SHARE PROGRAM LIBRARY AGENCY

PROGRAM NUMBER

068003

University of Miami

1365 MEMORIAL DRIVE - CORAL GABLES, FLORIDA
(305) - 284-6257

SHARE Program Library Agency
Triangle Universities Computation Center
P. O. Box 12076
Research Triangle Park, N. C. 27709

CONTRIBUTED PROGRAM LIBRARY SUBMITTAL FOR
(for IBM S/360, 1130 and 1800)

This form should be completed and submitted with the program package to PID at the address shown above. Standards and instructions for submitting programs are in your *User Group Reference Manual* or the *Contributed Program Submittal Standards Manual* available from PID.

(1) **Program Order Number** *(to be filled in by PID)* . . . . . ~~360D-06.8.003~~

(2) **System Type** *(machine)* S/360

(3) **Search Key** . . . . . 360 DATA STRUCTURES PROGRAMMING COMPILER AND PAGING SYSTEM

(4) **Programming Language** . . . . . 360 OS ALF

(5) **Author's Name and Address** . . . .

Professor F. W. Tompa
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

(6) **Direct Inquiries to Name and Address** *(if different than Author)*

(7) **Title of Program** . . . . The Data Structures Programming System

DISCLAIMER

Triangle Universities Computation Center (TUCC) serves solely as the distribution agent for contributed programs and does not test or maintain them. They are distributed essentially in the original form submitted by the author. Neither TUCC nor SHARE, INC., makes any warranty, expressed or implied, as to the documentation, function, or performance of the contributed programs.

(8) **Submitter's User Group Affiliation Code and Installation Code** . . . . . S BUC

(9) **Submitter's Own Program Identification and Suffix** *(optional)* . . . . .

(10) **Primary Subject Code** 06.8

(11) **Secondary Subject Codes** 03.2 01.6 03.8

(12) **Operating or Monitor System Required** . . . . . See Abstract

(13) **New or Revision Code** *(if revision, show prior Program Order Number in item 1)* . . . . . N

(14) **Year Completed** 69

(15) **Date of Submittal** 09 29 69

(16) **Documentation** *(number of original pages submitted)* 59

(17) **Abstract** *(should contain sufficient information for a reader to determine the value of the program). Listed on the reverse side of this form are subjects which may serve as a guide for a descriptive abstract.*

CONTRIBUTED PROGRAM LIBRARY SUBMITTAL FORM

Subject Guide

a. Purpose
b. Programming Language used
c. Version and modification level or release number of IBM Programming System used, or program order number for non-IBM authored program used
d. Field of application
e. Type of routine (main program, subroutine, etc.)
f. Specific description of machine requirements
g. Engineering Changes (EC) level of equipment (if pertinent)

ABSTRACT

The Data Structures Programming System (DSPS) allows a user to build and manipulate complex data (list) structures. The structures, written in the Data Structures Programming Language (DSPL) and translated into Assembler Language by a compiler, are designed completely by the user in order that they may best fit his particular application. The run-time paging component permits the structure to be arbitrarily large while imposing only minimal manipulation handicaps on the user.

Users of DSPS include those who need to design powerful data structures optimized for run-time speed. DSPL is completely compatible with Assembler Language in order to permit a user to conveniently intermix simple arithmetics, shifts, etc. for manipulating data.

DSPS was written for a System/360 Model 50 or higher, using 2314 or 2311 direct access devices (disks) for secondary storage. It has been run under releases 14, 15/16, and 17 of the 360 Operating System, using both MVT and MFT. The program is written in System/360 Assembler Language, and has been tested under IBM's Assembler F and Waterloo's Assembler G.

(Please attach additional pages if necessary) . . . . . **Total pages attached** _____

Permission to Publish

"I hereby give anyone permission to reprint, reproduce, and distribute this program to anyone else."

(18) **Signature of Submitter and Date** Frank Wm. Tompa 9/29/69

(19) **Signature of Installation Addressee** Roger J. Jette

T4SF

TABLE OF CONTENTS

LIST OF FIGURES

TAPE KEY

This volume contains three files and nine tape marks. All files are EBCDIC card image with block length 800 (10 cards/record). The arrangement of the tape is as follows:

STANDARD LABEL: VOLUME=SER=BUDSPS. 1 record; length 80.

HEADER: 2 records; length 80; last record length is 80.

T/M

FILE 1: Symbolics (source) for the DSPL compiler. 7567 cards; 757 records; last record length is 560.

T/M

TRAILER: 2 records; length 80; last record length is 80.

T/M

HEADER: 2 records; length 80; last record length is 80.

T/M

FILE 2: Symbolics for the paging component of DSPS. 2206 cards; 221 records; last record length is 480.

T/M

TRAILER: 2 records; length 80; last record length is 80.

T/M

HEADER: 2 records; length 80; last record length is 80.

T/M

FILE 3: Symbolics for the sample program for DSPS. 690 cards; 69 records; last record length is 800.

T/M

TRAILER: 2 records; length 80; last record length is 80.

T/M

Refer to "13 Job Control for DSPS" for information on unloading the tape, assembling the compiler and the paging component , entering the assembled decks into a load module library, and running the programs. These files can also be punched or listed using the IEBPTPCH utility as described in "IBM System/360 Operating System Utilities".

FILE NO. DSPS360-1

FORM FWT-6909-0

A MANUAL FOR THE DATA STRUCTURES PROGRAMMING SYSTEM FOR THE IBM SYSTEM/360

FRANK WM. TOMPA

CENTER FOR
COMPUTER & INFORMATION SCIENCES
182 GEORGE STREET
PROVIDENCE, RHODE ISLAND
02912

1 HISTORY AND ACKNOWLEDGEMENTS

The need for a low-level language in which to write data structure programs efficiently and easily is common to many applications including computer graphics. For this reason, a version of Carnegie-Mellon University's *1 macro processor was obtained by Brown University. The *1 language is a System/360 adaptation of Bell Laboratory's Low Level Linked List Language (L^6) which was developed by K. Knowlton [3]. Since size limitations imposed by core boundaries are too restricting for extensive amounts of data, work was simultaneously begun on a software paging mechanism to interface with *1. After using and debugging the macro processor for almost a year, it was decided that it was too slow to be practical for long programs, and thus compiling would be much more economical.[1]

The Data Structures Programming System (DSPS) was thus begun to combine a paging system (with hooks for record (information) retrieval) and a compiler for a *1-like language, the Data Structures Programming Language (DSPL). (In summary, then, one might say that since *1 is a modified L^6 with "blocks" in addition to L^6's own "fields", DSPS is paged, compiled L^6 with blocks.)

The research and writing of DSPS was under the supervision of Professor Andries van Dam and David Evans. The compiler was designed by Robert Sedgewick and Robert Letner. The implementation which they started was completed by Frank Tompa. The paging component was designed and implemented by George Stabler, with some programming assistance from John Gannon.

This manual is loosely based on Evans and van Dam's paper [1] and on Carnegie-Mellon's *1 manual [4]. Sections were rewritten from George Stabler's notes on the paging system. Other writing assistance was provided by Professor van Dam, Richard Bergeron, and John Gannon. The documentation was composed, edited, and printed using Brown University's Hypertext Editing System.

Any questions or suggestions concerning the Data Structures Programming System should be directed to the author.

---

1. Brown's language is processed by a compiler in order to shorten assembly time to one quarter of what would be used by Waterloo's Assembler G (i.e., one twentieth the time used by I.B.M.'s Assembler F).

2 INTRODUCTION

The need for compound data structures capable of representing complex interrelationships and associations between entities arises typically in computer graphics or file management areas. J. C. Gray [2], in a recent survey, categorizes a number of the more popular graphics data structures and points out that complex ring structures such as those of Sketchpad and CORAL have been used most often. Gray and A. van Dam and D. Evans [6] point out, however, that by trading off space for processing time, less complex data structures may be built which still incorporate graphical as well as applications data for hierarchical models. The design of the Data Structures Programming System described below was based on the premise that any search for an "ultimate" data structure which optimizes storage and time for a large range of dissimilar applications is, in fact, fruitless. Thus the Data Structures Programming System lets an applications programmer build a "customized" data structure which need not carry along the pointer structure overhead of a more general design. Designing and debugging data structures can be done quickly, as can the writing of a library of programs to build standard data structures, such as rings or simple lists, much as CORAL primitives can be used to build CORAL rings.

The decision to use a "low-level" Data Structures Programming Language in the system, as opposed to embedding primitives in a higher level language such as PL/I or FORTRAN was made to provide the most efficient object code and storage use, and the greatest flexibility in designing data structures. (The run-time environment does not contain any basic list manipulation routines, but is restricted to only those functions which directly involve paging.) Thus DSPL, while oriented towards designating data structures, is based on the same premises as the more recent crop of systems programming languages [5,7]; it is close to machine language (an expansion of five to eight machine instructions per DSPL command is typical), and yet as machine-independent as FORTRAN or PL/I. The DSPL compiler is completely compatible with System/360 Assembly Language and can easily be used to build and manipulate arrays, lists, stacks, symbol tables, garbage collectors, and standard list processors (e.g., for instructional use).

While most data structures operate strictly in core, realistically large entities and entity complexes involve the use of auxiliary storage such as disks or drums. This is especially true in computer graphics, in which a single picture may be "worth a thousand words" just for its graphical representation, in addition to which storage is required for all the applications information necessary to process the picture. The programmer should not have to worry about this problem; the system should provide the necessary swapping between core and external storage, and let

him program in a large "virtual" memory (or at least in logical units of arbitrary size).

In the Data Structures Programming System (DSPS), written for a non-hardware paged machine, the user constructs entities without being aware of core boundaries, and his DSPL-coded processes are executed on paged entities which are already in core, or are brought into core as needed[2]. The word "page" used in this context has a somewhat non-standard meaning. Instead of being merely a fixed number of characters or words, a page here is a segment or logical unit whose extent is partially described by the programmer himself and partially controlled by the system. Because of the techniques used for swapping the entities (pages) between high-speed core and external bulk storage, however, the term "paging" is not entirely without some of its standard meaning.

DSPS has in it the possibility of interfacing with a record (information) retrieval system. A page could be given a name or Boolean function of keywords and attribute-value pairs at the time of its creation, in order that it could be referenced without an explicit pointer. The information retrieval system used at Brown has never been interfaced with DSPS because of severe core restrictions on all graphics programs, and thus such a system is not included in the package. However the manual does describe the format for a page retrieval by name, as well as the parameters for generating the paging system with an IR interface. [A cursory description of the use of page names as paging references can be found under "5.1 Sequence Evaluation" below.]

DSPS has been used successfully for several applications at Brown. All the users required a great deal of list processing with some computations. (In some cases only sections of the system which were applicable to the particular user's requirements were used.) The major application was a piping design system which uses the data structure manipulation facilities to manage the abundant supply of computer graphics needed to construct and manipulate three-dimensional, (stereo perspective) displays of piping networks on an I.B.M. 2250 graphics console. The resulting package is a mixture of FORTRAN stress analysis subroutines, GPAK graphic subroutines, and DSPS data structure routines.

DSPS was also used to write run-time graphics data structures subroutines to replace those of I.B.M.'s Problem Language Analyzer (PLAN). The paging system for the program had to be altered slightly to allow a user to have 128K pages, but other than that, programming was confirmed to be

---

2. Originally a "look-ahead" mechanism was designed by which every page which is referenced by a pointer on the page under consideration would be brought into core before it was actually required by the user's program. However this was found to be impractical as considerable I/O time would be utilized on pages which were not needed, and bumping strategies became too complex and costly.

much easier in DSPL than in some other languages, with more efficient code produced than in the previous FORTRAN implementation.

A flow-charting program, using the 2250 for input as well as output, was written using just the paging component of DSPS, as the compiler had not yet been completed. Another user implemented a "virtual sketchpad" which used an earlier version of paged DSPS to manage the data structures for the graphical display part of the program.

DSPS has also been used for instructional purposes in undergraduate and graduate courses at Brown University, the University of Michigan, and Washington State University.

A program which illustrates the properties of DSPS is included on the tape along with this manual [see "13 Job Control for DSPS"]. The example is an implementation of an information retrieval system, which uses data cards originally intended for a KWIC (Key Word In Context) sorting program to set up a data base for subsequent information retrieval. A second pass through the sample program, with an appropriate change of data, prints out all the titles of works, with their respective authors and keywords, satisfying each request item. The example is rather extensive, and therefore it should be consulted in order to understand the concepts and constructs presented in this manual.

The remainder of this paper describes the two parts of the Data Structures System, namely the language (DSPL) and the paging mechanism.

3 DATA STRUCTURES

DSPL provides basic data structure elements which may be compounded into arbitrarily complex high-level data structures, as well as the facilities to manipulate the data and pointers. Data within these structures is referenced by indexing or by chains of pointers which lead to desired storage locations. The basic elements in DSPL are fields, which are analagous to pointers, and blocks, which represent data areas. A concatenation of fields and blocks defines a unique pointer sequence through an arbitrary data structure. All declarations cause compilation only of system (compile-time) symbol tables, not run-time code.

3.1 THE FIELD

The basic type of data element in DSPL is the field. Although the primary use of the field is to contain a pointer, it may also be used for storing data such as flags. Several formats of fields are available to the DSPL user.

The base field (BFIELD) is a full word in core. It is declared in the following format:

```
BFIELD    name[,loc][3]
```

The name is the one which will be used in any DSPL statement in which this BFIELD is referenced. The location is an assembler label defining the word to be reserved for the BFIELD.

In order to increase the efficiency of the code produced, a user may define a BFIELD to be located in a register by the following command:

```
BFIELDR   name,reg
```

The user should be aware that the compiler uses several registers in its generated code, and thus he should make sure that enough registers are available (i.e., not too many are used as BFIELDRs).

---

3. The macro-like syntax of DSPL is inherited from *1 in order to keep the compiler compatible to programs already written for the macro processor. Note that brackets in a syntax description indicate an optional parameter; braces indicate a choice of parameters.

As opposed to a base field, a FIELD is a template-like structure which can be used at any and all addresses in core. The field is a "mask" of contiguous bits within one full word which is referenced at run-time in relation to some address specified explicitely or by a pointer. That is, the FIELD is a structure which exists at a given displacement from any address specified by the user at run-time [see "4 Implied Outer Block"].

The declaration of a FIELD is as follows:

```
FIELD    name,disp,(lowbit,highbit)
```

Again the name is the one by which this FIELD is referenced. However, the location is given in terms of a displacement from some address to be specified at run-time rather than an absolute address in core. Furthermore the extent of the FIELD is given as the lowbit (leftmost) to the highbit (rightmost) inclusive.

Thus, whereas the declaration

```
BFIELD   D,PLACE
```

defines a base field named D to be located at the full word starting at PLACE, the declaration

```
FIELD    C,3,(0,13)
```

defines a masklike structure named C to be located in the first fourteen bits of the full word located at a displacement of three words from some specified address in core. As a particular example, the FIELD C can specify either of the units of memory diagrammed in Figure 1. If the address 1324 is specified [see "4 Implied Outer Block"], C will be located as in the example on the left. On the other hand, if FF038 is specified, the diagram on the right is applicable.

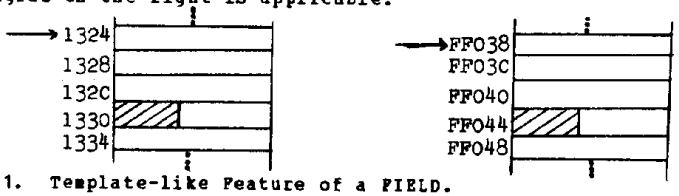

Figure 1. Template-like Feature of a FIELD.

3.2 THE BLOCK

The basic element for storing data is the block, which is called an "n-component element" or a "bead" in some other systems. A block is

defined to be a set of contiguous words which are to be treated as a unit. Again two basic types of blocks are available to the DSPL user.

A base block (BBLOCK) is located at a specified address in core. The declaration is similar to that of a BFIELD in that a name and a location must be defined, but in addition a user must specify the number of words in the BBLOCK.

```
BBLOCK   name,(loc,size)
```

As in the case of the difference between a FIELD and a BFIELD, a BLOCK differs from a BBLOCK in that it is declared as being at a displacement from some explicit or implicit (i.e., specified through the use of a pointer) location specified at run time. The declaration of a BLOCK is as follows:

```
BLOCK    name,(disp,size)
```

Whereas

```
BBLOCK   B,(LOC,8)
```

declares a base block B extending for eight words (thirty two bytes) from the full word aligned assembler label LOC,

```
BLOCK    C,(4,40)
```

declares a block named C to extend for forty words beginning at the fourth full word past any location to be specified at run-time.

3.3 PAGING STRATEGIES AND THE PAGING REFERENCE

A page is a logical unit, i.e., a self-contained segment of the user's global data structure. It can be of arbitrary size (up to 8000 words) and is used, for instance, to store the entire data structure for a single picture. Thus it is a collection of blocks and fields, some of which may contain pointers to other pages.

When a paging system was first planned at Brown, the philosophy was to permit the user to program in an environment which closely resembled a "virtual memory", letting him use the entire address space at will. Naturally, space was to be divided into fixed-length system pages, invisible to the user. A pointer from such a current, system page to an area on another page was to be trapped by an operating system addressing exception to check for in-core residency of the referenced page and to call a routine to do appropriate page-to-page linking. That is, this trap would lead to an invocation (via a "SPIE" error handling routine) of the

DSPS paging mechanism. However it was soon realized that since the user would have no control over the location of his various data items (i.e., the system would locate the items at the most convenient place at the time of the items' creation), any pointer could require such an exception. In fact, in the absence of any segmenting information supplied by the user, every pointer could conceivably cross page boundaries.

For example, if a ring of items would be expanded dynamically (i.e., more elements added during the execution of the program), it might have each of its items on a separate page since the creation of other data items would use up space on the pages already in use. Thus each pointer to the next element of the ring would cross page boundaries, rather than the whole ring being on one page. Whenever the ring would be searched for a particular item, the overhead of involving the operating system to catch addressing exceptions would be highly impractical, as every pointer would require an interrupt.

Then it was realized that if the user himself segmented his virtual memory into self-contained logical units (called user pages) which could be made as large as necessary, (e.g., to contain the complete data structure for an entire picture), he could control the location of his data on those pages. Furthermore, even though they could be arbitrarily large (up to sixteen bits of addressing), such pages would be very small compared to a twenty or twenty-four bit virtual memory address space; a user page would therefore not need to be further subdivided into systems pages, since the direct access I/O methods could handle it as one unit. Thus the ratio of pointers which might cross user page boundaries (e.g., those which refer to sub-pictures) to those used internal to a page could be made to be quite small. As a result, it was decided to have the user himself declare those relatively few special-purpose pointers which might be used to cross page boundaries (i.e., which might reference areas located on other pages) rather than having any pointer be a candidate for external referencing.

Such a specially declared pointer, called a paging pointer (or paging reference) must provide the disk address and the length of the referenced page in addition to the desired area from the top of that page. However since the paging reference is declared to be a pointer, it may not be used to contain data in the way that FIELDS may be used to contain flags. Furthermore, a paging reference is always two full words long in order to contain the information necessary for the paging component of DSPS. (In order to provide the user with efficient code, the compiler does not generate any checks for a valid pointer. If either a FIELD or a paging reference is used as a pointer, and at run-time it does not contain valid information for calculating an address, the operating system will terminate the program with an appropriate completion code [see "11.3 Debugging"].)

The format of a paging pointer is as pictured in Figure 2. The first half-word, a pointer to the next paging reference on the current page, is used for compacting and updating pages off-line. The second half-word is the displacement on a (possibly the same) page of the particular field or block to which the pointer refers. The second full-word is used for the disk address[4] of that page and its length.

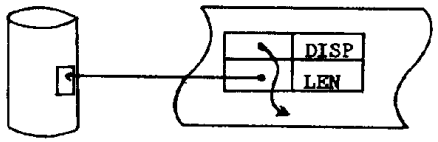

Figure 2. A Paging Reference

Whenever such a pointer is used in a DSPL statement, the compiler generates code to call the paging subsystem which handles the pointer and its requisite paging and linking appropriately, transparently to the user.

At run-time, the paging system thus first checks whether the pointer is being used internally (i.e., whether it points to an area located on the same page on which the pointer itself resides). If not, it finds the Page Control Block (PCB), a special page which serves as a "handle" for all other pages in core. The paging system then uses the Page Control Links (PCLs)[5] to search through the pages in core until the specified one is found, or until the search again reaches the PCB.

If the page specified is not in core, the paging system retrieves it from disk using the disk address and length specified by the second word of the pointer. In the case that there is no room in core for the newly referenced page, pages are swapped to disk (starting with the least recently used page) until enough room is available. In either case, the page is re-linked to the top of the lists, and the data is accessed as needed.

---

4. See "9 File Management and Page Updates" for a discussion of the trade-offs between absolute disk addresses and symbolic names or keys for page locations.
5. All the PCLs, which are located in the six words preceeding every page in core, are threaded by two circular linked lists (rings) which start with the PCB. The first of these rings links the pages from the most recently referenced (at run-time) to the least recently referenced page; the second, from the least recent to the most recent.

As in the case of the BFIELD and the FIELD, two types of paging references may be declared.

```
BFIELDX name[,loc]
```

This declaration states that the two words starting at loc will be used as a paging base field and referred to as name.

```
FIELDX name,disp
```

This second declaration states that the two full words located at a displacement disp from some point to be specified at run-time can be referred to as a paging pointer called name. Note that this extension of the definition of a field as a paging field does not define the extent to be a set of contiguious bits within a full word, but rather is always two full words long; thus the first-bit and last-bit are not specified.

Storage is defined in the user's program for all base fields and base blocks declared. That is, the compiler will generate appropriate storage and any needed labels for all those declarations which have a location identical to its name. If the user wishes to define his own storage (e.g., in order to initialize an item), he may, by declaring a location different from the name. [Note that in the declaration of a BFIELD or a BFIELDX, if the location is omitted, it is assumed to be identical to the name, and thus the compiler will generate an appropriate DC.]

On the other hand the declaration

```
BFIELD    SMALL,TINY
```

will cause the compiler to generate the statement

```
SMALL    EQU      TINY
```

and thus assume that the label TINY has been defined by the user.

3.4 CONVENIENCE OF DECLARATIONS

One of the advantages that DSPL has over other systems in which list-processing can be coded is that list structures can be altered very easily from one run to the next. This is accomplished by changing appropriate field or block definitions or by adding new ones for further versatility. One point to bear in mind is that, since as many as fifty of each type of definition is allowed in one assembly, it may be convenient to declare additional field or block definitions in order to use mnemonic names to the best advantage. That is, a user may want to declare both

```
FIELD LENG,2,(8,31)
```

and

```
FIELD FORWRD,2,(8,31)
```

for use within one program or even within one structure.

4 IMPLIED OUTER BLOCK

As opposed to base blocks and base fields, BLOCKs, FIELDs, and FIELDXs are not specific sections of memory, and therefore storage is not defined for these items. Instead, when one of these is referenced at run-time, some address must be specified as the beginning of an "implied outer block". This implied outer block can be imagined as an arbitrarily large, global, rectangular (one word wide) template, whose upper left corner is placed at any address specified at run-time. All the FIELDs, FIELDXs, and BLOCKs declared in the user's program would then be cut-outs (masks) in the template. Thus any one of them could be accessed by the user, no matter where the template is located, and irrespective of whether such an access would make sense.

The start of the implied outer block is determined by the last data structure item referenced by the user. If the last item was a BLOCK or a BBLOCK, the implied outer block begins at the same word as that block did; if it was a FIELD, BFIELD, BFIELDB, FIELDX, or BFIELDX, the implied outer block is located starting at the address specified by the contents of that field or base field. In either case, the implied outer block is of arbitrary length.

[For examples of the code generated by determinations of successive implied outer blocks, see "5.1 Sequence Evaluation".]

The concept of a template-like structure is not unique to DSPL. System/360 Assembler Language permits a user to have a "block" through the use of a DSECT. The format of a DSECT is fixed at assembly time by the declarations following the DSECT statement. The DSECT is located at run-time by the contents of a general register. The DSECT may be located anywhere in core, and may be moved during execution, according to the contents of that register. Similarly, PL/I provides the "based variable" and the "based structure" which have fixed formats at compile time, but which can be located at any arbitrary spot in core by the contents of some "pointer variable" at run-time.

DSPL has several advantages over these two languages. Both permit the user to reference only one level of pointers, and thus a linked list cannot be implemented in this way. Furthermore the structure of a DSECT or of a PL/I structure cannot be changed conveniently at run-time as is the case with DSPL, which provides facilities for specifying the displacement of a part of a structure from the top of the structure at run-time [see "5.3 Dynamic Origins and Lengths"]. Finally neither PL/I nor DSECTS provides a simple, built-in communication language for using a paging system.

5 SEQUENCES

A sequence is a chain of block and field names which ultimately leads to a particular block or field. The first (left-most) name in the sequence and only the first must be a base field[6] or base block name. After the evaluation of each item in the sequence, an implied outer block is established for determining the location of the next item. If a field is to be used as a pointer and it's extent is less than that needed for a full address, it is extended with 0's on the left.

5.1 SEQUENCE EVALUATION

The value of a sequence is calculated in the following way:

(1) First Name:
   a) If the sequence starts with a base field, we access (set a pointer to) the implied block to which it points.
   b) If it starts with a base block, we access that block.

(2) Middle Names:
   a) If the next name is of a field, we access that field within the outer block and then access the implied block pointed to by the field.
   b) If the next name is a block name, we access that sub-block of the previously accessed block.

In either case we now have a block again and step (2) is repeated until the last field or block name is encountered.

(3) Final name:
   a) In the case that a field is last, we do not access its implied block, but rather access that field itself. This is called a "field sequence".
   b) If the last name is that of a block (i.e., for a "block sequence"), we access this sub-block as before.

---

6. Unless otherwise specified, the generic term base field will be used for a BFIELD, BFIELDB, or BFIELDX. Similarly, a field will refer to a FIELD or a FIELDX.

Thus by using a field sequence the user obtains a field within some explicit or implied outer block. On the other hand, by using a block sequence, the user obtains a sub-block of an outer block.

Examples:

```
BFIELD    W
BBLOCK    C,(LOC,5)
FIELD     A,1,(0,15)
FIELD     Z1,0,(21,26)
BLOCK     BIG,(2,4)
BLOCK     Q,(2,2)
```

To calculate the block sequence WA(BIG)Q, DSPL goes through the following chain:

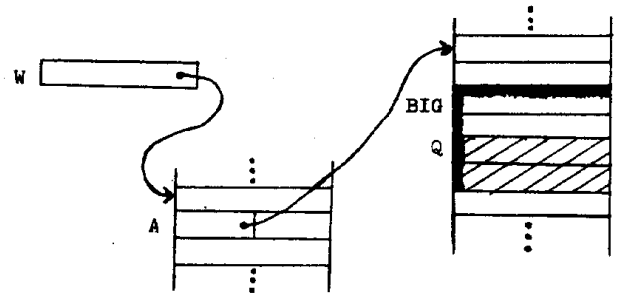

Figure 3. Sequence Evaluation
a) Block Sequence

The code generated to find the origin of the block might be

```
L     3,W              Evaluate the BFIELD W
LH    3,4(00,03)       Get the contents of the FIELD A
LA    3,16(00,03)      Add on the displacements of the BLOCKs BIG and Q
```


To calculate the field sequence CAQZ1, DSPL processes this chain:

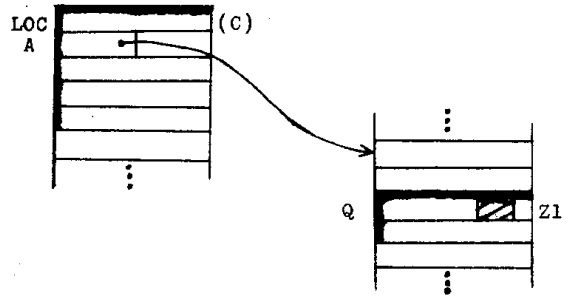

Figure 3. Sequence Evaluation
b) Field Sequence

In order to access the contents of Z1, the following steps would be executed

```
LA    2,C              Get the address of the BBLOCK C
LH    2,4(00,02)       Pick off the FIELD A (length in words)
L     2,8(00,02)       Get the FIELD Z1 after the BLOCK Q
N     2,=X'000007E0'   Isolate the field
SRL   2,5              Right-justify the FIELD Z1
```

Note that one may concatenate single letter names (or single letters followed by digit strings) to form sequences, without punctuation. Other names, however, require parentheses around them.

5.2 PAGING SEQUENCES

A paging sequence (xref) is a field or block sequence which starts with a BFIELDX and/or has one or more FIELDXs imbedded in the sequence. All paging references must point at some area on a page; that is, no BFIELDX or FIELDX may point at any resident, in-core area. Note that a paging pointer may refer to an area on any page, whether that area is internal or external to the current page.

As opposed to a FIELDX, the contents of a FIELD on some page may be either data or a pointer to an area on that same page. In the latter case, the FIELD contains the relative displacement of the area from the top of the page. Needless to say, when a sequence which contains such a FIELD is evaluated, the appropriate absolute address is calculated by the code generated by the compiler.

Following some definitions are some examples of paging sequences:

```
BFIELDX   X
FIELD     F,3,(0,31)
FIELDX    G,2
FIELDX    FX,8
BBLOCK    K,(K,8)
BLOCK     L,(2,4)
```

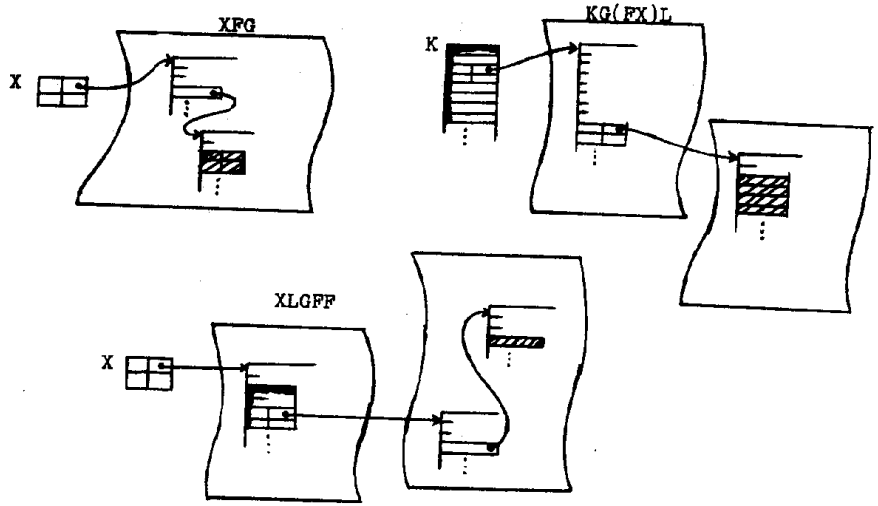

Figure 4. Paging Sequence Evaluation
a) Use of Paging Reference Externally

But G could also be used internally:

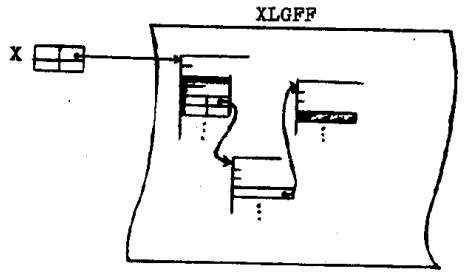

Figure 4. Paging Sequence Evaluation
b) Use of Paging Reference Internally

If an interface with a page retrieval system has been made, the compiler must recognize a page name by which the page can be retrieved. This is done by having a sequence start with the name of a page -- instead of a BFIELDX pointing at the top of that page. A page name is indicated to the compiler by enclosing it within slashes: /name/ .

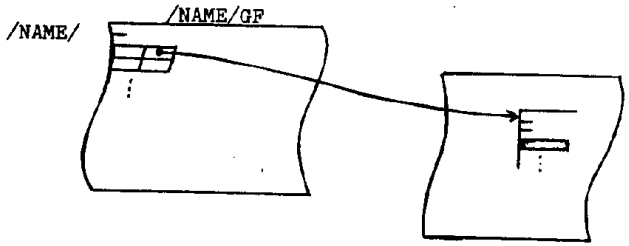

Figure 4. Paging Sequence Evaluation
c) Use of Page Names for Information Retrieval

Instead of specifying a name between slashes, a field sequence pointing at the name of some page may be used. Thus DSPS provides the facility of specifying at run-time which page is to be accessed.

5.3 DYNAMIC ORIGINS AND LENGTHS

The origin of a block or field and the size of a block or base block may be specified either statically or dynamically. If an origin is specified by a number (as in all the examples so far), it is static, the value being fixed for the assembly. Thus, as in Figure 1., a field declared to begin at a displacement of three words from an implied outer block will be located at that displacement whenever it is referenced. On the other hand, if it is specified by a field sequence, the contents of the field accessed by that sequence is interpreted as the origin (in words) of the block or field under consideration. If B and F are declared by the following statements,

```
BFIELD    B
FIELD     F,B,(0,31)
```

then the origin of the FIELD F is determined by the run-time contents of the BFIELD B each time F is referenced. Thus at a given evaluation of F, if B contains 5 and the implied outer block is at 100, then F can be represented by the diagram on the left of Figure 5. If B contains 1, then F is as pictured on the right.

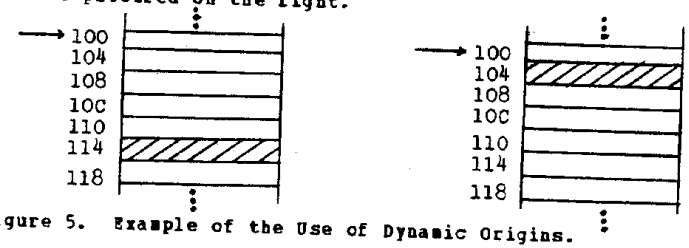

Figure 5. Example of the Use of Dynamic Origins.

Similarly, the length of a BLOCK or of a BBLOCK may be specified by a field sequence, as in the following example.

```
BFIELD    B
BLOCK     K,(3,B)
```

In this way, the sizes and locations of structure items can be varied as the program is executed by merely changing the contents of some field.

Whereas the static declaration is used for a fixed data structure element, the dynamic origin is very practical for branch tables (transfer vectors), displacements into arrays, etc. [see "11.1 Use of the Dynamic Origin"].

Consider

```
BBLOCK    C,(C,3)
BFIELD    D
FIELD     ABC,4,(0,31)
BLOCK     B,(C(ABC),D)
```

When BLOCK B is required by the program, the field sequences C(ABC) and D are evaluated, and the contents of C(ABC) are used to define the origin (in words) and the contents of D are used to define the length (in words). To "redefine" B, the user need only change the contents of C(ABC) or D at run-time.

For example, the sequence D(ABC)B(ABC) would generate the following code:

```
    L      2,D            Evaluate the BFIELD D
    L      2,16(00,02)    Evaluate the Sequence D(ABC)
    LA     3,C            Evaluate the dynamic origin-BBLOCK C
    L      3,16(00,03)                              -Sequence C(ABC)
    SLL    3,2
*                         Multiply by 4 to change dynamic origin
*                         from words to bytes
    L      2,16(03,02)    Evaluate the sequence D(ABC)B(ABC)
```

Neither the origins of base blocks and base fields nor the extent of a field may be specified dynamically. When specifying the length of a base block, a third parameter must be specified to give the maximum length to be allocated by the compiler.

Example: BBLOCK K,(K,ABC),25 where ABC is some field sequence.

Any name used in a dynamic origin or dynamic length must be already declared. For static lengths or static origins, only numbers may be used.

6 DSPL COMMANDS

6.1 OPERATIONS

There are two formats for basic instructions in DSPL. A binary operation is indicated in the following manner:

DO (sequence,binary-operator, { sequence / assembler-expression[7] })

The syntax of a unary operation is as follows:

DO (unary-operator,field-sequence)

The following operations are valid in DSPL:

OPERATOR	MEANING
BINARY OPERATIONS:	
_ (underline)	LOGICAL MOVE OR STORE
_@	STORE POINTER
FIELD OPERATIONS:	
+	ADD
-	SUBTRACT
*	MULTIPLY
/	QUOTIENT AFTER DIVISION
MOD	REMAINDER AFTER DIVISION
%	LOGICAL AND
"	LOGICAL OR
X"	LOGICAL EXCLUSIVE OR
B_D	DECIMAL TO BINARY
D_B	BINARY TO DECIMAL
UNARY OPERATIONS	
¬	LOGICAL COMPLEMENT
##	SHIFT RIGHT
__ (underline)	SHIFT LEFT

---

7. An assembler expression is limited to an assembler constant, an assembler symbol as defined by the System/360 Assembler Language Manual, or a four-byte, full-word aligned literal. Usually these will be decimal numbers or assembler names which have been assigned locations in the assembly.

EXAMPLES:

DO	(ABF,+,4)	Adds 4 to ABF
DO	(ABF,",AZF)	Ors the contents of AZF and ABF into ABF
DO	(¬,ABCFG)	Complements ABCFG
DO	(##3,AB)	Shifts the contents of AB 3 bits to the right

Any sequence used on either side of a field operation must be a field sequence. In addition, the left sequence of a STORE POINTER must be a field sequence.

The STORE operation will store an assembler expression, or the contents of the field or block on the right, into the block or field on the left. The quantity[8] is stored right-justified in a field, and right-justified beginning with the last word in a block. If the field or block into which information is to be stored is larger than the quantity to be stored, the beginning of that area is zeroed. If a quantity is stored into a block or field which is too small, the high-order end is truncated.

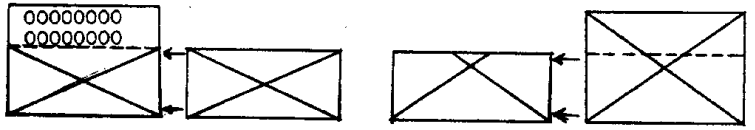

Figure 6. STORE Operation

The STORE POINTER operation makes the left hand side point at the location specified by the right hand side. This operation is used to set up a pointer in a field or base field, whether or not it is a paging reference. It is assumed that the right operand contains the address of a location in memory. When the right operand is a block sequence, it causes the left field to point to the beginning of the block accessed by the sequence. When the right operand is a field sequence, the left field is made to point at the implied outer block to which the right field is pointing.

When a field operation is performed, the field sequence and quantity are combined according to the field operator, and the result replaces the contents of the field specified by the sequence. The arithmetic operations will work correctly for non-negative integers, and for negative integers which are kept in half-word or full-word fields.

The B_D operation converts the contents of the right-hand field from the zoned decimal format to binary format and stores the result in the left-hand field. D_B converts the right field from binary to zoned

---

8. The term "quantity" will be used to mean either the contents of some field or block sequence or an assembler expression.

decimal and stores the result in the left-hand field. [Note that B_D is an operator and is not (B,_,D) which is a store operation.]

For shifts, the number of bits to shift the field may be specified statically (as an assembler expression) or dynamically (as a field sequence). This quantity is specified directly after the operator as:

```
DO      ($$3,ABC)
DO      (__FS,A)
```

Shift operations cause the contents of the field specified by the field sequence to be shifted by the number of places specified by the quantity. Zeroes are supplied when bits are vacated by the shift.

In order to use the contents of a field specified by some sequence, or to use the quantity specified by some assembly expression, the compiler will generate either a LOAD or a LOAD ADDRESS of the appropriate term. That is, the instruction

```
DO      (W,_,expr)
```

will generate either

```
L       reg,expr
```

or

```
LA      reg,expr
```

It is very important to note that if a series of characters specified by a quantity has a valid interpretation as a sequence, then it is interpreted as a sequence. Assembler expressions which might be misconstrued as a sequence must be preceeded by a dollar sign($) for a LOAD or an ampersand(&) for a LOAD ADDRESS. Thus if NAME is an assembler label, and N is a base field and A, M, and P are blocks or fields then

(W,_,NAME) treats NAME as a sequence,

while

(W,_,$NAME) treats it as an assembler label to be loaded and
(W,_,&NAME) treats it as a load-addressable assembler expression.

For non-sequences preceded by neither $ nor & the compiler generates a LOAD of the expression. Note that for a number [such as (W,_,3) or (W,_,51347)], the compiler generates a LOAD ADDRESS if the number is less than 4095; otherwise it generates a load of a full-word literal.

For any DO statement a sequence of operations may be specified in the following manner:

```
DO    (operation1),(operation2),...,(operationn)
```

The operations in the operation sequence are performed unconditionally from left to right.

6.2 TESTS

In almost any data structures program, one has to test various pieces of data for some specific value or in comparison to some other data. DSPL provides a user with the possibility of making a series of such tests and branching according to the results. The basic format is:

$$\left\{\begin{matrix}\text{IF}\\ \text{IFANY}\end{matrix}\right\}\ \text{(test1),(test2),...,(testn),THEN,(operation)}$$

When the operator IF is used, all subsequent tests must be true for control to be passed to the operation following THEN. When IFANY is used, at least one test must yield true for the same transfer of control. [Note for the case of only one test, transfer of control is the same for IF or IFANY.] If the tests cause control not to be passed to the operation, it is instead passed to the next sequential statement of the program.

There are two possible formats for tests:

```
the binary test
    (sequence,binary-test-operator,quantity)
and the logical test
    ([¬]sequence)
```

The binary test checks to see if the contents of a field or block is in some relation to a given value or the contents of some other block or field. In case the areas to be compared are not of the same size, the smaller one is considered as if it were padded on the left with zeroes. This is analogous to the store operation. (That is, after storing one field or block into a larger one, an equality test on them will yield true.) The following binary tests can be made in DSPL:

=	¬=	equals
>	¬>	greater than
<	¬<	less than
=@	¬=@	equals pointer

The check for equals pointer requires that the left side be a field sequence. As in the operation STORE POINTER, if the right hand side is a field sequence, a test is made whether the left hand side points at the same spot as the right. If the right hand side is not a field sequence, the test =∂ checks whether the left side points at the right side itself. [Note that the need for this test arises from the use of paging references.]

A logical test is true if the designated field or block is non-zero. As in the case of the binary test, the ¬ symbol indicates reversal of the sense of the test. Thus, when checking for zero, use

```
IF (¬ABC),THEN,...
```

instead of

```
IF (ABC,=,0),THEN,...
```

Similarly, when performing the opposite check, use

```
IF (ABC),THEN,...
```

instead of

```
IF (ABC,¬=,0),THEN,....
```

in order to generate more efficient code.

Any legal DO sequence is also valid after THEN in a test statement.

6.3 BRANCHES

An unconditional transfer of control can be made by a statement in any of the following formats:

```
GOTO    { assembler-expression }
        {   field-sequence     }

DO      ( { GOTO  }, { assembler-expression } )
          { CALL  }  {   field-sequence     }
          { SCALL }
```



```
DO      (RETURN)
```

The control operation GOTO specifies an unconditional branch to the storage location designated by the operand. Normally this will be the identifier of a labeled statement, but this address is computed whenever a field sequence is used as the operand (i.e., control is passed to the address contained in the field specified).

The CALL and RETURN commands provide a subroutine mechanism. CALL saves the address of the operation following the CALL and transfers control to the designated subroutine. Addresses are saved in a stack so that recursive calls of subroutines (up to twenty levels) are allowed. RETURN is specified in the subroutine; it causes a transfer to the last address saved by a CALL and the popping of the stack. A subroutine requires no special declaration and its body need not be contiguous. It is merely entered by a CALL and left by a RETURN. In fact, the same program label may be used for both subroutine calls and GOTOs.

The SCALL operation loads register 15 with the V-type address constant of the assembler expression or with the address specified by the field sequence. Next a BALR 14,15 is generated in order that the code produced by the compiler may follow I.B.M.'s subroutining conventions instead of using its own push-down stack. Note that a "BR 14" is required in order to branch back from the subroutine and continue processing. [This could be supplied in a FINAL statement, see "7 Program Delimiters".]

6.4 INPUT/OUTPUT

Any program which is to use DSPL I/O routines must execute the appropriate I/O declaration <u>once, and only once</u> before using the I/O operation. Before a card is read, the program must execute a statement as follows:

```
INPUT     eodad
```

The identifier (eodad on the input statement is the label of a statement to which a branch is to be made on end of file. [A default option is provided; thus if the parameter is omitted, control passes to the FINAL statement [see "7 Program Delimiters"] on end-of-file.]

Before a line is to be printed, control must pass through the instruction

```
OUTPUT
```

The commands for the actual I/O are

```
IN      quantity
OUT     quantity
```

The IN operation causes an 80 column card to be read and stored into the area specified by an assembler expression, or into the calculated address for a sequence. The OUT operation causes the first 133 bytes of the specified area to be printed on the system print device, the first byte being used as an ASA control character. (As with other operations, no check is made to see that the I/O area is of the proper size, or that a valid carriage control character is provided.)

6.5 REGISTER ALLOCATION

The instructions produced by the compiler make extensive use of the general registers. For this reason it is necessary to follow special procedures when using registers in assembly language within a DSPL program. Since the compiler does its own register allocation, when the programmer wants to use specific registers himself, he must declare that the compiler is not to use those registers. This is done by the instruction

```
XSAVE     reg1,reg2,...,regn
```

It is also possible to free previously XSAVEd registers by the instruction

```
XRET      reg1,reg2,...,regn
```

The user must make sure that the compiler always has several registers free to use as work registers. It is highly recommended that at least three or four registers are available at all times.

Although the compiler automatically XSAVEs the registers declared as base registers by INITIAL [see "7 Program Delimiters"], those declared as BFIELDRs are not XSAVEd in order that the user may make them available to the compiler when he does not need them.

Neither XSAVE nor XRET generate any code, but rather merely update compile-time tables.

6.6 PAGE FORMAT AND MANAGEMENT

Throughout this section, the word "xref" will mean a field sequence which ends in an paging pointer, i.e., with a FIELDX or a BFIELDX.

6.6.1 CREATE -- CREATE A NEW PAGE

A page of any length up to 32,000 bytes, i.e., 8000 words, can be created by use of the instruction

```
CREATE   size,xref[, {'pagename'} ][,EOV=addr]
                     {   lab    }
```

The first parameter is the length in words of the page desired. This length can be an assembler expression or a field sequence containing the length in words.*

"Xref" is a field sequence which must end with a paging reference. During the execution of the CREATE instruction, this paging reference will be made to point at the top of the newly formed page.

If the paging system has been generated with IR capabilities, 'pagename' is an optional twelve character name which will be assigned to the page. If preferred, the label of a twelve byte name can be specified. It is this pagename which is enclosed in slashes when referring to a page by name for information retrieval [see "5.1 Sequence Evaluation"].

At the completion of CREATE, a return code is supplied in register 15.

```
0 Creation completed
4 Insufficient space for new page -- xref is zeroed.
```

If the EOV parameter is included, the compiler will generate code to test the contents of register 15 and issue a GOTO [see "6.3 Branches"] in case of end-of-volume. (This "addr" can be a label or a field sequence pointing to the branch spot.) If this operand is omitted, the system assumes that the user will perform his own check.

6.6.2 ACQUIRE AND RELINQ -- GET AND FREE SPACE

Available space can be managed through the use of the instructions ACQUIRE and RELINQ.

---

9. In all instructions, whenever "size" is required the length must be specified in the number of words which may be represented in either of these forms.

```
ACQUIRE     size,pntr[,OUT=addr]        and

RELINQ      {size }
            {'ALL'},pntr
```

These routines manage free space on pages if the second parameter is a paging reference, as well as free space in core if it is not. (In the former case, a call is made to the paging routines ACQUIRE and RELINQ instead of to GETMAIN and FREEMAIN.)

When it is a paging reference, the 'pntr' in ACQUIRE (which may end in a paging reference or a FIELD) must point somewhere onto the page on which space is requested; at the completion of ACQUIRE it is altered to point at the new area.

A condition code is returned in register 15:
0 Request fulfilled
4 Insufficient space

If the OUT parameter is specified, a GOTO is generated in the case that the ACQUIRE was not successful [see "6.6.1 CREATE"].

RELINQ has the option of returning all space on the specified page to the free area queue. This is accomplished by specifying 'ALL' (written as shown) instead of a size as the first parameter. [Note: 'ALL' may be specified only when a paging reference is used as the second parameter.]

When returning only part of a page (i.e., a size is specified instead of 'ALL'), it is important to note that the area to be freed cannot overlap one which has not been previously ACQUIREd. When this requirement is not met the program will terminate abnormally.

At the completion of RELINQ, register 15 contains an appropriate return code as follows:
0 User still has space ACQUIREd on the page
4 The page is empty after the RELINQ

Note: On any particular page, a user should either use ACQUIRE and RELINQ or do his own space management. One cannot do both since a free-area queue, which is not protected in any way, is kept on each page. Of course, different methods of space allocation can be used on separate pages within one program.

6.6.3 HOLD -- PREVENT A PAGE FROM BEING REMOVED FROM CORE

```
HOLD        xref
```

Since any page may be "bumped" out to disk to make room for another referenced page, DSPS provides a facility for specifying that a page be

kept in core. For this purpose xref is a paging reference pointing to some location on the page to be held.

If the page to be held is not in core, it will be fetched and subsequently held. Holding a held page is an effective no-op.

Note: More than one page may be held, but whenever a page is retrieved from disk, there must be sufficient space in core for locating it. If even after swapping all non-held pages, the page provided for paging is insufficient for storing the requested page, the program will terminate with a S 851. Thus the user is responsible for insuring that there is sufficient "pageable" space for any page that is referenced.

6.6.4 RELEASE -- RELEASE A PAGE FROM HOLD STATUS

```
RELEASE     xref
```

Xref is a paging reference pointing at some location on the page to be released. Note: Releasing a non-held page is an effective no-op.

6.6.5 DEFDICT -- DEFINE A DICTIONARY

```
DEFDICT     xref,num
```

A dictionary is a page which has been declared by the user as one which he wishes to reference without an explicit pointer. Dictionaries serve the purpose of retrieving pages by number when an information retrieval system is not being used. The DEFDICT instruction is used to declare a page to be used as a dictionary and the corresponding number with which it will be referenced.

When using DEFDICT, xref is a paging reference pointing to some location on the page to be defined as a dictionary. Num is the number of the dictionary being defined, i.e., its identifier. This number must be greater than zero, but less than or equal to the DICTS operand of the corresponding PAGEINIT instruction [see "7 Program Delimiters"].

A return code is posted in register 15:
0 Definition successful
4 Invalid dictionary number

6.6.6 GETDICT -- RETRIEVE A DICTIONARY

```
GETDICT     xref,num
```

Xref is a paging reference which is made to point at the in-core address of the dictionary. Num is the number (ID) of the dictionary to be retrieved.

A return code is posted in register 0:
0 Retrieval successful
4 Undefined dictionary

As in all paging routines which retrieve a page from disk, if a copy of the page requested is already in core, the xref will be made to point at that copy (i.e., a new copy is not brought into core).

6.6.7 PCOPY -- COPY THE CONTENTS OF ONE PAGE ONTO ANOTHER

```
PCOPY     xref1,xref2
```

Xref1 is a paging reference pointing to some location on that page (page1) onto which a copy is to be made. Xref2 is a paging reference pointing to some location on the page to be copied (page2). The first operand page must be at least as large as the second operand page. The copy of page2 occupies the top of page1. That is, a field located at the nth word of page2 is at the nth word of page1 at the completion of PCOPY.

Copying includes transferral of hold status and updating of the free area queue on the first operand page. (That is, if page1 is larger than page2, the remainder is added on to the free-area queue.) [Note that both pages must be using ACQUIRE and RELINQ [see "6.6.2 ACQUIRE and RELINQ"] rather than the user's own sub-allocation routines.]

Any paging reference used internally on page2 is updated on page1 to become internal there also. The second operand page is left as is.

6.6.8 PDELET -- DELETE A PAGE FROM THE PAGING DATA SET

```
PDELET    xref
```

Xref is a paging reference pointing to some location on the page to be deleted.

7 PROGRAM DELIMITERS

7.1 FOR ALL PROGRAMS

7.1.1 INITIAL -- START A DSPL PROGRAM

All DSPL programs must begin with an INITIAL statement:

```
INITIAL [BASEREG={(reg1,reg2,...,regn)}][,CSECT=NO]
                 {      NO          }
```

This statement, in addition to setting up base registers, provides save-area chaining and prepares some areas for the compiler's use. The registers specified by the BASEREG parameter are assigned as base registers, and are therefore also automatically XSAVED [see "6.5 Register Allocation"]. By specifying CSECT=NO, the code generated will not include a CSECT statement, which is otherwise generated. If BASEREG=NO is specified, only the compiler areas are generated (i.e., no save-area chaining, etc.). If no parameters are specified, the compiler assumes

```
INITIAL  BASEREG=(13)
```

[Note: the compiler assumes that register 15 will point at the INITIAL statement on entry. Furthermore, if save-area chaining is to be done, register 13 must point at the higher save-area.]

In order to enable the user to have multiple DSPL csects in one assembly, all labels generated by the compiler are indexed (the index being incremented in the INITIAL statement). If a user wants to address base fields or base blocks generated by a FINAL [see "7.1.2 FINAL"] of a previous csect, he should use the following instructions:

```
INITIAL
L     REG,=A(LABEL)
USING LABEL,REG
XSAVE REG
  .
  .
  .
```

where LABEL is the label of the previous FINAL and REG is the base register he wishes to use (not 0,1,12,13,14,15, nor the one specified on this INITIAL).

7.1.2 FINAL -- END DSPL PROGRAM

Every program must execute a FINAL statement which reserves space for base blocks and base fields as well as closing DCBs and defining system's areas. The format of a FINAL statement follows:

$$\text{FINAL } [\text{ret}][,(\text{reg1}[,\text{reg2}])][,\text{T}][,\text{RC}=\begin{Bmatrix}\text{NUM}\\(15)\end{Bmatrix}]$$

The first parameter on FINAL is an unconditional branch as in GOTO [see "6.3 Branches"]. If this parameter is omitted, the FINAL statement may take parameters similar to I.B.M.'s RETURN macro (see I.B.M.'s manual "Supervisor and Data Management Macro Instructions"). Any trailing commas may be omitted.

Examples:

```
FINAL     BRANCH
FINAL     ,(14,8),T
FINAL     ,,,RC=(15)
FINAL
```

When no parameters are specified, the compiler assumes

```
FINAL     ,(14,12),,
```

FINAL generates a LTORG so that all literals will be addressable in the CSECT. If the compiler is to save space for the user's base fields and base blocks, it is done in FINAL. For those base fields and base blocks for which the user reserves space, the FINAL statement generates an EQU in order to define the name assigned. [Note that this requires a user to define all such base block and base field locations before FINAL.]

7.2 FOR PAGING PROGRAMS

7.2.1 PAGEINIT -- INITIALIZE PAGING SYSTEM

This instruction initializes the paging system and must be included at least once at the beginning of all programs using DSPS with paging.

```
                'pagedd'
PAGEINIT                    ,[DISP=NEW][,DICTS=NUM]
            field-sequence
```

'Pagedd' is the DD name associated with the data set to be used for paging. Alternatively, a user can specify a field sequence pointing at that DD name.

DISP=NEW (written as shown) directs the paging system to treat the paging data set as new although it may be, in fact, old. Thus an old data set can be reinitialized without reallocation of space. In case DISP=NEW appears on the DD card used, only one call to PAGEINIT may be made in the job step.

However, in order to free all the space used by the paging system, a user may want to call PAGEEND and then reopen the same data set later in the job step. Thus, if a user wishes to call PAGEINIT more than once, he must create the data set in a previous job step, so that he can specify DISP=SHR on the DD card in the present job step, and DISP=NEW only on the first PAGEINIT instruction.

DICTS=num directs the paging system to reserve space for the definition of num dictionaries in the Page Control Block [see "6.6.5 DEFDICT" and "6.6.6 GETDICT"]. 'Num' should be in the range 1-255; omission is equivalent to no dictionaries needed. This operand must be coded whenever a data set with dictionaries is being used, and for an old data set 'num' must correspond to the number of dictionaries previously requested. A field sequence containing the number, or the label of a location containing that number, may be used instead of the numerals themselves.

7.2.2 PAGEEND -- TERMINATE PAGING SYSTEM

```
PAGEEND
```

This instruction, which must be executed at the end of all programs using DSPS with paging, causes all necessary updates to be made to pages and the Page Control Block; data sets are closed; all space used for paging is freed. This instruction may be used as many times as needed within one job step as long as the conditions for the use of multiple PAGEINITS are met.

[Note that if the program terminates before the execution of PAGEEND, the pages in the data set specified by 'pagedd' in the PAGEINIT statement no longer contain valid results. It is therefore suggested that when such loss of information cannot be afforded, the user's data set be copied into a temporary work data set in order to provide one level of protection.[10]]

---

10. Because the paging system depends on track and record numbers for its retrieval system, any change in the arrangement of the records (pages) in the data set will invalidate the data set. Both IEHMOVE and IEBGENER (the I.B.M. utilities) may alter the relative locations (TTRs) of the records when copying a data set which resides on multiple extents. (This occurs on track overflow data sets with a split allocation.) For this reason, all DSPS page data sets which are to be moved or copied must be created with CONTIG specified in the SPACE parameter and without any secondary allocation.

8 STATEMENT FORMATS

DSPL allows any assembler label to be a label of any DSPL statement other than a declaration or register allocation. Labels begin in column one and operations may begin in any column between two and seventy one. One or more blanks must be present before and after the op code, but no more blanks are permissible until the end of the statement (before a comment).

When commenting a program, a user may not put a comment on INITIAL or FINAL [see "7 Program Delimiters"] if none of the parameters are being used. Such comments will lead to syntax errors since the compiler accepts parameters in a free format.

8.1 CONTINUATION CARDS

A continuation card is indicated in DSPL by the use of a non-blank character in column 72 of an input card. The next card may begin in any column other than column one.

However, no DSPL instruction other than DO, IF, and IFANY may be continued; and these may not be broken at any place except after a comma outside a major set of parentheses, or after the comma either directly before or after a THEN in an IF or IFANY statement. An example of the correct usage of continuation cards follows:

```
IF(P01(XREF),=,AB),((ZERO)),                              X
     THEN,(AB,_,P01(XREF)B),                              X
     (GOTO,OUT)
```

On the other hand, a user may not use either of the following:

```
DO (P01(XREF)B,_a,AB),(AB,_,                              X
    ABC)

DEFDICT XREF,                                             X
    NUM
```


8.2 LISTING FORMAT

In order to improve the appearence of DSPL listings, carriage control commands have been included in the compiler.

- EJECT

- SPACE num

behave similarly to the corresponding assembler language commands. [The minus sign in column one, suppresses the printing of the command itself.]

9 FILE MANAGEMENT AND PAGE UPDATES

Several problems regarding file management arise for a system which provides variable length pages for the user. For example, all pages cannot reside in fixed length areas in core; after the deletion of a page, the disk space can only be reused by another page of the same size or smaller; and pages cannot be relocated into larger areas at will, without adjusting all the pointers which refer to that page (either directly or through some sort of directory). In designing a paging mechanism, therefore, the basic needs of the user must be carefully considered in order to settle the system design trade-offs.

For DSPS, these assumptions include that user programs run in a real-time environment which requires fast response (e.g., less than one second response time in graphics). Furthermore a user is assumed to deal with a constantly expanding file (i.e., one in which new pages are added at run-time) rather than one which requires many deletions. The designers of the paging component of DSPS therefore chose to incorporate a method of paging which would use track and record numbers for addressing and update pages in place, rather than use a special page(s) to map some sort of keys into actual disk addresses. Any mapping function involves a level of indirectness for all pointers which cross page boundaries, which would thus increase search time and decrease response time during execution. On the other hand, using disk addresses implies retrieving the requested page with only one disk access.

In other words, the classical space-time trade-off was settled in favor of time: whereas a user's execution speed cannot afford to be limited by the time required for page retrieval, a user can afford enough disk space to allocate the maximum amount of space need for a page immediately upon creation. In this way all pages may be updated in place without the possibility of extending past the allocation provided. Even though some secondary storage space may remain unused while the page is still growing, the inefficiency in space allocation will be well compensated for by the increase in response time resulting from the elimination of any mapping function.

In addition to wasting some space, the disadvantage of updating pages in place is that after a page has been deleted through the use of PDELET, the space formally used by that page on the disk is not available to any other page until an off-line updating (i.e., garbage collection) program is run. This program requires that all pages in the data set be compacted to the beginning of the space, and thus all paging references are no longer valid until the disk address in the second word is updated to the new disk address.

The algorithm then requires that all pages be read one at a time, and each reference be changed according to a table that was set up at compaction time. This implies quite a high overhead to update, but, again, this overhead was felt to be more agreeable than using a mapping function.

The DSPS design criteria place it at one end of a spectrum of philosophies of garbage collection methods applicable to a paging system. For a highly static file (or at least, a simple growing file such as the one in the sample program included on the tape), the above method would be used, in which pages are updated in place and entire data sets are compacted in an off-line program. To provide for a more dynamic application, one would keep track of reusable space within the program, instead of returning such space to the system. Finally, the most dynamic method would be to use a mapping function, so that pages could be deleted and data sets compacted "on the fly".

Thus, in accord with the philosophy of a "minimal paging system", it is left to the user to implement the garbage collection routines which best fit his own application. A particular user might feel his application involves a basically expanding file with relatively few deletions required, and he is therefore content with CREATE and PDELET as it exists. One user at Brown, however, has need of a more dynamic file, and thus has implemented the second method in which he can reuse "deleted" pages at his own discretion. That is, whenever a page is no longer needed, he does not use the command PDELET, but rather calls his own routine which flags the page as available. When he wants to create a new page, he calls another of his routines which checks whether he already has a page of appropriate size available; and only if there is not, does he use the command CREATE to get more space. Finally, some other user may feel it is better for his application to implement his own mapping function as described above in order to provide for a highly dynamic file.

For files which may be segmented into a large static subset (e.g., completed drawings) and a relatively small dynamic subset (e.g., those few pictures currently being drawn, plus work pages), there is another technique for effecting a compromise between the drawbacks of the strictly expanding file and those of a highly dynamic one. The user may keep an inverse list of all pointers referring to a particular, dynamically changing page(s). That is, since only a small number of pointers will reference the page (e.g., only the very basic subpictures of a graphical file are referred to more than a few times), a list of these pointers may be kept on a associated page (or on that page itself) in order that all such pointers may be found immediately for updating in case the page is relocated. Through this compromise, only those pages which can change most radically need cause on-line relocation, and thus universal updating may be avoided without resorting to a global mapping function.

[Warning: Although it is possible to implement a mapping function using DSPS, a user requiring a highly dynamic file should reconsider his choice of the DSPS paging system for his application in light of the run-time overhead on updating involved.]

10 RESTRICTIONS ON USING THE COMPILER

10.1 The biggest restriction on paging pointers is that they may never point at an area in resident storage instead of a location on some page. That is, all paging references within the resident section must be used externally and no paging reference on an external page can be made to point back into the user's program. The following diagrams illustrate the two illegal pointers:

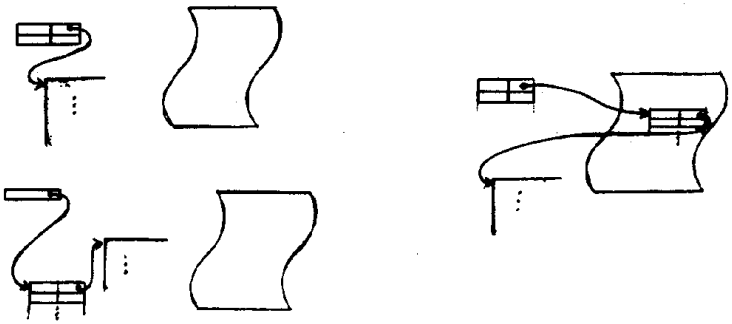

Figure 7. Illegal Use of Paging References

10.2 A field sequence which is to be used for a dynamic origin or length [see "5.3 Dynamic Origins and Lengths"] may not end with a FIELDX or a BFIELDX, as these are two-word pointers. The following is an example of this error:

```
BFIELD B
FIELDX X,3
FIELD  A,BX,(0,31)        or
BLOCK  K,(4,BX)
```

However the dynamic field sequence may refer to a field on another page. For example,

```
BFIELDX B
FIELD   X,3,(8,23)
```

```
FIELD   A,BX,(0,31)        or
BLOCK   K,(4,BX)
```

are legitimate definitions.

10.3 DSPS is not a re-entrant system. Since the DSPS users at Brown do not need to run concurrently, it was felt that run-time routines could be written more efficiently if re-entrancy were eliminated. [If the users at some installation would like to have a re-entrant version of DSPS (requiring at most two man-months of work), the author will be glad to discuss the writing of such a modified system.]

10.4 Any data element must be declared before it can be used in an instruction.

10.5 If a user wishes to run DSPS without the paging component, he may do so by specifying PARM='PAGING=NO' on the EXEC card for the COMPILE step of his job. If this option is specified, the paging system may no longer be referenced in the LINK-EDIT step, nor may a data set containing pages be used in the GO step [see "13.3 Running User Programs"]. No paging references or routines which call the paging system will be recognized by the compiler.

10.6 In general, a user cannot depend on the contents of registers 0,1,14,and 15 remaining unchanged during any DSPL instruction. Furthermore, whenever the paging component is being used, register 12 is also reserved by the system. However, register 1 is kept intact during an INITIAL instruction so that parameters passed using that register may still be accessed through register 1 immediately after INITIAL. One method suggested for this is as follows:

```
        INITIAL
        BFIELDR   R1,1
        BFIELDR   R2,2
        BPIELD    NEWPARM
        FIELD     PARAM1,0,(0,31)
        XSAVE     2
        DO        (R2,_,R1)      Save the pointer to the parms
*                      and pick off the first parm
        DO        ((NEWPARM),_,R2(PARAM1)(PARAM1))
            .
            .
            .
```


11 DSPL CODING

In order to use DSPL most effectively, the user should attempt to "think in DSPL" instead of coding in assembly language and then translating the results into DSPL. It is very wasteful to implement assembly language using DSPL, as the list processing language has advantages of its own. Remember that DSPL code is excellent for programming pointer chasing, list organization and manipulation, and other such list processing features; but that the option of coding assembly language within DSPL is also included so that simple adds, shifts, etc. can be written as concisely as possible.

Although in many systems it is necessary for a user to be aware of the limits of the pages on which he is working, such restrictions do not exist to the user of DSPS. There is no need for a user to check flags in pointers on some page to determine if that pointer crosses page boundaries. By using paging references on all pointers that may cross to another page, the paging system will check whether or not the pointer under consideration is used internal or external to that page.

11.1 USE OF THE DYNAMIC ORIGIN

One of the most powerful features in DSPL is the dynamic origin for a field or block. An obvious use of a field having a dynamic origin is in a branch table (transfer vector). Coded here is an example of assembly language implemented in DSPL, followed by a more efficient use of DSPL.

a)

```
        BPIELD FLAG
          .
          .
          .
        IF ((FLAG),=,0),THEN,(GOTO,LAB0)
        IF ((FLAG),=,1),THEN,(GOTO,LAB1)
          .
          .
          .
        IF ((FLAG),=,15),THEN,(GOTO,LAB15)
          .
          .
          .
```

b)

```
        EBLOCK TABLE,(LABS,15)
        BFIELD FLAG
        FIELD BRNCH,(FLAG),(0,31)
         .
         .
         .
        DO (GOTO,(TABLE)(BRNCH))
         .
         .
         .
LABS    DC A(LAB0)
        DC A(LAB1)
         .
         .
         .
        DC A(LAB15)
         .
         .
         .
```

Notice that the "computed GOTO" calculates the value of FLAG (which would be assigned 0,1,2,...,15), uses that value as the dynamic origin (in words) of the field BRNCH, and thus takes the appropriate branch in the table called LABS. This second method of coding uses significantly less core and execution time by taking advantage of a field's capabilities.

Similarily, a hash table may be implemented by storing the hash code in a BFIELD which can be used as the displacement of a BLOCK or FIELD in the table.

11.2 USE OF LONG SEQUENCES

When a long sequence is to be used repetitively, it may be advantageous to use a temporary BFIELDR to save code. For example, for

```
        BFIELD A
        FIELD B,0,(0,31)
        FIELD C,AB,(8,31)
        FIELD D,5,(24,31)
```

instead of


```
        DO (ABCD,_,AB),(AB,_@,ABC)
        DO (ABD,_@,ABCD),(ABC,_@,A)
```

it may be better to write

```
        BFIELDR T,2
        XSAVE 2
        DO (T,_@,ABC)
        DO (TD,_,AB),(AB,_@,T)
        DO (ABD,_@,TD),(ABC,_@,A)
        XRET 2
```

11.3 DEBUGGING

11.3.1 DETERMINING ERRONEOUS STATEMENTS

Whenever a DSPL program is run, the user is advised to scan through the DSPL listing first, in order to find errors detected by the compiler. Next he should check the assembly listing for any unassembled statements, which may have been caused by an error not detected by the compiler.

In addition to using the contents of registers 14 and 15 to determine which routines were being accessed at the time of an interrupt, a DSPL user also has the facilities of the push-down stack and its pointer for the instructions (CALL, LAB) and (RETURN). The pointer into the stack can be found in a word labeled $MS; the stack is at $PSTCK. Both $MS and $PSTCK are generated by FINAL. Thus the user may examine the subroutine nest to determine his state at termination.

Several completion codes may be issued by the paging routines. The analysis of the errors is as follows:

S 851 Not enough pageable space (i.e., too many pages held, too small a region size provided, or an invalid pointer passed to the paging system)
U 0001 I/O error in the data set containing the pages

11.3.2 DSPS CALLING CONVENTIONS

The most common error when using DSPS results from using a paging reference in a sequence evaluation before it is made to point at some area. Although the user does not need to know the details of the paging component of DSPS, using such a paging reference will usually cause an abnormal termination within the code of the paging subroutines. Thus, knowing the calling conventions for the paging system will help a user to

determine the error which caused a program interrupt in some paging routine.

1) Register one points at the paging reference to be evaluated when using any of the routines requiring such evaluation.

Note: In general, the user's registers are stored in his own save area by the paging subroutines. The one exception to this rule is that during the evaluation of a sequence (i.e., in PCATCH), the user's registers are saved in the eighteen word area after the identifier INITSAVE within the paging routines in core.

2) Register twelve points at the top of the page last addressed.

3) Register zero has the length specification in the case of CREATE, ACQUIRE, and RELINQ; and it has the number of the dictionary in DEFDICT and GETDICT.

11.3.3 IN-CORE PAGE FORMAT

In order to examine a user's pages in core after an abnormal termination of the job, one should know the following conventions.

1) The pointer to the Page Control Block (used by the paging system only) is located after the identifier DATAADR at the end of the paging routines in core.

2) The six words preceeding every page are used as the Page Control Link for that page. These words are attached to the user's page, and are retrieved in addition to the number of words requested by the user.

3) The first two words of a PCL (including that of the PCB) are pointers in the ring of all the pages in core.

4) The fourth word of a PCL is the disk address and length of the page. That is, (except for the last two bits) it is equivalent to the second word of any paging pointer referencing that page.

5) Space allocation by ACQUIRE begins at the bottom of the page. If ACQUIRE and RELINQ were used for space management on the page under consideration, the first two words of the user's page will be the beginning of the free-area queue for that page. (If these first two words are no longer part of the free-area queue, the sixth word of the PCL will have the address of the top of the free-area queue.)

6) The user's page starts after the PCL (i.e., at the seventh word after the location referenced by the linked rings).

11.3.4 SCAN

A routine SCAN is provided to snap the pages in core.

```
SCAN     num[,BASE=(addr1,addr2[,ONLY])]
```

Num is the identification number (between 1 and 255) which is printed out at the start of the SNAP.

When BASE is not specified, all the pages which are in the core at the time of the execution of the instruction are dumped.

If BASE is specified, the section of core between the absolute addresses of addr1 and addr2, as well as the pages, are dumped. In the case that the parameter ONLY is included, just the area from addr1 to addr2 is dumped and <u>not</u> the pages.

12 SETTING UP THE PAGING SYSTEM

12.1 SVC FOR MVT USER'S

In order to run the paging component of DSPS, installations running under MVT will need to add a Type III SVC to their SVC library. This supervisor call is used to attach a Start I/O appendage to the paging system in order to use BSAM for paging.

The system manager should be consulted to link-edit the SVC whose source code found under SYS1.PAGEMACS(QSVC) into the SVC library. He will assign an SVC number (such as 228 for the system at Brown) and a member-name under which the load module should be stored in the SVC library.

[Note that installations under MFT have no need for this SVC.]

12.2 PAGING OPTIONS

The paging component of DSPS provides for many options at the time of generation in order that the user may have the best system for his application. The following table gives each alternative.

KEYWORD	VALUES	ROUTINES GENERATED	FUNCTION
ALLOC	INT,any		Generate GETMAIN and FREEMAIN for space management in core
ALLOCN	null,name		Name of user's GETMAIN routine (see ALLOC)
COPY	YES,NO		Provide for copying one page onto another
CREATE	YES,NO	CREATE	Provide the capability for creating new pages
DEL	YES,NO	DELETE	Provide for deletion of pages

DICT	YES,NO	DEFDICT,GETDICT	Allow dictionaries
DSPL	YES,NO	PAGPTR,PCATCH, SETSTAR,GETPAGE	Provide interface with DSPL calling conventions
HOLDREL	YES,NO	HOLD,RELEASE	Allow a user to control page swapping from core
ID	YES,NO		Generate an identifier (DC) of paging routine names
IR	NO,YES	PAGE	Provide the link for an information retrieval interface
LINKXBS	YES,NO		Link paging references on each page
NOSPRT	null,name		Abend when insufficient space for user's pages in core, or address of user's SPIE routine to which control should be passed when insufficient space is detected
SCAN	NO,YES	SCAN	Do not provide the facility of the page snapping routine
SPIE	YES,NO	SPIETRAP,POINTX	Allow for virtual memory in paging (not DSPL)
SUBALOC	YES,NO	ACQUIRE,RELINQ	Provide DSPS page management capabilities
SYS	MFT,(MVT,num)		Attach a Start I/O appendage
UNALLOC	null,name		Name of user's FREEMAIN routine (see ALLOC)
XRP	NO,YES		Do not provide a third word in all paging references to give address of the page in core (must be NO if DSPI=YES)

13 JOB CONTROL FOR DSPS

13.1 RELOADING FROM TAPE

The following control cards transfer the data sets from tape to disk.

```
// EXEC PGM=IEHMOVE
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=2314,VOLUME=SER=diskname,DISP=OLD
//DD1 DD UNIT=2314,VOLUME=SER=diskname,DISP=OLD
//TAPE DD UNIT=2400,VOLUME=SER=BUDSPS,DISP=(OLD,PASS),                  X
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN DD *
    COPY     PDS=SYS1.DSPL,FROM=2400=(BUDSPS,1),TO=2314=diskname,      X
               FROMDD=TAPE[,RENAME=newname1]
    COPY     PDS=SYS1.PAGEMACS,FROM=2400=(BUDSPS,2),                   X
               TO=2314=diskname,FROMDD=TAPE[,RENAME=newname2]
    COPY     PDS=SYS1.PGM,FROM=2400=(BUDSPS,3),TO=2314=diskname,       X
               FROMDD=TAPE[,RENAME=newname3]
/*
```

SYS1.DSPL and SYS1.PAGEMACS will be blocked 3200 after reloading from tape; SYS1.PGM will be blocked 80. Installations requiring alternate blocking should use IEBCOPY as described in "IBM System/360 Operating System Utilities".

13.2 SETTING UP DSPS

The following control cards assemble the DSPL compiler and the paging system. The parameters used on the macro PAGING are those suggested for running DSPS programs [see "12.2 Paging Options"].

```
// EXEC ASMFCL
//ASM.SYSLIB DD DSNAME=SYS1.MACLIB,DISP=SHR
// DD UNIT=2314,VOLUME=SER=diskname,DISP=SHR,                           X
//             DSNAME=SYS1.PAGEMACS
//ASM.SYSIN DD *
         PAGING SYS=(MVT,220),SPIE=NO,SCAN=YES
         END
/*
//LKED.SYSLMOD DD UNIT=2314,DISP=(NEW,KEEP),SPACE=(TRK,(10,10,1)),     X
//             DSNAME=SYS1.LINKLIB(PAGING),VOLUME=SER=diskname
```



```
/*
// EXEC ASMFCL
//ASM.SYSLIB DD DSNAME=SYS1.MACLIB,DISP=SHR
// DD UNIT=2314,VOLUME=SER=diskname,DISP=SHR,DSNAME=SYS1.DSPL
//ASM.SYSPRINT DD SYSOUT=A,SPACE=(TRK,(30,30)),                        X
//             DCB=(RECFM=FBSM,LRECL=121,BLKSIZE=1936,BUFL=1936)
//ASM.SYSIN DD *
         DSPL
         END
/*
//LKED.SYSLMOD DD UNIT=2314,VOLUME=SER=diskname,DISP=OLD,              X
//             DSNAME=SYS1.DSPL.LINKLIB(DSPLPGM)
/*
```

13.3 RUNNING USER PROGRAMS

These control cards must be used for running any DSPS programs.

```
//JOBLIB DD DSNAME=SYS1.DSPL,VOLUME=SER=diskname,UNIT=2314,            X
//             DISP=(SHR,PASS)
//COMP EXEC PGM=DSPLPGM,REGION=150K[,PARM='PAGING=NO']
//RHLETNER DD UNIT=2314,DISP=(NEW,PASS),SPACE=(3360,(100,50)),         X
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3360)
//OUTPUT DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=111,BLKSIZE=111)
//SEDGEDDD DD DDNAME=SYSIN
//SYSIN DD *
        .
        .  (user's program)
        .
/*
// EXEC ASMFCLG
//ASM.SYSIN DD UNIT=2314,DISP=(OLD,DELETE),DSNAME=*.COMP.RHLETNER
/*
//LKED.PAGE DD DSNAME=SYS1.PAGEMACS,UNIT=2314,VOLUME=SER=diskname,     X
//             DISP=SHR
//LKED.SYSIN DD *
    INCLUDE  PAGE(PAGING)
/*
//GO.SYSPRINS DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSUDUMP DD SYSOUT=A
//GO.DSPSPAGE DD DSNAME=your.paging.set,UNIT=2314,                     X
//             VOLUME=SER=diskname,DISP=(NEW,KEEP),                    X
//             SPACE=(TRK,(100),,CONTIG)
//GO.SYSIN DD *
        .
        .(user's data)
```

```
/*
```

In order to run the test program, replace

```
//SYSIN DD *
```

and the user's program by

```
//SYSIN DD DSNAME=SYS1.PGM(PGM),VOLUME=SER=diskname,UNIT=2314,          X
//            DISP=SHR
```

and replace

```
//GO.SYSIN DD *
```

and the user's data by

```
//GO.SYSIN DD DSNAME=SYS1.PGM(DATA1),VOLUME=SER=diskname,UNIT=2314,     X
//             DISP=SHR
```

in order to first use the data stored under the membername DATA1 of SYS1.PGM. This creates an information retrieval data base. Requests may then be processed by rerunning the sample program with DATA2 substituted for DATA1 on the GO.SYSIN card, and the disposition of GO.DSPSPAGE changed to SHR. As noted in "7 Paging Delimiters", DSPSPAGE must be as written in order to be the same as the pagedd given in the PAGEINIT statements.

14 REFERENCES

1. Evans,D. and van Dam, A., "Data Structure Programming System", Proceedings of IFIP Congress 68, (Edinburgh, 1968), p. C67.

2. Gray, J. C., "Compound Data Structure for Computer Aided Design; a Survey", Proceedings of 22nd National Conference of the Association for Computer Machinery, (Thompson Book Co., Washington, D.C., 1967), p. 355.

3. Knowlton, K. C., "A Programmer's Description of L*", Communications of the ACM, (The Association for Computing Machinery, Inc., New York, 1966), Vol. 9, No. 8, p. 616.

4. Newell, A., Earley, J., and Haney, F., "360 Reference Guide: *1 Manual", (Carnegie Institute of Technology, Pittsburgh, 1967).

5. Richards, M., "The BCPL Reference Manual", Project MAC Memorandum-M-352-1, (Massachusetts Institute of Technology, Cambridge, 1968).

6. van Dam, A. and Evans, D., "A Compound Data Structure for Storing, Retrieving, and Manipulating Line Drawings", AFIPS conference Proceedings, (Thompson Book Co., Washington, D.C., 1967) Vol. 30, p. 601.

7. Wirth, N., "PL360: A Programming Language for the 360 Computers", Journal of the ACM, (The Association for Computing Machinery, Inc., New York, 1968), Vol. 15, No. 1, p. 37.

DATA STRUCTURES PROGRAMMING SYSTEM

SEPTEMBER 1969

COMPLIMENTS OF THE

HYPERTEXT EDITING SYSTEM

CENTER FOR
COMPUTER & INFORMATION SCIENCES
BROWN UNIVERSITY
PROVIDENCE, RHODE ISLAND

29 SEPTEMBER, 1969