to array	N	
R	reference to text	N	

Figure 1: Allowable Datum Forms

2.1 Allowable Datum Forms and Pairing of Numbers

In Fig.1, all allowable datum forms are shown, where a vertical bar is used to indicate OR in the sense of union. Thus the set of alphanumeric characters, A, is the union of the period, the letters and the digits. Parentheses enclose actual members of a set, abbreviated by use of dots between lowest values and the highest value. The order shown is the sort order. A symbol (form N) must start with a letter and may optionally have up to seven additional characters from the set A.

Note that all sets are finite, even though the full set of

printable characters is not shown and hence the cardinality of forms W and S cannot be precisely determined. More importantly, the floating-point forms are finite in both range and precision. Form E, for example, has a cardinality of

$$11*2^{27}-83 = 1.4*10^{9} \text{ approximately}$$

or less than one and a half billion values to cover a range of about $2*10^{36}$. By contrast, form N has on the order of 10^{12} members and yet any two are clearly distinguishable. This distinction between representation and so-called "real" numbers should never be forgotten. Note also that use of form F (REAL*8 in IBM FORTRAN) gives about $4*10^{12}$ additional values between every adjacent pair in set E. This makes clear the importance of "double precision" in many kinds of numerical calculations.

The forms defined in Fig.1 are internal forms but they are essentially the external forms also with the following exceptions.

(1) The single quotes around a word are external, they are dropped internally; the same is true of a string but the internal string has one more byte than the number of external characters. The notation LJ means "left-justified, blank filled on the right".

(2) All numbers are written in decimal notation externally. The notation RJ for forms X and I means "right-justified, zero filled on the left". However this and the internal 2s-complement notation for I are really of no concern to the user. They work the way one expects.

(3) Floating-point forms are written externally in a subset of FORTRAN forms: signed or unsigned integers, mixed numbers (with decimal point), or exponential forms consisting of either an integer or a mixed number followed by E followed by a signed or unsigned exponent of one or two digits. The form "Ebdd" where "b" means blank and "d" means digit is also accepted for compatibility with FORTRAN formatted output.

(4) Sets P and R have a different interpretation explained below.

The following conventions are in effect for the typing of blanks and numbers. With the exception of enquoted strings (forms W and S) and the form "Ebdd" above, wherever one blank may be used, several may be used. Blanks may surround separators (operators, commas, etc.) or not as desired. No blanks may appear within the representation of a number except the special form "Ebdd". Generally speaking, however, any external form of number may be used for any form of internal number, provided it is in the allowable range. All numbers are converted to F form on input and then fixed and/or truncated as appropriate to the specified internal use. However, for direct arithmetic, the external form is remembered and honored as though the value had already been internalized to the apparent form. Thus the effect of the following expressions should be carefully noted.

1/4 gives zero

1.Ø/4 or 1/4.Ø or 1.Ø/4.Ø give .25

4/1 gives 4

4.Ø/1 or 4/1.Ø or 4.Ø/1.Ø give 4.Ø

That is, pairing an integer with a "real" produces a "real". This holds for intermediate results as well. Thus

(1+3)/8 gives zero

(1+3.)/8 or (1.+3)/8 or (1+3)/8. give .5

For variables, the above rules hold also, where the type of variable (forms B, X, I, E, F) determines the implied form. Conversion to proper form for a result is automatic provided it is within specified range. An exception exists for form B, however. Only the parity of the whole number part of a result is used to assign a value to a bit. Thus

1, 3, -5, 3.14, -1

if assigned to a B variable, all produce a value of 1.

Ø, 2, -4, 2.739

all produce a value of Ø.

A B-variable may be used either as an integer (Ø or 1) or as a boolean variable (1=true, Ø=false). Thus the OR of Ø and 1 B-values gives 1 and the AND of them gives Ø. The result of a relational operation is also a B-value. Thus the expressions

3.5+(x<y)

3.5+(a AND b)

where x,y are numeric variables and a,b are B-variables are legal and produce either 3.5 or 4.5. The parentheses are necessary; without them the result is a B-value equivalent to the expressions

(3.5+x) < y

(3.5+a) AND b,

respectively. In the latter case, the parity of the whole number part of (3.5+a) is used as a B-value, that is, a number paired with a B-value for a boolean operation is first converted to a B-value.

An R-datum is a symbol used as a reference to an enfiled body of text which is stored in a separate section of the data bank. The body of text is given a name (same as the external R-datum in N form) which is recorded both as the datum value and in the directory of the special section of the data bank. Defining an R-datum does not define the text or cause any other action. It is for reference only.

A P-datum differs completely in external and internal forms. It refers to one of four forms of data arrays which can be attached to a searchable entity. The external P-datum in N form is a temporary symbolic name for the array which must be entered in formats to be explained in the sequel. It is first necessary to explain the concept of nodes.

2.2 Node Structure

The data bank contains two kinds of data: node-structured and attached. Although all attached data can be accessed down to the datum level, primary searches are made and implied sets defined on the basis of a node structure. The latter is the subject of this section.

The node structure is based on the notions of attributes, items, entities and relationships. A set of items is always de-

fined by attributes but not all items are entities and not all entities are items. An entity is a data structure that is entered (or transferred) as a unit, though it may subsequently be altered in content (but not in structure). When an item is also an entity, it is called a record. The datum values for its attributes are regarded as residing in fields which collectively form the record. A record may also have entities (arrays) attached to one or more fields (form P) but these do not enter into the definition of a node. (This is further elaborated under search levels in the sequel.)

A record as defined above does not constitute a node. In order for a set of records to become a set of nodes, one or more relationships must be defined among them. These relationships are called primary relationships and any one node Q can have at most four:

its downward link, D(Q)
its left peer, L(Q)
its right peer, R(Q)
its backward link, B(Q).

We also define an upward link, U(Q), for the purpose of definitions. If Q1=U(Q2), then Q2=D(Q1) and B(Q2)=Q1. Every Q except the root has a B(Q). The use of L(Q) and R(Q) require an orderable attribute called the key attribute. If Q2=L(Q1) or R(Q1), then B(Q2)=Q1. In any peer group {Qi}, one and only Qi, called the hook, has a U(Qi). If Qh is the hook and U(Qh)=Q, then Q is the superior of {Qi} and any Qi is an inferior of Q. If Q is at level v, then {Qi] constitutes the part of level v+1 hooked to Q.

Every set {Qi} has a numeric key, k, constant over the set. At level v, $k \geq v$ but the k-values need not be consecutive. It is usual but not strictly necessary to have

$$\min k(v+1) > \max k(v)$$

and similarly that the k-values for different sets {Qi} at one level be distinct. Following these rules permit the programming of built-in checks against errors in assignment or search specification. The range of k-values assignable by the user is 10 to 255, the first ten digits being reserved for the system.

It is possible that a set of records has no orderable attribute since, although every datum form is orderable, uniqueness of values is required for one attribute of a set {Qi}. If no attribute can be well-ordered, then no inferior set can be defined meaningfully. Such data is either meaningless or it should be put in an array which is attached to a meaningful node.

Every set of records has one implied order, namely the order of creation. This order is not remembered explicitly by TEXTAG but all records entered consecutively are in fact stored in the same order. The prototype system utilized essentially the above-described node structure and takes advantage of this feature for special purposes but this is not done in TEXTAG since no general utility is apparent. However, a record of date of creation or last modification is provided for nodes as described in the sequel.

It should be noted that records should not be entered in order on the ordering attribute. Doing so results in all R(Q) and no L(Q) relationships so that all searches are linear. On the other hand, all members of one set {Qi} should be entered together if possible so that searches do not jump around throughout the physical file. Sometimes it is necessary to reconstitute one or more major branches to improve retrieval characteristics. The language provides facilities for this.

2.3 Search Paths, Identifiers, Locators and Implicit Sets

Although not strictly necessary, it is usual for a set of peer nodes to be ordered on an attribute whose datum values are symbols (form N). Forms B, E, F, P and R may not be used. For any one node, its value of the attribute is called its referent. However, a referent is only meaningful and necessarily unique within its peer group. Furthermore, the group's key attribute needs to be known by name, and also the numeric key to control some searches.

Suppose a node structure has been created and it is desired

to find some particular node. This is done (in the first instance) with a command called FIND. Although this is the simplest search instruction, it already shows considerable complexity; for example, it is important where one starts. The system maintains three pointers called ROOT, START, STOP. (These symbols are preempted only in commands where ambiguity is easily avoided, in fact impossible.) Initially the system is located at the root node and all three pointers point to it. ROOT is never again changed, either by the system or by the user (who is blocked). Associated with the three pointers are the k-values (Ø for the root) belonging to the nodes.

The definition of the structure of each record (i.e., the attribute names and forms) in a peer group is kept with the D(Q) which leads to it; the key attribute is always defined first and hence its name is in a known position. (How attribute sets are defined is given in section 2.5.) Suppose the data bank has three main branches, or sublibraries, for North America, Western Europe and Eastern Europe. Say the main nodes for these branches have a key attribute named WRLD.RGN, form N, with the values N.AMER, W.ERUOPE and E.EUROPE, respectively.

Perhaps the user has forgotten the record layout and key attribute. If START points at the root, he can have it displayed with the command

DISPLAY LOWER

However, he need not do this to find, say, W.EUROPE if he knows that symbol. It is sufficient to issue the command

FIND W.EUROPE

The STOP pointer will now be set to this node and its k-value recorded. If the user wants to know what has been recorded about Western Europe as a whole, he can issue the command

DISPLAY NODE

If he now wants to see the record layout, he can issue

DISPLAY LOWER(START)

When no subargument is given to the main argument for DISPLAY, the subargument STOP is implied. Thus the prior command is equivalent to

DISPLAY NODE(STOP)

i.e., STOP is "where we are" and START is the superior.

To investigate the substructure under W.EUROPE, the following command must first be issued

SET START = STOP

Although general assignment statements require no verb, the verb SET is necessary here; it notifies the system that we are referring to system variables, not user-defined ones. Only START or STOP may appear on the left, never ROOT. The START and STOP pointers may be held with the command

SET HOLD(i)=START|STOP

where i is an integer literal or variable. Up to 16 pointers may be held (i=1,...,16). If anything but ROOT, START or STOP appears on the right of such a SET phrase, it must be of the form

SET STOP|START=HOLD(i)

This is not an ordinary vector and has no correspondence to any vector or list of the same name defined by the user.

Suppose the main branch nodes have inferior sets with a symbolic key attribute COUNTRY and each of the countries represented has an inferior set with the key attribute MAJ.SUBD (for major subdivision). To find the "bundesland" Niederosterreich in Austria, starting again from the root, the following commands could be issued:

```
FIND W.EUROPE
SET START=STOP
FIND AUSTRIA
SET START=STOP
FIND NIEDEROSTERREICH
```

However, it can all be done with one command

FIND W.EUROPE_AUSTRIA_NIEDEROSTERREICH

After both cases, START is set to AUSTRIA and STOP to NIEDEROSTERREICH. The argument to the last FIND is called an identifier, which is a chain of attribute values concatenated with underlines. If NIEDEROSTERREICH did not exist under AUSTRIA, the STOP pointer would end up at the node which would be the nearest peer, i.e., the one for which NIEDEROSTERREICH would be L(Q) or R(Q) if

it did exist. The message

SEARCH FAILS AT LEVEL 3

would be displayed. If AUSTRIA had no inferior set, the START pointer would remain at W.EUROPE and the STOP pointer at AUSTRIA. The following message would be displayed

SEARCH IMPROPER AT LEVEL 2

(The failure of a search can be tested in a program as explained subsequently.)

Note that the attribute MAJ.SUBD must be defined as having S-form with a length of at least 16 to hold the long name. This must not be confused with the name of the attribute, i.e., MAJ.SUBD. Attribute names must be of form N.

Several other search commands are defined in the sequel but further capabilities of FIND need to be pointed out here. Suppose the above search was successful and we are next interested in Bavaria in West Germany. Let us suppose GERMANY is used under both W.EUROPE and E.EUROPE. We could start again from the root (which might be the most efficient here) with the following two commands.

```
SET START=ROOT
FIND W.EUROPE_GERMANY_BAVARIA
```

However, in more complex structures we might want to do it in a different way. One way is to use the command FINDSUP (find superior). The two commands

```
SET STOP=START
FINDSUP
```

would get us back to START=W.EUROPE. The command

```
FIND GERMANY_BAVARIA
```

would then get to the desired node. However, this can also be done in one instruction using a "backup" referent in the identifier, viz.,

```
FIND -2 GERMANY_BAVARIA
```

Note that the instruction

```
SET STOP=START
```

used above is equivalent to backing up one level. Using FINDSUP twice would have had the same effect but been much less effi-

cient. The referent -2 used to begin the identifier in the last FIND command above thus properly means "back up two levels". In other words, FINDSUP starts from STOP rather than START and the backup referent for FIND, while interpreted as meaning to begin with that many FINDSUP commands, takes advantage of the known relationship between STOP and START. The use of -1 is always redundant. For example, the instruction

FIND -1_OBEROSTERREICH

has the same effect as

FIND OBEROSTERREICH

since START remains at AUSTRIA in either case.

Before ending this section, one more example will be given to illustrate the concept of implied sets of items. Suppose each country node for each region has an attribute LANGUAGE (to record the main language spoken) and it is desired to find all countries where the main language is German. There is a command SEARCH which accepts a variety of arguments; only four forms will be shown here. Starting from the root, any of the following might be used (the operator <EQ> meaning "equal" in explained in section 4.3):

(i) SEARCH WRLD.RGN_COUNTRY [FOR] LANGUAGE <EQ> GERMAN

(ii) SEARCH DOWN 2 [LEVELS][FOR] LANGUAGE <EQ> GERMAN

(iii) SEARCH TO K [FOR] LANGUAGE <EQ> GERMAN

(iv) SEARCH AT K [FOR] LANGUAGE <EQ> GERMAN

Note that (i) uses a chain of attribute names, not values. This is called a locator rather than an identifier. In the present case, it is redundant but would not be if, for example, some region had an attribute STATE instead of COUNTRY. The form (ii) simply searches down two levels looking for the attribute LANGUAGE, regardless of the key attributes (which are defined with the D(Q)s). Form (iii) will search down as far as the level with the specified k-value for the attribute LANGUAGE. Depending on how k-values were assigned, this might not be the same level for all node sets. Form (iv) restricts the search to a specified k-value.

Levels are processed in order on key attributes, in the

present case alphabetically. Something like the following output would be produced.

```
WRLD.RGN=E.EUROPE
     COUNTRY=GERMANY
WRLD.RGN=N.AMER
     (NONE)
WRLD.RGN=W.EUROPE
     COUNTRY=AUSTRIA
     COUNTRY=GERMANY
     COUNTRY=SWITZ.
```

This output amounts to an implicit set of items whose attribute set might be called GERMAN with the attributes WRLD.RGN and COUNTRY. (Note that, being implicit, it has no key attribute.) More elaborate forms of SEARCH can further restrict or enlarge the set and also permit full examination of nodes, with further disposition of the information as desired.

2.4 The Nature of Information

Referring to the previous subsection, one can note the three-tiered nature of any data which can be regarded as information. First is the name or other designation of the set of defining attributes. Second are the names of the attributes which are the values of the defining set and which are assumed to be meaningful to a reader. Third are the values of the attributes occurring in the items. This is the absolute minimum for meaning. Stated differently:

1. What class are we talking about?
2. What attributes are we interested in?
3. What values of those attributes occur?

These are not independent. The class is defined by at least some attributes, and attributes are abstractions based on values or qualities that have been observed and given some general label. Nevertheless, understanding or knowledge is based on an organization like that above which is independent of the particular repo-

sitory of information or the system used to manipulate it. For example, if our attention is directed to the class of objects known as humans, they are distinguished by various attributes. One attribute is known as language and one of the particular languages spoken is German. Additionally, the information is organized by where people live, etc. The complete structure of such knowledge is exceedingly complex and the extent often vast. In any practical use of it, we must restrict our attention to only a few attributes and a modest number of distinguishable grades of these attributes. Even when we consider a continuum, this becomes a discrete subject of study; no one supposes that every possible value will ever be observed, much less recorded.

This is perhaps a good place to point out again the types of applications for which TEXTAG is suitable, or rather unsuitable. Bibliographic data is, in anything like pure form, not a suitable body of material to be organized with a system like TEXTAG. The scope of literature in almost any field has reached such proportions as to approach a continuum. Until and unless such a body of literature has been extensively, expertly and consistently abstracted -- a gargantuan task in most cases -- no system designed for elaborate organization with precise referents and other labels can deal with it. There is simply no way to find, for example, all references to a particular subject area by a particular author or school unless someone has laboriously dug out and organized such information and systematically recorded it. Such a task could be greatly aided and made more fruitful by a system like TEXTAG, once a human being had done the initial investigative work. Users of bibliographic data banks do not always seem to understand this or else assume the necessary work has been done. In only a few specialized cases is the latter true.

TEXTAG provides some capability for storing unorganizled text but huge volumes are not contemplated. This will be explained in the sequel.

2.5 Defining an Inferior Set

The first task in constructing a data bank is to establish its main branches. The bare system provides a root node with nothing hooked to it. The definition of the main branches is very important since all substructures will be influenced by it to some degree. This is, of course, true at any level but the first breakdown affects more nodes, obviously.

In the previous example, for instance, does one really want to segregate data by world regions or by main subject areas. This will depend on the purpose and intended use. It is a decision the person who constructs the data bank must make and will have to make again and again at lower levels. TEXTAG does provide commands for reconstituting all or part of a data bank into different organizations so mistakes need not be lived with indefinitely. However, the necessary instructions can be intricate and time consuming to execute so one should use as much foresight as possible in the original breakdown.

Starting from any node pointed to by STOP (the root the first time), one defines an inferior set with the following instruction:

DEFINE LOWER K=k [NP] [SOURCE=filename,deckname]

The option NP means "no prompt". The SOURCE option indicates that necessary definitions are to be read from a deck in a card-image input file and implies NP. If neither option is used, the following prompt is displayed:

TYPE IN ATTRIBUTE DEFINITIONS, KEY ATTRIBUTE FIRST.
TERMINATE WITH ":END" ON A NEW LINE.

Input lines, only the first of which starts with ATTRIB: and containing phrases as indicated are then expected.

ATTRIB: symbol [AS] form [,symbol ...]

where the symbols are in N form and "form" may be any of the fol-

lowing:

B|X|I|E|F|L|D|A|C|N|W|P|R

without further specification; or

X|I|E|F = (min, max)

showing allowable range of values; or

S=length|C='string'|W=symbol

However, for the key attribute (first one), the forms B, E, F, P and R may not be used. The last three forms require explanation. If it is desired to use S form for an attribute (for instance, MAJ.SUBD in the earlier example), the field length must be defined. This is the only form whose length is not fixed. The phrase C='string' defines in the string the particular characters which are allowable; otherwise C form permits any printable character.

The phrase W=symbol permits the field to have the form of a general word but be restricted to specified values. The symbol refers to a form of array called a LIST which must either be defined by a packet of lines for a list following the last attribute definition or have been entered previously into temporary storage (explained later). The format for a list is explained below. This use of a list should be employed sparingly since it adds considerably to the size of the D(Q) substructure. Although the same list can be used with different inferior sets during definition, it must be copied into each D(Q) for subsequent use.

The form :END beginning a line is used to terminate all packets of related lines input to TEXTAG, whether from the terminal or from a file.

Note that several attributes can be defined on one ATTRIB: line by separating them with commas. Continuation lines can be used as required or desired but the break must be at a comma. If input is from a card-image file, all the above conventions apply just as though the lines were typed. The deck structure in such a file is explained in section 6.1.

To illustrate the foregoing, the first level of the earlier

example might be defined as follows:

```
DEFINE LOWER K=10 NP
ATTRIB:  WRLD.RGN N, POP.MILL I, AREA.THO I, NO.CNTRY X
:END
```

Note that DEFINE adds the D(Q) substructure to the node pointed to by STOP. If a D(Q) already existed, an error message is displayed.

It is now necessary to create the nodes themselves. This is done with the command CREATE. For checking purposes, this command requires that K be specified again. It permits two forms of input: either a free-form list of attribute datums in the order specified by DEFINE, starting with the key attribute and complete up to right-hand dropoff, or a list of identified phrases which may or may not be complete. Unspecified attributes are set to their null values (0 or blank). However, the key attribute value must be specified. The general form is

```
CREATE INFERIOR K=k [ID] [SOURCE=filename, deckname]
```

The argument ID indicates that identified phrases of the form

```
attribute-name=value
```

will be used. In either case, the values are entered on one or more lines following of the form

```
NODE:  attribute-name=value [,...]
```

if ID was specified or

```
NODE:  value1 [,value2 ... valuen]
```

if ID was not specified and the node has n attributes. The lists are terminated as usual with :END on a new line. The indicator NODE: occurs only on the first line.

Tne superior to the node being created is pointed to by STOP both before and after execution. Several nodes can be defined in succession by starting each one with NODE: on a new line before :END appears. Note that "NODE:" cannot be a value; it is a preempted construction. The same is true of ":END". If for some reason these must be used as datums, they must be enclosed in single quotes.

An entire set of ATTRIB:, CREATE and NODE: lines, followed by appropriate :END lines, can be executed from a card-image file

with one DEFINE using the SOURCE option. Similarly, such a SOURCE phrase can be used with the CREATE command. More generally, any number of commands can be enfiled in a deck and executed with the RUN command, explained in the sequel.

2.5.1 Attachments to a Node; Annexes and Arrays

Two special cases occur with attributes of form R and P. The symbol for an R form refers to a body of unorganized text called an annex which either has been or will be entered into an auxiliary file of the data bank. These annexes are stored in a separate file, simple in structure but possibly voluminous. Each annex must have a unique referent symbol. No check is made that the annex exists when a node referring to it is created. An annex is not uniquely associated with one node. If it is desired to simply reserve space for a later reference to some annex when a node is created, the attribute datum can be left blank. Note that an R-form attribute really accomplishes nothing that an N-form couldn't; it is just a convenience and allows searching on a known attribute type. The enfiling of annexes is explained in section 8.

The symbol for an attribute of form P is entirely different; it is strictly local and temporary, used merely to identify a following packet of lines which define an array. The lines defining all arrays for the node must appear after all node lines but before the :END line. Order of the packets need not be correlated with order of node fields; each packet ends with its own :END line nested inside the one for the node as a whole. If any P-form attributes remain undefined, the missing arrays are looked for in temporary storage provided the proper reference form is used for the attribute value. If not found, an error is declared. Rules for this reference form are given in section 3.2.

Arrays are of four kinds:

(1) A vector, introduced by the following line

VECTOR: symbol(n) [,I|E|F][,PACK][,INDEXED]

where

(a) "symbol" is the P-form attribute referent;

(b) n is the vector length;

(c) I|E|F defines the number form, E being the default;

(d) PACK may be used only with F; only nonzero values are stored and each one is truncated to a precision of 16^{-10};

(e) INDEXED indicates that only nonzero values are supplied with phrases of the form

index=value

The actual values are entered in following lines in free form, either

value1, value2, ..., valuen

or

index=value, index=value, ...

if INDEXED is specified. If the indexed form is used for a vector to be packed, enough values should be given to allow for the maximum number of nonzeros expected in later revisions. If an additional index is subsequently specified for an already full packed vector, an extension will be created but this is very likely to be in a remote section of the data bank which will lead to great inefficiency. Zero values may be used to reserve positions. This cannot be done with unindexed input for which only nonzero values will be packed.

(2) A matrix, introduced by the following line:

MATRIX: symbol(m,n) [,I|E|F] [,PACK] [,INDEXED]

where the meanings are the same as for VECTOR but m,n are the two dimensions. Each row is entered as a vector, starting with ROW: on a new line. Every row must be represented, whether in full or indexed form. Only one :END is used to terminate the entire matrix, not each row.

(3) A list which is a vector of words (enquoted if necessary) which must be given in full and cannot be packed. A list is introduced with

LIST: symbol(n)

(4) A table which is a rectangular array with a symbolic stub and head. The stub is regarded as column Ø and the head as row Ø and these are not counted in the dimensions m,n. Tables are of three kinds: those with numeric elements, those with symbolic elements, and those whose rows consist of character strings. All tables are stored in full array format.

(a) Numeric tables. There are three sizes, form I, form E or form F. The stubs are always 8 characters in length but the heads contain 2-character, 4-character or 8-character strings, respectively. These characters may be form C but enquoted if not like truncated or full symbols. A numeric table is introduced with the line

TABLE: symbol(I|E|F)=head1, ... headn

defining n columns and each row is introduced with the line

ROW: $stub_i$=$elem_{i1}$ [, $elem_{i2}$, ...]

that is, right-hand dropoff is permitted.

(b) Symbolic tables. These are also of three sizes, indicated by a number. The introductory line is

TABLE: symbol(2|4|8)=head1, ..., headn

Otherwise the forms are the same as for numeric tables except that elements are character strings (of length 2, 4 or 8) and not numbers.

(c) Character string tables. These tables contain up to 8 strings in each row where each string contains up to 64 characters in increments of 8. The head is stylized to define these strings, as follows.

TABLE: symbol(S)= Aa [, Bb ... Hh]

As many (capital) letters, in sequence, are used as the number of strings per row. The lower-case letters here stand for digits from 1 to 8, independently. The digit specifies the number of 8-character increments (words) in

the string. For example,

A3, B1, C2

specifies three strings: the first 3*8=24 characters long, the second 1*8=8 characters long, and the third 2*8=16 characters long. In defining the rows, strings must be enquoted and separated by commas. The strings may be shorter than the specification in which case the recorded string is blank filled on the right. Right-hand dropoff is also permitted in which case missing strings are set to blanks. Each row is specified by

ROW: $stub_i$ = 'string A' ['string B' ...]

Strings are addressable down to individual words. A table reference index-pair such as ($stub_i$,C1) for the above example of a head definition would get the first 8 characters of the third string in the i-th row. This is also the fifth word in row i so the index pair could be given numerically as (i,5). Table references are explained more fully later.

3.0 WORKING STORAGE, VARIABLES AND ARRAYS

The data bank proper is a file which is buffered through high-speed (virtual) storage where it is used and modified. Additionally, however, a large block of high-speed storage is used as working storage to hold variables, arrays and other information. This working storage is actually organized in threaded sets of similar entities but the user need not be concerned with this. However, the user must be aware of the distinction between the data bank, which has a permanent status, and working storage, which is transient and disappears on exit from TEXTAG. (Arrays may be saved in and restored from an auxiliary file.)

Most of the work done with and by the system is actually handled in working storage. A fairly complete programming language is provided which can be used for calculations either on operands from the data bank or independently of it. (Indeed, TEXTAG is a good general-purpose computing system for interactive

or semi-automatic use.) In this section, the kinds of variables and arrays which may be defined and used in working storage are described. The set of basic operations available are described in section 4.Ø.

3.1 Variables and Prefixes; References to Node Values

Six types of single-valued variables are provided. These correspond to a subset of the datum forms and the type of variable to be used or created is always indicated by the datum form letter used as a prefix to its name. With one exception, variable names are always symbols (form N) but, within that restriction, are arbitrary. Variable names must be unique within a type. The exception occurs for bits (form B); only 26 of these are available and they have the predefined names A,B,...,Z. A 27-th one with the name $ is set by find and search commands to indicate sucessful (1) or unsuccessful (Ø) completion. It may be tested but not assigned.

A prefix is a letter followed by a full colon which immediately precedes the symbol it specializes. The six types of variables and their reference forms are as follows:

B:letter	a bit	(B form)
I:symbol	an integer	(I form)
E:symbol	a short floating-point value	(E form)
F:symbol	a full floating-point value	(F form)
N:symbol	an 8-byte word	(N or W form)
S:symbol	a character string of defined length.	

With the exception of a bit or an S-variable, a variable is defined when it first appears on the left of an assignmnet statement. If it occurs on both left and right on first appearance, the value in the expression on the right will be zero for a numeric variable and blanks for an N-variable. The predefined B-variables are initially all Ø.

An S-variable must be defined with the command FORM,

FORM S:symbol= n|I:symbol

where n or the I-variable defines the length of the string. The string is initially all blank. The length of a string previously defined can be recovered with the command DIMEN (meaning "get the dimension") in a statement of the form

DIMEN I:symbol=S:symbol

The I-variable may also be defined with such a statement. FORM and DIMEN have more general uses for arrays. DIMEN could have been defined as a function rather than a command but would have been more awkward in practice that way even though more "readable" in this case, i.e. the (incorrect)

I:symbol=DIMEN(S:symbol)

In the more general uses, the nonstandard syntax in a functional form would lead to undue complexity in the parser and no significant benefit to the user.

There is a seventh prefix to refer to the attribute values of tne node pointed to by the STOP pointer. This can be regarded as a generalized variable prefix. It has the form

D:attribute

where "attribute" is the name of the attribute. The meaning of such a reference obviously depends on what node is currently available. (If none, the reference results in an error being declared.) Since not all attribute forms have corresponding variable types, it is necessary to define how different attributes are treated by a D-reference. This is given in the following table.

attribute form	equivalent variable to	type
B		B:
X		I: (nonnegative only)
I		I:
E		E:
F		F:
L|D|A|C		N: (left justified)
N|W		N:
S		S:
P|R		(see next subsection)

These equivalences are reversible but not one-one. The possible (automatic) reverse conversions are given in the following table.

variable type	possible attribute datum forms
B:|I:|E:|F:	B (standard rule)
B:|I: (0,...,255)	X
I:	I|E|F
E:|F: ±(0,..,32767)	I (whole number part only)
E:|F:	E|F
N: (first character)	L|D|A|C (if proper)
N: (form N)	N|W|R
N: (general)	W|S
S:	S

When filling an S-form datum, the source string is either truncated or extended with blanks if the lengths differ. The above rules can be roughly summarized by the statement: node datum values can be assigned from variables wherever it makes sense within established rules.

3.2 Arrays, Node Attachments and Annexes

There are likewise six types of arrays which may be used in working storage. These correspond to the six types of arrays which may be attached to a node with a P-form attribute. However, since single-letter designators are not used for such attached arrays, prefixes for working storage had to be assigned. They are as follows:

V:symbol	vector	(I|E|F form)
M:symbol	matrix	(I|E|F form)
L:symbol	list	(W form)
T:symbol	numeric table	(I|E|F form)
A:symbol	symbolic table	(2, 4 or 8 byte)
H:symbol	character string table	

(The last prefix is H both as a reminder of the special head format and also because such strings are often used for headings of various kinds.)

All arrays in working storage are unpacked and must be explicitly created. They may be created in two ways: either by direct definition which establishes both structure and some or all values, or by use of the command FORM which establishes only structure and leaves null values in the body. (The head and stub of a table are part of its structure.) No matter how a table is created in working storage, it must be done while outside the span of any other command, such as DEFINE or CREATE.

Arrays may be directly defined in either of the following ways:

(1) By use of in-line packets of lines exactly as used for attached arrays under CREATE except that PACK is ignored. For example, the lines

```
MATRIX: symbol(10,5) E
     ROW: elem(1,1), elem(1,2), ..., elem(1,5)
          :
     ROW: elem(10,1),elem(10,2),..., elem(10,5)
:END
```

define a 10 by 5 matrix whose elements can then be refer-

enced with the form M:symbol(i,j). The elements are E form. One or more packets can be executed from a card-image file by use of the command RUN.

(2) By use of the READMOD command which uses similar packets but can also modify existing arrays in various ways. It is explained in Appendix B due to its variety of rules and formats.

The command FORM permits a variety of options including use of existing heads and stubs to form a new table, generation of stylized head and stub symbols, and boolean functions on existing heads and stubs regarded as sets. A simple use would be

FORM M:symbol(I)=m,n

which would form an m by n matrix of integers with zero values. FORM is fully explained in Appendix A due to its many options.

An existing array can be copied to a node attachment under the command CREATE merely by reference as mentioned in an earlier section. The known definition of the array in working storage can be used instead of explicit instructions under CREATE. A difficulty arises, however, since the type of array is unspecified. This is overcome by using a prefixed form to specify a P-form attribute value:

attribute = p:symbol

where p=V|M|L|T|A|H. If a packet for an array named "symbol" follows, and it is of a different type, the packet specification overrides and the above prefix is ignored. Indirect references as explained in the following subsection may be used, e.g.

attribute = M:N:ABC

means the attribute is a matrix whose name is the value of the variable N:ABC.

The use of D:symbol to refer to an attribute of type P imposes an unavoidable burden on the user, namely, he must know it is a P-form attribute and what type of array it refers to. The referent forms used in expressions require this and there is no way to avoid it. Moving the array to working storage does not help since the user still must know what it is. Due to the flexibility in evaluation of expressions and assignment of values in

the TEXTAG language, it is not even possible to guarantee that all errors will be caught. To alleviate this situation somewhat, the QUERY command is provided. (It is also used for other purposes.) For the present purpose, the following form of the command is sufficient:

```
QUERY TYPE N:symbol1 = D:symbol2
```

The value assigned to N:symbol1 will be one of the following:

VECTOR.k	Array is a vector
MATRIX.k	" matrix
LIST...8	" list
TABLE..s	" numeric table
ATABLE.s	" symbolic table
HTABLE.8	" character string table
bbbbbbbb	Attribute form is P but array is undefined
REFER...	Attribute form is R and symbol is not blank
REFERbbb	" " " is blank
********	Attribute form is neither P nor R

where

k=I|E|F

s=2|4|8

b=blank character

Note that initial letters, except for errors, correspond to array types and the eighth character gives the kind or size of elements. These can be extracted with standard symbolic functions. The DIMEN command, further explained in the sequel, can be used to obtain array dimensions if necessary.

When the type of array is known, the D-prefix is used as though it were the proper type of prefix. For example, if the array is a matrix, a referent to an element of the matrix would be of the form

```
D:symbol(i,j)
```

If the array is packed, necessary searches are made. For example, the phrase

```
M:WORK(2,3)=D:ATTACHED(2,3)
```

is valid even though the node-attached matrix is packed and the working storage matrix is not. This continues to be true with

automatic indexing features introduced in the sequel. However, the reverse phrase

D:ATTACHED(2,3)=M:WORK(2,3)

can lead to difficulty. The packed matrix must have an assigned position for element (2,3) or either an error is declared or an extension to the array must be made. (Which is controlled by a mode switch.) Extensions can lead to great inefficiency in file handling. Although the use of packed vectors and matrices can lead to great space saving in some cases, they must be used with foreknowledge of what is to be done with them.

A reference to an R-form attribute presents a possible misconception. First, a phrase of the form

D:refer=N:symbol

where "refer" stands for an R-form attribute name, is perfectly valid to define or redefine the name of an annex for reference. (The value of N:symbol must be form N.) Similarly, the phrase

N:symbol=D:refer

is valid but it merely assigns the annex name to N:symbol. It does not access the annex or even determine if it exists. An annex can be displayed or printed with appropriate commands, for which purpose saving its name with the above command may be useful.

3.3 Indirect References

Any of the prefixes previously introduced may be appended with N:symbol to form an indirect reference. The variable N:symbol must have already been defined, and its value must be form N. This value is then the name of the variable or array whose type is denoted by the first prefix. For example, if the value of N:IND is VAR, then

I:N:IND is the same as I:VAR

T:N:TND is the same as T:VAR

and so one. In the case of a boolean variable, only the first letter is used; thus with N:IND as above

B:N:IND is the same as B:V

Tnis capability is critical for effective use of the language. It permits a referent symbol to be itself a variable. The concept is not extended to further levels; doubly indirect references would have to be programmed. However, it is the first indirection which is the critical capability.

An indirect reference may be used on the left of an assignment statement as well as on the right. For example, if I:VAR has not been defined,

I:N:IND = expression

will define I:VAR under the above assumptions. An array may be indirectly referenced on the left provided it has already been formed.

4.Ø BASIC OPERATORS AND FUNCTIONS

Most of the usual elementary operators and functions and some less common ones are provided in the TEXTAG language. They can be classified as

arithmetic operators,
elementary (numerical) functions,
relational and boolean operators, and
symbolic operator and functions.

4.1 Arithmetic Operators and Form Conversions

The standard four binary arithmetic operators and the unary minus sign are provided in normal fashion. If a and b are single-valued operands and c is a single-valued result, then all

the following assignment statements are recognized:

```
c=a+b ,        c=-a+b
c=a-b ,        c=-a-b
c=a*b ,        c=-a*b
c=a/b ,        c=-a/b
c=a   ,        c=-a
```

The restriction to single-valued quantities is rigid but the effect of multiple-valued quantities is achieved with automatic indexing features presented in the next section. Exponentiation as a binary operator is not provided but the equivalent capability is presented in the next subsection.

Automatic conversion of quantities in mixed-form expressions and in assignment to a result is provided in all circumstances subject to valid ranges and the rules of pairing. The latter was discussed in preliminary fashion in section 2.1. The full set of rules is summarized in the next two tables.

paired forms	calculated as
B,B|B,I|I,I	I
B,E|B,F|I,E|I,F	F
E,E|E,F|F,F	F

calculated value	assignable to
I|F (units position)	B
I|F (mag < 32768)	I (whole no. part)
I|F (I floated)	E|F (truncated for E)
boolean B	B|I|E|F (floated for E|F)

Attempted division by zero is detected and results in an error being declared. Underflow in multiplication or division of F forms gives a zero result. Overflow in multiplication of I or F forms and division of F forms give a system error. Attempted assignment of a magnitude equal to or exceeding 32768 to an I-form is detected and results in an error being declared.

Four "finite limits" are provided for E and F forms and may

be referenced with the following preempted constructions:

+INF.	$16^{3\emptyset}$	(plus "infinity")
-INF.	$-16^{3\emptyset}$	(minus "infinity")
+NIL.	16^{-15}	(plus "nil")
-NIL.	-16^{-15}	(minus "nil")

Values within these magnitudes cannot cause machine overflow in one operation.

Parentheses may be used in standard fashion to control pairings and order of computation. Thus

a+b/c is equivalent to a+(b/c)

(a+b)/c must be used if so intended.

Note that division of an integer by a larger integer gives zero, and by a smaller integer gives the content not the residue. Thus

3/5=Ø , 5/3=1

The residue must be programmed; for example

5-(5/3)*3=2

A pair of vertical bars is also recognized as indicating absolute value and they rank as a pair of parentheses. Thus, the expression

7-(6-|1-|3-5||)

has value 2.

4.2 Numerical Functions

Arguments to all numerical functions are converted to F form before evaluation of the function and the result is in F form.

Exponentiation has traditionally been a source of nuisance in programming. The use of ** to denote it in FORTRAN was due to unavailability of a suitable character. The double asterisk is both visually displeasing and a bothersome special case in scanning. The up-arrow is better but still not universally available. If these objections are thought trivial, two others are not. First, an integer exponent is handled much faster by multiplication but finding out it is an integer is itself expensive; a non-integer exponent requires use of a logarithm. Second, ex-

ponentiation is not truly an operator but a function.

In TEXTAG, exponentiation is treated as a function but two additional functions are provided for the most common integer exponents, squaring and cubing.

The functions provided are the following where "arg" is any numeric-valued expression (including B form) within the ranges shown.

SQRT(arg)		arg≥Ø
SQUARE(arg)		|arg|<16^{15}
CUBE(arg)		|arg|<$16^{1Ø}$
LOG(arg)	log on e	+nil.<arg
EXP(arg)	e to arg	-36.Ø<arg<72.Ø
LOGlØ(arg)	log on lØ	+nil.<arg
EXPlØ(arg)	lØ to arg	-18.Ø<arg<36.Ø
SIN(arg)	arg in radians	(any)
COS(arg)	arg in radians	(any)
ATAN(arg)	result in radians	(any)

The last gives a result in the first or fourth quadrant. Arguments outside specified ranges result in an error being declared.

Obviously, fourth and fifth powers, etc., can be formed by nesting. For example,

SQUARE(SQUARE(F:X))

gives the fourth power of F:X.

4.3 Relational and Boolean Operators

The specification of relational and boolean operators has also traditionally been a nuisance in programming. Almost none of the commonly available keyboards have an adequate set of graphics and internal codes are not fully standardized. Attempting to define queer combinations of standard special characters is unsatisfactory. The use of short abbreviations delimited in some way seems to be the only practical solution. The scheme used in FORTRAN is satisfactory in principle but the use of the period as a delimiter leads to difficult-to-read and all-but-

ambiguous constructions (which would be ambiguous with other TEXTAG conventions) such as

IF(A.LT..5.AND.B.NE.1.Ø)

Two relational operators, < and >, are provided on all keyboards and, since they are not a complete set but make excellent enclosers, they are used in TEXTAG to enclose relational and boolean operators.

The unary boolean operator NOT is not provided separately. It also leads to awkward constructions which create ugly exceptions in scanning and parsing. (The unary minus sign does too but one never writes a+-b.) Instead, the function of NOT is combined with the other operators to refer to the second operand. (Also, both conditionals IF and IFNOT are provided.)

The relational operators apply to either numeric or symbolic operands except S-form strings but both operands must be either numeric or symbolic. The result is a bit, Ø (false) or 1 (true).

a <LT> b	is a less than b?
a <LE> b	is a less than or equal to b?
a <EQ> b	is a equal to b?
a <NE> b	is a not equal to b?
a <GE> b	is a greater than or equal to b?
a <GT> b	is a greater than b?

An additional operator is provdided for strings. The left operand is either a literal string or an N-variable of which only the initial characters up to the last consecutive nonblank are used. The right operand is an S-variable. For example,

N:TEST <IN> S:WHAT

means "does the string of characters in N:TEST occur in S:WHAT?". The <IN> operator is regarded as a relational operator.

The boolean operators, strictly speaking, apply only to B-form operands. However, since numbers have the parity of their whole number part converted automatically to a bit for use as a

boolean operand, effectively any numerical values can be used.

a <AND> b	logical product of a and b
a <ANOT> b	logical product of a and complement of b
a <XOR> b	exclusive OR of a and b
a <XNOT> b	exclusive OR of a and complement of b
a <OR> b	logical sum of a and b
a <ONOT> b	logical sum of a and complement of b

4.4 Order of Execution of Unparenthesized Expressions

The order of execution of operators and functions is as follows from first to last:

unary minus to function argument
function
unary minus outside function
multiplication and division
addition and subtraction
relational operators, indifferently
<AND>, <ANOT>
<XOR>, <XNOT>
<OR>, <ONOT>

Lacking any other distinction, operations proceed from left to right. Note that definition of order for <XOR> separately is important, a point often overlooked. For example,

(1 <XOR> 1) <OR> 1 = 1
1 <XOR> (1 <OR> 1) = Ø

The order given implies that

1 <XOR> 1 <OR> 1 = 1

Similarly,

(Ø <AND> 1) <XOR> 1 = 1
Ø <AND> (1 <XOR> 1) = Ø

The order given implies that

Ø <AND> 1 <XOR> 1 = 1

4.5 Symbolic Operator and Functions

These functions are different in nature from the others and very few languages provide them. They are extremely useful and powerful for some kinds of work.

The arguments to these functions always include at least one character-word and such arguments can be obtained from any suitable source. The possible operand sources are:

literal input strings up to 8 characters
N-variables
elements from lists
elements from A-tables
8-character increments from H-tables
result of a symbolic function
stub or head value from any table except H-table heads

If the source does not provide 8 characters, it is overlaid on the left end of 8 blanks.

Final results may be assigned to an N-variable, list element, A-table element or H-table increment. If the destination is less than 8 characters long, the result is truncated on the right. Intermediate results may be used wherever a character-word is required in an expression. Table stubs and heads are not assignable except by table creation commands. Use of S-variables is explained below.

The functions are as follows where "arg" is an operand as described above.

MASK(arg,mask)

where "mask" is also an 8-character word in which the zero and blank characters are "deleters". For every deleter in "mask", the corresponding character of "arg" is set to blank. Unless nested within the LEAVE function, the result is closed up to the left, blank filled

on the right, called normalized.

FILL(arg1,arg2)

For every blank character in "arg1", the corresponding character in "arg2" replaces it. Result is normalized if not nested within LEAVE.

SHIFT(arg,n)

where n is an integer literal or variable. The "arg" word is left circularly shifted by n character positions. Result is normalized if not nested within LEAVE.

BUMP(arg [,n])

where n is an integer literal or variable, 1 implied if not specified. The "arg" word is processed from right to left as follows. Every character not a digit or letter (A through Z only) is left alone. If a digit not '9', the character is replaced with the next higher digit. If '9', it is replaced with 'Ø' and a carry-left generated. If a letter not 'Z', the character is replaced with the next higher letter. If 'Z', it is replaced with 'A' and a carry-left generated. This continues until no carry-left is generated or until the first (left-most) position has been processed. The whole procedure is done n times. The result is normalized if not nested within LEAVE.

LEAVE(funct(args))

The result of the function is not normalized. LEAVE is effective only for the first nested level. Thus in

```
LEAVE(fnl(fn2(args),...)
```

result of fn2 is normalized but that of fnl is not.

A few examples will clarigy these function. Literals will be used for simplicity. The equals sign here means "gives".

```
MASK(ABCDEFGH,XØX) = 'ACbbbbbb'
LEAVE(MASK(ABC,XØX)) ='AbCbbbbb'
FILL(LEAVE(MASK(ABC,XØX)),'....') = 'A.C.bbbb'
MASK(ABCDEFGH,'ØØØØØØØX') = 'Hbbbbbbb'
SHIFT(ABCDEFGH,7) = 'HABCDEFG'
```

```
MASK(SHIFT(ABCDEFGH,4),X) = 'Ebbbbbbb'
BUMP(A$9Z7,3) = 'B$ØAØbbb'
     skip 3 blanks at end
     7+3=Ø, carry one
     Z+1=A, carry one
     9+1=Ø, carry one
     skip $ but carry still in effect
     A+1=B
```

One symbolic operator is provided: concatenation denoted by the ampersand &, e.g.

```
argl & arg2
```

Tne "argl" string is scanned from right to left for the first nonblank (i.e., on the end), and then "arg2" is appended up to a maximum of 8 characters. Result is normalized if not nested within LEAVE. The following examples should make the action clear.

```
ABC & XYZ = 'ABCXYZbb'
'AbbC' & XYZ = 'ACXYZbbb'
LEAVE('AbbC' & 'XbZ') = 'AbbCXbZb'
ABCD & '12345' = 'ABCD1234'
'bABCD' & '1234' = 'ABCD123b'
```

The last illustrates that truncation occurs before normalization. Normalization is actually a symbolic function itself but is automatic if not inhibited. It can be forced as follows:

```
'bAbBbC' & 'b' = 'ABCbbbbb'
```

S-variables cannot be processsed as units by basic operators and functions. However, two devices for handling them are provided. The referent form for an S-variable uses two indices which are integers:

```
S:symbol(i,k)
```

where i indicates the first character to be used and k the number. Thus if S:LETTER has length 26 and the value of all the letters, 'AB...Z',

```
S:LETTER(1Ø,4)  produces 'JKLM'
```

Hence any part of a string up to 8 characters can be a symbolic operand or result. Shortest length appearing, explicit or impli-

cit, controls. Thus

```
N:ALPHA = S:LETTER(24,3)
```

assigns 'XYZbbbbb' to N:ALPHA.

```
N:ALPHA = S:LETTER(1,10)
```

assigns 'ABCDEFGH' to N:ALPHA; the overflow is ignored. If the value of S:HEX is all blanks with a length of 16, and the length of S:OCT is 8 with value '01234567', then

```
S:HEX(1,10) = S:OCT(1,10)
```

is an error since S:OCT is only of length 8, but

```
S:HEX(1,10) = S:OCT(1,8)
```

is valid; the ninth and tenth characters of S:HEX remain blank. If N:OCT has the same value as S:OCT, then

```
S:HEX(1,10) = N:OCT
```

has the same effect as the previous statement. The statements

```
S:HEX(9,2) = '89'
S:HEX(11,6) = S:LETTER(1,6)
```

would finally result in a value of

```
'0123456789ABCDEF'
```

for S:HEX. Extraction from a string is not normalized before use and, if no symbolic function or operator is used, final result remains unnormalized.

Note that substrings up to 8 characters long from an S-variable can be compared.

Strings are treated as units by the command ASSIGN which takes what appears to be a normal assignment statement as its argument fields. The operand(s) and result are full strings, however. They may be either S-variables or H-table strings. The referents for the latter require a nonstandard column index. The concatenation operator (only) is recognized but may be repeated. The forms of the command are as follows:

```
ASSIGN S:symbol = arg1 [& arg2 ...]
ASSIGN D|H:symbol(s,h) = arg1 [& arg2 . .]
```

where

arg_i is a literal string up to 50 characters (all on 1 line)
an S-variable
an H-table string (see indexing below)

s is any standard table stub designator, i.e., a row index, stub symbol, or automatic index (all explained in the next section).

h is A|B|C|D|E|F|G|H referring to the corresponding string in the row of the table.

An H-table may be in working storage or attached to current node (STOP pointer). In the latter case, the D-prefix is used as explained earlier.

The "column" designator h is nonstandard because reference is usually to a word (increment) requiring forms such as A3, B1, etc. The letter used alone for the whole string is valid only with ASSIGN.

Concatenation is done blindly; no normalization takes place. When the destination field is full, remaining source characters or terms are ignored. If the destination field is not filled, remaining positions are filled with blanks regardless of their prior values.

Macros (explained in Appendix C) are easily written for more complicated symbolic functions. Such macros become essentially new commands. Since they take substitutable arguments, their use can be generalized to a considerable degree. A preliminary example of a macro for numeric functions appears in the next subsection.

4.6 Other Functions

Functions for special purposes are easily created by packets of command lines. If these are put into the form of macros, they become generally available. A few such macros are provided as standard adjuncts to the system but the user may modify them if he wishes. The general concept will be illustrated with three

elementary statistical functions. (Details of syntax and creation of macros are given in Appendix C.)

Suppose we define three F-variables

F:SUM, F:MEAN, F:STD.DEV

and point to a vector V:X The vector may be I|E|F form and of unstated length. We will also assume a vector in F-format of length 1 or more, called say V:WORK, has been formed and is available for use. (It is necessary to formally use an array and not a simple variable, a trick explained in the next section. The trick can be avoided but would require more commands.) We anticipate the next section by introducing an automatic index denoted by !1. This means to run over all indices of a vector. We also need to be able to use an I-variable, say I:WORK. The sum, mean and standard deviation are then computed as follows.

```
V:WORK(1)=Ø.Ø
V:WORK(1)=V:WORK(1)+V:X(!1)
DIMEN I:WORK=V:X
F:SUM=V:WORK(1)
F:MEAN=F:SUM/I:WORK
V:WORK(1)=Ø
V:WORK(1)=V:WORK(1)+SQUARE(V:X(!1)-F:MEAN)
F:STD.DEV=SQRT(V:WORK(1)/(I:WORK-1.Ø))
```

The variables F:SUM, F:MEAN, F:STD.DEV and the vector V:X (also V:WORK and I:WORK if desired) can be replaced with substitutable arguments in a macro. If the macro is called STATS, the above computation could be done with one statement such as

```
STATS F:SUM, F:MEAN, F:STD.DEV, V:X
```

More clever arrangements can be easily devised, especially by using a conventionalized vector or T-table instead of the F-variables. The last statement could then be reduced to something like

```
STATS OF V:X
```

5.Ø REFERENCES TO ELEMENTS OF ARRAYS

Much of the power of the TEXTAG language derives from the flexibility of referent forms for arrays and node structures. The latter was explained in section 3.1 but a few additional points will be brought out here.

In effect, the last node found -- the one pointed to by STOP -- constitutes a temporary set of variables which may be referenced via the D-prefix. This is made possible by the fact that the only way to reach a node is via the D(Q) of its superior. This is always pointed to by START and contains the definition of inferior nodes -- essentially a "symbol table". (The root node is an exception but its structure is known to the system.) Actually, D(Q) of the superior is retained in working storage until START is changed.

Arrays attached to nodes (but not annexes) may be referenced just as arrays in working storage except that the user must know which D-reference points to an array attachment and what type it is. The QUERY command can be used to check on this. A form of the DISPLAY command will also display the D(START), i.e., the names and forms of the attributes of the inferior set, but of course this does not help in automatic execution.

The indices into tables may be of four types: numeric, automatic indexing, symbolic, and head and stub matching. The first two apply to all forms of arrays -- vectors, matrices, lists and tables -- and will be presented first.

5.1 Numeric and Automatic Indexing (Except Tables)

A numeric index into, say, a vector means essentially what it has traditionally meant. If V:X denotes a vector X, the first element is referenced by V:X(1). If the length is m, the last element is referenced by V:X(m). In TEXTAG, a vector is by definition numeric, although three number forms may be used for different vectors. The user need not pay any particular attention to the number form, once it is defined, except for the gen-

eral limitations on integers if I-form is used and the lack of precisiion if E-form is used.

The index need not be an integer, no matter what form the elements. A general numeric-valued expression is allowed so long as the value is not less than one and less than m+1 where m is vector length Thus,

V:X(I:IND), V:X(E:END), V:X(F:IND)

are all valid provided $1 \leq I|E|F:IND < m+1$. Expressions and nesting of referents are allowed, e.g.,

V:X(I:IND+1), V:X(V:Y(I:IND))

provided all resulting ranges are valid. An index out of range results in an error being declared.

A matrix is a two-dimensional array of the same kinds as for vectors and customary indexing is used with the same freedom as above. Thus

M:X(2,3)

is the element of matrix X in the second row and third column. Such forms as

M:X(I:ROW,I:COL), M:X(V:R(2), I:COL)

and so one, are allowed. A matrix element may be used, e.g.

M:X(M:INDEX(I:ROW,I:COL),2)

and, in general, any numeric valued expression.

The symbolic form of a vector is the list but each element is always 8 bytes (characters) long. A list is referenced just like a vector, e.g.,

L:NAMES(1), L:NAMES(I:PTR), L:NAMES(I:IND+2)

subject to the same index limits as for a vector of the same length.

There is no symbolic form of matrix; a table must be used instead. As will appear, a symbolic matrix would be of minimal utility. Lists, on the other hand are very useful and much less of a nuisance than a table with only one row or column. There is a fundamental distinction between a vector and a list; a value obtained from a vector is treated as a number, a value from a list as a character-word. This is the same distinction as between an I-variable, say, and an N-variable. In several uses

the difference must be known by the system. Prefixes also permit the same symbol to be used for different forms. I:X, F:X, L:X, M:X, for example, may all exist simultaneously without confusion.

In using arrays, one frequently wants to loop over an entire index range. TEXTAG does not provide a DO-loop in the usual sense although a LOOP command exists. However, looping through an array is usually done with automatic indexing. This is not quite as flexible as a DO-loop but much briefer and more convenient in most cases. When automatic indexing is inadequate, a loop is easily programmed with other instructions.

An automatic index is indicated by a flagged integer of the form !n where n=1,...,9. There is no difference in meaning among the digits except, if more than one occurs in the same expression, the smallest digit indicates the outermost loop. Generally speaking, a row should be outer to a column (for efficiency) but the difference is usually minor.

An automatic index runs over all values from 1 to m where m is the shortest vector length to which it refers. Thus

```
M:X(!1,!2)=Ø.Ø
```

clears the entire matrix to zero. The statement

```
M:X(!1,!1)=1.Ø
```

puts ones down the main diagonal. The matrix need not be square; the shorter dimension governs. The statement

```
M:X(!1,!2)=M:Y(!1,!2)
```

puts as much of matrix Y as possible in the upper left hand corner of matrix X (possibly filling it or possibly using all of matrix Y). Rows and columns of the result matrix not referenced because of shorter limits on the right are not changed. Transposition lcan be used; thus

```
M:X(!1,!2)=M:Y(!2,!1)
```

transposes some or all of Y into some or all of X. In effect, one almost has matrix operations without DO-loops or subroutines. There is a difference, however. If one is alccumulating values, the addition must be explicitly stated. For example, for matrix

multiplication,

```
M:PROD(!1,!2)=M:LEFT(!1,!3)*M:RIGHT(!3,!2)
```

is incorrect. The proper statements would be

```
M:PROD(!1,!2)=Ø.Ø
M:PROD(!1,!2)=M:PROD(!1,!2)+M:LEFT(!1,!3)*M:RIGHT(!3,!2)
```

Note that these two statements amount to generalized multiplication of (possibly) nonconformal matrices.

An automatic index can be used on the right of a phrase, e.g.,

```
V:X(!1)=!1
```

fills vector X with its row indices, in appropriate form. Such expressions are meaningful only when the automatic index is active. The statement

```
I:X=!1
```

is meaningless and results in I:X being assigned the value Ø (unless the statement is in the range of a LOOP command, in which case it is meaningul). An automatic index may never appear alone on the left, that is, it may not be explicitly assigned a value.

A limited form of arithmetic is permitted with an automatic index (used as an index). This may take only one of four explicit forms:

```
!n+k
!n-k
!n+I:symbol
!n-I:symbol
```

where k is a literal integer. Thus

```
M:X(!1,!1+1)=1.Ø
```

puts ones down the superdiagonal and

```
M:X(!1,!1-1)=1.Ø
```

puts ones down the subdiagonal. In the first case, the column index exceeds its valid range and, in the second, it has value Ø the first time. These exceptions are automatically blocked and no error results.

The above limitations do not apply to an automatic index used as a variable (when this is meaningful). It may then be used like an I-variable. Further examples are given in the sub-

section on LOOP.

5.2 Indexing of Tables; Automatic Symbol Matching

Any of the three forms of tables can be indexed numerically and automatically just like matrices. A T-table is essentially a matrix with symbolic row (stub) and column (head) names. An A-table contains character words (2, 4 or 8 bytes in length) but can be treated like a matrix with respect to indexing. A T-table element can be used to index another table, or a vector, matrix or list, for that matter. Thus

V:X(T:IND(!1,1))

will use all elements of column 1 of T-table IND in succession as indices into vector X. Of course, they must all be in the range of the vector length. By having different permutations of the indices in various columns of T:IND (presumably form I), one can select elements of V:X in different orders by changing the column index of T:IND.

There would be little point in having tables, however, if only numeric indexing were used. (An array of symbolic values could still be useful.) The main point of tables is their stubs and heads which, among other things, permit one to index symbolically. Thus if the third row of T:PROD has the stub COAL and the fifth column the head YR1967, either of the referents

T:PROD(COAL,YR1967), T:PROD(3,5)

give the same element, but the first is more readable and one need not know the numeric indices. Any valid symbolic expression can be used for a symbolic index. Thus if N:TYPE has the value COAL and N:YR has the value '67',

T:PROD(N:TYPE,YR19&N:YR)

gives the same element as the prior two referents.

Undefined symbolic indices give a different result than numerical indices which are limited by a shorter vector length in another reference or are out of range. If T:X has a stub value A and a head value B but T:Y lacks one or the other or both, then

T:X(A,B)=T:Y(A,B)

results in zero being assigned to T:X(A,B); no error is declared. This point should be carefully noted with automatic symbol matching or nested table referents. The presumption is that a nonexistent element should be taken as zero for a T-table and blanks for an A- or H-table. This is consistent with the purpose of symbol matching but can be a trap.

Tables can also be degenerate. A table with only a stub but no head is call a stub table; if it has a head but no stub, it is called a head table. In either case it has no body and is called null. If it has neither a stub nor head, the table is called void. The use of lists obviates the need for null or void tables in most cases but occasionally one is used to give a zero or blank result for any possible symbolic reference, as in a general program where other tables to be processed are meaningful. Also a 2-byte or 4-byte head table takes less storage than a list. An H-table, however, must have a head. An attempt to define or form an H-table without a head is treated as an error.

The stub of a table is referenced with an explicit zero for the head index (i.e., column Ø) and the head of a table is referenced with an explicit zero for the stub index (i.e., row Ø). Such referents may never be used in a left member -- stubs and heads may not be assigned new values -- but may be used in an expression on the right. (The use of (Ø,Ø) is illegal.) The value obtained is a character-value for any type of table referenced.

Tne following example of a nested referent using automatic indexing shows one possible use of such symbols for selection.

Example:

T:PART(A:SELECT(Ø,!1),!2) = T:WHOLE(A:SELECT(Ø,!1),!2)

Explanation: The head of A:SELECT should be a subset of the stub of T:PART and have a non-empty intersection with the stub of T:WHOLE. For each head symbol of A:SELECT, the row of T:PART with that stub symbol is either:

(a) set to zero if the symbol is not in the stub of T:WHOLE

or

(b) set to the values of the row of T:WHOLE with the same symbol.

If T:PART and T:WHOLE do not have the same number of columns, the shorter head limits the range of !2 in both cases. The order of the head of A:SELECT and the stubs of T:PART and T:WHOLE need not be the same. The heads of T:PART and T:WHOLE are irrelevant except for length, i.e., a row or partial row is copied in its physical order. Operations similar to this example are sometimes better done with direct symbol matching.

Automatic symbol matching is specified with a flagged integer in the form "n where n has the same meaning as for automatic indexing. If both !n and "n appear in the same statement with the same value of n, indexing is stepped before symbol matching. However, nine values of n have never been known to be used at one time so that it is better practice to use different values for each purpose.

Symbol matching has quite a different effect from automatic indexing and the two should not be confused, although the previous example has the effect of symbol matching. If that example were changed to the following

T:PART("1,!2) = T:WHOLE("1,!2)

then all rows in T:PART whose stub symbol occurs in the stub of T:WHOLE would be set to the values of that row of T:WHOLE. Any other rows would be set to zero with the same limit on !2 as in the prior example. The hierarchy of control is left to right, that is, unmatched rows are not left alone but cleared.

Essentially symbol matching for T-tables permits generalized matrix algebra for nonconformable tables. For A- and H-tables, analogous symbolic manipulations are possible but are difficult to describe. The reader should test the effects for himself after symbolic operations have been carefully studied.

The following example does generalized matrix multiplication. It is assumed that the tables have appropriate non-empty intersections of stubs and heads.

```
T:PROD(!1,!2) = Ø.Ø
T:PROD("1,"2) = T:PROD("1,"2)+T:LEFT("1,"3)*T:RIGHT("3,"2)
```

The product table, T:PROD, is first cleared to zero. Then for each stub symbol in T:PROD which occurs in T:LEFT and each head symbol in T:PROD which occurs in T:RIGHT, the inner product of the corresponding row of T:LEFT and column of T:RIGHT is put in the body element of T:PROD (summed product by product). Each inner product is itself subject to symbol matching between the head of T:LEFT and the stub of T:RIGHT. Order of occurrence of symbols in all stubs and heads is immaterial.

Symbol matching nearly always requires at least two tables whereas automatic indexing is often useful for one. (Symbol matching could conceivably be used on one table whose stub and head contain some common symbols but it is hard to imagine for what purpose.)

Use of the LOOP verb is often required for full control of both automatic indexing and symbol matching.

5.2.1 Use of Dummy Arrays for Summing

In section 4.6, the vector V:WORK was used for summing V:X with an automatic index. The reason is as follows. A variable is evaluated only once for an entire assignment statement since it can only change if it appears as the left member after all computation is complete. An array, on the other hand, may be subject to looping due to an automatic index or name matching. Hence any referenced element of an array is evaluated for each

pass of the implied loop and, if the array appears on the left, it is assigned its new value at the end of each pass. The statement

```
F:WORK=F:WORK+V:X(!1)
```

results in F:WORK being assigned the sum of its original value and the last element of V:X which not what is wanted. By using a dummy numeric array, e.g.,

```
V:WORK(1)=V:WORK(1)+V:X(!1)
```

the intended summation is performed since V:WORK(1) is evaluated on the right and assigned on the left for each pass over !1 even though the index for V:WORK is constant. No test is made to see if an index actually changes; an element of an array is evaluated and assigned if necessary for each pass of any implied loop.

5.3 The LOOP and CONTINUE Commands

Several commands are meaningful only in the context of a program entity since they involve transfers or spans of control. These are called execution control commands. Any other commands may be executed from immediate type-ins but, in fact, one usually executes programs or parts of programs automatically from a pre-typed card-image file. Execution from such a file is initiated by either the command RUN or a SOURCE phrase in other statements. The location of program files is explained in the next section.

Two execution control commands are LOOP and CONTINUE which form "begin-end" pairs which may be nested. LOOP is not the same as DO in a DO-loop. A DO statement in FORTRAN has the implication "for" referring to an index range. A LOOP statement has the implication "if" referring to a boolean expression which must involve either automatic indexing or name matching in an array in working storage. For example,

```
LOOP  M:X(!1,1) <GT> Ø
```

means to execute all statements down to the matching CONTINUE for all values of !1 for which the corresponding element of matrix X is positive. When the range of !1 is exhausted, i.e. would next

exceed the row order of X, control passes to the statement following the matching CONTINUE.

It is illegal to use a statement within the span of control of a LOOP statement (i.e. before the matching CONTINUE) which transfers control out of the loop. Doing so will give unpredictable results. TEXTAG programs are not compiled but are executed interpretively no matter what the source of the statements. Consequently, it is not possible to detect the kinds of loop errors which a compiler would catch.

It is perfectly legal to transfer control around an entire LOOP-CONTINUE sequence, so long as the LOOP statement has not been executed. This is often done with nested loops where an inner loop is bypassed within an outer loop. Each LOOP must have a matching CONTINUE, even if nested loops logically end at the same point. An example appears below.

If no meaningful condition can be stated for LOOP, a dummy one can be used. A very common one is of the form

```
LOOP  T:name(Ø,!1) <NE> ' '
```

which means to loop over all head symbols of a table which are not blank (presumably none are). A name-matching flag is much less common but a statement like the following is possible:

```
LOOP  T:A(Ø,"1) <EQ> T:B("1,Ø)
```

which means to execut the loop, in order on head symbols of table A, whenever the symbol also occurs anywhere in the stub of table B. The order of B's stub need not match A's head in any sense.

Within the span of control of a LOOP, the automatic indexing or name matching indices are fixed. If referenced by statements in the loop, they do not automatically loop within the statements. An inner LOOP statement must have at least one flagged index not occurring in any outer LOOP statement. Consider the

following sequence:

```
LOOP T:CONTROL(!1,!2) <NE> Ø.Ø
     F:DIV=T:CONTROL(!1,!2)
     N:ID=T:CONTROL(!1,Ø) & T:CONTROL(Ø,!2)
     LOOP T:RESULT(!3,Ø) <NE> N:ID
          T:RESULT(!3,!4)=T:RESULT(!3,!4)/F:DIV
     CONTINUE
CONTINUE
```

The outer loop is executed over all columns within all rows of table CONTROL for which the element is nonzero. When it is nonzero, it is saved as F:DIV (not strictly necessary but faster in execution). A symbol is also formed as N:ID by concatenating the current stub and head symbols (presumably short) of table CONTROL. Both these statements would be nonsense outside of a loop since there is no index in the left members. However, !1 and !2 are fixed within the outer loop.

The inner loop is then executed for all rows of table RESULT whose stub symbol is not equal to N:ID. The !3 index is fixed in this loop. For all other rows, all elements are divided by F:DIV, by running over the free !4.

Note that each loop requires a CONTINUE even though they are logically at the same point. The indenting is done to make the program more readable, a practice to be recommended.

If the required operation had been to divide only the row of table RESULT whose stub symbol has the value of N:ID (instead of all other rows), the inner loop would be unnecessary and inefficient. It could be replaced by the single statement

```
T:RESULT(N:ID,!4)=T:RESULT(N:ID,!4)/F:DIV
```

In this case, !3 could be used here instead of !4 but it is a matter of indifference.

6.0 FILES: CARD-IMAGE, MACROS, INTERNAL OTHER

In addition to the data bank proper and the auxiliary file for annexes, several other files may be used during a TEXTAG session. At any moment during normal use of TEXTAG, a maximum of seven files may be in use simultaneously. (See Appendix F for maintenance mode.) These seven files all have functional names which are convenient for reference and are listed below. These functions are really system variables whose values are the actual file names. (File names depend on conventions of the host system and will not be elaborated here. See Appendix D.)

As usual in computing, there are two kinds of files: card-image (like formatted in FORTRAN but generally free form) and binary (or unformatted) files. The first designation is awkward and the second imprecise. They will be referred to as external and internal files, respectively. External files are prepared by the user with an Editor (or conceivably a card-punch). Internal files are controlled by the user but only via system commands. He cannot edit or otherwise change them directly (except for changing the name with host system commands).

The seven standard file functions are:

(1) DATABANK, the main data bank file. The user specifies the name as an argument to the last command issued to the host interfacing routine used to invoke TEXTAG. This is an internal file. It is fixed for the session.

(2) ANNEX, an internal file containing text annexes. The name is specified with the SET command and may be changed during execution.

(3) ARRAYS, an internal file in which arrays from working storage may be enfiled and recalled. The name is specified with the SET command and may be changed during execution.

(4) MACLIB, an external file containing macros in TEXTAG language. It is specified with the USE command which must be the first command executed (except LOAD if used). It is

fixed for the session and may not be changed.

(5) READMOD, an external file containing decks which are processed by the READMOD command. The file and deck names are arguments to the command and may differ on different statements.

(6) PUNCH, an external file produced by the PUNCH command. The file name is specified with the SET command and may be changed. The file may or may not be available for reading during the session depending on host system conventions.

(7) PROGRAM, an external file of decks of statements which may be initiated by parameters to TEXTAG, a RUN command or a SOURCE phrase. Any number may be used but only one at a time. Default is the user's terminal keyboard.

Three other kinds of files are also involved with use of TEXTAG. The printed output file is the standard spool file of the host system. It is made available by a command in the host-interfacing routine used to invoke TEXTAG. In general, all internal files must be declared to this routine. This is over and above use of the SET command or other specification techniques noted above, all of which depend on the file being accessible.

There are also internal scratch files used by certain commands. The user need not be concerned with declaring these; it is done automatically. However, the user takes advantage of them with the commands DUMP, RETRIEVE and IDENTIFY. Within TEXTAG proper, their names are predefined.

The final form of file is a subroutine library. Subroutines may be called with TEXTAG statements provided they were loaded initially with a LOAD command which must be the first command executed. The LOAD command is explained in Appendix E. This function is very important for extending the scope of TEXTAG capabilities. This approach was used very successfully -- though not with complete generality -- in the prototype system. The main difficulty was the sheer size of the system library invoked by FORTRAN (and PL/I) routines. Using assembler language routines, no difficulty at all was encountered. They have been used for

special functions, random number generation, graphic output, and similar purposes. The space problem can be overcome by using sufficient main virtual storage, readily obtained under CP/CMS (but at extra charge).

Another style of external files is used in connection with TEXTAG but not by TEXTAG itself. A standard interfacing routine is provided which is FORTRAN-callable and can access the ARRAYS file. The input to the FORTRAN program is presumably in external form, though not necessarily. A program could generate data directly to load into ARRAYS. However the data is obtained, the ARRAYS file must already exist with appropriate (empty) arrays. This is easily created using TEXTAG interactively in a preliminary session. A FORTRAN or other program can also unload arrays from ARRAYS to use for some other purpose.

6.1 The Concept of Decks in External Files

A useful convention has long been used in certain application areas. This is the concept of decks in an external file. A deck is simply a sequence of lines in a card-image file which begin and end with lines of pre-defined and unique format. Several schemes are in use for these demarcation lines, all of which are arbitrary. The one used in TEXTAG is the same as that used for many years for input files to mathematical programming systems. For simplicity and standardization, the same forms have been used for a variety of purposes and the scheme is about as good as any, better than some.

The reason for using decks is to avoid the proliferation of files and to keep related material together. Frequently, one creates a file with decks sequenced in expected order of use. However, the file will be searched if necessary (once circularly in a forward direction) for a requested deck.

A deck is identified by its first line (card) which has a very rigid format:

```
NAME      deckname
```

where NAME starts in position 1 and "deckname" in position 15. (The reason for the latter position is lost in unrecorded history but there is no compelling reason to change it.) The deck is terminated with an equally rigid format:

ENDATA

starting in position 1. (The word was invented when computers only handled six characters at a time.) Each deck must have one and only one (each) NAME and ENDATA lines.

An asterisk in position 1 of any line indicates a comment. These are usually ignored and are intended for use in a listing of the file. TEXTAG has an option for displaying such lines within a program deck which is sometimes useful in debugging. Comment lines may also appear between decks but they are always ignored.

The :END lines used by TEXTAG are within a deck (unless typed at the terminal) and are a sub-demarcation. Commands for terminating execution of a program deck deck are given in section 9. However, if control ever reaches the ENDATA card, execution is terminated anyway. The action depends on circumstances and hence ENDATA is explained as a command in section 9. It has the force of "RETURN" or "QUIT".

A TEXTAG macro library (MACLIB) is simply a file of decks, each macro constituting one deck. Flags for substitutable arguments are recognized but otherwise the statements are like other TEXTAG statements. The file as a whole is treated specially, however. To eliminate searching, the file is scanned when the command USE is executed and a directory is created in main storage. Hence invoking a macro never requires a search. Either a MACLIB file or a program file can be edited and is then immediately available and suitable for use in a TEXTAG session.

Files used by READMOD have the same deck structure. These decks are not executed but rather processed by READMOD. The distinction is minor in a practical sense. READMOD may be regarded as simply another processor level.

7.0 SEARCH CONSTRUCTIONS

We return now to the subject of searches in the data bank, essentially continuing from section 2.5 but with various needed facilities explained in the intervening sections now available.

In section 2.3 the commands FIND, FINDSUP and SEARCH were introduced. The first two are members of a set of six commands, called FIND commands, which will be explained in the first subsection below. They are all simple in concept and are actually carried out by the file system management routine itself, appropriately interfaced. The SEARCH command, on the other hand, is much more complex in both concept and execution. The TEXTAG processor utilizes the file system with appropriate FIND commands and others not directly available to the user to carry out the search. Search constructions and their actions are explained in the subsection following the FIND commands.

7.1 FIND Commands

Of the six commands in this set, only FIND itself takes arguments, the meaning of the others being unambiguous. All rely on the inferior-set structure and use of key attributes. Most of the options for FIND were presented in section 2.3. A FIND statement has the following general format where action starts from the node pointed to by START.

FIND [-n|katr_]identifier_chain

where n is an integer indicating the number of levels to back up

katr is a symbolic expression indicating the name of the key attribute to which to back up

identifier_chain is a chain of key attribute values, which may be symbolic expressions, connected by underlines if more than one, indicating the path of referents to be taken.

If the only argument is a single referent, FIND looks for it among the value of the key attribute for the inferior set of the

START node. Referring to the example in section 2.3, starting from START pointing to AUSTRIA and N:COUNTRY=GERMANY, the following statement could be used

FIND -WRLD.RGN_E.EUROPE_N:COUNTRY

to get to the node for East Germany. Assuming success, START would then point to E.EUROPE and STOP to GERMANY. Note that one need not know how many levels up WRLD.RGN is.

The FINDSUP command was fully explained in section 2.3. It is rarely needed by the user. If for some reason STOP is set to an old value (e.g., SET STOP=HOLD(i)) and its superior is unknown, then FINDSUP must be used. Resetting START automatically clears the ISD for STOP. Resetting STOP causes the values to be available. Arbitrarily resetting STOP without executing FINDSUP may leave the node incorrectly defined even though available and neither D:symbol referents nor other commands will execute properly.

The other four FIND commands are for accessing nodes of an inferior set in ascending or descending sort order on their referents. The first two start from the START node which must have a D(Q).

FINDLO

This finds the referent in lowest sort and sets STOP to point to it.

FINDHI

This finds the referent in highest sort and sets STOP to point to it. The other two commands start from the STOP node (but START must not have been changed).

FDHIER

("feed higher") gets the node with the next higher referent in sort order. If none exists, B:$ is set to Ø. The last command is just the opposite

FDLOER

("feed lower") which gets the node with the next lower referent in sort order. Both commands set B:$ to 1 and STOP to the node fed is successful. If not, STOP points to the node with highest or lowest referent, respectively.

The purpose of making the last four commands available to tne user is primarily so he can write his own search macros. The SEARCH command makes extensive use of them.

7.2 The SEARCH Command

Several commands (e.g. HOLD, SET, DISPLAY introduced in section 2.0) use different rules from those explained in preceding sections for general assignment statements and expressions. This is particularly true for SEARCH. The differences lie primarily in the recognition of reserved or key words in different contexts where they make sense. In a general expression, an unprefixed symbol is taken as a literal whereas, in many commands, the meaning is implied by usage. This is not really ambiguous or confusing when once understood. Indeed, it would be a nuisance to have to specially flag key words in a natural context.

Some reserved words are really useless but are allowed for readability and "correctness". Thus a statement such as "DO X for n=1,10" is no more meaningful than if "for" were omitted, as in FORTRAN. Other cases are not so clearcut. Thus the statement

```
SEARCH DOWN 2 LEVELS FOR LANGUAGE <EQ> GERMAN
```

is not so clear if "LEVELS FOR" are omitted, even though the parsing routine just throws them away. We like a little verbosity, but not too much. The phrase

```
TO 20
```

is undoubtedly preferred by most people to

```
UNTIL NUMERIC KEY EQUALS 20
```

even though the first is not understandable until the definition of the form has been found. Somehow, a little key doesn't deserve a big long phrase. (As seen below, the form TO K=20 could lead to ambiguity.)

The use of unflagged symbols does pose some problems. In the first example above, LEVELS, LANGUAGE and GERMAN are all nouns, but are used in completely different senses. The first is a reserved word for readability, the second is a user-assigned

name of an attribute, and the third is a value to be looked for. The operator <EQ> looks a bit out of place but we also need to permit such constructions as

POPUL <GE> 1ØE6

and who wants to type out

POPUL GREATER THAN OR EQUAL TO TEN MILLION

Now LANGUAGE and GERMAN can be substituted with N-variables or symbolic expressions, which might be very long if written out. LEVELS and <EQ>, on the other hand, are not substitutable but must be written literally as shown. The statement formats for SEARCH are designed so that, if used correctly, no ambiguity results and most errors will be detected. It is conceivable, however, that some unanticipated error could slip through and give incorrect results. This is the price for convenience and some readability without extensive syntactical and contextual analysis (which might make the wrong conclusion anyway).

The general formats for SEARCH are

SEARCH locator|depth-limit [FOR] boolean.exp [,
REPORT] [,HALT [ON HIT]]

or

SEARCH RESUME

The arguments are explained separately.

(1) locator

A locator is a chain of key attribute names connected with underlines. Action always starts from the START node which is not itself counted. This node must have a D(Q) defining the key attribute name as the same as the first name in the locator. The inferior set is then searched in ascending sort order for a node with a D(Q) defining a key attribute with name equal to the second name in the locator. This position is remembered and the process continued from this node until the end of the locator is reached. The last inferior set is then searched in ascending order on key attribute values for the required condition. If found it is output as directed and unless HALT is specified, searching continues. When the inferior set is exhausted, the procedure

backs up one level and continues the search there for the last name in the locator. This process continues until all nodes at all levels which "stay on a path" have been searched.

(2) depth-list

A depth limit is a phrase of one of the following four kinds:

DOWN TO name

DOWN d [LEVELS]

AT k

TO k

The first form is almost like a locator except all key attribute names are used to form paths until either "name" is reached (success) or a path stops (failure). This search is effectively a double one: first to find all possible locators which terminate with "name" and then all nodes satisfying the condition stated. It can, consequently, be very long if peer groups have many inferior peer groups.

The second form simply proceeds along paths for all possible locators having length d+1 using the known key attributes.

The AT form proceeds along paths for all possible locators which terminate with the numeric key equal k. Encountering key>k before key=k terminates a path search. An integer variable may be used for k.

The TO form is the most general of all. It proceeds in the same way as the AT form but looks for the condition at every level on the way. This involves a search at the level only if attribute names matching those in the condition occur in the ISD. This is also true for the end level of the three other forms of search.

Note that the reserved words DOWN, AT and TO are meaningful; they are preempted from use as key attribute names. If K=k were allowed instead of just k for TO or AT and either of these were misspelled, ambiguity could result. For example,

SEARCH TOO K=15 - - -

where TO is misspelled looks like a one-word locator followed by some kind of condition (which is incorrect but is interpretable and could waste a lot of time). Since k is an integer, not a symbol, it is immediately recognized. This is true even if k is represented by an integer variable which is allowed (but not a general expression).

(3) boolean.exp

This is a boolean expression stating the condition looked for. The expression is not completely general. This is not a strong restriction in capability but mainly a practical one to avoid unnecessary complexity and length in the statement and its execution. The most general form allowed is

rel.exp [<bool> rel.exp ...]

where "rel.exp" is a simple relational expression of the form

attribute name <rel> comparator

and "attribute name" and "comparator" are either literals or simple variables, N-form for the first. (Indirect addressing may be used.) These are not ultimate restrictions since attribute names and comparator values can be precomputed if necessary and, by using HALT, any calculations on the node values can be done. The one capablity that is blocked is masking of a symbolic value. However, initial characters can be found by using an S-form for the comparator. For example

attribute <EQ> S:COMP(1,3)

limits the comparison to three characters.

Note that such relational expressions are really conventions, not true phrases, even though they appear natural. For example, if the attribute name is LOCATION and the comparator is VIENNA, the phrase

LOCATION <EQ> VIENNA

does not mean that the left and right members should be equal, but rather that the attribute whose name is LOCATION should have a value which is VIENNA. This important dis-

tinction is often overlooked by such people as model builders who glibly mix the names of some things with the values of others. In FORTRAN, it is handled by requiring that both sides be variables or, in some dialects, that a literal comparator be in quotes. It could be done in the TEXTAG language with the standard phrase

D:LOCATION <EQ> VIENNA

which is properly what is intended but it would be a bit of a nuisance and not natural-looking. Within the condition expression of SEARCH, left members are simply understood to be interpreted as if prefixed with D:. However, they may be prefixed normally if a variable is to be used; thus, if N:ATTR has the value LOCATION, the phrase

N:ATTR <EQ> VIENNA

gives the same effect as above, i.e. as though the left member were D:N:ATTR.

A further restriction is imposed on boolean expressions, namely, parentheses are not recognized. Instead, execution is left to right. Thus a<AND>b<OR>c is interpreted as (a<AND>b)<OR>c. If a<AND>(b<OR>c) is intended, it must be written b<OR>c<AND>a. The reason for this is that the general expression evaluator is not available during the search and making it so would lead to great inefficiency.

(4) [,REPORT]

This option controls disposition of results. The default is to display them at the terminal. If REPORT is specified, results are spooled to the output print file. (Printed output can be duplicated at the terminal or sent only there, and vice-versa, by using the command MSGCLASS for controlling output dispositions.) Format is automatic using predefined conventions. The result is entirely readable and this approach save a lot of fuss about defining formats and then

invoking them. Indenting is used to indicate levels

(5) [,HALT [ON HIT]]

This option (with its trailing phrase if desired) causes the search to be interrupted each time the condition is satisfied. The option must be used if anything other than simple displayed or printed output is required. For example, it may be desired to display the entire node, or to examine attached arrays or to extract data to put in other arrays in working storage. All facilities of the language which do not involve changing the STOP and START pointers may be used.

(6) RESUME

This form of the command simply continues the search after a prior SEARCH with HALT has halted. If no such prior SEARCH has been executed or if the data bank pointers have changed, an error is declared.

RESUME is a key word but not a reserved one since it preempts nothing. Since nothing follows it, it cannot be construed as a locator and the word can be the name of an attribute (a very unlikely one) in other contexts.

Order of the words and phrases is fixed and they must appear in the order shown. However, HALT can be used without REPORT and the filler words FOR and LEVELS can be omitted or not

When HALT is used. the automatic display will be intermixed with other type-ins and displays at the halts. If a clean hierarchical output is desired, REPORT should be used to produce one off-line. Other printing commands and options should not, of course, be used at the halts unless so intended.

7.2 1 Illustrative Example of SEARCH

To overcome the difficulty of getting a feel for a statement from its general format description, a small artificial example is given here. (Real examples that are manageable on a page are hard to find.) To simplify identification nodes will be labelled

(not really done) as follows:

N1, N2, N3 for the first level

N11, N12, ..., N21, .. for the second level

N111, N112 .., N211, .. for the third level.

Structure will be indicated by indenting and order will be by key attribute names. Attribute names will be indicated by Tn, Un, Vn, Wn where n=1,2,... . An ISD is indicated by a list of attribute names followed by K=k all enclosed in parentheses and indented to the level which it defines. Attribute value will be denoted by small letters followed by digits.

Consider then the following structure.

```
ROOT
     (T1, U1, V1, W1  K=12)
     N1=t1, u1, v1, w1
          (T2, U2, V2, K=14)
          N11=t21, u21, v21
          N12=t22, u22. v22
               (T3, U3, K=21)
               N121=t31, u31
               N122=t32, u32
          N13=t23, u23, v23
     N2=t2, u2, v2, w2
          (U2, V2, W2, K=15)
          N21=u21,v21,w21
               (T3, V4, K=22)
               N211=t31, v31
               N212=t32, v32
          N22=u22, v22, w22
     N3=t3, u3, v3, w3
          (T2, W2, K=15)
          N31=t25. w21
          N32=t26, w22
          N33=t27, w23
               (T4, V5, W5, K=23)
               N331=t41, v5, w5
               N332=t42, v5, w5
```

The results of the following searches are as shown. It is assumed they all start from the root.

SEARCH T1_T2_T3 FOR U3 <EQ> u32

Unique result is N122.

SEARCH DOWN TO T4 FOR V5 <EQ> v5 <AND> W5 <EQ> w5

Both N331 and N332 are found.

SEARCH DOWN TO T3 FOR V3 <NE> X <OR> V4 <EQ> v31

Nodes N121, N122, N211 are found.

SEARCH DOWN 3 FOR W2 <EQ> w21

Nodes N21 and N31 are found.

SEARCH TO 22 FOR V5 <EQ> v5

No match is found. (One occurs at 23).

SEARCH AT 15 FOR W2 <EQ> w2

Nodes N22 and N31 are found.

Determination of the search paths followed to find these is left to the reader.

8 Ø REPORTING FACILITIES

TEXTAG provides very flexible reporting facilities. The commands DISPLAY and REPORT output current values of working storage or the STOP node and its ISD (superior's D(Q)) or the ISD for its inferior set. These are all output in predefined formats which, however, are usually adequate for working purposes and even intermediate reports. For more complex or precise formatting, the PRINT (also PUNCH) and FORMAT statements can be used. The latter are slightly modified versions of FORTRAN FORMAT statement, familiar to most computer users. Finally, there is a powerful command called AUTOFORM for producing either numeric or symbolic tabular output with full stub and head, and long and wide page, facilities. In addition to tabular heads, page and section heads and a footnote are provided with the commands HEADING and FOOTING. There are also a set of NOTE commands for outputting arbitrary lines of text with single or double spacing. The command PAGE both starts a new page and defines page length.

Consequently, virtually any type of report can be produced.

The reporting facilities may be extended to graphs by using a subroutine available (loaded with LOAD command) and the use of TEXTAG macros. In one standard use of the prototype system, six graphs are output on one page with this approach. (Resolution is only fair; output is to a standard line printer.)

Results of various kinds may also be enfiled in arrays and then accessed with a FORTRAN (or other) program by means of a standard interfacing routine which accesses enfiled TEXTAG arrays. This is also, probably, the best way to output results to drive a plotting or other graphic device.

8.1 Use of Annexes

Facilities for handling textual material in annexes are rather meagre in TEXTAG. No attempt is made to duplicate the facilities of a context editor or page formatting system. Such processors are widely available and specialized for the purpose. An annex is loaded into the ANNEX file with the command READ. The only rules applied are the following:

A. Input is stored in lines of 64 characters.
B. If neither the 64-th character is punctuation, blank or last on input line nor the following character is blank, the line is cut back to a preceding blank and the remainder put on the next line.
C. An input line starting with one or more blanks causes any prior input to be stored and a new line started.

If the input has been produced by a formatting routine with page width of 64 characters, then TEXTAG will store it as is. Otherwise, the result will be ragged on the right.

An annex may be displayed or printed with the command SCAN. No output editing is done; the 64-character stored lines are just printed verbatim.

Character strings or lines can be found in an annex with the command LOCATE and strings in a line can be replaced with strings

of equal length with the command CHANGE. An option for substituting an entire line is provided. Deletions and insertions are not possible.

9.Ø COMPLETE LIST OF COMMANDS

TEXTAG is a two-level system (which sits on top of the two-level CP/CMS host). The first TEXTAG level is called the Interface Level. From the user's viewpoint, only five standard commands are recognized at this level and, excepting QUIT, only at this level (Some additional maintenance commands appear in Appendix F.) These commands are always typed at the terminal. From the standpoint of program structure, the interface level is considerably more important. It contains many routines and functions which are either executed automatically or provide services to the second level. In particular, the File Management Program resides at the first level.

The second level is called the Operational Level and commands may be executed from the user's terminal or from an external file, and these modes may be intermixed. The operational level recognizes 65 commands, subcommands and special lines (subject to later expansion). These commands are presented in seven functional groupings in the subsections below. The grouping is arbitrary in some respects. There are no restrictions between or among groups except two commands which must be executed first if used and those which can only be executed from an external file. These all occur in the Execution Control group, presented first, but some commands in this group can be issued from the terminal as well.

All command names longer than four characters except FORMAT, FINDSUP. FINDLO, FINDHI and variants on NOTE may be abbreviated

to four characters. These commands must be spelled in full.

The level-1 commands are presented first. They are

TITLE	Define page title and activate output spooling.
DECLARE	Declare internal user files.
EXECUTE	Invoke the operational level.
READ	Load an annex.
QUIT	Return to CMS.

9.1 Invoking and Initiating TEXTAG

TEXTAG is invoked by simply typing its name at CMS level. (This assumes the TEXTAG residence disks are accessbile.) It first announces itself with a display at the terminal and then gives the prompt

READY:

awaiting a typed-in command. (Prompts at the operational level consist of simply ">".)

The first command normally issued is called TITLE. It serves two functions: to define a title line (first header) for all printed output, and to activate the spooling mechanism. The title line can be changed subsequently but unless TITLE is issued, all offline printing will actually be output only at the terminal. There are three classes of output, called message classes, which are explained in section 9.8 under the command MSGCLASS. TITLE also sets these classes to default dispositions. The format of TITLE is simply

TITLE 'any text up to 64 characters in length'

The surrounding single quotes are mandatory and single quotes may not appear within the title. The entire statement must end by position 72 and no continuation is allowed. (At the operational level, up to 80 characters can be used for a new title line.)

The second command, which must be issued, is DECLARE which is used to declare all the user's internal files to be accessed. The file functions DATABANK, ANNEX and ARRAYS are the usual ones. If more than one ANNEX or ARRAYS file is to be used, they must

all be declared. The DECLARE command is explained in Appendix D since it involves host system conventions.

Unless one is loading an annex, the next command is simply

EXECUTE DATABANK=filename [,SOURCE=file,deck]

This invokes the operational level for the named data bank (filename). If the optional SOURCE phrase is used, the first commands at the operational level will be taken from the source specified. This is not necessarily the same as issuing a RUN command from the terminal at the second level. If LOAD and/or USE commands are required, they must be issued first. When it is desirable to incorporate such commands in program decks, the operational level must be entered with the above SOURCE phrase.

The operational level will eventually be exited and return is to the interface level. Another EXECUTE statement may then be issued for a different data bank. This is necessary in some restructuring sessions when parts of an old data bank are to be used in a new one. See Appendix F for details on this and also certain other maintenance functions at the interface level.

The command QUIT causes all open files to be closed and control to revert to CMS.

9.1.1 Loading an Annex

Annexes are used at the operational level but they may be loaded to the ANNEX file only at the interface level. The command to do this is

READ filename,deckname TO annexfile

where "filename,deckname" specify an external file and a deck in it containing the text, and "annexfile" is the ANNEX filename which must have already been declared. The external source file has a standard deck structure. The first line of the deck (after the NAME card) must be as follows:

ANNEX: symbol [LINK TO oldsymbol]

where "symbol" is the unique name by which the annex is to be known (the name used for R-form attributes). More than one annex can be loaded by ending each one with a :END line followed by

another ANNEX: line. The last one need not have a :END line since the ENDATA will terminate loading anyhow.

If the optional LINK phrase is used, "oldsymbol" must be the name of an existing annex. The new annex will be linked to the old one and the two may be accessed in sequence with LOCATE and SCAN commands at the operational level.

There is no absolute limit on length of text but one annex should not contain more than about 2000 characters (about 30 lines) for convenient and efficient use. See section 8.1 regarding line packing.

9.2 Execution Control Commands

There are 14 true commands, one immediate instruction and three special line formats in this group. Eleven of these may be used only in a program deck or macro, but all may be except as explicitly noted. To clarify the different senses in which "execution control" is used here, the group will be presented in four subgroupings.

9.2.1. Subroutine Loading and Macro File Declaration

The two commands in this subgroup must be issued first on entering the operational level if they are to be used. If both are used, they must be issued in the order here presented. They may not be repeated.

The LOAD command is used to obtain subroutines in assembled or compiled form which are to be used. The format is simply

LOAD subroutine1 [,subroutine2 ...]

Appendix E gives details on the source of the subroutines and various considerations about order and number of subroutines loaded.

The USE command specifies an external file containing macros which are to be available for the session. Each macro is a deck.

The format is

USE MACLIB=filename

An in-core directory is built to avoid further searching of the file and to establish limits for each macro. Macros are then invoked by using their names as commands. See Appendix C for rules for creating and using macros, and Appendix D for file naming conventions.

9.2.2 Transfers of Control in a Deck

All seven commands in this subgroup, plus label lines, are errors if issued from the terminal.

Since TEXTAG is completely interpretive -- no compilation phase is used -- the only possibility for transfers of control is by skipping forward or backward over the lines constituting a program deck or macro. To make this have the appearance of transfers to labelled statements separate label lines are used since most TEXTAG statements are free form and no label field is available. Transfer of control to a label line really means to the line following. (The use of some queer mark or spacing convention on a statement line to indicate a label is undesirable for a number of reasons.) A label line is one of the few rigid formats (along with NAME, :END, ENDATA and three more presented below). The format of a label line is

//label

where "label" is limited to six characters constituting a symbol. In referring to a label line, the slashes are not used, just "label".

Transfer to a label may be done both conditionally and unconditionally, both forward and backward. There are two unconditional forms, one of which is completely different in execution from true commands, and it will be explained first. The instruction format is

GOTO label

where GOTO is spelled as one word in positions one to four. GOTO is executed directly by the line scanning routine and is very

fast. It is often exactly what is wanted, in spite of its rigid format. For example, at the end of a section of code whose execution is conditional, it is often required to jump over a following section. GOTO is the better choice for this. Furthermore, in a macro, GOTO is the only means of transferring to a label.

The other unconditional transfer of control is a true command called SKIP. Its format is the basis for the conditional forms as well. The format is

SKIP n|-n|TOP|label

where

n (an integer) means to skip the next n lines

-n means to skip back n lines

TOP means to start again from top of deck

label has the meaning "GOTO label".

The last form may not be used in a macro. This applies to the conditional forms as well.

Note the difference in counting for n and -n. If the skip is on line 15, SKIP 5 skips lines 16-20 and transfers control to line 21. SKIP -5 skips to 15-5=10 and execution begins there. The n may be any integer-valued expression. In counting lines, all lines must be used including comment and label lines.

The arguments for SKIP will be referred to as "skip" in the conditional commands which take the same forms. The three commands are as follows:

IF boolena-expression, "skip"

If the boolean-expression has value 1 (true) the skip occurs, otherwise control passes to the next line.

IFNOT boolean-expression, "skip"

If the boolean-expression has value 0 (false) the skip occurs, otherwise control passes to the next line.

TALLY I:symbol, "skip"

The I-variable is decremented by 1. If still positive, the skip occurs, otherwise control passes to the next line. Note that an (almost) endless loop does not occur if the I-variable is initially zero or negative.

In program decks (but not macros) a dynamic location table with up to 16 entries is maintained. Whenever a label line is encountered in passing (but not skipping), it is ignored but the label and line position are recorded. If the table fills, the oldest entry is replaced. (A label is not entered again if it is already in the table.) Thus a backward GOTO or skip is usually found in the table. If not, a forward circular search is done, at most once around the deck. If not found, an error is declared and control returns to the terminal. A GOTO in a macro always involves a search but these decks are usually short and most references are forward.

The LOOP and CONTINUE commands, discussed in detail in section 5.3, are also in this subgroup. Execution control is maintained separately for loops, using a stack. The maximum nesting level is five. (This is easily expanded if it becomes necessary.) Skips or GOTOs which transfer control outside the loop are not only illegal but fatal errors. They would hopelessly confuse the processor and the session would probably have to be aborted. (Essentially, the stack would never clear.)

9.2.3 RUN, QUIT, EXIT, ENDATA and Interactive Responses

In this group, only ENDATA and interactive responses are illegal from the terminal but the interpretation of these instructions may depend on where they are issued.

The RUN command is equivalent to a SOURCE phrase and takes the format

RUN filename, deckname

Execution always starts from the top of the program deck. If a RUN command (or an equivalent SOURCE phrase) is executed within a program deck, the existing program file is closed and the new one opened. There is no provision to return to the first program deck. RUN or a SOURCE phrase is illegal in a macro. However, READMOD may be used.

The interpretation of QUIT depends on where it occurs. The

general format is

QUIT [OPTION]

where the option (named OPTION) applies only to a program deck. (It is ignored elsewhere.) The interpretations are as follows.

(a) In a macro, QUIT means "return to main program" whether this is the terminal or a program deck. Macros may not be nested.

(b) In a program deck, QUIT means to return to the terminal. If OPTION is not specified, this is absolute and the program file is closed. If OPTION is used, the following display occurs at the terminal

QUIT COMMAND IN PROGRAM. SHALL WE CONTINUE? (Y/N)

If the response is Y, execution continues from the program deck; if N, the file is closed and the next command taken from the terminal.

(c) At the terminal, QUIT = EXIT.

(d) At the interface level, QUIT means to return to CMS; the TEXTAG session is finished.

The command EXIT has no arguments and is absolute wherever it occurs. The line

EXIT

causes the operational level to be closed out and control to return to the interface level. If any internal files are open, any modified pages are first rewritten to the file and then all files are closed. All of working storage is released and lost.

ENDATA is not properly a command but an end-of-deck but it is interpreted if control ever reaches it:

(a) In a macro, ENDATA = QUIT (i.e. "return").

(b) In a program deck, ENDATA = QUIT with no option.

(c) From the terminal, ENDATA elicits the response "?".

It is often desirable to be able to make decisions or to enter parameters from the terminal while a program is executing. Two special line types are provided for this purpose. These are in the program deck but have the effect of taking input from the terminal. A NOTE command (see section 9.8) should precede to indicate what is expected, for example,

NOTE 'DO YOU WANT TO ... (Y/N)'

```
NOTE 'TYPE IN NAME|VALUE OF ...'
```

The first would be followed by the line

```
?Y/N
```

in positions one to four. This line is interpreted as

```
"IF RESPONSE <EQ> Y, 1"
```

By phrasing questions properly, the action for a negative response can often be put in the following line. Any response but Y (yes) is interpreted as N (no).

The second special form is equivalent to a literal expression of one term. The second NOTE statement above would be followed by something like

```
p:symbol = ?INP
```

where p is an appropriate prefix (I|E|F|N but not S). The value typed in replaces "?INP".

9.2.4 Using Subroutines and Host Facilities

Subroutines made available with the LOAD command can be executed in two ways: with a CALL command or with the prefic C, i.e., C:routine where "routine" is the name of a loaded routine. The prefix C can be thought of as an abbreviation for CALL but routines so used must be single-valued like function subprograms in FORTRAN. The result is returned in "floating register Ø" and is treated either as F-form (consistent with IBM's FORTRAN) or N-form. A CALL statement can be used for multiple-valued functions, or almost anything, in usual fashion. However, subroutines called in either way should not do I/O. This may, in fact, be possible if sufficiently extensive arrangements have been made but is not guaranteed to work. (An exception exists for printed output produced by an assembler-language subroutine using the standard TEXTAG output routine. Linkage for this is provided and is used by some standard subroutines available.)

A call statement has the format

```
CALL routine(arguments)
```

and a function call the format

```
C:routine(arguments)
```

The latter can be embedded in a general expression, for example

```
F:result=SQRT(EXP(C:routine(F:argument)))
```

but function calls may not themselves be nested. (The subroutine may call other subroutines in normal fashion however.)

Arguments can be almost any quantities in working storage except tables. Vectors, matrices and lists may be passed and individual elements of tables may be passed, even in a loop. For example,

```
T:name(!1,ANS)=C:routine(T:name(!1,ARG))
```

is legal but not

```
T:result=C:routine(T:argument)
```

The former results in as many calls of "routine" as the row dimension of T:name. The latter implies multiple values and is nonsense in the context of the language. Nevertheless, in the case of a vector, matrix or list used as an argument, the entire array is passed. Two situations occur, for example:

```
CALL routine(M:symbol(1,3), F:result)
CALL routine(M:symbol, V:result)
```

Both are legal but the first does not accomplish what is intended (presumably). The values of the element of matrix M:symbol in row 1, column 3 and the variable F:result will be passed to the routine but in temporary locations. The routine may put new values in these locations but they are ignored. The proper statement (assuming the subroutine appropriately written) would be

```
F:result=C:routine(M:symbol(1,3))
```

In the second CALL statement above, the locations of the matrix and vector themselves are passed to the routine. Assuming the dimensions are understood, the routine may put new values directly into the arrays. A matrix in working storage is ordered as in FORTRAN.

Note: This is the transpose of everything else including a matrix attached to a node and a PL/I matrix but FORTRAN programming predominates. This decision is subject to change. If an array is only for the called routine's own use, it makes no difference which way it treats it. This occurs in existing

routines which only need so much working space.

The origin of TEXTAG's own print and punch output routine is automatically passed with every CALL statement but this is only usable at assembler language level.

A number of CMS commands of the host system can be executed while in TEXTAG environment. This is done with the command CMS, as follows:

CMS cms-command and arguments

One CMS command recognized is CP so the following is possible

CMS CP cp-command and arguments

The list of CMS commands which may be executed is given in Appendix G, along with certain restrictions and added syntax necessary to prevent double conversions. (Essentially, everything but symbols must be enclosed in quotes.) The CP level can always be reached by pushing an interrupt key on the terminal.

9.3 Array Packet Commands and Subcommands

The line introducing a packet of array definition lines for working storage is actually a command even though under CREATE it appears to be merely a packet header. The ROW: lines are subcommands. All these commands have a colon as the last character of their name. As commands, they may be abbreviated to four characters but under CREATE they must be spelled in full since the colon is meaningful. The complete list follows.

```
VECTOR: symbol(n) [,I|E|F] [,INDEXED]
MATRIX: symbol(m,n) [,I|E|F] [,INDEXED]
     ROW:  (as appropriate)
LIST: symbol(n)
TABLE: symbol(I|E|F)=head1,...,headn    (a T-table)
TABLE: symbol(2|4|8)=head1,...,headn    (an A-table)
TABLE: symbol(S)=Aa [,Bb,...,Hh]        (an H-table)
     ROW:  (as appropriate for any table)
```

Note carefully that the sequence for a table differs from that for a vector, matrix or list. The type indicator for a table is

essential whereas for a vector or matrix it is optional and for a list merely implied The FORM command, however, uses the table sequence for all arrays, since it is natural in that context. (In the above, a construction such as LIST: symbol=n would appear awkward or incorrect.)

All the packets following any of the above except ROW: are terminated with

:END

Section 2.5.1 gives full details on the formats as used under CREATE. They are the same for entering arrays into working storage except that PACK is not recognized. (If it appears, it is ignored, not treated as an error. Hence the same packet may be used for both purposes.)

The command READMOD is also considered to be in this group. It has the format

READMOD A:symbol, filename,deckname

This command may be considered a subenvironment. It takes input from the file and deck named and either enters arrays into working storage or modifies those already there. It can add or delete rows in tables (only) but not columns. The A-table specified is created with a stub consisting of the names of all (other) arrays created or modified and one column showing, in coded form, what action occurred. Appendix B gives full details.

The elements used in an array definition may be in form of expressions in variables or arrays already defined in working storage. There are restrictions on this feature under READMOD however.

9.4 Node Definition Creation, Deletion and Searching

The following commands and subcommands in this group have

already been fully explained in prior sections as indicated:

See section 2.5 for the following:

```
DEFINE LOWER K=k [NP] [SOURCE=file,deck]
    ATTRIB: symbol [AS] form [,...]
CREATE INFERIOR K=k [ID] [SOURCE=file deck]
    NODE: [attribute-name=]value [,...]
```

See section 7.1 for the following:

```
FIND  [-n|katr_]identifier_chain
FINDSUP
FINDLO, FDHIER, FINDHI, FDLOER
```

See section 7.2 for the following:

```
SEARCH locator|depth-limit [FOR] boolean.exp [,
    REPORT] [,HALT [ON HIT] ]
```

or

```
SEARCH RESUME
```

There are four further commands in this group which have not been mentioned thus far. They are for modifying the structure. as opposed to the content, of an existing data bank. The first one is PRUNE which has the format

```
PRUNE  STOP|LOWER|D:attribute
```

The first option deletes the STOP node and all its substructures of any kind Its neighboring peers are re-linked. The second option deletes the D(Q) and the entire inferior set of the STOP node but leaves the latter and its own arrays intact. The third option refers to a P-form attribute and prunes only the corresponding array attached to the STOP node.

The PRUNE command leaves dead space in the DATABANK file which may not be reusable. This can become a dead weight on file handling, using up disk space and slowing down execution. Appendix F discusses further commands at the interface level for correcting this condition and other purposes.

The other three commands are for "taking a data bank apart and putting it back together". They utilize two temporary (scratch) files available to the system and take advantage of a special argument to DEFINE, which has the format

```
DEFINE REBUILD=name [,NP] [,SOURCE=file,deck]
```

followed by the usual ATTRIB: lines. These new attribute names are presumed to contain a subset of the old ones in the D(Q) of the STOP pointer. Only one such definition may exist at a time.

The name specified must be a unique label. The following command can then be issued:

DUMP dumpname [,OLD]

The branch consisting of the inferior set of the STOP node will be written to the first scratch file, restructured according to the latest REBUILD definition and identified by (name,dumpname). If no REBUILD definition exists or OLD is specified, the existing node structure is kept and "name" will be the key attribute value of the STOP node.

The next command forms the complete identifier for the STOP node and this and the node itself can be saved on the second scratch file. It has the format

IDENTIFY [SAVE][, DISPLAY| REPORT]

The identifier and node can be displayed or reported with the options, DISPLAY being the default if no arguments are given. If the SAVE option is used, the identifier chain and end node are built into an abbreviated tree on the scratch file which is, in fact, a new data bank. (See below.)

After having issued appropriate DEFINE, DUMP and IDENTIFY commands (after any required PRUNE commands), the operational environment should be exited. The scratch files will remain intact at the interface level. One then initiates a new data bank by returning to the operational level with an EXECUTE command. The command RETRIEVE can now be used to retrieve dumped, rebuilt branches and graft them onto a new structure. The format is

RETRIEVE (name,dumpname), K=k

The retreived branch will be grafted onto the current STOP node as though DEFINE and CREATE commands were issued. The K number redefines the numeric key for the first inferior level. Lower levels will be left as they were unless their k-values are equal or lower than the specified k in which case they will be incremented to make the lowest value one greater than the specified number.

The scratch files do not stay around for ever. They can be saved as explained in Appendix F. However, one special case occurs for the second one which is explained below.

9.4.1. Using the IDENTIFY File

The second scratch file written by the IDENTIFY command with SAVE option does not appear to serve much purpose as described above. In order to utilize it, it is necessary to return to either the operational or CMS level and issue the following CMS command, where the options must be used at the operational level:

[CMS] COPY[FILE] SCRATCH2 DBFILE D1 filename2 DBFILE A1

(If the first scratch file is also to be retained, then

[CMS] COPY[FILE] SCRATCH1 DBFILE D1 filename1 DBFILE A1

must also be issued before returning to TEXTAG.) See Appendix F for details; the D1 and A1 arguments may change in some circumstances. Then one can invoke TEXTAG again, using filename2 in the EXECUTE command.

The reason for this seemingly round-about approach is due to limitations on the number of files which may be declared at once and to free the user from concern about whether he needs too many. Furthermore, if he had forgotten to declare a needed file until the time the IDENTIFY command is to be issued, a great deal of work might be lost. The scratch files are always available but they cannot safely be renamed within TEXTAG or copied at the operational level for immediate use. Since only a minimum of effort is required at the interface level to re-invoke TEXTAG, this seems the lesser of two evils.

The structure of the file is a legitimate data bank but each superior node which does not lead directly to an end node (in its inferior set) is abbreviated to its key attribute only. The D(Q) of the superior to an end node (the STOP node when IDENTIFY was executed) is recorded in full along with the full node. Several end nodes may in fact be peers, in which case only one identifier chain and one D(Q) are required for the set.

9.5 General and Special Assignment Statements

A general assignment statement has the form

result = expression

where the result is single-valued but, if automatic indexing or name matching is involved, this is interpreted as applying to each indexing combination. In principle, there are three kinds of quantities -- arithmetic, symbolic and logical (boolean) -- but the distinction is largely blurred due to TEXTAG's liberal interpretations. The following combinations are possible with the results shown.

(1) arithmetic result = arithmetic value

Form conversion is automatic, the only restriction being that if the left member is an integer, then |value|<32768 must hold.

(2) arithmetic result = symbolic value

This is illegal. However, for a specific case, see the CVTR command below.

(3) arithmetic result = logical value

The Ø or 1 is floated if necessary.

(4) symbolic result = arithmetic value.

This is illegal only if $|value| \geq 10^{n}$ where n is the length of the result in characters (n=2,4,8). Otherwise, the whole number part of a nonnegative number is converted to character code with leading zeros if necessary. For example, if n=8 and value=45132.67, the value is converted to

'ØØ045132'

If the number is negative, tens complement form is used. For example, the negative of the above number is converted to

'99954868'

This feature is sometimes handy for creating coded forms. For example, if I:NO is between 1 and 99, the following statements create a symbol NO.Ø1---NO.99 in N:NO.

```
N:NO = I:NO
N:NO = NO. & MASK(N:NO,'ØØØØØØXX')
```

(5) symbolic result = symbolic value
This is always valid but truncation may occur if the result is a shorter form than the value.

(6) symbolic result = logical value
This always results in '0' or '1' preceded by the necessary number of '0' characters, i.e., the same as (4).

(7) logical result = arithmetic value
The bit (B:letter or a B-form attribute) is set to the parity of the whole number part of value.

(8) logical result = symbolic value
This is illegal.

(9) logical result = logical value
Obviously unambiguous.

An expression may contain subexpressions of all kinds. For example,

T:ARG(N:PREFIX & N:I,I:J) <GT> E:TEST+1

has a symbolic expression to denote a row, an arithmetic expression for a comparand (in fact both comparands are), and the whole thing is a logical expression. Consequently, it makes little sense to talk about kinds of expressions, except with respect to final value.

Four commands may be regarded as specifying the use of special assignment statements. These are

ASSIGN, CVTR, DIMEN, TIME

(The command SET uses phrases like assignments but they refer to system quantities.) The command ASSIGN applies only to strings and has the form

result.string = arg.string [& arg.string ...]

as already fully explained in section 4.5.

9.5.1 CVTR Command

The CVTR command (ConVert To Real) converts a symbolic value which looks like a number to a real number. The statement structure is

CVTR arithmetic.result = symbolic.expression

If the left member requires an integer, the real number is converted to integer format in the usual way. The symbolic value must conform to the following rules:

(a) Maximum total length is 8 characters.

(b) Either + or - may be the first nonblank character. At most one of either but not both may appear and must be immediately followed by a digit or a decimal point.

(c) One decimal point may appear anywhere after a sign, if any.

(d) Leading blanks are ignored; first blank after a nonblank terminates the string.

(e) Any other character except the digits Ø,1,...,9 is an error.

Note that exponential format is not allowed.

9.5.2 DIMEN Command

The DIMEN command takes the following formats:

(a) DIMEN I:[N:]symbol = V|L:[N:]symbol

The length of the vector or list is assigned to the integer variable. If the vector or list does not exist, the result is -1.

(b) DIMEN I:[N:]symbol = M:[N:]symbol(ROWS|COLS)

the row or column order of the matrix is assigned to the integer variable. If the matrix does not exist, the result is -1.

(c) DIMEN I:[N:]symbol = T|A|H:[N:]symbol(ROWS|COLS)

The row or column order of the table is assigned to the integer variable. A result of Ø indicates a head or stub table, respectively. A result of -1 indicates the table does not exist. (The synonyms STUB=ROWS and HEAD=COLS may be used.)

(d) DIMEN I:[N:]symbol = T|A|H:[N:]symbol(stub,Ø)
where "stub" is a stub symbol. The result is one of the following:

-1 the table does not exist;
Ø the stub symbol does not occur;
i the stub symbol is for row i.

(e) DIMEN I:[N:]symbol = T|A|H:[N:]symbol(Ø,head)
Same as (d) reading "head" for "stub" and "column" for "row". For an H-table, "head" must be of the form

A1|A2|...|A8|B1|...|B8|C1|...|H8

(f) DIMEN I:[N:]symbol = T|A:[N:]symbol(SIZE)
Result is 2, 4 or 8 if table exists, else -1. This refers to the size (number of bytes) of the elements.

(g) DIMEN I:[N:]symbol = H:[N:]symbol(A|B|C|...|H)
The result is one of the following:

-1 the table does not exist;
Ø the indicated string does not occur;
j the indicated string is j words long.

(h) DIMEN I:[N] symbol = S:[N:]symbol
The result is the number of characters in the string unless it does not exist; then the result is -1.

For any of the above except (h), the prefixed symbol on the right can be replaced with D:[N:]symbol to refer to an array attached to the STOP node. If the reference is not to a P-form attribute, the result is -2; if it is P-form but the array does not exist, the result is -1.

9.5.3 TIME Command

The TIME command assigns the time of day to an N-variable. The statement is simply

TIME N:[N:]symbol

The N-variable is defined if necessary. The value assigned is in the form

hh:mm:ss

(hour:minute:second) on a 24-hour clock.

Note: The full time and date can be displayed, but not stored, with the command DISPLAY TIME.

9.6 Miscellaneous Commands

Seven commands are included in this group:

MSGCLASS for output disposition.
SET and QUERY for a variety of functions.
DISPLAY and REPORT for preformatted output.
LOCATE and CHANGE for annexes.

9.6.1 MSGCLASS

Three classes of printed output are predefined in the system:

ERR[OR]	Reporting of detected errors
LOG	Short informational messages and dialogue
REP[ORT]	Printed material usually wanted off-line

The definition of these classes is not a prerogative of the user but he can control their disposition and the command REPORT is simply a synonym for DISPLAY which changes the latter's disposition from LOG to REP.

There are three dispositions of printed output:

ON[LINE]	to the terminal display unit,
OFF[LINE]	to the output spooling file which is eventually routed to a line printer,
BO[TH]	both ON and OFF.

Until a TITLE command is issued at the interface level, or until a MSGCLASS command is issued at the operational level, all output is to the terminal. (If MSGCLASS is used without TITLE to activate OFF, a canned title is used.) When TITLE is executed, it sets the standard dispositions as follows:

ERR class to BOTH
LOG class to BOTH
REP class to OFF

The MSGCLASS command is used to change these. It has the format

MSGCLASS class = disposition

(Any class or disposition name can be abbreviated to 2 characters.) The only unallowed combination is

ERR = OFF

MSGCLASS can be used as often as desired. A title line can be changed at the operational level with the command HEADING explained in section 9.8. However, this command does not activate output spooling of itself.

9.6.2 SET and QUERY

These two commands, which have been mentioned a number of times, are used for setting a variety of system variables and controls and for finding out the status of some of them. SET always implies an action; QUERY never implies any action except to display information. (All its output is LOG class.)

The functions performed by SET will be given separately since a general format is meaningless. QUERY will be discussed with SET where appropriate; its independent functions will be given last.

(1) SET PRINT = ON|COMM[ENT]|OFF

This action refers to the printing of lines executed or passed in a program deck or macro. The default setting is OFF (don't print lines). If ON is specified, every executed line, including comment lines (* in position 1) will be printed LOG class. This is only useful for debugging as it will normally be very voluminous. If COMM[ENT] is specified, only comment lines will be displayed. This is sometimes useful in monitoring progress of a job if comments have been put at the right place in the deck. (Don't put them in an inner loop unless debugging.) The option OFF cancels either or both the others. (If ON and COMM are specified in succes-

sion, comments appear twice.)

(2) SET EXTEND = YES|NO

This action refers to extending packed arrays attached to nodes. The default is YES. If NO is specified, a new entry for an already full packed vector causes an error instead of an extension. The command

QUERY EXTEND

displays YES or NO according as which is currently set. This is sometimes necessary to determine if extensions may have been created (inadvertently).

(3) SET HOLD(i) = START|STOP

SET START|STOP = HOLD(i)|STOP|START|ROOT

These were fully explained in section 2.3.

(4) SET ARRAYS|ANNEX = filename

If an existing ARRAYS or ANNEX file is open, any changed pages are rewritten and the file closed. The new file is then opened. The command

QUERY ARRAYS|ANNEX|DATABANK

displays the filename currently in effect. Note that DATABANK may not be set.

(5) SET PUNCH = filename

Here filename refers to an external file used for output from the PUNCH command. If an old file is open, it is closed. The new file is then made available for punch output. (A new one is not actually created until PUNCH is executed.) A punch file cannot be read on the same session. (This is a deficiency which may be correctable.) An old file is extended, not rewritten. An open file may be closed without specifying a new one by using

SET PUNCH = ' '

(One or more blanks between single quotes.) If the PUNCH command is executed without a filename being set, output goes to the card punch at the computing center. The command

QUERY PUNCH

displays the filename currently set.

(6) QUERY TYPE N:[N:]symbol = D:[N:]symbol

This was explained fully in section 3.2.

(7) QUERY class

This command is used to get a list of all names of a class of variables or arrays in working storge. "class" may be any of the following:

INTE[GERS]	all I-variables
SHOR[T]	" E- "
FULL	" F- "
SYMB[OLS]	" N- "
STRI[NGS]	" S- "
VECT[ORS]	all vectors
MATR[ICES]	" matrices
LIST[S]	" lists
TTAB[LES]	" T-tables
ATAB[LES]	" A- "
HTAB[LES]	" H- "

The optional endings may be omitted or spelled differently. Thus MATRIX and MATRICES are equivalent.

The DISPLAY command is used to display values and REPORT to print them.

Several phrases can appear on one SET line and several arguments on one QUERY by separating them with commas. They are processed left to right as encountered.

9.6.3 DISPLAY and REPORT

These two commands are identical except that DISPLAY produces LOG class and REPORT produces REP class output. Only DISPLAY will be shown.

The command has four different formats:

(i) DISPLAY class [,...]

(ii) DISPLAY class=symbol [,...]

(iii) DISPLAY D|p:[N:]symbol [,...]

(iv) DISPLAY LOWER[(STOP|START)]|NODE[(STOP)]

where

class is any of the classes used with QUERY under (7) in subsection 9.6.2, or BITS

symbol is the unprefixed name of a variable or array in working storage, or a letter for BITS

p is B|I|E|F|N|S|V|M|L|T|A|H corresponding to BITS and the QUERY classes

The different formats can be intermixed on one line, for example:

DISPLAY SHORT, MATRIX=name, B:A, NODE

will display all defined E-variables, the named matrix, the bit B:A, and the current STOP node.

All displays are in formats fixed in the system, all quite neat and readable except that matrices and tables with more columns than will fit across a page produce multiple (indented) lines for each row. Where this is not satisfactory, the commands AUTOFORM or PRINT must be used.

Further explanations follow.

(1) The format (i) should be clear. All defined variables or arrays in the class are printed out in succession.

(2) Formats (ii) and (iii), except the D-prefix, are equivalent except the prefixed name is output for identification with (iii). When it is desired to produce output without prefixed forms, (ii) should be used.

(3) Format (iii) with D:symbol produces the output implied by the attribute form equivalent to format (ii), i.e., unprefixed. The following have no equivalent in working storage and are displayed as shown:

X-form	attribute=integer (in decimal)
L|D|A|C-form	attribute=character
R|W-form	(same as symbol)
P-form	the implied array format

(4) Format (iv) without the parentheses always means the STOP node. The three combinations are as follows:

DISPLAY LOWER(START)

produces the definition of the structure of the STOP node with the phrases

attribute-name = form

DISPLAY NODE[(STOP)]

displays the node as though all D:symbol specifications had been listed.

DISPLAY LOWER[(STOP)]

produces the definition of the structure of the STOP node's inferior nodes.

9.6.4 LOCATE, SCAN and CHANGE

These three commands refer only to an annex. A SET ANNEX command must have been executed previously. Then the following command must be issued:

LOCATE R:annex|R:N:symbol

where "annex" or the value of N:symbol is the name under which the annex was enfiled by READ. LOCATE is then used in one of the following forms:

LOCATE n

where n is line number;

LOCATE $+k

move k lines forward;

LOCATE $-k

move k lines backward; or

LOCATE 'string'

find the first occurrence of the string, looking forward only. The quotes are required and the string may not contain a quote.

The SCAN command has the format

SCAN n|'string' [,REPORT]

Either n lines will be displayed or as many as necessary until 'string' occurs. The starting line is not checked for 'string'.

REPORT produces REP class output.

The CHANGE command is used after a line is located and has the format

CHANGE 'string1','string2'

meaning replace 'string1' with 'string2'. The two strings must be the same length. To replace an entire line, write

CHANGE $, 'new line text'.

The line found or changed is displayed after any of the above except the first LOCATE which displays the name and creation or last modification date of the annex.

9.7 Array Forming and Management Commands

The six commands in this group perform the following functions:

FORM	Form an array (or string) in working storage
DELETE	Delete an array in working storage
ENFILE	Enfile one array, or all of a class, or all arrays on ARRAYS.
REPLACE	Replace an array or arrays in situ on ARRAYS (leaves no dead space)
RECALL	Recall one, several or all arrays of a class, or all arrays from ARRAYS to working storage. Also used to list arrays existing on ARRAYS.
ERASE	Erase a specific array on ARRAYS.

The FORM command has the following general format

FORM p:[N:]symbol[(type|size)] = [stub.exp][,head.exp]

but the above is so general that it says very little. A full discussion is found in Appendix A which should be consulted.

The other commands in this group will be presented in appropriate subgroupings.

9.7.1 The DELETE Command

This command deletes arrays in working storage. The statement format is simply

DELETE p:[N:]symbol$_1$ [,p:[N:]symbol$_2$ ...]

where p=V|M|L|T|A|H.

DELETE does not guarantee that released array space is reusable. When working storage is assigned for various requirements, an attempt is made to re-use space from deleted entities but it may not exist in suitable blocks. There is no facility for compacting working storage.

9.7.2 The ENFILE and REPLACE Commands

The command ENFILE has the general format

ENFILE p:symbol [AS new-symbol]

or

ENFILE class|ALL [,LIST]

With the first form, one array is written to the file. It may optionally be renamed (but not changed in type). The second form writes all arrays of a class or all arrays of any type (V|M|L|T|A|H) in working storage to the file. A list of names within types may optionally be displayed.

In both cases, an existing array of the same name and type in the file is logically, but not physically, replaced. This leaves dead space in the file which can grow to troublesome size over time. When this happens, all arrays should be recalled and written to a fresh file.

When modifying only the values of an old array or arrays, and not changing their dimensions or types, the verb REPLACE should be used instead of ENFILE. (Syntax is otherwise identical.) This avoids the dead-space problem by rewriting arrays to the same physical space in the file.

The file of arrays to be written to is specified by use of the verb SET, with the ARRAYS phrase. Every file to be used must be declared at the interface level with DECLARE. Several files

may be used by repeated use of SET provided all have been declared at the interface level.

9.7.3 The RECALL Command

Arrays may be recalled from ARRAYS to working storage with the verb RECALL, or all arrays in a file may be listed. The statement has three forms:

RECALL p:symbol [AS new-symbol]

This recalls a single array with optional name change. The "new-symbol" may be a symbolic expression.

RECALL class|ARRAYS [= 'mask'] [,LIST]

This recalls all arrays of a class or all arrays of any type. If the optional mask is used, only those in a class or in all classes whose names match the selection mask will be recalled. The asterisk is used as a universal character. A list of all arrays recalled may optionally be displayed. The mask must be a literal enclosed in single quotes.

RECALL ,LIST

This does not actually recall any arrays but only lists all those in the file, by name within type.

If an array of the same name and type as one recalled already exists in working storage, it is first deleted. If "new-symbol" is used, this rule applies to it, not to the original name on file.

The file or files accessed are governed by the same rules as for ENFILE.

9.7.4 The ERASE Command

The ERASE command logically deletes a single array from the file of arrays currently set. This is its only function and it does nothing to working storage. It has only one rigid form:

ERASE p:symbol

The symbol must be a literal, not indirectly referenced, and only one may be specified per statement. This rigidity is intentional

to prevent inadvertent destruction of a file. ERASE leaves dead space in the file.

9.8 Report Generation

In this section, options will be denoted vertically within braces (like a set) instead of being separated with a vertical bar. This is necessary to reduce line length in some format descriptions.

The commands DISPLAY and REPORT were given in section 9.6.3 but may also be considered as Report Generation commands.

Recalling the definition of message classes, it was stated under the interface level command TITLE that initial defaults of all three classes are actually on-line but execution of TITLE establishes an output page heading and sets the normal defaults. TITLE is almost always executed before calling the operational level unless it is desired to use TEXTAG in an ad hoc way. There are also commands at CMS and CP levels which control the actual disposition of output print files but we can assume here that these have been appropriately used to deliver the output where the user expects it.

The page header defined by the TITLE command is regarded as "Header Ø". Seven more header levels can be defined for subtitles, column identification, etc. All eight levels may be set, changed and cancelled with level-2 commands.

There are essentially only seven different report writing commands although three have alternative symbols for variants of the funtions, giving ten altogether. Going from the simplest to the most complex happens to be a logical order of presentation.

9.8.1 The PAGE Command.

This command has two purposes: to start following output on a new page and to set page length (an option rarely used). The statement has the form

PAGE [{n / I:[N:]symbol}]

If the integer argument is used, it sets the number of lines per page to that value. Default is 54 lines per page, including all headers and spaces. Maximum paper size is 66 lines for American, 72 lines for European.

9.8.2 The NOTE Command.

This command is used to write notes or comments as LOG class. It has the basic format

NOTE 'any printable string without single quotes'

The single quotes are mandatory; neither they nor the command will be printed. (Single quotes are not printable except with the command PRINT.) Normally, NOTE double spaces but there are three variants:

NOTE1	Single space
NOTE1R	Single space and output as REPORT class
NOTE2R	Double space and output as REPORT class

The maximum length of the string is 69 characters minus the length of the command. A blank must appear between the command and leading quote; the final quote must not be farther right than position 72.

NOTE is often used with an interactive response sequence, described in section 9.2.3. To be effective, LOG class must include on-line.

9.8.3 HEADING and FOOTING Commands.

The HEADING command establishes a new title line (level 0) or page headers and sub-headers. It has the format

HEADING level,'any text not containing single quotes'[,]
['continuation']

where "level" is a literal integer 0,1,...,7. The level number establishes the heading for that level and cancels all prior headers with a higher level number. A level 0 header may have up

to 80 characters between quotes, all others may have up to 127 characters.

Since maximum-length text will not fit on one line, it may be continued on one more line as indicated.

The FOOTING command defines a footing for use with AUTOFORM. It is not recognized elsewhere. The format is

```
FOOTING 'any text not containing single quotes'[,]
        ['continuation']
```

A footing may have a maximum length of 127 characters and uses up three vertical spaces on the page (triple spacing).

9.8.4 FORMAT Statements.

FORMAT statements are used with PRINT and PUNCH; they are almost identical to those of FORTRAN but AUTOFORM uses a stereotyped form.

FORMAT is exceptional in TEXTAG in four ways:

(a) FORMAT statements may be continued over as many lines as desired. A FORMAT statement is considered to be complete when left and right parentheses balance. If this never occurs or if right parenthese are left over, action is unpredictable.

(b) A single quote may be represented by two adjacent single quotes in conventional fashion.

(c) FORMAT statements may appear anywhere before they are used (in time). They are executed, even though in a special way, and hence may not be in a loop. Encountering the same FORMAT label twice is an error.

(d) A label symbol is used which is limited to a length of 4 characters, the first of which must be a letter.

The general statement structure is

```
FORMAT label,(s1,s2,...)
```

where s_1, etc. are format items. The following format items are

identical to standard FORTRAN:

aIw, aFw.d, aEw.d, aAw, wX, Tw, 'text', a(...)

where

a is an optional repeat count; if omitted, the repeat count is taken to be 1. It must be an unsigned integer constant.

w is an unsigned non-zero integer constant which specifies the total number of characters in the field.

d is an unsigned integer constant specifying the number of decimal places.

The following are also allowed:

aCw.d Like aFw.d but with commas after thousands, millions, billions, etc.

a$w.d Like aCw.d but with leading currency sign. (Requires 1 more position.)

aLw This is not logical conversion (T or F) as with some systems. It is used only with AUTOFORM and produces "picture" output, i.e. 1 and -1 print as such, 0 is not printed, and all other values are encoded in the form '1' or -'1' where

represent <10, <100, <1000, <10000, ≥10000,

respectively and

represent ≥.1, ≥.01, ≥.001, ≥.0001, <.0001,

respectively.

For output to printer or terminal, the first character is interpreted as a carriage control in almost standard fashion and is not printed:

b advance one line and print

0 advance two lines and print

- advance three lines and print

1 throw to new page and print, but for terminal, same as -

Note that the usual + code for printing on the same line is not recognized since this is incompatible with system details.

Other restrictions are that a format item must not specify either an integer or a fractional part in excess of 9 digits, that not more than 132 characters per line (not including car-

riage control) must be required for use with PRINT, and not more than 8Ø characters for use with PUNCH.

9.8.5 The PRINT Command.

A statement with this command is essentially like the FORTRAN statements PRINT or WRITE(6,fmt) but the TEXTAG syntax imposes some differences. The statement structure is

PRINT fmt, data list

where "fmt" is a label defined in a FORMAT statement. Valid members of the data list, which are separated by commas, are the following:

Source	FORMAT Specification
I:symbol	I-format
E|F:symbol	F, C, $ or E formats, or I-format if only integer part wanted.
B:letter	I-format
N:symbol	A-format
L:symbol(i)	A-format
{T/A}:symbol(stub,Ø)	A-format
H:symbol(stub,Ø)	A-format
{T/A}:symbol(Ø,head)	A-format
A:symbol(stub,head)	A-format
V:symbol(i)	Same as E:symbol
T:symbol(stub,head)	Same as E:symbol
M:symbol(row,col)	Same as E:symbol

Any of the above may be indirectly referenced. Automatic indexing may be used (!n) for any stub or head, the order being determined by the n, higher n changing faster. However, automatic indexing will give multiple output lines over all index combinations (except for flags fixed within the range of a LOOP command). This is not ordinarily what is desired, but rather the kind of implied DO-loop used in FORTRAN. This is accomplished by use of dummy indices of the form <l> where l is a letter. Thus the referent

T:A(<I>,<J>)

will give the list

```
T:A(1,1), T:A(2,1), ..., T:A(m,1),
T:A(1,2), T:A(2,2), ..., T:A(m,2),
    .
    .
T:A(1,n), T:A(2,n), ..., T:A(m,n)
```

(Note that T:A with no indexing is illegal, unlike FORTRAN.) The indexing may be further controlled by forms such as

```
((T:A(<I>,<J>),<I>=b,e,d),...,<J>=b',e',d')
```

where b (beginning index), e (ending index or limit), d (increment) must be integers or I-variables. All table and array indexing (whether dummy indices or not) must consist of single terms, i.e. no expressions will be evaluated. No further parentheses other than those indicated are allowed and those implied are required. The increment d may be omitted, in which case 1 is inplied. In this case, the limit e must be followed by a closing parenthesis.

A dummy index may appear by itself and its current value will be output. However, the definition of the range of a dummy index must appear last and terminates the range of the loop. For example, the following statement

```
PRINT fmt,((<I>,M:A(<I>,<J>),<I>=1,5,2),<J>=2,4)
```

produces the list

```
1,M:A(1,2),3,M:A(3,2),5,M:A(5,2),1,M:A(1,3),...,5,M:A(5,4)
```

A check is made for balancing of parentheses at the end of the statement as well as for conformity with the associated FORMAT statement. Any discrepancies result in an improper-syntax error.

9.8.6 The PUNCH Command.

This command is identical to PRINT with the following exceptions:

(a) Line length is limited to 80 characters.

(b) Output is either to the virtual punch or to a card-image file of type DATA.

(See also SET PUNCH in section 9.6.2)

9.8.7 The AUTOFORM Command.

This command automates tabular reports. Using it is the easiest way to prepare such output once the command is understood, but it is the most difficult to explain.

First, one must prepare a T-table containing all the numerical values to be printed. It is possible that one might wish to print only symbolic values in which case an A-table would be prepared instead. In either event, no arithmetic or symbolic manipulations are done within AUTOFORM except those described for formatting purposes. To simplify discussion, it will be assumed a T-table is used for values.

Essentially, AUTOFORM prints the T-table as class REPORT but, if it did only this, it would be no different from the REPORT command. AUTOFORM has the following features:

(1) A FORMAT statement is used for external field definitions.

(2) Both long-page and wide-page extensions are automatic. For example, suppose a report requires 120 lines vertically and 200 positions horizontally. If page length is 54, this requires 3 pages vertically and 2 pages horizontally. The 3 vertical pages for the left-most 130 positions or less (depending on the FORMAT statement) are printed first. This is call a strip. Then the strip for the rightmost positions is printed.

(3) Both head text for tabular columns and stub text for rows of values are inserted automatically. Head text is inserted on the first page of every strip and stub text is repeated on all strips.

(4) One or two H-tables may also be used, one for stub text and one for head text. Head text may have multiple lines. (This is in addition to the eight levels of heading available with the HEADING command.)

(5) A footing line may be inserted at the bottom of all left-most strips. This is done automatically if the FOOTING command has been executed.

(6) If H-tables are used, the order in which tables are specified determines both selection and sort order for output, either by rows or by columns or both.

(7) Both the AUTOFORM statement and the FORMAT statement it references are relatively simple.

The AUTOFORM command (and its predecessors in other systems) is the outgrowth of extensive experimentation and discussion to try to achieve a reasonable compromise between elegance of reports and simplicity of specification. In principle, one could produce an AUTOFORM report using PRINT statements but the programming would be horrendous. On the other hand, some types of reports are not handled by AUTOFORM at all or only with unsatisfactory compromises in page arrangement. AUTOFORM represents about the most that can be done with one command for a fairly broad class of requirements. When it is not suitable, PRINT may be used for full generality.

A FORMAT statement used with AUTOFORM has the following stereotyped structure:

FORMAT label,([left margin,]stub,[reset,]body)

where

left margin is either Tn or nX. By using Tn for both left margin and reset, stub text could be placed on the right.

stub is either Aw or nAw which is interpreted as An*w. This must accommodate maximum stub-text width.

reset is either Tn or nX and provides extra space between stub-text and first value, or for resetting first value position.

body is any of the following: aIw, aLw, aFw.d, aCw.d or a$w.d for T-Tables, or aAw for A-tables. The initial "a" in these items determines the number of columns per strip. These format items may also take the form a(nX,Fw.d), etc. to provide more space between columns. As previously noted, aLw produces "picture format" with AUTOFORM. A typical specification would

be 50L2. Vertical headers can be created by programming with an H-table.

The format item for body also determines maximum column header width. Multiple-line headers are specified differently but each line must fit within body width.

The structure of the AUTOFORM statement is

AUTOFORM fmt,BODY={T/A}:symbol [,STUB=H:symbol][,HEAD=H:symbol]

If only the BODY phrase is used, the table is printed in natural order using its stub symbols for stub text and head symbols for head text. (This amounts to REPORT with a FORMAT statement.) If STUB or HEAD phrases are used, order of all three phrases is meaningful. The scheme works as follows.

1. The stub of the STUB table must contain a subset of the stub of the BODY table. Automatic name matching is implied. Only those rows whose names appear in the stubs of both the STUB and BODY tables will be printed.
2. The rows of the STUB table with matching names are used for stub text.
3. The stub of the HEAD table must contain a subset of the head of the BODY table. Only those columns whose names appear in both the stub of the HEAD and the head of the BODY tables will be printed.
4. The rows of the HEAD table with matching names are used for head text.
5. The order in which the phrases appear determines which table rules the order of stub and head:

BODY only	Natural order, all rows and columns.
STUB,BODY	Stub order for rows, BODY order for columns, row selection only (all columns).
HEAD,BODY	HEAD order for columns (based on stub of HEAD), BODY order for rows, column selection only (all rows).
X,X,BODY	STUB order for rows, HEAD order for columns (based on stub of HEAD), selection of both rows and columns.

BODY,X,X Full selection but BODY order for rows and columns.

STUB,BODY,HEAD Same as STUB,BODY but using HEAD for head text with column selection.

HEAD,BODY,STUB Same as HEAD,BODY but using STUB for stub text with row selection.

Thus the form of the command which gives the most selection and reordering is:

AUTOFORM label,STUB=H:symbol$_1$,HEAD=H:symbol$_2$,BODY={$^{T}_{A}$}:symbol

The use of the H-tables for stub and head text can be further controlled by appending specifications to the H-tables names in the form

H:symbol(hlist)

where hlist can have the following forms:

head	e.g.	A1
head-head	e.g.	A1-A3
head&head	e.g.	A3&B1

and, for HEAD only,

hlist/hlist	e.g.	A1-A3/B1-B3

In the examples to the right above, the meanings are as follows:

A1	use the 8 characters of A! for text
A1-A3	use the 24 characters of A1,A2,A3 for text
A1-A3/B1-B3	use A1-A3 then space a line and use B1-B3

These can be combined, as for example, for a HEAD

A1-A3&B1/C1-C4/D1&D3

9.8.8 AUTOFORM Variants

Two variants for AUTOFORM exist. If A-format is specified for a T-table BODY, this is interpreted as a predefined field format similar to G-format in FORTRAN. The minimum field size is 10 positions, as follows from left:

position 1: sign

positions 2-10: most significant 8 digits with embedded point if magnitude is in range [10^{-5},$a0^{8}$-1]

Larger magnitudes are printed in E-format with five significant

digits, i.e.

±.xxxxxExx

and smaller values as

±.xxxxE-xx

However, very small values are printed as

±.0000bbbb

and identical zero as 'b0.0bbbbbb'. If more than 10 positions are specified, the above forms are right justified in the field.

The other variant is the command AUTOFILE. This punches to a file (SET PUNCH must have been previously executed). Line length is 130 and no carriage control is created. The BODY table name is placed in the stub position for the (first) head line. AUTOFILE with A-format is very useful for preparing files for special transmissions.

APPENDIX A: Details of the FORM Command

The FORM command is used to create an empty string or array in working storage. In all cases, if a string or array of the same type with the same name already exists, it is first deleted just as though DELETE had been used. The space which was occupied by a deleted array may or may not be reusable. An attempt to use such space is made when new space must be allocated, even to the extent of trying to find an old block which most nearly fits the new requirement. However, an exact fit is often impossible and an accumulation of small "scraps" of storage may gradually eat into available space. No compacting is done.

It should be noted that the above rules apply only to arrays and strings. Other types of variables, all of which are fixed length, are never deleted or have their space reallocated. The user should avoid proliferating ad-hoc variables for temporary use. A better practice is to use a few standard names for such purposes, e.g., I:WORK, F:WORK, etc. The same applies to arrays if the same dimensions can be used. For example, if is often handy to form a vector, say V:WORK, with perhaps five elements or so for general use, particularly for accumulations.

Strings are somewhat of an exception in that DELETE does not apply to them but an old one of the same name will be deleted by FORM. A working string is very suitable since substrings can be specified in referents. Thus a string called, say, S:WORK with a length of 32 bytes, can be used in pieces for different requirements. For example,

```
S:WORK(9,3)
```

uses the 9-th through 11-th characters.

It is particularly important to adopt some convention for working variables and arrays for use in macros. A macro should almost never contain a FORM statement (unless its primary function is to set up a number of arrays initially). Similarly, a macro should not cause definition of a new working variable every time it is invoked. In addition to wasting space, it is a nui-

sance to have to specify a substitutable argument for what should be a common work variable. It is important to realize that macros are not like FORTRAN subroutines; they have no local variables. All variables and arrays are global.

The FORM command uses several formats and conventions. The simplest use is to define a string:

FORM S:symbol=n|I:[N:]symbol2

A string of n blank character is created; n may be specified with an integer variable, indirectly referenced if desired. The maximum length is 255. The space actually occupied by a string, as with all variables and arrays, is more than the length. A string, for example, occupies (n+13+r) bytes, including its name, where r is the number of bytes necessary to make the entire length congruent to zero (mod 8). The r bytes are wasted space. Thus only strings of length n congruent to 3(mod 8) waste no space. (Such wastage does not occur in nodes except possibly for the last attribute since nodes are "assembled" in an order which guarantees appropriate boundaries.)

The uses of FORM for lists, vectors and matrices are the next simplest cases, in that order.

FORM L:symbol=n|I:[N:]symbol2|dexp

where n or the integer variable here refer to the number of 8-byte words in the list. Thus

FORM L:WORK = 1Ø

forms a list of ten 8-character blank words which may then be referenced as L:WORK(i).

The "dexp" form needs more explanation. It is used in further cases and its full scope will be presented first. The abbreviations stands for "dimension expression", as with the command DIMEN, and refers to another array. Thus for example,

FORM L:WORK = L:SOURCE

forms L:WORK with the same length as L:SOURCE which must already exist. Any array referenced in a "dexp" must already exist. It need not be the same kind of array. Thus

FORM L:WORK = V:REFER

forms L:WORK with the same number of word elements as the number

of elements in the vector REFER, regardless of whether the latter is of type I, E or F.

When a "dexp" refers to a matrix or table, it must be specialized to refer to number of rows or number of columns. The various forms used are

M:symbol(ROWS), M:symbol(COLS)

T:symbol(STUB), T:symbol(HEAD)

A:symbol(STUB), A:symbol(HEAD)

H:symbol(STUB), H:symbol(HEAD)

However, it is sometimes desirable to be able to use a "dexp" without knowing whether it refers to a matrix or a table. Consequently, ROWS and STUB are treated as synonyms, and likewise COLS and HEAD. The reason one may be uncertain is that a D-prefix referring to a P-form attribute is allowed in a "dexp". Thus

FORM L:WORK = D:attribute(ROWS)

gives L:WORK the same length as the number of rows in the array attached to the STOP node regardless of whether it is a matrix or a table.

The use of H:symbol(HEAD) refers to the number of 8-character increments, not the number of strings. (When forming an H-table, the use of this form involves further rules described below.)

For vectors and matrices, the element form must be specified unless the default E-form is desired. This is done by use of a parenthesized form designator following the name of the array. Thus

FORM V:WORK(I) = V:REFER

forms the vector of integers named WORK with the same number of (zero) elements as the vector REFER, regardless of whether the latter is form I, E or F. (These are the only three possibilities.) Names of vectors must be unique regardless of element form; the same is true of matrices and lists and of each of the three types of tables.

The general format for forming a vector is

FORM V:symbol(f) = n|I:[N:]symbol2|dexp

where f=I|E|F. The general form for a matrix is

FORM M:symbol(f) = r, c

where either of r and c (number of rows and columns, respectively) may be any of the right members used for a vector. Thus

FORM M:WORK(F) = I:ROWS, T:REFER(STUB)

creates the matrix WORK with the number of rows equal to the variable I:ROWS and as many columns as the T-table REFER has rows. The elements of WORK are F-form, all zero as formed.

Forming tables involves not only dimensions and element form but also creating stubs and heads. The general formats for the three types of tables are:

FORM T:symbol(I|E|F) = [s] [,h]

FORM A:symbol(2|4|8|) = [s] [,h]

FORM H:symbol = [s],hlist

where s denotes a stub expression, h denotes a general head expression, and hlist denotes one of two special head expressions for H-tables. Tables may be null which is why either s or h is optional but an H-table must have a head. If both s and h are omitted, then the table is void, an option rarely if ever needed. Null tables are usually better replaced with lists but not invariably.

Both s and h may take several forms. The simplest is to create a stereotyped list,

characters & integer

This is a conventionalized expression and means to form all symbols starting with "characters" suffixed with '1', Ø1' or 'ØØ1' up to "characters" suffixed with "integer". For example,

FORM T:WORK(F) = R&1Ø, C&5

forms a 1Ø by 5 T-table of F-form elements (all Ø.Ø) with stub symbols

RØ1, RØ2, ..., R1Ø

and head symbols

C1, C2, C3, C4, C5.

Note that the size of "integer" determines how many characters are used for the "running index". Maximum length is 999 and total symbol length must not exceed 8 for the stub or 2, 4 or 8 for the head depending on element type or size. "characters" may be

an N-variable and "integer" may be an I-variable.

It is frequently convenient to use a blank string for "characters". For example, if twelve time periods are to be referred to symbolically by number, the expression

```
' ' & 12
```

forms the list '01', '02', ..., '12'.

Either s or h may be a "dexp" provided it refers to another table or a list (not a vector or matrix). Thus

```
FORM A:WORK(8) = T:REFER(STUB),A:UNITS(HEAD)
```

creates the A-table WORK with 8-character (blank) elements having the same stub as T-table REFER and the same head as A-table UNITS. An H-table head may not be used this way except for the head of an H-table being formed.

A "dexp" for a list is interpreted differently. The statement

```
FORM A:WORK(8) = L:ITEMS, L:MAKES
```

uses the values of the list elements. Since a list may be computed (symbolically), this is a useful convention for preforming the stub and head of a table with programming.

Finally s or h may be a conventionalized boolean expression in stubs or heads regarded as sets. The possible forms are:

```
dexp1 <OR>   dexp2
dexp1 <XOR>  dexp2
dexp1 <AND>  dexp2
dexp1 <ANOT> dexp2
```

where dexp1 and dexp2 are one of the forms

```
T|A|H:symbol(STUB)
T|A:symbol(HEAD)
L:symbol
```

The interpretation of the boolean operators is as follows:

<OR> all symbols in the first list followed by all those in the second list not also in the first (original list order in both cases).

<XOR> all symbols in the first list not in the second followed by all in the second not in the first.

<AND> all smbols in the first list which are also in the second list.

<ANOT> all symbols in the first list which are not in the second list.

(Note that <ONOT> is undefinable since the universe is unknown.)

For H-table heads, only the following two forms are admissable for hlist:

Aa [,Bb...,Hh]

just as for the TABLE: command, or

H:symbol(HEAD)

which copies the head of another H-table. The latter form is not admissable for any stub or the head of a T- or A-table. (The first form would be interpreted as a list of up to 8 symbols for a stub or a T- or A-table head.)

An abbreviated format is also recognized. This is simply one of the following:

FORM p:symbol = p:symbol2

FORM p:symbol = D:attribute

where the attribute must be an array. The array named on the left is formed as an empty duplicate of the one named on the right. This is particularly useful for saving an array attached to a node while further searching is done. For example, the statements

FORM T:WORK = D:TABLE

T:WORK(!1,!2) = D:TABLE(!1,!2)

saves the table attached to the STOP node for attribute TABLE in working storage as table WORK. All features are copied exactly except the name. (A packed vector or matrix is expanded in working storage.)

The prefix on the left (new array) must be the same as the

one on the right or implied by the D-prefix. That is, there is no way to avoid knowing whether it is a list, vector, matrix, T-table, A-table or H-table. Element sizes and dimensions need not be known.

APPENDIX B: The READMOD Command: Formats and Rules

READMOD creates or modifies arrays in working storage and, for tables only, can add or delete rows (but not columns). This facility serves three major purposes, here listed in order of increasing importance:

1. For initial loading of either a voluminous or a special set of arrays, using regular formats without individual commands. In fact, however, this is almost equally well done with a standard RUN command or an initial SOURCE phrase.
2. For reading in modifications and extensions to a standard set of arrays for special runs or analyses. For example, one might want to replace certain values in enfiled arrays, after executing RECALL, and to load special comparands.
3. If regular runs are to be made with standard programs, as for periodic updating of the data base, the standard program can incorporate checks on the validity or reasonableness of values read in via READMOD. Actual preparation of the input deck can then be more safely entrusted to clerical personnel or others not fully familiar with conventions used in the particular data bank.

The format for READMOD is

READMOD A:[N:]symbol, filename, deckname

Either or both filename and deckname may be literals or variables N:[N:]symbol. The A-table specified will be created by READMOD. If a table by this name already exists in working storage, it will be deleted and a new one created. This same rule is also applied to any entire array read in by READMOD but not, of course, to modifications. On the other hand, if an array to be modified does not already exist, the modifications are ignored

but an entry is made in the created A-table.

One can most easily regard READMOD as a subenvironment which recognizes only the following commands and subcommands:

VECTOR:	Define a new vector
MATRIX:	" matrix
LIST:	" list
TABLE:	" table
ROW:	Start row of matrix or table
:END	Terminate a packet
UPDATE:	Update an existing array
ADDROW:	Add one or more rows to an existing table
DELROW:	Delete one or more rows from an existing table
ENDATA	Terminate READMOD execution

All of these except UPDATE:, ADDROW: and DELROW: are exactly the same commands as given in section 9.3 with the formats detailed in section 3.2. The ENDATA line terminates READMOD execution and returns control to the next line of the program source which called it.

The three new commands use a prefixed format since they refer to existing arrays. The latter two are most easily disposed of first. They are single-line commands.

ADDROW:

ADDROW: T|A|H:symbol = $stub_1$ [,$stub_2$ ...]

The new rows are defined as an extension to the row order of the existing table specified. They are set to zero (T-table) or blank values of appropriate length. Note that ADDROW: only defines the stub, not values. It may be followed by UPDATE: to specify values for new rows already added.

DELROW:

DELROW: T|A|H:symbol = $stub_1$ [,$stub_2$ ...]

The named rows of the existing table specified are deleted from the table. The row order is modified to reflect the deletions. Deletions should be done before additions since the same space is guaranteed reusable.

UPDATE:

This command introduces a packet and has two formats; for arrays other than tables

UPDATE: V|M|L:symbol

and for tables of any type

UPDATE: T|A|H:symbol = $head_a$ [,$head_b$ ...]

The following lines of the packet depend on the type of array. For vectors and lists, indexed notation is used in the form

i_1 = value [,i_2 = value ...]

The values are adjusted to the proper form, I|E|F, for a vector (and also a matrix below). Size of list elements is unique.

For a matrix, each row to be modified is introduced with the line

ROW:=i, j_1 = value [,j_2 = value ...]

where j_k are the indices of the columns (in row i) to be changed.

Since indices are used for all the above, they need not be in monotonic order, either by i or j. In preparing the deck, changes can be written down as they are determined. The same i can appear more than once for a matrix.

The style for tables is entirely different. The head symbols specified on the UPDATE: line are a subset of those existing in the table. The rows are then specified as

ROW:=$stub_i$, $value_a$ [,$value_b$ ...]

where the values are for the columns denoted by the head subset, in the same order. The rows ($stub_i$) can be presented in any order.

As with the standard commands, a ROW: line both terminates a previous row (if any) and starts a new one. A :END line is used only at the end of all modifications for an entire array.

The Created Table

The stub of the created table consists of the names of all arrays defined or modified, in order of first appearance in the deck. Note that duplicate names may occur for arrays of different types. The table has one column with head symbol MODIFIED and 8-character elements. At least the first two characters will

be nonblank for each entry. The first character is simply the array prefix, V|M|L|T|A|H. After this first character, the remaining character(s) and their order depends on what was encountered in the deck. The codes are as follows.

N array is new and no old one existed

R array is new replacing an old one

U an UPDATE: line appeared for the (existing) array

X an UPDATE: line appeared for a nonexistent array

Y an ADDROW: or DELROW: appeared for a nonexistent table

Z a mismatch occurred (see below)

A an ADDROW: line appeared (properly) for the table

2 an ADDROW: line appeared for an existing row

D a DELROW: line appeared (properly) for the table

Ø a DELROW: line appeared for a nonexistent stub name

A mismatch means an index was out of range or a head or stub symbol for a modification did not exist.

Multiple occurences of the same (kind of) event are not shown. A new array, not later modified, will show either pN or pR.

APPENDIX C: Rules for Creation and Use of Macros

The structure of a file of macros is more fully described in sections 6.1 and 9.2.1 but each macro is a "deck" in the file which starts with a NAME card specifying the name by which the macro is to be invoked. A deck ends with a card (line) containing ENDATA in positions 1-6. Each macro has exactly one each of these lines. Exit from a macro occurs when either the ENDATA line is reached or a QUIT command is encountered. QUIT is described in section 9.2.3.

In between NAME and ENDATA, lines in a macro are standard TEXTAG statements with two exceptions: use of flags to denote substitutable arguments and certain restrictions on transfers of control explained in section 9.2.2. Here we are concerned only with substitutable arguments.

A substitutable argument is indicated with a flagged integer of the form %n where n=1,2,...,9 and refers to the position in the string of values supplied when the macro is invoked. (Hence one macro can have at most nine such arguments.) The question here is what can be substituted. To begin with, a command may not be substituted. Otherwise, the restrictions are of a different sort.

What a substitutable argument represents is a field in the statement. A substituted argument in a macro call overrides most parts of the corresponding field(s) represented by %n in the macro. A field consists of the following parts:

Prefix. Any of the 15 letter prefixes or double prefixes for indirect addressing, or any of the following:

0-4 for literals of various kinds

5 for !n

6 for "n

other codes for special forms not important to detail here.

Begin. A code representing leading plus or minus signs, left parentheses or leading absolute value bars.

End. A code representing right parentheses or ending abso-

lute value bars

Term. A code representing any of the following:

(a) a following arithmetic, concatenation, relational or boolean operator.

(b) a following equals sign, comma, underline or end-of-line mark.

Length attribute. Number of characters for any form of alpha, otherwise conventionalized.

Value. The numerical, logical or symbolic value of the referent proper. Some fields have a null value and zero length.

The corresponding parts of a substituted argument always override all the above parts of a %n field in the macro except for (b) under Term. Equals signs, commas, underlines, end-of-line marks and blanks never override.

Consider the following table reference:

```
T:TAB1(!1,N:COL) = ...
```

This breaks into three fields as follows:

	Prefix	Begin	End	Term	Length	Value
Field 1:	T:				4	TAB1
Field 2:	5	(	,		8	1
Field 3:	N:		)	=	3	COL

As an example of a macro, consider the following:

```
NAME          REPLACE
CALC %1 = FILL(LEAVE(MASK(%2,%3)),%4)
ENDATA
```

Then the statement

```
REPLACE  N:RESULT  N:ARG  '*000****'  XABC
```

replaces the 2-nd to 4-th characters of the value of N:ARG with ABC and assigns the result to N:RESULT. Note that the macro call could equally well be written in either of the two following forms:

```
REPLACE N:RESULT=N:ARG,'*000***','XABC'
REPLACE N:RESULT = (N:ARG, '*000****', XABC)
```

Clever design of macro model statements can make their invocation

appear like natural phrasing.

Macros without substitutable arguments are often useful too. All previously defined objects are available to the macro and definitions of variables, arrays, etc. made in a macro are available afterwards. There are no symbols local to a macro except the %n fields. (Macros differ in this respect from ordinary subroutines, which must be kept in mind.)

Note the use of CALC in the second line of the example. This is a dummy command. It is needed only when a line would start with a flag which can only occur for a substitutable argument in a macro. A line may not start with a flag. The dummy command CALC simply indicates an ordinary assignment statement. It could be used for any such statement but is ordinarily unnecessary.

Note that the following statement does not require CALC,

```
T:RESULT(%1,%2) = ...
```

It is required only when the first field is flagged.

APPENDIX D: Declaring User's Internal Files

All files used in connection with TEXTAG are CMS files. (For archiving or transferring files on tape, the CMS command TAPE is used. Refer to CMS documentation for this.) All CMS files have a three part designation

filename filetype filemode

abbreviated to

fn ft fm

(When the entire designation is referred to without distinction of its parts, it is called merely "file-id")

Of the three parts, only fn is completely arbitrary and both fn and ft are always restricted to a symbol containing nothing but letters, digits and the currency symbol ($). The first character must be a letter or $ but $ should not be the first character since this has a special meaning in the CMS editor.

The fm has the form "dn" where

d=A|B|C|D|E|F|G|H|S|X|Y|Z

and n=1|2|3|4|5. The letter refers to an accessible disk and the digit to access restrictions. Almost all user files have fm=A1 although B|C|F|G|H followed by 1|2 are possible. Fixed conventions are as follows:

A-disk	the main user disk normally available
D-disk	a temporary scratch disk
E-disk	residence disk for TEXTAG modules
S|X|Y|Z-disks	system disks

The digit following has nominal meanings for access control but these are not very effective. (See below.)

If not specified otherwise, all files are created with fm=A1. The TEXTAG scratch files are normally specified automatically as fm=D1. If no temporary disk space is available, the user is asked if he wishes to proceed using his A-disk. If yes, the scratch files have fm=A5 and the user is reminded to erase

all such files after the session. D-files disappear at log-off.

The actual access restrictions are controlled in a different way from the nominal meanings for n above. The user should normally use only n=1 for his own files. He may use B|C|F|G|H-disks only if they have been linked and/or accessed with appropriate CP and CMS commands. That subject is outside the scope of this paper; see CMS documentation or consult the responsible account manager. In any event, no user account can write on disks other than its own without explicit permission.

Although ft is in reality arbitrary, i.e., vis-a-vis CMS, in fact both CMS and TEXTAG use a number of conventions which have meaning at the corresponding system levels. An arbitrary ft should be used only for purposes of temporary manipulation (saving a file under a concocted file-id) and should be a garbled or specialized mnemonic which does not conflict with standard conventions.

For most purposes in TEXTAG, only fn need be specified, and assigning these symbols is the user's prerogative. The only exceptions are for the DECLARE command when fm must be specified or in using CMS commands where CMS rules apply.

The ft conventions used in TEXTAG are as follows:

TXTRUN	A TEXTAG program, external file.
TXTMAC	A TEXTAG MACLIB, external file.
DATA	Input to READ or READMOD or output from PUNCH, external files.
DBFILE	Any internal TEXTAG file.
TMODULE	The executable modules for TEXTAG.

The user has no access to the last except as they are invoked by commands. The DBFILE files are not editable nor printable. All the external files are. Tab conventions (horizontal spacing) are the same for all editable files and are those of ft=DATA which is a type recognized by the CMS Editor.

Only internal files need be declared. The format for the DECLARE command is

DECLARE ft1 fm1 [ft2 fm2 ...] $

where the terminating $ is required and must be preceded by at

least one space. For example, suppose the user needs the following files:

```
DATABANK    fn = RESOURCE
ANNEX       fn = RESREFS
ARRAYS      fn = RESWORK
```

Then, assuming all have fm=A1, the following statement is required

```
DECLARE RESOURCES A1, ANNEX A1, ARRAY A1  $
```

(The commas are optional.) They must all have ft=DBFILE.

In using the CMS command RENAME, one of the three following forms is sufficient.

```
RENAME  fn1 ft1 fm  fn2  =  =
```

The filename only is changed, from fn1 to fn2.

```
RENAME  fn1 ft1 fm  =  ft2  =
```

The filetype only is changed, from ft1 to ft2.

```
RENAME  fn1 ft1 fm  fn2 ft2  =
```

Both filename and filetype are changed. The filemode must not be changed. (The "=" means "same as before".)

A file may be copied to another disk with the CMS command COPY[FILE]. It does not admit the "=" convention. The format is as follows where the options must be used when in TEXTAG and [CMS] must not be used when in CMS.

```
[CMS]  COPY[FILE]  fn1 ft1 fm1  fn2 ft2 fm2
```

The user must have read access to the fm1-disk and write access to the fm2-disk.

The following filetypes have special meanings in CMS and should not be used (unless so intended).

Language Input: ASSEMBLE, MACRO, FORTRAN, FREEFORT, PLI, PLIOPT, BASIC, BASDATA, COBOL

Data input and output: FTnnF001, LISTING, MEMO, TESTFORT

Special system files: MACLIB MAP, TEXT, TXTLIB, MODULE, SYSUTn, CMSUT1

Special application files: DATARUN, DATAMAC, MPFILE, SMODULE, TMODULE (the TEXTAG system)

Miscellaneous: ASM3705, AUXxxxx, CNTRL, COPY, DIRECT, LKEDIT, LOADLIB SCRIPT, SYNONYM, UPDATE, VSBASIC, VSDATA.

DBFILE should never be used except for renaming or copying.

EXEC filetype is a special case. These files are the CMS equivalent of TEXTAG's program files and macros. Use them only if you know what you are doing. However, they soon become almost a necessity if one does much work with CMS.

APPENDIX E: Use of the LOAD Command

If machine-language subroutines are to be used, they must be loaded with the first TEXTAG statement executed at the operational level. The command used is called LOAD and takes the statement structure

LOAD $subrtn_1$[,$subrtn_2$,...]

The subroutine names must be literals and refer either to TEXT files on an accessible CMS disk or to members of a TXTLIB (text library) which has been declared at CMS level. A maximum of 19 subroutines may be listed but this number should probably never be approached.

If any of these subroutines refer to other subroutines, the latter will be loaded and linked automatically, subject to the same accessibility rules as the ones named. (The indirectly loaded subroutines cannot be called directly with TEXTAG statements.) If routines from a large integrated library are specified, they may cause indirect loading of a large number of subroutines which may exceed available space or leave insufficient room for TEXTAG to operate. (Order of specifying routines can also be critical in case a named routine is itself indirectly referenced. Lowest levels should be specified first.)

Routines specified by LOAD may be used either as functions, with C:subrtn(args) expressions, or with the CALL verb which has standard FORTRAN format. Arguments may be any single-valued TEXTAG referents or TEXTAG arrays identified by the prefixes L:, V: or M:. Tables cannot be passed as entities. Elements of tables may be passed.

TEXT files are created by the assembler or one of the compilers available in CMS. Note that this use of the word "TEXT" is misleading; it is not text in the ordinary sense but actually executable machine code. It is usually better to put TEXT files in a "text library" (TXTLIB) which can be declared with the CMS command GLOBAL which controls order of search. Consult CMS documentation or the account manager for further details on this.

APPENDIX F: Restructuring an Old Data Base

Remodelling an old structure is frequently more difficult than building a new one. This is certainly true for data banks. Computerized files have one great advantage, however; they can be easily duplicated. So the first thing to do in restructuring an old data bank is to copy it and then work from the copy. This provides two safeguards:

A. The old data bank can still be used while the new one is being created.

B. If a fatal error is made, all is not lost. One can start again from a fresh copy.

Next, one should use the PRUNE command to get rid of as much of the old structure as necessary and desirable. If the pruning is extensive, this will leave a great deal of dead space in the file which will impair its efficiency of use. If this is the case, or when any data bank file has accumulated too much dead space, the following command should be used at the interface level:

```
COMPRESS  oldfile TO newfile
```

This will create a new version with no dead space. This is for DATABANK files only and both must have been DECLAREd.

COMPRESS does not apply to ARRAYS or ANNEX files (or any other kind but data banks). To compress an ARRAYS file, the following commands can be used at the operational level:

```
SET  ARRAYS = oldfile
RECALL ALL [.LIST]
SET  ARRAYS = newfile
ENFILE ALL [,LIST]
```

The files must have been declared, of course.

In order to remove annexes from an ANNEX file, the command PURGE is used at the interface level. The format is

```
PURGE  oldfile  TO  newfile
```

(after declaring the files). This causes the name of each annex to be displayed at the terminal followed by a question, viz.,

annex-name PURGE? (Y/N)

Any response but Y keeps the annex. A Y response causes it to be purged. If another annex is linked to it and is to be retained, the link will be broken and a message to this effect displayed at the terminal. New annexes can be added at the end with normal use of READ.

Returning to a data bank file proper, suppose it has been copied, pruned and compressed. One can now use the commands DUMP and IDENTIFY at the operational level to create the scratch files described in section 9.4 if this is appropriate. (This may have been done while pruning, depending on circumstances.) These two files can be used to create a new data base or the pruned original file can be used with the dump file to put substructures back in different places. Sometimes one may have to iterate on this process to achieve the final structure desired.

The IDENTIFY file, while formally a data bank, is mostly useful for reviewing old relationships. The FIND and SEARCH commands can be used in normal fashion for this. It is also frequently useful for preparing reports, that is, it is not treated as a restructuring but a subset of a larger data bank.

APPENDIX G: CMS Commands Executable from TEXTAG

A number of CMS commands may be executed from the operational level of TEXTAG with the pre-command CMS, e.g.

CMS cms-command arguments

The available commands, not to be confused with TEXTAG commands of the same name in some cases, are:

ACCESS — access a disk already linked

DEBUG — enter debug environment; generally not useful

ERASE — erase a file; be careful with this

EXEC — execute a CMS program ft=EXEC; may be useful but can be dangerous

FILEDEF — equivalent to an OS DD-card; needed for FORTRAN but unlikely in TEXTAG

GLOBAL — declare one or more TXTLIBs; logically impossible of use in TEXTAG

LISTFILE — lists (i.e., shows existence and other info) of one, several or all files of various types on various disks; often highly useful

PRINT — prints a CMS (printable) file off-line; sometimes useful

PUNCH — punches a CMS file; can be useful if one knows how to use but be careful

QUERY — obtain info on CMS and some CP system status; very useful if one is familiar with the system

READCARD — reads a file from virtual card reader (spooled); sometimes very useful after a transmission of data from some source but usually this requires editing with CMS Editor

RELEASE — releases a device; might be needed sometime but requires knowledge of CP and current status

RENAME — rename a CMS file; discussed in Appendix D and may be useful but use with care

SET, STATE, SVCTRACE, SYNONYM, TAPPDS; these require extensive knowledge of CMS and are unlikely to find a

use within TEXTAG environment

TAPE used for archiving and transmitting on tape but should not be attempted within TEXTAG except under very unusual circurstances

TYPE type all or part of a (printable) CMS file; sometimes very useful

The CMS command CP is also available, e.g.

CMS CP cp-command arguments

This makes the entire range of CP commands available. The CP level is always available anyway, however, by pushing the interrupt button.

All arguments except symbols must be enclosed in single quotes when using the CMS command in TEXTAG. This is necessary to prevent the TEXTAG scanning routine from converting them before passing to CMS or CP. For example, to type the first five line of file MYDATA (ft=DATA, fm=A1), the following is necessary

CMS TYPE MYDATA DATA A1 '1' '5'

since CMS expects the numbers in character form.

Immediate and Emergency Commands

There is a set of commands somewhere intermediate between CP and CMS called "immediate" commands. To get at these commands, push the interrupt key once to get to CP level, and then once again. The useful commands are:

HT halt typeing at the terminal. Notning else if effected. The command may not appear to work because of several lines being already stacked for output but eventually it will stop typing.

RT resume typing at the terminal.

HX halt execution. The TEXTAG session is abruptly terminated and control reverts to CMS. Files may be left in partially updated state.

If HX does not work for some reason, the following can be done. Push the interrupt key to get to CP and then type I CMS (initial load of CMS). This kills the entire session and leaves one in the same state as at log-in except that any linked disks remain

linked (but not accessed). This is the most drastic action possible and should only be used in an emergency.

Sometimes line noise causes CP to be entered, as though the interrupt button had been pushed. To resume execution, type B (begin).

One other emergency action is possible but may not always be effective. If in some kind of loop in a TEXTAG program, push interrupt and at CP level type EXT (external interrupt). This has approximately the same effect as a QUIT command in the deck. This interrupt is also checked in the File Management Program in case of a long loop in one instruction. Control returns to the terminal after the program file (if any) is closed.