OSEDIT: AN INTERACTIVE EDITOR FOR OS/360

A thesis presented to

The Faculty of Graduate Studies

The University of Manitoba

In partial fulfilment of the requirements

for the Degree Master of Science in Computing Science

by

K. B. Rugger

May, 1974

ABSTRACT

The work of this thesis represents the design, implementation, and documentation of an interactive source-file editing package, called OSEDIT. In order to perform some of the functions desired in such a package, considerable interaction with the operating system must be done. These interfaces are general, and may be useful in other and similar applications. The data structures used to do the file editing, and the command language that the user has available at his terminal are also described.

ACKNOWLEDGEMENTS

The author wishes to thank the members of the Computer Science Department and of the Computer Centre at the University of Manitoba for their continuing assistance in the implementation of OSEDIT. In particular, Bill Reid has provided many refinements and improvements to the original code. Because of his position in the Computer Centre, he has also undertaken the sometimes distasteful task of dealing with the local suzerains. Mike Doyle has proven himself to be a very useful advisor: his suggestions have been frequent and always technically correct.

TABLE OF CONTENTS

OSEDIT

I. INTRODUCTION

OSEDIT is an implementation of an interactive source-file editing package. This system has been successfully implemented under the Manitoba University Monitor (MUM) system [9] as one of the MUM application programs, and under the Time Sharing Option (TSO) of the System/360 operating system. OSEDIT allows for online updating of System/360 files which reside on direct-access storage and are organized as physical sequential or partitioned files. These files must also be in one of the System/360 "fixed" record formats[6].

The design criteria that were initially decided upon for OSEDIT were primarily motivated by the failings of earlier and similar text-editing systems. The foremost objective was to provide a system which would be extremely reliable, not only from the operating system viewpoint, but also in protecting the integrity of users' data. This goal led to designing a system which kept changes to the file being edited completely separate from the original file.

Normally, in the event of a system failure, only the changes are susceptible to damage. If OSEDIT is not the cause of failure, changes can usually be recovered.

Our second objective was to provide good response time for the interactive user. Good response time is a somewhat subjective term, dictated objectively by such things as total system load, the complexity of a particular request, etc. MUM users typically expect response time of less than one second; consequently, this was the chosen design-point for the most commonly used OSEDIT commands. As happens in any system of this type, there will be a strong correlation between response time and total system load. Response time is also affected greatly by the necessary serialization in sharing certain resources. This serialization occurs in two places insofar as OSEDIT is concerned: in input/output channel and device contention, and in the acquisition of core storage for buffers. It was decided that core storage was the more critical of the two; hence, OSEDIT is very conservative in this manner. The status of a particular user is maintained in a record which is kept on disk by MUM. While the user's status is on disk, no core storage is being used for him. This reflects the desire to keep OSEDIT core usage to a minimum--a goal in direct conflict with good response time.

The command language and the facilities implemented were chosen with the above criteria in mind: reliability, low response time, and low core usage. Many additions are possible, but we believe that those which have been implemented fulfill most editing requirements.

This documentation details the implementation of OSEDIT in a more or less "bottom-up" fashion, starting with the OS data management interface, and ending with the command structure that the user sees. An OSEDIT user's guide is supplied as appendix A. It is intended that each section can be read as a complete unit, with little dependence on previous sections. Here we present an overview of OSEDIT operation:

(1) The user signs on to OSEDIT. A disk data set is allocated, called the work data set, which will be used to store card images entered by the user during the session.

(2) The user enters the FETCH command. This command specifies a data set, called the fetch data set, which the user wants to edit.

(3) Editing commands are entered, causing a repetoire of changes to be accumulated in the work data set, but causing no changes to the fetch data set. The user at his terminal, however, sees a virtual data set formed from the fetch data set and the changes he has applied. This virtual data set is called the change data set.

(4) A SAVE command is then used to write the change data set into a real OS/360 data set, called the save data set.

(5) The user may return to step (2) to edit another data set or he may sign off. The sign off processing deletes the work data set from disk.

The commands available to the OSEDIT user can be classified into two categories according to their function. The first category comprises those commands which take a data set name as an operand, and which are therefore called data set oriented commands. These commands include FETCH and SAVE as described above, and also ALLOCATE and SCRATCH, which create and destroy disk data sets, respectively. The second category consists of the text editing commands. All of these operate on the FETCHed data set, and all of them take a range of sequence numbers as an operand. LIST allows the data set to be listed to the terminal. DELETE specifies the deletion of one or more card images in the data set. FIND and AFIND allow the data set to be scanned for the occurrence of some particular character string. CHANGE and ACHANGE operate in much the same manner, except that the character string being sought is replaced by another character string. The algorithms used to implement these commands are discussed in section V.

II. OS/360 DATA MANAGEMENT INTERFACES

The data management interfaces that OSEDIT uses were developed because the standard IBM interfaces were unsuitable. Actually, they were written twice. The "practice" implementation differed very significantly from the final version in that it attempted to use conventional data management services supplied by the operating system. These services do not provide any form of dynamic device allocation, except under the Time Sharing Option (TSO). The TSO services, which are not available to batch jobs, are implemented in such an ad hoc manner that they cannot easily or safely be extended to batch jobs. One can approach a solution to providing dynamic allocation in two ways, given that the TSO code is not used.

The first solution is to use those facilities already provided, and to resort to "illegal tricks" only when required. For example, one could avoid the problem of not having dynamic data set allocation facilities by providing OS Job Control Language "DD cards" [4] for every pack on which editing is to be done. This type of solution has obvious shortcomings, such as only a permanently mounted volume may be accessed. This particular limitation is not serious in many installations, including our own. Other problems accompany this type of solution, however, and they are not quite so obvious. For example, the STOW facility of the operating system operates only with the partitioned data set access method supplied by the operating system. OSEDIT uses its own access methods to attain a much higher level of performance than is available otherwise. This makes the OSEDIT access method incompatible with STOW and precludes the use of partitioned data sets. Because we are not willing to give up partitioned data set support, we must resort to a considerable amount of coding effort to overcome the poorly designed STOW facility. This type of problem is usually solved by acquiring a very good understanding of the operating system code, and handing it such parameterization that it is tricked into performing the desired function. Such tricks are very often dependent on the release of the operating system, and may not provide adequate error checking. One of the worst problems is the operating system's predilection for issuing ABEND requests from service routines such as OPEN, CLOSE, and EOV. These may be trapped via the STAE mechanism, but the side effects of ABEND proceedings are very hard to withstand.

Rather than attempt to solve the many problems of interfacing with the existing data management services, it was decided that OSEDIT would directly handle some of the services itself. The OSEDIT-supplied facilities include:

(1) volume allocation/deallocation,
(2) data set OPEN/CLOSE services,
(3) an End-Of-Volume (EOV) service, and
(4) Partitioned Data Set (PDS) support.

The data management facilities of the operating system used by OSEDIT are:

(1) the Execute Channel Program (EXCP) level of I/O supervision [8],
(2) catalog management [8],
(3) data set creation and deletion [3], and
(4) the STOW facility for partitioned data sets [6].

OSEDIT contains a module named COM which supplies the equivalent services of the operating system's job management, device allocation routines, and the EXCP level OPEN/CLOSE/EOV services. COM accepts, as input, a parameter list which is detailed by the COP dsect [Appendix B]. This parameter list is also used by other routines which do operations at the data set level. It contains information about the data set, and a code which tells COM what operation is to be performed. We now give a detailed description of the services provided by COM.

OPEN

A data set is opened by COM in the following manner. First of all, COM establishes and initializes a work area (described by the COD dsect [Appendix B]) for its internal operations. This area contains the necessary control blocks to do operations on the Volume Table of Contents (VTOC) [7] for a direct-access device. COM then attempts to enqueue on the data set's name, using the operating system's ENQ macro [6]. Exclusive or shared control of the data set is requested, based on whether OSEDIT will write into or read from the data set. COM will return an error code to the caller if the data set is not available at this time.

COM then ascertains whether or not the direct-access volume requested is available. This consists of a number of tests on the status of the device which is maintained by the operating system in the Unit Control Block (UCB) [7]. A device is considered available by OSEDIT if the following conditions hold:

(1) the device is a direct-access device,
(2) the device is "ready" (turned on),
(3) the device is logically "online" to the system,
(4) an operator requested "mount", "dismount", or "unload" operation is not pending,
(5) the device is not marked "non-sharable", and
(6) the device is not the IBM 2321 data cell.

If the volume is not available, COM dequeues from the data set (using the DEQ macro [6]), and returns an error code to the caller. Otherwise, a count field in the device's UCB, the data management count, is incremented by one to indicate that there is a new user on the device. The operating system will permit such operations as changing removable packs on the device only when this count is zero. Normally, this allocation function is supplied by the job management routines which ensure that a disk pack will remain mounted for the duration of a job by incrementing the same field at job initiation time. If the device has in fact been allocated to OSEDIT by job management, the data management count is not incremented by COM.

Exclusive control of the VTOC is then requested (via the ENQ macro), and a read operation is started which searches the VTOC for an entry corresponding to the name of the desired data set. If this search fails, the OPEN fails at this point; if the search succeeds, the VTOC entry (called a Data Set Control Block or DSCB [7]) for the data set is read into the COM work area. The important data set attributes are copied from the DSCB into the caller's parameter list for later use when processing the data set. If the data set is being opened for output, COM will make sure that the expiry date for the data set has passed before proceeding with the OPEN (unless this check is specifically overridden by the caller). If this check succeeds, the OPEN will succeed provided that there are no errors in the DSCB for the data set. This is normally a very serious situation because the VTOC may not reflect the true allocation of space on the device.

All that remains to be done to complete the OPEN is to build an operating system control block, used by the I/O supervisor, called the Data Extent Block (DEB) [7]. The contents of the DEB are used by the I/O supervisor to check the validity of disk addresses supplied by the requesting program during an I/O operation. The address of the DEB is returned to the user via the parameter list as part of his DCB.

CLOSE

Close processing chiefly undoes the work performed by the OPEN function. COM first of all obtains and initiates its work area (described by the COD dsect). If the VTOC entry for the data set is to be updated, the entry is read from the VTOC, updated, and rewritten. This update consists of moving information from the user supplied parameter list to the DSCB, and is done only for an output data set. It is necessary to enqueue on the VTOC during the updating operation. The area of core occupied by the DEB, which was constructed when the data set was opened, is freed, and the data management count for the device is decremented by one. This last operation releases OSEDIT's claim on the device which was required to prevent the volume from being dismounted while OSEDIT was processing the data set. COM then dequeues from the data set to allow other jobs to access the data set.

END OF VOLUME

The End of Volume function is performed by COM for the purpose of increasing the space allocated to a data set. The operating system allows a data set to consist of up to sixteen extents, each representing a block of contiguous storage on the direct-access device. The normal operating system routines for extending a data set are an integral part of OS data management, and cannot be entered from

OSEDIT. COM, therefore, implements an almost equivalent function for OSEDIT routines. EOV shares many functions with OPEN and CLOSE since it is necessary to build a new DEB for the extended data set.

COM proceeds with EOV by first allocating and initializing the standard work area. The DSCB for the concerned data set is read into the work area and the old DEB is freed. If there are already sixteen extents in the data set, or if no secondary quantity was specified for the extension when the data set was created, COM proceeds as it would for CLOSE, and returns with an error indication.

The space allocation SVC is issued to obtain space for the new extent. This space is always requested in a contiguous block; consequently, COM will always add one extent. This is different from the operating system extend function which allocates up to three extents totalling the required amount if there is not a sufficiently large block of space to satisfy the request. COM does not follow this convention mainly for programming convenience and reliability, but the redefinition of this rule should not normally cause any problem when operating on small source data sets.

The space allocated by the allocate SVC represents a new data set on the volume. OSEDIT reads this VTOC entry and merges the space into the VTOC entry for the data set being extended. The VTOC entry for the data set just created is zeroed (ie. the data set ceases to exist) and the entry for the data set being extended is rewritten. This merging of space is actually somewhat more complicated than it sounds because a data set may have one or two VTOC entries, depending on the number of extents it has, and because it is necessary to update information in the DSCB describing the VTOC extent.

Extend finalizes by building a new DEB for the data set which contains the new extent information.

III. OSEDIT INPUT/OUTPUT OPERATIONS

The design criteria for OSEDIT make the input/output operations provided by the normal access methods too inefficient for the retrieval and writing of source data set records. This section describes how these operations are done in terms of the channel programs [5] built to do them. Additionally, the work data set, in which OSEDIT checkpoints changes as they are entered online, is described.

READING FROM A SOURCE DATA SET

OSEDIT must be able to read data sets with small or large blocking factors. Clearly, the normal access method technique of reading one physical block into a buffer has limitations. For example, consider scanning 1000 80-byte source records in a data set on the IBM 2314 disk-storage device [3]. If the records are completely unblocked, the records will span 25 tracks. If the records are read one at a time, the device will do a minimum of 1025 rotations (about 25 seconds). If the records are blocked at the maximum amount permissible (32K-byte blocks), 11 tracks will be required, and the data can be read in a minimum of 13 rotations (about a quarter of a second). This method would require a 32K-byte buffer, however, which is larger than the total core requirements of OSEDIT. For small block sizes, we wish to read several physical records to save time; for large block sizes (eg. such as those which occur with track overflow [6]), we wish to read a partial block to save buffer space. OSEDIT builds channel programs which do exactly this. Regardless of what the physical blocksize is, OSEDIT builds channel programs which fill a buffer which is a constant size for any data set. This buffer is referred to as a logical buffer and its length is referred to as a logical blocksize. Note that the logical blocksize will seldom be equal to the physical blocksize of the data set. Continuing the above example, if a 2K-byte logical buffer is used, the reading can be done in a minimum of 65 rotations for the unblocked data set and 51 rotations for the blocked data set (one or two seconds).

This scheme also greatly simplifies the problems of buffer control, directory maintenance, etc., because to modules other than the I/O module, all data sets have the same logical block size.

The I/O module for reading data sets, called READBLK, accepts as input a logical disk address which is composed of a physical disk address and a displacement into that physical block. It then constructs a channel program which will fill the in-core buffer. This channel program consists of the following parts:

(1) A search sequence is built to position the device to the start of the physical block specified.

(2) A read data CCW with the SKIP bit on (to suppress data transfer during the read) is constructed if

the displacement in the logical address is non-zero. This dummy read will position the device to the required displacement into the physical block.

(3) At this point, a read data CCW is constructed to either read the rest of the physical block or to read sufficient data from it to fill the in-core buffer, whichever is the smaller of the two.

(4) A read count CCW is generated. The count will be used in case of problems as described below. This is the end of the channel program if sufficient reads have been generated to fill the in-core buffer.

(5) A read data CCW is generated to read either a physical block or a partial physical block. In the latter case, a read data CCW with the skip bit on is also generated to space over the end of the physical block if it has not already been reached. Step (4) is then repeated.

All CCW's above are generated with the multiple-track bit [2] on in the read CCW modifier fields. This permits the channel program to cross track boundaries without program intervention and to process track overflow data sets without any special programming. Note that the channel program generated in the cases of data sets with small block sizes will be very long (over one hundred CCW's in practice), but that the data transfer times will be commesurable with those for larger block sizes.

A number of error or exceptional situations can occur during the execution of such a channel program. Some of these are handled directly by the I/O supervisor in the operating system, and the rest by OSEDIT itself. We summarize these conditions below [6].

I/O Supervisor Error Recovery

Command Reject -- This results from an OSEDIT channel program attempting to cross a track boundary in a data set which does not span the entire cylinder. The protection check bit in the channel status word will also be on. The channel program is restarted if the operation is not violating the data set's boundaries.

Cylinder end -- System/360 hardware provides for automatic track switching within a cylinder, but will not allow a channel program without explicit seek operations to cross from the bottom of one cylinder to the top of the next cylinder. The I/O supervisor provides this operation in software by restarting the channel program in the next cylinder.

I/O errors for which retry is possible -- These are handled by the operating system. If retry fails, OSEDIT will not use the data set.

OSEDIT Error Recovery

End of file -- The last logical block returned by OSEDIT's READBLK routine may be a short block.

End of extent -- This is handled as either end of file (no further extents in the data set) or a new channel program is built to continue filling the in-core buffer from the next extent.

Incorrect length -- It is permissible to have short physical blocks in the data set being read. Their presence can be detected only by not suppressing the incorrect-length I/O interruption. This is because OSEDIT may be reading several blocks in one operation. The channel status word contains a residual length, but this applies only to the last block read. If the incorrect-length exception were masked, it would be possible for OSEDIT to accept a short block without knowing it. The short block is accepted and a new channel program is built to continue filling the buffer from the record following the short block. The physical address is known because of the read count CCWs in the channel program.

Upon successful execution of the block read, OSEDIT either returns indication of end of file or the address of the next logical block. This permits sequential or direct retrieval of logical blocks from the data set.

WRITING A SOURCE DATA SET

OSEDIT writes a sequential data set or a member of a partitioned data set in response to the SAVE command. For the sake of efficiency, it is desirable to minimize the number of I/O operations required to write the data set. The approach taken here is similar to that taken for reading a data set: a fixed-size buffer is filled in core and is written to the device. To support arbitrarily large block sizes, this operation requires a buffer whose capacity is that of a track on the device. It is not possible to write part of a physical block in the manner that it is possible to read one. Consequently, this module generally writes a track at a time.

The algorithm is similar, once again, to that used for reading a data set, because long channel programs are built. The essential CCW in the chain is the write count key and data CCW. Addresses for the records written are allocated using the standard IBM formulae for disk space [8].

There are no error or exceptional conditions which arise during the writing of a data set which concern OSEDIT, other than a permanent I/O error. This is a reflection of the fact that OSEDIT controls the format of an output data set in contrast to the fact that there is a lot of latitude in the format of an input data set.

WORK DATA SET ORGANIZATION

The OSEDIT work data set is allocated when a user signs on to the system, and is used to store changes as they are being applied to a source file. It is organized into two areas. The first track is reserved for writing a checkpoint record. This record is used only in the event of a system failure during a session to allow the session to be resumed later. The balance of the data set is used to hold source images which are either additions or replacements to be applied to the source data set.

The checkpoint area contains all information necessary to resume a session. It is automatically written to disk after every ten alterations to the change data set. The checkpoint area consists of the pointer structures detailed in Section IV along with the data set names and various flags necessary to restore the previous change data set. When a user signs on to OSEDIT, the work data set is allocated. If this allocation fails because the data set already exists, the cause is assumed to be that a previous session did not terminate normally. The checkpoint record can then be read and used to resume the session.

The additions and replacements area is accessed through a routine which keeps the records in that area blocked. This results in increased expense to add a record to a block, because the block must be read, updated, and rewritten. However, access to records in this area is greatly improved in the event that large blocks of data are added to the file. This will happen if the user is creating a file or making extensive additions to an existing file. It was felt that this is a normal occurrence and that good access to these records should be provided to help the data set scanning commands.

IV. CHANGE FILE ORGANIZATION

OSEDIT updates source data sets in the following manner. First a FETCH command is used to indicate which data set is being operated on. FETCH performs a number of operations related to checking authorization, opening the source data set, resetting buffer pointers, etc., but after this is done, FETCH builds a directory, the entries of which correspond directly to the logical blocks in the fetch data set (see description of READBLK routine for a definition of "logical block"). As the user edits the change data set, the directory is updated to indicate deletions and additions. A replacement requires no special treatment; internally it appears as an addition.

There are four entry points to the logic which maintains the superposition of the changes on the original fetch data set. These allow the command processors to be completely independent of the access method used to maintain the changes. These entry points are as follows.

SETL -- This entry point operates in much the same manner as its counterpart in the ISAM access method [6]. SETL accepts a pair of sequence numbers which comprise the range over which subsequent READFs will operate.

READF -- This routine is entered after a SETL operation in order to read the change data set. READF returns an end-of-file indication in the event that either the last card has been read or that the next card available has a sequence number which is greater than the upper limit specified in the SETL command.

DLT -- Deletion of cards is accomplished by giving DLT a range of sequence numbers over which deletion is to be done. A single card image is deleted by making both ends of the range equal.

ADD -- This routine allows insertion of new card images or replacement of existing ones. It accepts the card image and its sequence number as parameters. The card image is written into the work data set.

In order to explain how these routines work, it is first necessary to explain the organization of the directory.

There are two types of entries in the directory, Block Definition Elements (BDEs) and Addition ELements (AELs). A BDE is created for each logical block in the source data set by the FETCH command. There is also a "dummy", end-of-file BDE which is useful to terminate searches and for the case in which the source data set is empty.

```
+-----------------------------------+
|                                   |
|             BDE AREA              |
|        (allocated by FETCH)       |
|                                   |
+-----------------------------------+
|                                   |
|             AEL AREA              |
|  (allocated by editing commands)  |
|                                   |
+-----------------------------------+
|                                   |
|             FREE AREA             |
| (allocated for AEL's as required) |
|                                   |
+-----------------------------------+
```

ORGANIZATION OF OSEDIT DYNAMIC CORE

The BDEs are created by FETCH in a linear order, and can be searched quite easily to determine the logical block address in which a card with a given sequence number would be found. This is the key to random access of the change data set. Deletions and additions are indicated in the directory by AELs which are chained from the BDE for the logical block to which they apply.

```
+------------------------------------+
|                                    |
|  HIGHEST SEQUENCE NUMBER IN        |
|   LOGICAL BLOCK                    |
|                                    |
+------------------------------------+
|                 |                  |
|  FIRST AEL      |  LAST AEL        |
|  FOR THIS BLOCK |  FOR THIS BLOCK  |
|                 |                  |
+-----------------+------------------+
|         |                          |
|  FLAGS  |   DISK ADDRESS FOR       |
|         |   BLOCK (CCHHR FORMAT)   |
|         |                          |
+---------+       +------------------+
|                 |                  |
|                 |  DISPLACEMENT    |
|                 |  IN BLOCK        |
|                 |                  |
+-----------------+------------------+
```

FORMAT OF BLOCK DESCRIPTOR ELEMENT

The flag bits in the AEL indicate in which of four possible formats a specific AEL has been built. An addition (or replacement) is indicated by no flag bits in the AEL. The sequence number of the card and an index into the work data set are specified. For deletions, either a single card or a range of cards may be deleted by the presence of deletion AELs. The single card case is indicated by a $DEL1 flag, and of course, the sequence number of the deleted record is in the AEL. A range of card images is deleted by constructing AEL markers at the boundaries of the deleted range. The start and finish of the deleted range are indicated by the $DELON and $DELOFF flags, respectively.

```
+-----------------------------------+
|                                   |
|     SEQUENCE NUMBER               |
|                                   |
+-----------------+-----------------+
|                 |                 |
| DISPLACEMENT TO | DISPLACEMENT TO |
|  NEXT AEL       |  PREVIOUS AEL   |
+--------+--------+-----------------+
|        |                          |
|  FLAGS |   LOCATION IN WORK DATA  |
|        |    SET OF ADDITION       |
+--------+--------------------------+
```

FORMAT OF ADDITION ELEMENT

The AELs which apply to a particular logical block of the data set are chained in sequence number order from the BDE for that block. It is clear, then, that the process of sequentially retrieving records from the change data set is one of merging the fetch data set with the card images in the work data set. The AELs serve to access the records in the work data set, and to indicate the deleted portions of the FETCH data set.

Considerable effort has been expended to ensure that AEL's are not created needlessly, because of the requirement of doing updating in a fairly small amount of core storage. Thus, a $DEL1 element will not be constructed if either the referenced card image does not exist in the source data set, or if the image is already deleted, or if deletion can be done by removing an addition AEL. Similarly, overlapping deletion ranges are merged to minimize the number of $DELON and $DELOFF elements. If an AEL for a card already exists, that AEL will not be reallocated.

The core storage used for BDEs and AELs is shared from one area. The size of this area dictates the ultimate limit on the size of source data set which can be processed or the total number of changes that can be applied to a data set. If the entire area is devoted to BDEs, we have an upper limit on the size of data set that can be FETCHed. This upper limit is based on the fetch data set logical blocksize and the number of 16-byte BDE entries that can be stored. One BDE entry is created for each logical block in the data set. The number of logical blocks is equal to:

$$\lfloor NREC * LRECL / LBSIZE \rfloor$$

where

NREC = the number of records in the data set,

LRECL = the logical record length of the data set,

LBSIZE = the logical buffer size, and

[ ] denotes the integer part of a number.

For example, in a 4800-byte pointer storage area it is possible to store 300 BDEs. If the fetch data set buffer is 2000 bytes and 80-byte card images are being processed, the logical blocking factor is 25. This allows a 7500 card data set to be FETCHed.

Similarly, the number of AEL's which can be allocated limits the number of changes which can be put in the change file, prior to each save. An AEL occupies twelve bytes. Using the figures in the above example, it can be seen that the 4800-byte pointer area will hold 400 AELs. This means that if source is being entered to create a new data set, 400 lines may be entered before a SAVE operation is done. If changes are being applied to a 2000-card data set, 80 logical blocks are in the data set. This requires 1280 bytes of BDE space, leaving 3520 bytes for AEL allocation, or space enough for 293 AEL's.

V. COMMAND PROCESSING LOGIC

All commands in OSEDIT are entered into a central command scanner which does a simple lexical analysis of the command entered at the terminal. The actual command processor has this input available to it in addition to the services of the OSEDIT access method plus several other miscellaneous service routines. Many command processors are quite short as a result.

COMMAND SCANNER

The command scan routine processes positional and keyword operands along with the delimiters which separate them. The following rules apply to command syntax.

(1) Operands are separated by blanks or commas. A missing operand may be indicated by consecutive commas.

(2) Keyword operands have the format 'KEYWORD=VALUE'. Thus the sequence 'DSN=A.B.C' specifies 'A.B.C' as the 'DSN' keyword value. Keyword operands may be specified in any order, except that if one is specified twice, the second occurrence overrides the first.

(3) Positional operands are scanned and passed to the command processor with the indication of the order in which they occurred. The first occurrence of a positional operand which scans successfully as a sequence range is passed to the command processor as a sequence range. It does not appear as a positional operand as well.

(4) A sequence range operand is either a single sequence number or two sequence numbers separated by a slash (/). If the sequence number preceding the slash is omitted, the smallest sequence number

is assumed (ie. 00000000) as the start of the range. If the sequence number following the slash is omitted, the largest sequence number is assumed (ie. 99999999) as the end of the range. A slash alone represents the entire range of possible sequence numbers.

(5) A sequence number is specified scaled by ten thousand to allow a large inter-record sequencing increment to be used without the penalty of typing low-order zeros. Thus 2.3 represents the actual sequence number 00023000.

(6) Operand values may be specified in one of several formats.
   (a) A string of characters excluding parenthesis, commas, single quotes, and space characters is converted to upper case and is passed as such.
   (b) A string of characters surrounded by single quotes allows parentheses, commas, and space characters to be passed. A single quote may be specified as two consecutive single quotes. Alphabetic characters are converted to upper case if the data set is being processed in upper-case mode.
   (c) The three modifiers U, L, and X may be followed by a quoted string which is to be translated to upper case, lower case, or hexadecimal digits, respectively. This facility allows, for example, insertions of lower case characters into source data while using a terminal which has no lower case facility.
   (d) The modifier M allows a mixture of upper or lower case characters and hexadecimal digits. A hexadecimal digit is entered as an ampersand (&) followed by the two characters which comprise the digit. An ampersand is entered as two consecutive ampersands in this format.

(7) Abbreviations of command names and keywords are allowed by permitting truncated versions of the full spellings. This usually means that only one or two characters need be entered to provide an unambiguous abbreviation for a word.

Keyword values are edited by routines associated with each keyword. This permits keywords whose values are in the form of an operand list. The attempt here was to make certain keywords appear exactly as their OS/360 Job Control

Language [4] counterparts (eg. SPACE and DCB operands). Although some of these may have a rather perverse format, it was felt that most terminal users are also JCL users and are likely to remember the JCL formats.

DATA SET ORIENTED COMMANDS

There are a number of commands which require that a data set name be supplied to the command processor. Notably, the FETCH and SAVE commands, which effect the reading and writing of data sets being edited, fall into this category. Also provided are the ALLOCATE and SCRATCH commands which create and destroy disk data sets.

FETCH COMMAND PROCESSOR

FETCH has been discussed to some degree in the sections on the OSEDIT access method. Hence we will not redescribe here the data structures that FETCH builds while operating as part of the access method.

FETCH is entered from the command scanner. It first determines if a fetch data set is already open because of a previous FETCH. If so, this data set is closed immediately. A service routine to both FETCH and SAVE, namely DEFDSN, is entered to validate the operands given in these commands, and which reads the VTOC entry for the target data set. The DCB attributes of this data set are overridden by those supplied by the terminal user (if any). The data set is opened by calling the COM module, described in Section II. If the data set is of the partitioned organization, a directory search is also performed. The data set is then read to build the directory for the OSEDIT access method. If sequence numbering is present on the source records, they are checked for validity and ascending sequence. If all of the above succeeds, the directory is flagged as being valid for subsequent commands. The FETCH command may fail because of such things as the data set not existing, the data set not being cataloged, the data set having bad DCB attributes, I/O errors, etc. If one of these occurs, an error message is produced and the directory is marked as containing invalid information.

It is also possible that the user wishes to create a new data set, rather than to edit an existing one. This is facilitated by means of the CLEAR option in the FETCH command. The processing in this case consists only of building an empty BDE directory.

SAVE COMMAND

SAVE writes an OS data set from the change data set. The channel programs built have been discussed in the OSEDIT data management chapter. The goal is basically to write one track at a time to the output data set. SAVE is a fairly complicated operation: it is organized into six internal routines which have been designated as SAVE1 through SAVE6.

SAVE1 uses the subroutine common to FETCH and SAVE, DEFDSN, to read the save data set VTOC entry [7] and merge DCB attributes. It also if the SAVE operation is directed to the FETCH data set which is currently opened because of a previous FETCH operation. If this is the case, special processing is required. In the case of a partitioned data set, it is necessary only to get exclusive access to the data set as opposed to the shared access requested by FETCH. In the case of a physical sequential data set, the problem is much greater because the changes would, in general, overwrite information which is part of the change data set (ie. the fetch data set). This problem is remedied by allocating a new data set in which to perform the actual SAVE.

SAVE2 opens the data set and performs initialization functions. A buffer pool is built, based on the block size of the save data set. The OS/360 device table [8] is consulted to get device information required to do track balance calculations in SAVE4. The address of the first write operation is calculated, and will either be the start of the data set or the next available position in the data set. Control then passes to SAVE3.

SAVE3 allocates and fills single buffers which will be written to the save data set. The minor editing functions that are available in SAVE are performed at this point. When a buffer is full, SAVE4 is called to schedule it to be written. SAVE3 terminates by passing control to SAVE6.

SAVE4 is passed buffers from SAVE3. It has the problem of assigning a record address to the block being scheduled. In doing this assignment, SAVE4 queues blocks until a complete track of information has accumulated. If the block passed from SAVE3 fits on the track currently being built, a channel program to write to this track is modified to write this block. Should the record not fit on the present track, SAVE5 is called to write the track using buffers queued because of previous calls to SAVE4. When control returns to SAVE4, the current block will be scheduled as record one on the next track of the data set. In allocating the next track to be written, the end-of-volume function of COM might be used to get additional space in the save data set.

SAVE5 starts the channel program that has been built. It then returns to the buffer pool the buffers that were written.

SAVE6 terminates SAVE processing. A STOW [6] is issued to update a partitioned data set directory. The data set which was SAVEd into is closed, updating the VTOC entry's last record written information.

ALLOCATE COMMAND

The ALLOCATE command issues a request via the space management SVC [2] to effect the creation of a new data set. The parameters to the SVC are the address of a Job File Control Block (or JFCB [7]) describing the data set which is to be allocated, and the address of the UCB for the device on which the data set is to be allocated.

SCRATCH COMMAND

The SCRATCH command uses the operating system's SCRATCH SVC to accomplish deletion of a direct-access data set.

EDITING COMMANDS

OSEDIT supplies commands to change data residing in the change data set. All of these commands accept either a single sequence number, or a range of sequence numbers, for which the indicated operation is performed.

DELETE COMMAND

DELETE uses the OSEDIT access method to remove card images from the change file. Most of the work involved in the deletion is done by the access method (see section III).

LIST COMMAND

LIST allows records of the change data set to be displayed. A small amount of formatting is done in the command processor.

SCAN COMMANDS

Four OSEDIT commands cause the change data set to be scanned for the occurrence of specified phrases. The CHANGE and ACHANGE commands permit replacement; the FIND and AFIND commands only list records in which the phrases occur. The CHANGE and FIND commands operate on all occurrences of the phrases in the supplied range; ACHANGE and AFIND stop after the first occurrence.

Each of these commands accepts multiple phrases which may be searched for in the change data set. Because it is possible to specify a large range, these commands must operate fairly efficiently in order to provide adequate response time. Consequently, a translate-and-test table is built which will allow stopping on the first characters of the supplied phrases. The function value stored in the table indexes a list of phrases being sought which start with that first character. After each first character match found by the translate-and-test instruction, the remainder of the characters in each phrase which start with this character is compared against the source for a match.

The CHANGE and ACHANGE commands follow a successful match by a replacement operation. Because the replacement field may have a length different from that of the field being replaced, it is necessary to construct the changed image in a second buffer. After all possible changes have been made, this card image is replaced in the change data set.

AUTOLINE COMMAND

In order to facilitate adding large blocks of source statements, the AUTOLINE mode is provided. In this mode, the user is prompted with sequence numbers to which the user responds by entering card images. AUTOLINE uses the access method to add the cards at the indicated sequence positions. AUTOLINE mode is terminated by entering a null line.

Changes of mode on a terminal system are usually somewhat inconvenient to a terminal user. This is because the mode change implies that the user has lost some of the facilities available to him before the mode transition. To lessen this annoyance, OSEDIT allows a normal command to be entered in AUTOLINE mode by entering a dollar sign as the first character on the line. If the line cannot be parsed as a command, it is added to the data set.

SINGLE CARD INSERTIONS

A single card may be inserted by entering a sequence number followed by the card image. The card image is right-padded with blanks and is inserted into the change data set.

VI. MUM INTERFACE

OSEDIT has been incorporated under the Manitoba University Monitor (MUM) [9] system as one of its application programs. This results in an additional 15K bytes of code being core resident and a further 9K bytes for buffers. It is possible to generate a MUM system including OSEDIT with a total core requirement of approximately 65K bytes. This is extremely reasonable since comparable systems today often require three to six times this amount of core.

MUM manages the terminal input/output operations and provides a "roll area" for each user. The contents of this roll area reside on disk storage while the user is inactive (eg. waiting for terminal input/output operations to complete). OSEDIT is completely reentrant, and maintains all status concerning a given user in the roll area.

MUM makes no attempt to time-slice the use of the roll area. Instead, an application program, such as OSEDIT, must issue a PAUSE request to MUM to indicate that it may be swapped to disk if there is a queue for the roll area. This is done by OSEDIT during the processing of commands which might be excessively long in duration.

The SAVE command, being by far the most expensive command in terms of core storage and disk operations, relies on an enqueue facility added to MUM to queue SAVE operations. SAVE shares the roll area during a SAVE operation with other users by issuing the PAUSE request; however, only one SAVE operation will be in progress at any time among users. This restriction can be eased by allowing several buffers for SAVE.

In summary, MUM and OSEDIT co-exist quite well since both have been designed to achieve high performance in a small amount of core. The interfacing was accomplished in a very small amount of time, since only the PAUSE request and the enqueue facility had to be added to MUM.

Another area which had to be handled in the MUM environment is that of data set security. MUM account numbers are numerous and easily acquired in our installation. This poses the question of how to limit the facilities offered by OSEDIT to protect against accidental or malicious damage to data sets on the system. This must be done with the realization that the operating system itself offers very little facility in this area (as one might expect).

It was decided that a fairly flexible scheme of access was required because of the great variety of users on MUM. A small field that is maintained on the accounting file is used to indicate what facilities of OSEDIT are available to a given account number.

The information in this accounting field classifies an account number in three ways. First, there is an indication of whether or not the account number may use OSEDIT. This allows the installation to totally deny access to a given account number. Secondly, there is an indication of what list of volumes the account number may access. The actual lists of volume serial numbers are maintained in OSEDIT and may be modified through a small assembly or by usual system's means. The idea here is that volume access privileges usually apply to a group of users, rather than to a specific user. The "zeroth" list allows for unregulated access. This type of access would be given to systems programmers, for example. Thirdly, the type of access allowed is specified. There are four levels of privilege associated with this specification. These are summarized in the following table.

USER TYPE	I	II	III	IV
SYSTEM DATA SET	W	R	R	R
USER'S DATA SET	WF	WF	WF	WF
OTHER DATA SET	WF	WF	RF	none

R -- read access only
W -- read/write access
RF -- read only access, if user can use volume
WF -- read/write access, if user can use volume

The table shows that any user may read any system data set, but only a type I user (eg. a system programmer) would be allowed to write into a system data set. Similarly, any user may have read/write access to his own data sets. The classification of user type indicates whether or not access is allowed to someone else's data set, and if access is permitted, whether or not writing is permitted.

It is expected that the majority of users will be of type II, with access to only a specific list of volumes. This type of classifciation is well suited to the situation in which a given group of people use an entire volume.

VII. CONCLUSIONS

OSEDIT has been implemented and compared with the IBM supplied editing package for TSO. OSEDIT offers few improvements in the editing facilities actually supplied: the major difference is in the response times realized by both systems under heavy loading conditions. OSEDIT appears to reduce response time to approximately one-quarter of that given by TSO EDIT. This is most appearant on FETCHes and SAVEs.

Most of the improvement in response time can be attributed to the highly efficient channel programs that OSEDIT uses. Similar approaches have been used in the past to achieve the same end results: eg. HASP builds long channel programs to control unit-record devices. The central idea is to get best possible use out of the channels and the devices attached to them. This approach tries to optimize usage of the hardware without placing any restrictions on how the device is used. to change the usage patterns of the hardware.

A totally different way to attack the problem would be to change the physical organization of the data on the device to allow efficient updating using considerably simpler channel programs. Thus, rather than attempting to make channels work harder, one chooses an organization that is appropriate to the problem. Unfortunately, this is almost impossible to do under the present operating system, because there is no facility to insert a user-specified access method between data and an arbitrary program that processes that data. Without this facility, one cannot easily have an assembler or compiler process the file in a sequential manner.

It is to be hoped that with the current interest in "structured programming" and "software engineering" will come a second generation of software that will influence the design of IBM's operating systems. The term "modular programming" applied to OS denotes the fact that the operating system was written in one-kilobyte transients. It does not in any way mean that one section of code handles one function, making it easy to redefine or to expand that function. The OS logic for OPEN/CLOSE/EOV is very poorly designed and implemented, and is therefore almost impossible to alter. The potential market in data base systems has caused some changes in this area (eg. the introduction of VSAM as a new access method), but has not restructured data management. Hopefully an OSEDIT approach to providing access to data will not have to be used within a few years.

A second major area, which OSEDIT did not address, is that of providing a sophisticated command language. The command language provided does little more than to provide an interface between the user and the OSEDIT access method. It has the advantage of being extremely easy to learn and use, and fulfills many of the common editing requirements.

However, we do admit that in the design of OSEDIT too much time was spent developing facilities to do functions which should have been supplied by an operating system.

We do not argue that a command language should be arcane and difficult to learn, but that it should be more powerful than that facility which was provided in OSEDIT. There are two good reasons for this. First, a powerful command language enables a user to quickly specify exactly what he wants done. Although the execution of a given command might be expensive, savings will be realized in lower line connection times, lower swapping loads, elimination of human errors, etc. Secondly, we contend that a relatively small subset of the total users of an editing system will provide most of the usage on the system. In other words, we claim that there are "everyday" and "occasional" users of a system. Very often salesmen, administrators, Data Processing managers, and other marginally interested persons belong to the second group. This has lead to the requirement that such systems be designed for the occasional user only. We claim that such reasoning is every bit as invalid as would be that of designing a system which required a long time to master.

An effective command language would be composed of at least two parts: an index system and a data parser. We will consider these two separately.

An index system should provide many aids to a programmer in the area of data housekeeping. The OS utility programs have remained essentially constant since the early releases of the operating system, and do not provide sufficient support for a terminal user (or anyone else, for that matter). It is ludicrous that a terminal user is forced to create and submit a batch job to obtain a listing of his data set on the high speed printer. The index system mentioned would prepare such a job for the user as well as handle all utility functions. For example, it could also handle backups on demand, accounting for disk space, release unused space in data sets, provide annotation of the contents of data sets, provide a certain amount of security, etc. In short, the objective is to relieve both the programmer and the installation of the chore of preparing and running utility programs.

The data parser provides commands to edit within a given data set. Just how this section of the command language should be designed is not immediately obvious. We have simulated a few possibilities by means of a SNOBOL [1] program, but it seems very difficult to resolve a number of questions about the design. For example, consider the differences between an editor which operates on line numbers (eg. OSEDIT) and one which operates by scanning for a given context (eg. TSO EDIT with TEXT files). The line number is to be preferred when it is difficult to specify context, eg. when locating one of several END statements in a PL/1 program. However, line numbers are not at all natural or convenient when editing English text. The solution is probably to allow both means of specification, and maybe a few more.

Having studied several source editing packages, we would conclude that none of them have really solved the problem which they set out to solve. We have also concluded that the major reason for this is the great dependency that an editor, and indeed, every program run on a machine, has on the operating system and its data management components. OS/360 does not provide a good basis; hence any attempt at an editor will be at an immediate disadvantage.

APPENDIX A

OSEDIT USER'S GUIDE

INTRODUCTION

OSEDIT is a utility program available under MUM for the purpose of editing OS/360 data sets online. The commands available under OSEDIT fall basically into two varieties: those involving data management operations (eg. allocating a new data set on disk), and those which are used to edit data. Because many of the commands have similar operands, this introduction will describe some of the operands available.

The syntax for describing a command consists of the command name, followed by a list of possible operands for this command. Command names have long and short forms: the shortest form is underlined in this description. If an operand is optional, this will be indicated by enclosing the operand description in brackets [ ]. Operands can be of two types, keyword or positional. A keyword operand has an equal sign imbedded in it (eg. DSNAME=XXX). The order in which keyword operands appear in the complete command is irrelevant. Operands are delimited by either commas or blanks. If an operand contains one of the punctuation characters comma, equal sign, parenthesis, or single quote, this operand must be enclosed in single quotes. If a single quote appears in such a quoted string, it must be represented by two single quotes.

The syntax of certain operand fields follows.

dsname - represents a data set name with an optional member name enclosed within parentheses following it. If dsname starts with a period, a user supplied prefix will be prefixed to the dsname.

volume - represents a volume (or disk-pack) name. Volume names are six characters in length.

space - represents a space parameter for data set allocation. This parameter is coded in exactly the same manner as that in Job Control Language [4].

dcb - represents the DCB operand, and is coded in exactly the manner as in Job Control Language. Four sub-operands are available, namely LRECL, BLKSIZE, RECFM, and DSORG. These may be abbreviated to their first letters.

seq-no - represents a sequence number range. OSEDIT sequence numbers are specified in fractional form, with a decimal-point assumed in the middle of the eight-digit sequence number. Thus, 23.5 represents the sequence number 00235000. A range may be used in place of a single sequence number in all commands. A range is specified by specifying two sequence numbers separated by a slash. For example, 23.5/43.002 represents all sequence numbers in the range 00235000 to 00430020. Either end of the range may be omitted so that /5 represents 00000000 to 00050000, 5/ represents 00050000 to 99999999, and / represents 00000000 to 99999999 (ie. the entire data set). Sequence numbers in OSEDIT are kept from command to command, and do not have to be respecified if the range of operation does not change.

prefix - is used to specify a prefix that is applied to DSNAMEs.

column - is used to specify column limits over which scan commands will operate. For example, (3,45) denotes columns 3 to 45.

DATA SET COMMANDS

ALLOCATE - creates a new disk data set.

```
ALLOCATE  DSNAME=dsname
          VOLUME=volume
          SPACE=space
          [ DCB=dcb ]
```

eg. AL DSN=RUGGER.XXX,VOL=TS0001,SPA=(TRK,(5,2))

This command creates a data set on the disk pack TS0001 with the name of RUGGER.XXX. The data set allocated will have five tracks as a primary extent, and a secondary allocation quantity of two tracks. No DCB information will be supplied.

SCRATCH - deletes an existing data set.

```
SCRATCH   DSNAME=dsname
          VOLUME=volume
```

The data set identified in this command is permanently and irretrievably removed from disk. Both the DSNAME and VOLUME parameters are required.

FETCH - prepares a data set for updating.

```
FETCH     [ DSNAME=dsname ]
          [ VOLUME=volume ]
          [ DCB=dcb ]
          [ RENUM ]
          [ NONUM ]
          [ CLEAR ]
          [ LOWER ]
          [ UPPER ]
```

RENUM - data set is to be renumbered.
NONUM - data set is unnumbered.
CLEAR - used to FETCH an empty data set.
LOWER - terminal input will have lower case.
UPPER - terminal input will have no lower case.

The FETCH command must be used to retreive a data set that is going to be edited. If a data set is being created, the CLEAR option is specified in the FETCH command to indicate that no data set is to be FETCHed. The LOWER and UPPER commands have no effect on the data set being edited: they apply only to typewriter entries. Lower case items in the fetch data set will print as lower case in both LOWER and UPPER modes of operation. Note that UPPER is the default mode of operation.

SAVE - writes current work data set to disk.

```
SAVE      [DSNAME=dsname]
          [VOLUME=volume]
          [DCB=dcb]
          [seq-no]
          [RENUM]
          [NONUM]
          [UPPER]
```

DSNAME - required if not FETCH dsname.
seq-no - allows SAVEing only part of the data set.
RENUM - renumbers records in written data set.
NONUM - causes sequence number fields to be blank
UPPER - translate SAVEd data set to upper case.

SAVE is used to make the changes entered at the terminal permanent. If no data set is specified, the data set from which the previous FETCH operation was done is used. A certain amount of editing is possible during a SAVE operation on the sequence number fields. The RENUM option renumbers the sequence field, while the NONUM option overlays the sequence number field with blanks. The UPPER option translates all lower case characters in the data set to upper case characters. A range of sequence numbers may be specified to limit the amount saved. This has the effect of deleting all cards whose sequence numbers do not lie in the specified range. A data set may thus be broken up into several smaller data sets according to sequence number.

SET - set processing defaults.

```
SET       [PREFIX=prefix]
          [DSNAME=dsname]
          [VOLUME=volume]
```

The SET command allows a default data set name to be specified. This name will be used in all commands for which the data set name is optional. A default volume may also be specified. If it is, and no VOLUME operand is specified in a command for which this operand is optional, data set searches will be directed to this volume and then to the system catalog. A data set name may be prefixed by setting a prefix with the SET command.

Thereafter, all DSNAME operands which start with a period will have the prefix applied to them.

EDITING COMMANDS

LIST - list elements of work data set.

```
LIST        [seq-no]
```

The LIST command lists records from the work data set to the terminal. This allows the data set to be inspected before and during editing operations.

DELETE - delete records in work data set.

```
DELETE      [seq-no]
```

CHANGE - edit work data set.

```
CHANGE      [seq-no]
            scan-item replacement-item
            [scan-item replacement-item . . .]
            [COLUMN=column]
```

eg. C 3/5 ABC X 'M N' QQQQ COL=(2,56)

denotes change all occurrences of ABC to X and all occurrences of MbN to QQQQ in columns 2 to 56 of cards in the range 00030000 to 00050000 inclusive.

ACHANGE - changes first occurrence of character string.

```
ACHANGE     [seq-no]
            scan-item  replacement-item
            [COLUMN=column]
```

This command is identical to CHANGE, except that it stops after making one change.

FIND - scan work data set.

```
FIND        [seq-no]
            scan-item
            [scan-item . . .]
            [COLUMN=column]
```

The FIND command is useful for scanning a data set which is either not line numbered, or is unfamiliar to the user. Note that more than one item may be specified as scan items. Every card in the sequence range which contains any of the specified scan items is listed on the terminal.

AFIND - scan work data set for first occurrence.

```
AFIND       [seq-no]
            scan-item
            [scan-item]
            [COLUMN=column]
```

The AFIND command is identical to FIND, except that the first sucessful match of a scan item to a source record terminates the AFIND command.

AUTOLINE - supply sequence numbers for input.

```
AUTOLINE  start/incr
```

start - first sequence number generated.
incr - increment between sequence numbers.

In AUTOLINE mode, a line beginning with $ is treated as an ordinary OSEDIT command. AUTOLINE mode is not left during execution of this command. If a line begins $$, one of these $'s is deleted, and the resulting card image is entered into the work data set. AUTOLINE mode is terminated by entering a null line.

END - terminates OSEDIT session.

```
END
```

APPENDIX B

COM DSECT

The following assembler DSECT shows the parameter list used for calling COM, the OSEDIT data management interface.

```
COP      DSECT ,
COPBLKSI DS    H                  DATA SET BLOCKSIZE
COPLRECL DS    H                  DATA SET LRECL
COPIOB   DS    8F                 IOB
COPSEEK  DS    2F                 SEEK ADDRESS IN IOB
COPDEBP  DS    F                  POINTER TO DEB
COPDSN   DS    CL44               DATA SET NAME
COPVOL   DS    CL6                VOLUME NAME
COPMEM   DS    CL8                MEMBER NAME
COPDSORG DS    X                  DSORG
COPRECFM DS    X                  RECFM
COPTIONS DS    X                  OPTIONS TO COM:
$WRITE   EQU   X'80'               OPEN FOR OUTPUT
$OPEN    EQU   X'40'               DO OPEN
$NODATE  EQU   X'20'               OMIT DATE CHECK
$EXTEND  EQU   X'10'               DO EOV OPERATION
$NOENQ   EQU   X'08'               NO DSN ENQUEUE
$READ    EQU   X'00'               OPEN FOR INPUT
$CLOSE   EQU   X'00'               DO CLOSE OPER
COPLSTAR DS    CL3                LAST BLOCK IN DS
COPTRBAL DS    H                  TRACK BALANCE
COPDIRCN DS    X                  DIRECTORY COUNT
```

APPENDIX C

SAMPLE TERMINAL SESSION

The following is a sample of how the user interacts with OSEDIT. The material on the left represents OSEDIT commands; that on the right is documentation.

fe dsn=rugger.xxx.data	The FETCH command specifies what data set is to be edited.
l /	List the data set. The slash with no operands indicates that the entire data set is to be listed.
00010000 X: PROC; 00020000 DCL I EXTERNAL; 00030000 I = I + 1; 00040000 END;	Listing of the data set.
2.1 dcl j external;	Add card image 2.1, or actually 00021000 since OSEDIT scales sequence numbers.
c 3 1 j 00030000 I = I + J;	The CHANGE command instructs OSEDIT to change all occurrences of 1 to J on card image 00030000. The changed image is printed for verification.
save	Save the data set back.
end	End the session.

BIBLIOGRAPHY

1. Griswold, Poage, and Polonsky, "The SNOBOL4 Programming Language", Printice-Hall, 1971.

2. IBM System 360 SRL, "Component Descriptions: 2314 Direct Access Storage Facility", Form No. A26-3599.

3. IBM System 360 SRL, "Direct Access Device Space Management", Form No. GY28-6607.

4. IBM System 360 SRL, "Job Control Language", Form No. C28-6704.

5. IBM System 360 SRL, "Principles of Operation", Form No. GM22-6821.

6. IBM System 360 SRL, "Supervisor and Data Management Services", Form No. GC28-6646.

7. IBM System 360 SRL, "System Control Blocks", Form No. GC28-6628.

8. IBM System 360 SRL, "System Programmer's Guide", Form No. C28-6550.

9. Bill Reid, Master's dissertation, University of Manitoba, 1972.