LEVEL 68

MULTICS PROGRAMMER'S MANUAL

SUBJECT

Introduction to Programming in the Multics Operating System Environment

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics operating system. This information can be found in the 2-volume set, *New Users' Introduction to Multics*.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

AG90-02

May 1980

Honeywell

PREFACE

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. The MPM consists of the following:

Reference Guide	Order No. AG91
Commands and Active Functions	Order No. AG92
Subroutines	Order No. AG93
Subsystem Writers' Guide	Order No. AK92
Peripheral I/O	Order No. AX49
Communications I/O	Order No. CC92

This document provides an introduction to programming in the Multics environment. It is directed at persons who wish to develop programs which take advantage of the special features of the Multics System. For example, accessing of storage system segments and protocols for writing programs to be used as commands are discussed.

This document assumes knowledge of programming, specifically, in the PL/I language, in which all of its examples are coded. Only knowledge of (ANSI) standard PL/I is assumed; PL/I idioms peculiar to Multics are pointed out and discussed in the sample programs as they are encountered. The PL/I language as implemented on Multics is defined by the Multics PL/I Language Specification Order No. AG94.

Throughout this manual, the term Multics is used to refer to the Multics operating system.

Some general familiarity with the fundamental concepts and facilities of the Multics system is assumed as a prerequisite to this material. This information is available in the following publications:

New User's Introduction to Multics, Part I Order No. CH24
New User's Introduction to Multics, Part II Order No. CH25

CONTENTS

SECTION 1

PROGRAMMING IN THE MULTICS ENVIRONMENT

A programmer may, if he wishes, treat Multics as simply a PL/I, FORTRAN, APL, BASIC, COBOL, or Lisp machine, and contain his activities to just the features provided in his preferred programming language. On the other hand, much of the richness of the Multics programming environment involves use of system facilities for which there are no available constructs in the usual languages. To use these features, it is generally necessary to call upon library and supervisor subroutines. Unfortunately, a simple description of how to call a subroutine may give little clue to how it is intended to be used. The purpose of this document is to illustrate typical ways in which many of the properties of the Multics programming environment may be utilized.

The programmer choosing a language for his implementation should carefully consider the extent to which he will want to go beyond his language and use system facilities of Multics which are missing from his language. As a general rule, each of the Multics language implementations matches some well-known standard for completeness of that language (e.g., ANSI or IBM). However, in going beyond the standard languages, the programmer will find that Multics tends to be biased towards convenience of the PL/I programmer. For example, if a programmer plans to write programs which directly call the Multics storage system privacy and protection entries, he must supply arguments which are, in PL/I, structures. If he is writing in FORTRAN or BASIC, he has no convenient way to express such structures. Note that the situation is not hopeless, however. Programs which stay within the original language can be written with no trouble. Also, in many cases, a trivial PL/I interface subroutine can be constructed, which is callable from, say, a FORTRAN program, and goes on to reinterpret arguments and invoke the Multics facility desired. Using such techniques, almost any program originally prepared for another system can be moved into the Multics environment.

BASIC ADDRESSING TECHNIQUES

The most significant difference between the Multics programming environment and that of most other contemporary computer programming systems lies in its approach to addressing online storage. Most computer systems have two sharply distinct environments: a resident file storage system in which programs are created, and translated programs and data are stored, and an execution environment consisting of a processor (actually allocated in short time bursts) and a "core image", which contains the instructions and data for the processor. Supervisor procedures provide subroutines for physically moving copies of programs and data back and forth between the two environments.

In Multics, the line between these two environments has been deliberately blurred, so as to simplify program construction: most programs need to be cognizant of only one environment rather than two. This blending of the two environments is accomplished by extending the processor/core-image environment. In Multics, the share of the processor is termed a <u>process</u>, and the core image is abstracted into what is called an <u>address space</u>. Each user when he logs in is assigned one newly created address space, and a single process which can execute in it.

A Multics address space is not like the usual core image, however: it is larger, and it is segmented. A segment may be of any size between 0 and 256K 36-bit words and an address space may have a large number of segments -- a typical Multics process has about 200 segments. (The hardware places a limit of 4094 distinct segments, but table sizes in the current software limit an address space to a number closer to 2000.) Typically, each separately translated program resides in a different segment; collections of data which are large enough to be worthy of a separate name are placed in a segment by themselves.

The segment is also the unit of storage of the Multics catalogued file storage (the Multics storage system.) These two environments, distinct in many other systems, are automatically mapped together on demand, by the Multics virtual memory system. When a program already appearing in the current address space calls to another one which is not yet there, a linkage fault occurs, the supervisor locates the needed procedure, and maps it into the current address space, assigning it some as yet unused segment number. Similarly, data segments are mapped into the address space. This property eliminates the need for explicitly programmed overlays, chain links, or memory loads, and also reduces the number of explicitly programmed input and output operations.

In contrast to many other systems, this address space is retained throughout the login session, and its contents gradually are increased as different programs and data objects are accessed. (Facilities are also available for starting over with a new address space, or removing items no longer needed in the address space.) Finally, all supervisor procedures and commands called by the user are mapped into the very same address space. Thus, there is a great uniformity of access methods, to user-written programs, to data, to library or supervisor programs, and to items never before used but catalogued in the storage system.

As will be seen in the examples which follow, the effect of the mapping together of these two environments can range from the negligible (programs can be written as though there were a traditional two-environment system, if desired) to a significant simplification of programs which make extensive use of the storage system. We begin with seven brief examples of programs which are generally simpler than those encountered in practice, but which illustrate ways in which on-line storage is accessed in Multics.

1. Internal Automatic Variables. The following program types the word "Hello" on four successive lines of terminal output:

```
a:   procedure;
     declare i fixed binary;
     do i = 1 to 4;
          put list ("Hello");
          put skip;
     end;
     return;
     end a;
```

The variable i is by default of PL/I storage class "internal automatic": in Multics it is stored in the stack of the current process and is available by name only to program "a" and only until "a" returns to its caller. It is declared binary for clarity: although the default base for the representation of arithmetic data is binary, according to the PL/I standard, as well as in Multics PL/I, some other popular implementations have a decimal default. There is no need for decimal arithmetic in this program, and binary arithmetic is faster.

2. Internal Static Variables. The following program, each time it is called, types out the number of times it has been called since its user has logged in:

```
b:   procedure;
     declare j fixed binary internal static initial(0);
     j = j + 1;
     put list (j, "calls to b.");
     put skip;
     return;
     end b;
```

The variable j is of PL/I storage class "internal static"; in Multics it is stored in b's static section and is available by name only to program b. Its value is preserved for the life of the process, or until procedure b is recompiled, whichever time is shorter. The "initial" declaration causes the value of j to be initialized at the time this procedure is first used in a process.

3-4. External Static. Suppose we wish to set a value from one program and have it printed by some other program in the same process:

```
c:   procedure;
     declare z fixed binary external static;
     z = 4;
     return;
     end c;


d:   procedure;
     declare z fixed binary external static;
     put list (z);
     put skip;
     return;
     end d;
```

In both programs, the variable z is of PL/I storage class "external static"; in Multics it is stored in a particular segment where all such variables are stored, and is available to all procedures in a particular process, until the process is destroyed. External static is analogous to COMMON in FORTRAN, but with the important difference that data items are accessed by name rather than by relative position in a declaration. Multics calls such data items "external variables". There are commands (for example, list_external_variables) to list, reinitialize, and otherwise deal with all the external variables used by a process.

Each variable which is accessed in this form generates a linkage fault the first time it is used. Later references to the variable by the same procedure in that or subsequent calls do not generate the fault. A more complete discussion of dynamic linking appears in a later section of this document.

5. Direct Intersegment References. The following program prints the sum ofp the 1000 integers stored in the segment w:

```
e:   procedure;
     declare w$(1000) fixed binary external static;
     declare (i, sum) fixed binary;
     sum = 0;
     do i = 1 to 1000;
          sum = sum + w$(i);
     end;
     put list (sum);
     put skip;
     return;
     end e;
```

The dollar sign is recognized as a special identifier by the PL/I compiler, and code for statement 6 is constructed which anticipates dynamic linking to the segment named w. Upon first execution, a linkage fault is triggered, and a search undertaken for a segment named w. If one is found, the link is "snapped," which means that all future references will occur with a single machine instruction. The storage for array "w$" is the segment w.

If no segment named w is found, the dynamic linker will return to command level and report an error to the user. At this point, it is possible to create an appropriate segment named w, and then continue execution of the interrupted program, if such action is appropriate.

6. Reference to Named Offsets. The following procedure calculates the sum of 1000 integers stored in segment x starting at the named offset u:

```
f:   procedure;
     declare x$u(1000) fixed binary external static;
     declare (i, sum) fixed binary;
     sum = 0;
     do i = 1 to 1000;
          sum = sum + x$u(i);
     end;
     put list (sum);
     put skip;
     return;
     end f;
```

The difference between this example and the previous one is that segment x is presumed to have some substructure, with named internal locations, called "entry points". To initially create a segment with such a substructure, the compilers and assemblers are used, since information must be placed in the segment to indicate where within it the entry points may be found. Unfortunately, the PL/I language permits specification of such structured segments only for procedures, not for data. The create_data_segment command and create_data_segment_ subroutine (see the MPM Commands, Order No. AG92, and MPM Subroutines, Order No. AG93) are designed to be used to create such data segments. The ALM assembler can also be used for creating structured data segments.

7. External Reference Starting With a Character String. In many cases, a segment must be accessed whose name has been supplied as a character string. In those cases, a call to the Multics storage system is required in order to map the segment into the virtual memory and to obtain a pointer to it. The following program uses the supervisor entry hcs_$make_ptr to perform a search for a segment of a given name, identical to that undertaken by the linker in the previous examples.

```
g:   procedure(string);
     declare string character(*) parameter;
     declare hcs_$make_ptr entry (pointer, character (*),
          character (*), pointer, fixed binary (35));
     declare null builtin;
     declare p pointer;
     declare (i, sum) fixed binary;
     declare v(1000) fixed binary based(p);
     call hcs_$make_ptr (null (), string, "", p, (0));
     sum = 0;
     do i = 1 to 1000;
          sum = sum + v(i);
     end;
     return;
     end g;
```

The PL/I null string value ("") indicates that it is not a named entry point in the segment to which a pointer is wanted, but a pointer to its base. Perhaps the segment does not even have named entry points. The PL/I null pointer value (null()) and the zero passed by value ((0)) in the call to hcs_$make_ptr are relevant to its handling of error conditions and some of the parameters of the search for the segment. We will not deal with them here, although we will consider some of these issues in later sections. See the MPM Subroutines, Order No. AG93, for a full description of the hcs_$make_ptr subroutine.

Another method of accessing storage system segments is by means of the subroutine hcs_$initiate. When using hcs_$initiate, the pathname of the segment desired is specified directly. one directly specifies the path name of the segment desired: no search is undertaken for the segment as in the case of a linkage fault. This procedure differs greatly from the examples above, in which a search is involved. An intermediate situation, in which library routines are used to construct a pathname starting with an entry name, is found in the "simple text editor" example, which appears later in this book.

SECTION 2

DYNAMIC LINKING

A particularly potent programming tool of Multics is the dynamic linking facility. Dynamic linking consists of delaying the search for and mapping of a subroutine (or data segment) until the first call for that subroutine (or use of that data segment) occurs. Dynamic linking is accomplished by having the compiler leave in the object code of a compiled program a special bit pattern which, if used in an indirect address reference, causes a machine fault (trap) to the dynamic linker. The linker inspects the location causing the fault, and from pointers found there, locates the symbolic name of the program being called or the data segment being referenced. It then locates the appropriate segment, maps it into the current address space, and replaces the indirect word with a new one containing the address of the program or data entry point, so that future references will not cause a dynamic linking fault.

There are many ways in which dynamic linking can be used, but the following three are probably most significant:

- to permit initial debugging of collections of programs before the entire collection is completely coded.
- to permit a program to include a conditional call to an elaborate error handling or other special-case handling program, without invoking a search for or mapping of that program unless the condition arises in which it is actually needed.
- to permit a group of programmers to work on a collection of related programs, such that each one obtains the latest copy of each subroutine as soon as it becomes available.

The use of dynamic linking in program development can be shown by the following script. When the script starts, the program "k" and subprogram "y" have been written already and compiled.

```
 k: procedure;

      declare (x, y, z)               entry;
      declare i                       fixed binary;
      declare (sysprint, sysin)       file;

      put list ("Which option?");
      get list (i);
      if i = 1 then call x;
      else if i = 2 then call y;
      else if i = 3 then call z;
      else put list ("Bad option ");
      return;
   end k;

 y: procedure;
      declare sysprint                file;
      put list ("y has been called.");
      put skip;
   end y;
```

In these and all examples in this manual, typing by the user is prefaced by an exclamation point. The user does not type the exclamation point, nor did Multics. It serves only to distinguish typing by the user from typing by Multics.

Comments on the script are interspersed with the script itself, enclosed in square brackets.

[The program "k" is invoked by typing its name. The user calls for option 2, and the program "y" is called.]

```
! k
 Which option? !2
         y has been called.
 r 17:11 0.123 11
```

[The program ran even though two of the three subroutines it could call do not exist, because the subroutine it did need was in existence. Since linking is done on demand, and no demand for "x" or "z" occurred, their nonexistence did not keep the program from running.

In the next use of "k", the user asks for an option corresponding to the program "z," which doesn't exist.]

```
! k
 Which option? !3
 Error: Linkage error by >udd>States>Jackson>k|152 (line 11)
 referencing z|z
 Segment not found.
 r 17:11 0.283 90 level 2
```

[The attempt to call the nonexistent subroutine "z" failed. The linkage error handler has invoked a second, recursive invocation of command level, as indicated by the field "Level 2" in the ready message. The error message shows the full pathname of the program attempting to locate "z," and gives the name of the program that could not be found. The notation "z|z" means entry point "z" in segment "z." It is necessary to separate entry point name from segment name, since a PL/I program in a segment could have several entry points with different names.

Execution of "k" is suspended, since it cannot continue with the call. The user has the choice of giving up, or creating "z." The user invokes the qedx editor, creates the segment, and compiles it.]

```
! qedx
! a
! z: procedure;
!      declare sysprint file;
!      put list ("This is Z");
!      put skip;
!   end z;
! \f
! w z.pl1
! q
 r 17:12 0.382 48 level 2
```

[The source segment has been created, now it must be compiled to create a callable object segment.]

```
! pl1 z -table
 PL/I 25c
```

[Now that an object segment "z" has been created, the call from "k" can be restarted. This is done with the "start" command.]

```
! start
        This is Z
 r 17:12 0.166 27
```

[The program successfully finishes. It can now be run with option 3 without any additional intervention.]

```
! k
 Which option? !3
        This is Z
r 17:13 0.075 18
```

BINDING RELATED SUBPROGRAMS

Whenever related subprograms are separately translated, they are normally linked by the Multics dynamic linker at the time they are executed. If a set of related programs is known to always require certain links, then a program known as the binder may be used to pack them into a single segment, permanently link any cross references, and condense any common outward references into a single outbound link. In return for the loss of flexibility which comes with such permanent binding, one reduces both the space required for the programs and the number of library searches which must be undertaken to run the collection of programs. In addition, binding of separately translated subroutines retains most of the advantages of separate translation. (An alternative scheme would be to collect the procedures together into a single giant procedure, and then recompile. This alternate scheme has the disadvantage that a very long recompilation is needed for every one-line change to any part of the collection of programs.)

For more information on the details of dynamic linking and binding see the MPM Reference Guide (Order No. AG91) sections on object segments, system libraries and search rules, and the description of the "bind" command in MPM Commands, Order No. AG92.

SECTION 3

THE MULTICS PL/I COMPILER

Multics has one of the most powerful and complete implementations of PL/I available. The PL/I language is especially important to Multics, as most of the Multics system is written in PL/I. Almost without exception, all Multics commands are written in PL/I.

The most important service of the PL/I compiler is, of course, translation of the source program to produce machine code. On Multics, the machine code is placed into an object segment. An object segment is a segment like any other, but has a special format. One portion of it - the text section - contains instructions. Other portions describe the object segment itself. The most important of these descriptions is the definition section, which defines the names and locations of entry points present in the segment, and the names of external entry points used by the segment. Other sections contain templates for impure data used by the program (the static section) and the indirect words (links) used to implement dynamic linking (the linkage section).

A second service of the compiler is the creation of listing segments. A listing segment has the same name as the source segment, with the suffix "pl1" changed to "list". A listing segment contains a numbered list of the source program and information that is useful for understanding, debugging, and improving the performance of the program.

The PL/I compiler is invoked like any other command. The compiler takes one argument, the name of the source segment to translate. It also accepts several control arguments. The most common are:

-table
: augments the object segment with a symbol section. A symbol section is essential for debugging. It contains detailed information about the program in a form suitable for the Multics debugging tools.

-map
: causes a listing to be created.

-optimize
: causes the compiler to go to extra work to generate highly efficient code. Programs should only be optimized after they are fully debugged, since there is no reason to expend computer resources creating a highly efficient, yet faulty, program. It should also be mentioned that optimization reorganizes program and data flow in ways that may interfere with debugging; this is another reason why undebugged code should not be optimized.

For full information on the control arguments accepted by the PL/I compiler, see the discussion of the pl1 command in the MPM Commands manual.

The listing begins with a five-line summary of the circumstance of the compilation. For example, the following is extracted from the listing header of the compilation of the simple editor discussed in a chapter below.

```
COMPILATION LISTING OF SEGMENT eds
Compiled by: Multics PL/I Compiler, Release 25c, of February 18, 1980
Compiled at: Honeywell LISD Phoenix, System M
Compiled on: 04/21/80  1433.6 mst Mon
    Options: optimize map
```

The compiler both records here and encodes into the binary object program the date and time of compilation and the version of the compiler used. The date_time_compiled (dtc) command may be used to print the date and time of compilation stored in the object program. If that date and time are not identical to those printed at the top of a given listing, then that listing is for a different compilation, and should be suspected as being possibly for a different program.

A line-numbered listing of the source program follows the header. The line numbers are used by error diagnostics, and also by the Multics debugging aids.

Following the source listing is information about the program.

First comes a list of all the source files used in the compilation. This listing includes the full pathname of each file and the date and time that the file was last modified. This list can be used to verify that the most recent and proper versions of include files were used in the compilation.

The listing next gives a cross reference of all variables used in the program. This cross-reference listing may be used to discover unnecessary variables, which are set and never referenced, or perhaps never referenced at all. Any variable which is referenced only once is suspect, except for external subroutines which may happen to be called only once. Variables never referenced at all appear in the immediately following list.

For each variable, this listing gives its attributes such as data type, storage class, and the line numbers of all statements where it was referenced or set.

If there were any variables declared but unused, the compiler places their names in a separate section of the listing, under the heading "NAMES DECLARED BY DECLARE STATEMENT AND NEVER REFERENCED". No well-written program should declare unused names. The presence of a name here indicates the possibility of a bug.

The next section gives all "NAMES DECLARED BY EXPLICIT CONTEXT". This includes all the label and entry constants used in the program. The PL/I language considers the use of a name in the context of a label (on a statement or an entry) as an explicit declaration of the name.

The most significant warning in the listing is provided by the section "NAMES DECLARED BY CONTEXT OR IMPLICATION". This section lists every name that was used without being declared by either declare statement or explicit context. When a name is used without being declared, PL/I declares it with default attributes. Often, these will be inappropriate, since the compiler is only guessing. No well-written program should contain any names declared by implication. This is such a likely cause of error that the compiler will also issue a warning on the terminal when compiling a program that requires implicit declaration. If the program contains no implicitly defined names, then the section will be replaced by the message "THERE WERE NO NAMES DECLARED BY CONTEXT OR IMPLICATION".

The listing next gives information about the size of the object segment, under the heading "STORAGE REQUIREMENTS FOR THIS PROGRAM". Typical storage requirements might be:

	Object	Text	Link	Symbol	Defs	Static
Start	0	0	4276	4352	4045	4306
Length	4570	4045	54	201	231	0

All of the numbers describing storage requirements are printed in octal, so, for example, the binary machine instructions occupy 4045 (octal) locations or 2085 (decimal) locations.

Following the object segment description is a list of each block defined in the program. Internal procedures that are not recursive, and meet a few other requirements, can be called in a very efficient manner. These procedures are called quick procedures. A quick procedure shares the stack frame of some other, non-quick procedure. The block list tells why each block that is non-quick is non-quick (or, if the block is quick, which stack frame it shares). Significant performance gains can accrue if the programmer is able to make often-called internal procedures quick.

Following the block list are details about automatic storage allocation. "STORAGE FOR AUTOMATIC VARIABLES" describes the layout of the "stack frames" (procedure activation records, in which automatic variables are stored) of all of the non-quick procedures (including the main procedure itself). This information is useful in machine-level debugging.

The next section begins with "THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM." Many frequently used PL/I features are implemented in a library segment named pl1_operators_, and are used by fast "operator" calls compiled into the program. Certain PL/I constructs can only be implemented by using (comparatively) expensive operators. When performance is of great importance, the user should inspect this list for expensive operators. (See Section 5, "Performance Measurement Tools" for applicable performance evaluation techniques). It may be possible to avoid them by re-writing portions of the source code.

Following the operators used is a list of external entries called and external variables used. This information is also present in the symbol listing.

The final section gives the octal location of the first instruction generated for each statement. This section is known as the statement map. The statement map is also stored in the symbol section of the object map, when the -table control argument is given. It is this which allows the Multics debugging aids to determine the source line corresponding to an instruction when a fault occurs executing that instruction.

If the -list control argument is given, then the statement map is followed by an assembly-like listing of the detailed machine language program which it generated. Such a printout is useful for reviewing the performance of a program, since it may provide clues about use of PL/I constructs which are inherently expensive to implement.

SECTION 4

MULTICS DEBUGGING TOOLS

A variety of debugging tools are available on Multics. The most powerful of these is a program named probe, which permits source-language breakpoint debugging of PL/I, FORTRAN, and COBOL programs.

To understand the examples given below, one must first know a little about the Multics stack. The stack is essentially a push down list which contains the return points from a series of outstanding interprocedure calls. It also holds storage for automatic variables. If one were to stop a running program and trace its stack, he would find, starting at the oldest entry in the stack, a record of the procedures used to initialize the process, followed by the command language processor, followed by the procedure most recently called at command level and any procedures it has called. If an unexpected error occurs (or the user presses the "Quit" button), the system will save the current environment, mark the stack at its current level, push it down, and create a new activation of the command processor.

The new activation of the command processor accepts commands just as the original one did. It is possible to restart the suspended program, or to discard the saved environment, or to use one of the Multics debugging tools to examine the saved environment.

The release command causes the command processor to unwind the stack back to its own previous activation, and discard the intervening stack contents. The suspended programs cannot be resumed or examined after the stack has been released.

To attempt to resume execution of the suspended program, use the start command. The command processor then attempts to continue execution of the suspended program at the point of interruption. Depending on the nature of the error, and what the user has done since the error occurred, the restart attempt may or may not succeed. Programs may always be restarted after a QUIT, but only seldom after an error. If the program cannot be restarted, the error message will usually be repeated. An unsuccessful attempt to restart a program is usually harmless.

The probe command can be used to examine the saved stack and the current state of suspended programs. Probe can print the values of program variables and arguments, as well as reporting the last program location to be executed.

The use of probe is shown in a series of examples, first by the following program, blowup.pl1. This program has an illegal reference to the array a, and the subscriptrange condition occurs when it is run. Since subscriptrange checking is disabled by default, the error manifests itself as an out_of_bounds condition instead of a subscriptrange. Although this error is easy to spot, the behavior of the program is typical of other, harder to spot errors.

```
! print blowup.pl1

                    blowup.pl1          04/17/80  1332.0 est Thu

 blowup: procedure;

         dcl      j                             fixed binary;
         dcl      a                             (10) fixed binary;
         dcl      sum                           fixed binary;

         a (*) = 1;
         do j = -1 to -100000 by -1;
              sum = a (j);
         end;
    end blowup;


 r 13:32 0.110 20

 pl1 blowup -table
 PL/I 25c
 r 13:32 0.675 174
```

[The program is compiled with the -table control argument. This action causes a symbol table to be created and stored with the program in the executable object segment. This information is used by the Multics debugging aids. A symbol table should always be created while debugging, so that errors may easily be found.]

```
! blowup

Error:  out_of_bounds at >udd>States>Grant>blowup|24 (line 9)
 referencing stack_4|777777 (in process dir)
 Attempt to access beyond end of segment.
 r 13:32 0.228 32 level 2
```

[The program is invoked by typing its name. It takes an 'out_of_bounds' fault, because the subscript used in the reference to array "a" is invalid. The program does not use PL/I subscriptrange checking, so it attempt to calculate the address of the (nonexistent) element of "a" referenced. The resulting address does not exist, so the fault occurs.

This message shows the name of the error condition, the pathname of the program, the octal location in the object segment where the error occurred, the line number, and an additional message about the error. If the program had not included a symbol table, the line number would not have been part of the message.

The ready message has a new component. It says level 2. This number gives the number of activations of the command processor. There is always one command processor, and a second was added when the error occurred.]

```
! probe
 Condition out_of_bounds raised at line 9 of blowup (level 7).
```

[The user invokes the probe command. A message is given about the most recent error found in the user's process. The word "level" here refers not to command processor level, but to the number of programs saved on the stack. The error occurred in "blowup" which was the seventh program on the stack.]

```
! stack

  13          read_list|13400
  12          command_processor_|10301
  11          abbrev_|7507
  10          release_stack|7355
   9          unclaimed_signal|24512
   8          wall|4410
   7          blowup (line 9)                    out_of_bounds
   6          read_list|13400
   5          command_processor_|10301
   4          abbrev_|7507
   3          listen_|7355
   2          process_overseer_|35503
   1          user_init_admin_|40100
```

[The stack is displayed by the "stack" request. This request shows every program on the stack, in the order invoked. The numbers on the left show the order of activation. The entry for "blowup" shows the source line number corresponding to the last location executed, and the name of the error that occurred. The line number can be determined because "blowup" was compiled with a symbol table. The other programs have no symbol table, so the display shows the octal offset of the last instruction executed.]

```
!  source
  sum = a (j);
```

[Using the "source" command, the source statement for line 9 is displayed. This is the line that was being executed when the error occurred. More precisely, the error occurred executing the object code corresponding to this source line.]

```
! value j
 j = -2689
! symbol a
  fixed bin (17) automatic  dimension (10)
 Declared in blowup
```

[The value of the variable "j" is displayed with the "value" request. This request takes as its argument the name of a variable, and prints the value of the variable. Next, the "symbol" request is used, to show the attributes of "a."]

```
! quit
 r 13:33 1.080 129 level 2
```

[The last probe request used is "quit," which exits probe, and returns the the command level. The user is still at command level two, and the program is still intact. The next command typed is the "release" command, which discards the saved frames, returning to level one.]

```
! release
 r 13:33 0.057 16
```

SECTION 5

PERFORMANCE MEASUREMENT TOOLS

After a program is written and debugged it is often desirable to increase its efficiency. The first step in improving efficiency is to remove all PL/I condition checking prefixes and to compile with the -optimize control argument. Beyond that, Multics provides tools which identify the most expensive and most frequently executed programs in a given collection. Within these crucial programs, the most costly lines are found by using the "profile" facility.

To measure the performance of a program, compile it with the -profile control argument. This control argument causes the compiler generates special code for each statement, recording the cost of execution on a statement by statement basis.

The example that follows shows the use of profile with a very small sample program:

```
primep_: procedure (trial_prime) returns (bit (1) aligned);
         declare trial_prime            fixed binary (35) parameter;
         declare trial_factor           fixed binary,
                 last_factor            fixed binary;
         declare (mod, sqrt)            builtin;
         last_factor = sqrt (trial_prime);
         do trial_factor = 2 to last_factor;
              if mod (trial_prime, trial_factor) = 0
              then return ("0"b);
         end;
         return ("1"b);
     end primep_;
```

This subroutine cannot be called directly from command level, since only programs whose arguments are nonvarying character strings may be called directly. It is to be used with other programs. To test it, a simple command was written which accepts one argument, converts it to binary, and calls the "primep_" subroutine. The testing command is called "primep." It is not shown here.

```
!  pl1 primep_ -profile
 PL/I 25c
 r 17:44 0.699 140
```

[The profile control argument is used. Next the program is invoked, by means of a command "primep," which accepts one argument, converts it to binary, and calls the subroutine primep_ with it.]

```
! primep 3
   3 is  a prime.
 r 1744 .110 23
```

[To evaluate the performance of the subroutine, several hundred calls to it should be made, over a wide range of values. The next command line invokes primep 500 times, with values from 1 to 500. The "index_set" active function returns the numbers from 1 to 500, and the parentheses invoke primep once for each value.

The output from the program is not interesting, so the discard_output command is used. This command causes output from the program to be discarded, instead of printed on the terminal.]

```
! discard_output "primep ([index_set 500])"
 r 17:45 5.103 54
! discard_output "primep ([index_set 500])"
 r 17:45 5.088 40
```

[While the program was run, performance statistics were saved. Now the "profile" command is used to display those statistics. For each line, it displays the total times executed, an estimate of the cost, and the PL/I operators used.]

```
! profile primep_

          Program: primep_
             LINE STMT   COUNT       COST STARS   OPERATORS
                8         1000      34000 ****    fx1_to_fl2, dsqrt, fl2_to_fx1
                9         1000       3000
                9         4418      13254 ***
               10         4218      59052 ****    mod_fx1
               10          800       8800 **      return
               12         3418       6836 **
               13          200       2600         return
            -------
            Totals:      15054     127542
           r 17:46 0.368 51
```

[Note that some statements (those in the loop) were executed more than others. The COST for a statement is the product of the number of instructions for the statement and the number of times the statement was executed. This cost does not take into account the fact that some instructions are faster than others, or the time spent waiting for missing pages (page faults). The STARS column gives a rough indication of the relative cost of each statement.

The names of the PL/I operators used are also given. The operator fx1_to_fl2 is used to convert the fixed point number to float, so that its root may be taken. The dsqrt operator takes the square root. Finally the operator fl2_to_fx1 converts the result back to integer. The PL/I mod builtin is implemented by the mod_fx1 operator. These operators are the most expensive things in the program. Occasionally a program can be re-written to not require expensive operators.

When profiling large programs it is usually desirable to look only at the most expensive lines, since they are the only ones of interest. The profile command can be instructed to sort the lines by cost, and display them in order. The next command displays the five most costly lines.]

```
! profile primep_ -sort cost -first 5

Program: primep_
   LINE STMT   COUNT       COST STARS   OPERATORS
     10         4218      59052 ****    mod_fx1
      8         1000      34000 ****    fx1_to_fl2, dsqrt, fl2_to_fx1
      9         4418      13254 ***
     10          800       8800 **      return
     12         3418       6836 **

 -------
 Totals:       15054     127542
 r 17:46 0.205 49
```

More detailed records of execution are available when the program is compiled with the -long_profile control argument. When this is done, the program samples the Multics clock before every instruction, so the total time per statement is available to the profile command. The performance data from a program compiled with -long_profile is displayed with the profile command. For further information, see the MPM Commands description of the profile and pl1 commands.

Other Multics performance measurement tools include the "trace" command, which provides a record of procedures called, and time spent in their execution; the "page_trace" command, which lists page faults.

SECTION 6

A SIMPLE TEXT EDITOR

Our next sample program is a printing-terminal text editor similar to, but simpler than, the standard "edm" command (See MPM Commands for a description of the edm command.) It is a typical example of an interactive program which makes use of the Multics storage system via the virtual memory. In overview, the editor creates two temporary storage areas, each large enough to hold the entire text segment being edited. It copies the segment into one of these areas, so as not to harm the original; and then, as the user supplies successive editing requests, constructs in the other area an edited version of the segment. When the user finishes a pass through the segment, the editor interchanges the roles of the two storage areas for the next editing pass. When finished the appropriate temporary storage area is then copied back over the original segment. This example is not intended to be a model for designing or implementing text editors, but rather, an illustration of the techniques used in interactive Multics PL/I programs, particularly commands.

For this example, we have available a program listing as produced by the PL/I compiler. The program itself is derived from the edm command of Multics, and it exhibits several different styles of coding and commenting, since it has had many different maintainers.

The program is preceded by several pages of comments on the program. The comments appear in the same order as the item(s) in the program that they comment on. Where possible, they refer to line numbers in the program listing. Unfortunately, programs do not always invoke features in the best order for understanding, so the following strategy may be useful: as you read each comment, if its implications are clear and you feel you understand it, check it off. If you encounter one which does not fit in to your mental image of what is going on, skip it for the moment. Later comments may shed some light on the situation, as will later reference to other Multics documentation. Finally, a hard core of obscure points may remain unexplained, in which case the advice of an experienced Multics programmer is probably needed. Be warned that the range of comments is very wide, from trivial to significant, from simple to sophisticated, and from obvious to extremely subtle.

Finally, some comments provide suggestions for "good programming practice." Such suggestions are usually subjective, and often controversial. Nonetheless, the concept of choosing among various possible implementation methods one which has clarity, is consistent, and minimizes side effects is valuable, so the suggestions are provided as a starting point for the reader who may wish to develop his own style of good programming practice.

The reader will also notice that some comments appear to be critical of the program style or of interfaces to the Multics supervisor. These comments should be taken in the spirit of illumination of the mechanisms involved. Often they refer to points which could easily be repaired, but which have not been in order to provide a more interesting illustration. Most of the points criticized are minor in impact.

The program listing appears below, following the commentary.

Line number

fifth unnumbered line

The command "pl1 eds -map -optimize" was typed at the terminal. This line records the fact that the map and optimize options were used. The map line option caused a listing and variable storage map to be produced. A source segment named eds.pl1 was used as input; the compiler constructed output segments named eds.list (containing the listing) and eds (containing the compiled binary program.)

1 No explicit arguments are declared here, even though eds should be called with one argument. Instead, the keyword "options (variable)" appears, which indicates that this program can be called with a variable number of arguments. This is a Multics extension to ANSI PL/I. Since eds is used as a command, it is a good human engineering practice to check explicitly for missing arguments; the PL/I language has no feature to accomplish this check gracefully. Library facilities are available to determine the number and type of arguments supplied (See lines 91-95). All Multics commands are declared in this way.

5 To avoid errors when program maintenance is performed by someone other than the original coder, all variables are explicitly declared. This practice not only avoids surprises, but also gives an opportunity for a comment to indicate how each variable is used.

6 One default which is used here (and is subject to some debate) is that the precision of fixed binary integers is not specified, leading to use of fixed binary(17). This practice has grown up in an attempt to allow the compiler to choose a hardware supported precision, and in fear that an exact precision specification might cause generated code to check and enforce the specified precision at (presumably) great cost. In fact, the PL/I language does not require such checks by default (although they can be specified). Thus, it is usually wise to specify data precision exactly. In some cases (for instance, all of the fixed binary (21) variables used to hold string lengths) the compiler might attempt to hold these values in half-length registers were this precision not specified.

However, a large class of variables which will contain "small or reasonable size integers" can still be conveniently declared with the implementation's default precision.

7 All character strings in this program are declared unaligned, by the defaults of the language. Given the fact that the Multics hardware has extremely powerful and general string manipulation instructions, no advantage is to be gained in speed or length of object code by declaring strings (when they are over two words, or eight characters, long) with the aligned attribute.

Therefore, almost all supervisor and library subroutines which accept character string arguments require unaligned strings. By the rules of PL/I, aligned and unaligned strings may not be interchanged as parameters, and thus, there is incentive to avoid aligned character strings in all cases.

7 All line buffers are designed to hold one long typed line (132 characters for input terminals with the widest lines) plus a moderate number of backspace/overstrike characters. To support memorandum typing, the buffers permit a 70-character line which is completely underlined.

By use of temporary segments as work areas (see line 120), an almost unlimited number of nearly infinite work-variables can be constructed, virtually avoiding the "fixed length buffer problem." However, the acquisition and maintenance of such segments are not as cheap as PL/I automatic variables, and judgement should be exercised as to where traditional "fixed length" variables are appropriate.

9 The variable named code has precision 35 bits, since it is used as an output argument for several supervisor entries which return a fixed binary(35) value. Almost all supervisor and library subroutine entries return an "error code" value, which indicates the degree of success of the operation requested. The values of system error codes require 35 bits. It would seem appropriate, on a 36-bit machine, to use fixed binary(35) declarations everywhere. However, use of fixed binary(35) variables for routine arithmetic should be avoided since, for example, addition of two such variables results in a fixed binary(36) result, forcing the compiler to generate code for double precision operations from that point on. One must be careful of the PL/I language rule which requires the compiler to maintain full implicit precision on intermediate results.

10 Legal PL/I overlay defining can be an extremely powerful tool for increasing the readability and maintainability of code. The variable "commands" is declared here as occupying the same storage as the variable "buffer", but only being as long as that part of it which contains valid characters, as defined by the value of "count". Thus, we need only write "commands" when we want the portion of "buffer" that has valid data in it, instead of "the substring of 'buffer' starting at the first character for 'count' characters."

18,19 All editing is done by direct reference to virtual memory locations. The variable from_ptr is set to point to a source of text, and the based variable from_seg is used for all reference to that text. The number 1048576 (two to the twentieth power) is the largest possible number of characters in a segment.

19 The general operation of the editor is to copy the text from one storage area to another, editing on the way. The names from_seg and to_seg are used for the two storage areas.

38 One set of supervisor interfaces calls for 24 bit integers; this declaration guarantees that no precision conversion is necessary when calling these interfaces. (See line 104.)

51 The PL/I language provides no direct way to express literal control characters. The technique used here, while it clutters the program listing, at least works.

PL/I does not provide any "named constant" facility, either. The Multics PL/I implementation allows the "options (constant)" attribute for internal static variables, which instructs the compiler to allocate the variable in the pure (unmodifiable) portion of the object segment. This is advantageous for three reasons: First, if an attempt is made to modify such a variable, the hardware will detect an error, thus checking and enforcing its "constant" use. Second, it allows the variable to be shared between processes, conserving storage. Third, it is an indication to others reading the program that a "named constant" is intended.

59 Subroutines com_err_ and ioa_ are called with a different number of arguments each time, a feature not normally permitted in PL/I. The Multics implementation, however, has a feature to permit such calls. The "options" clause warns the compiler that the feature is used for this external subroutine.

60 All subroutines other than com_err_ and ioa_ are completely declared in order to guarantee that the compiler can check that arguments being passed agree in attribute with those expected by the subroutine. Warning diagnostics are printed if the compiler finds argument conversions necessary.

60 The procedure cu_ (short for command utility) has several different entry points. The Multics PL/I compiler specially handles names of external objects which contain the dollar sign character. The dollar sign is taken to be a separator between a segment name and an entry point name in the compiled external linkage. Thus, this line declares the entry point name arg_ptr in the segment name cu_.

61 For many procedures, the segment name and entry point name are identical, so the compiler also permits the briefer form cv_dec_, which is handled identically to cv_dec_$cv_dec_.

64 The hardcore (ring zero) supervisor entries ("hardcore gates") are all easily identifiable since they are entered through a single interface segment named hcs_. Segment hcs_ consists of just a set of transfers to the subroutine wanted. A transfer vector is used to isolate, in one easily available location, all gates into the Multics supervisor. Also, it is in principle possible to replace a supervisor routine dynamically, by changing a single transfer instruction. (There are in fact hardcore gate segments other than hcs_, but you will probably not have occasion to deal with them.)

83 The program will need to know what I/O switches will be used in order to perform certain I/O operations. I/O switches are the general source/sink I/O facility of Multics. Multics PL/I programs manipulate I/O switches as PL/I pointer values. The two external variables declared on this line contain the pointer values identifying the standard terminal input and terminal output switches.

84 As mentioned above, system error codes are returned by most supervisor and library subroutine entries. In one case, we will need to know if a specific error (see line 107) was returned by a supervisor entry. A segment (error_table_) exists which has entry point definitions for external static variables (see Section 1) containing all the possible values that can be returned as errors by system routines. The variable error_table_$noentry contains the value returned as an error code by system routines to indicate that "the entry you specified in the directory you specified does not exist."

91 The first order of business is to access the command's argument. As was pointed out above, this is done via library subroutine rather than PL/I parameter passing in order to diagnose the case of a missing argument.

Since the command argument is nominally unlimited in length, cu_$arg_ptr returns a pointer to the argument as stored by the command processor, and its length. The based variable "sname" will describe the argument once this pointer and length are obtained.

92 If for any reason the argument to the command cannot be accessed (most typically, because it does not exist), a nonzero value of "code" will be returned.

93 The subroutine com_err_ is called to print out the error message associated with the returned error code. This subroutine produces an English explanation of the error obtained from the value of the error code. It also causes terminal output to be produced even if the user is temporarily diverting output to a file. In general, com_err_ should be used to report all command usage and storage system errors.

94 A Multics command exits by simply returning to its caller. (See also line 432.)

99 Assuming that a pointer to an argument was returned, we must now convert that argument to a standard (directory name, entry name) pair. The subroutine expand_pathname_ implements the system-wide standard practice of interpreting the typed argument as either a pathname relative to the current working directory, or an absolute pathname from the root, as appropriate.

104 The supervisor entry point hcs_$initiate_count is invoked to map the segment specified by the (directory name, entry name) pair into the process's virtual memory. It returns a pointer to the segment, which it constructs from the segment number by which the segment was mapped into the virtual memory of the process (made known). If the segment was already "known", i.e., in the process' address space, the segment number from the existent mapping will be used to create a pointer to return.]

The PL/I null string ("") is a special signal that no (possibly additional) reference name is to be initiated for the segment.

106 Unfortunately, the zero/nonzero value of the return code from hcs_$initiate_count cannot be used to check whether the initiation (mapping into the address space) succeeded. In the particular case of this subroutine and hcs_$initiate, a nonzero error code is returned in the ostensibly successful case of the segment having already been in the address space or the process, a case which is rarely an error.

These two subroutines are documented to return a nonnull pointer value if and only if the segment has been successfully mapped into the address space, whether by prior act or anew. Thus, testing the return pointer for the PL/I null pointer value is an adequate test of success.

108 The program will soon acquire (on line 124) a process resource, namely two temporary segments from the process' pool of temporary segments. When the program is finished executing, it will return them (line 565) to the pool. However, the program may be interrupted (perhaps by a QUIT, or a record quota overflow), and the user may abandon its execution (perhaps via the "release" command). In this case, it would seem that the program would not get a chance to return its "borrowed" resources. However, Multics defines the "cleanup" condition, which is signalled in all procedures when their execution is about to be irrevocably abandoned. The handler for the cleanup condition invokes the procedure "cleanup", which relinquishes these resources.

The array "temp_segs" is initialized to null pointer values before establishing the cleanup handler, so that the contents of the array is well defined at all times. (The release_temp_segments_ subroutine checks for null pointer values, and performs no action if it encounters them.) Otherwise, if the cleanup handler were invoked before the temporary segments were acquired, the pointer array would have undefined, probably invalid values, and the call to release the temporary segments would have unpredictable results.

The cleanup handler is established before the temporary segments are reserved. This sequence guarantees that there will be no "window" in which the program can be abandoned between the time that the segments were acquired and the time that the cleanup handler was set up.

116 The editor (eds) will create a new segment (see line 471) if an attempt is made to edit a segment which does not exist. By comparing the value of the error code returned from hcs_$initiate_count with the system error code stored in the variable error_table_$noentry, we can differentiate the case of failure to initiate simply because the segment did not exist from all others (e.g., incorrect access to the segment specified).

117 The com_err_ subroutine (as well as the ioa_ subroutine, see line 137) allows conditional substitution of parameters. The construct "^[>^]" is used to prevent error messages containing two sequential >'s in error messages describing segments stored in the root directory whose name is (">").

124 A pool of segments in a process directory is maintained by the get_temp_segments_ and release_temp_segments_ subroutines. These segments are doled out to commands and subsystems which request them (via get_temp_segments_) and it is expected that they will be returned to the pool when there is no further use for them. This facility avoids the need for user programs to create and delete (or attempt to manage or share) segments needed on a "scratch" or "temporary" basis (for work areas, buffers, etc). Segments obtained from this facility are guaranteed to contain all zeros (truncated) when obtained.

The number of segments to be obtained is determined by get_temp_segments_ from the extent of the pointer array parameter. The name of the subsystem is passed to get_temp_segments_ both to facilitate additional checking by release_temp_segments_ and to support the list_temp_segments command, which describes which subsystems in a process are using temporary segments.

136 If the segment specified on the command line did not exist, the editor is to assume that it is creating an new segment, and go into input mode. The value of the variable "source_ptr" will be null if this is the case.

137 The ioa_ subroutine is a handy library output package. It provides a format facility similar to PL/I and FORTRAN format statements, and it automatically writes onto the I/O stream named user_output, which is normally attached to the interactive user's terminal. When used as shown, it appends a newline character to the end of the string given. Programmers who are more concerned about speed and convenience than about compatibility with other operating systems use ioa_ in preference to PL/I "put" statements, because ioa_ is cheaper, easier to use, and far more powerful.

The formatting facilities of ioa_ are used in a simple way in this example. The circumflex ("^") in the format string indicates where a converted variable is to be inserted; the character following the circumflex indicates the form (in this case, a character string) to which the variable should be converted. The first argument is the format string, remaining arguments are variables to be converted and inserted in the output line.

140 The storage system provides for every segment a variable named the bit count. For a text segment, by convention, the bit count contains the number of information bits currently stored in the segment. The bit count of the segment being edited was returned by hcs_$initiate_count (hence its name) on line 113.

This statement converts the bit count to a character count. Note that we have here embedded knowledge of the number of hardware bits per character in this program. If the system-wide standard had been to store a character count with a segment instead, it would not have been necessary to have an implementation-dependent statement here. Unfortunately, a stored character count would get the system into the business of maintaining an interpretation of the segment's contents, which it currently does not do.

140 The PL/I language specifies that the result of a divide operation using the division sign is to be a scaled fixed point number. To get integer division, the divide built-in function is used instead. Note that the precision of the quotient is specified to match its size.

141 Here, we invoke some of the most powerful features of the Multics virtual memory. This simple assignment statement copies the entire source segment to be edited into the temporary buffer named from_seg. A single powerful hardware string-copy instruction is generated for this code, copying data at processor speed. Note that we are regarding the entire text segment as a simple character string of length csize. We may regard it this way because the storage representation for permanent text segments is chosen to be identical to that of a PL/I nonvarying character string.

141 Be sure to read the comments embedded in the program, too.

150 The user-ring I/O system is being invoked to read a line from the user's terminal. The line is read from the I/O switch identified by the external pointer iox_$user_input. Although passing the buffer to be used as a character string would be more convenient this set of interfaces was designed with maximal efficiency in mind, and this form of call is more efficient. Note it would also be safer than passing a pointer to the character string, since that would allow PL/I to check that an appropriate character string was being passed, as opposed to a pointer, which can point to any data type. This design demonstrates the frequent tradeoff between efficiency and convenience.

144 Subroutine iox_$get_line is often used for input rather than the PL/I statement "read file (sysin) into ..." again because of efficiency and error-handling considerations. The PL/I facility ultimately calls on the Multics iox_ package anyway. (Again, if one wished to write a program which would also work on other PL/I systems, he would be better advised to use the PL/I I/O statements instead.)

151 It is highly unlikely that a call to read a line from the terminal will fail. Nevertheless, in cases of people debugging their own extensions to the Multics I/O system (a practice intended by the designers of the I/O system), it can occur. It is reasonable to abort the entire editor in this unlikely case rather than repeating the call: presumably that would repeat the error too.

155 For the sake of human engineering, the editor ignores blank lines. Since complete input lines from the typewriter end with a new line character, the length of a blank line is one, not zero.

157 The code to isolate a string of characters on the typed input line is needed in four places, so an internal subroutine is used. This subroutine is not recursive, which makes it possible for the compiler to construct a one-instruction calling sequence to the internal procedure. Certain constructs (e.g., variables of adjustable size declared within the subroutine) will force a more complex calling sequence. For details, one should review the documentation on the Multics PL/I implementation.

159 Although the dispatching technique used here appears costly, it is really compiled into very quick and effective code -- 2 machine instructions for each line of PL/I. For such a short dispatching table, there is really no point in developing anything more elaborate. If the table were larger, one might use subscripted label constants for greater dispatching speed.

164 Human engineering: the typist is forced to type out the full name of the one "powerful" editing request which, if typed by mistake, could cause overwriting of the original segment before that overwriting was intended.

175 Whenever a message is typed which the typist is probably not expecting, it is good practice to discard any type-ahead, so that he may examine the error message, and redo the typed lines in the light of this new information.

182 The general strategy of the editor is as follows: lines from the typewriter go into the variable named "buffer" (accessed as "commands") until they can be examined. Another buffer, named "line_buffer" (accessed as "line") holds the current line being "pointed at" by the eds conceptual pointer. Subroutine "put" copies the current line onto the end of to_seg, while subroutine "get" copies the next line in from_seg into the current line buffer.

200 The procedure get_num sets up the variable n to contain the value of the next typed integer on the request line. Such side-effect communication is not an especially good programming practice.

201 The delete request is accomplished by reading lines from from_seg, but failing to copy them into to_seg. If deletion were a common operation, it might be worthwhile to use more complex code to directly push ahead the pointer in from_seg, and thus avoid a wasted copy operation.

212 More side-effect communication: the variable edct is always pointing at the last character so far examined in the typed request line.

229,240 All movement of parts of the material being edited is accomplished by a simple string substitution, using appropriate indexes.

259 The locate request is accomplished by use of the index built-in function, used on whatever is still unedited in from_seg.

397 A negative number in the "next" request results in moving the conceptual pointer backward. The resulting code is quite complex because the eds editing strategy requires interchanging the input and output segments before backward scanning, so that the backward scan is with regard to the latest edited version of the segment.

402 This code to search a character string backward is recognized by the compiler as such. Extremely efficient object code to search the substring backward is generated, using a single hardware instruction. No copies are made in this fairly expensive-looking statement: it is in fact cheap. Combinations of reverse, index, substr, search, verify, etc. that seem that they ought to generate efficient code in fact usually do.

431 Before exiting from the editor, the temporary segments should be returned to the temporary segment manager, and the segment that was initiated terminated.

443 Another human engineering point: since the user may have typed several lines ahead, the error message includes the offending request, so that he can tell which one ran into trouble and where to start retyping.

444 Note a small "window" in this sequence of code. If the editor is delayed (by "time-sharing") between lines 443 and 444, it is possible that the message on line 443 will be completed, and the user will have responded by typing one or more revised input lines, all before line 444 discards all pending input. Although in principle fixable by a reset option on the write call, Multics currently provides no way to cover this timing window. Fortunately, the window is small enough that most interactive users will go literally for years without encountering an example of a timing failure on input read reset.

476 Note that we copy data into the original segment, set its bit count, and truncate it in that order. This provides for maximal data being saved should their be a system failure between any two lines. Common sense seems to indicate this order as "maximally safe", and analysis of the data involved will demonstrate this as well.

514-516 The input and output editing buffer areas are interchanged by these three statements. Here is an example of localizing the use of pointer variables to make clear that they are being used as escapes to allow interchange of the meaning of PL/I identifiers.

527 The I/O system provides this entry point to perform control operations (e.g., "resetread") upon the objects represented by I/O switches.

539 This editor considers typed-in tab characters to be just as suitable for token delimiters as are blanks. Ideally, tab characters would never reach the editor, instead having been replaced by blanks by the typewriter input routines. Such complete canonicalization of the input stream would result in some greater simplicity, but would require a more sophisticated strategy to handle editing of text typed in columns.

539 The PL/I search and verify builtins, which are very useful in circumstances like this (parsing lines) are compiled into very efficient single-instruction hardware operations by the Multics PL/I compiler.

556 The cv_dec_ library routine is used here rather than a PL/I language feature, because cv_dec_ will always return a value, even if the number to be converted is ill-formed (in which case it returns zero.) Thus the editor chooses not to handle ill-formed numbers. Had it wished to check, for them, it could have used the cv_dec_check_ subroutine. PL/I language conversion would cause an error signal which must be caught and interpreted lest PL/I's runtime diagnostic appear on the user's console. Thus, eds retains complete control over the error comments and messages which will be presented to the user. Such control is essential if one is to construct a well-engineered interface which uses consistent and relevant error messages.

565 The cleanup procedure calls the release_temp_segments_ subroutine to release the temporary segments acquired earlier. A binary zero is passed to release_temp_segments_ by value (by enclosing it in parenthesis) because the cleanup handler has no use for an error code. Cleanup procedures should never print messages, even error messages, because they are only invoked when exiting a procedure. There is no corrective action the user can take.

566 If the segment edited was not known before editing it, it should be unknown after the editor finishes as well. The supervisor maintains a reference count for each segment in the process. This count is incremented by the call to hcs_$initiate and decremented by the call to hcs_$terminate_noname. If the count goes to zero (i.e. the segment was made known by the editor) then the segment is made unknown.

```
COMPILATION LISTING OF SEGMENT eds
Compiled by: Multics PL/I Compiler, Release 25c, of February 18, 1980
Compiled at: Honeywell LISD Phoenix, System M
Compiled on: 05/06/80  1456.1 edt Tue
    Options: map optimize

eds:     procedure options (variable);

/*        internal variable declarations.  */

declare  break                  character (1);           /*  Holds break char for change */
declare  brk1                   fixed binary;
declare  buffer                 character (210);         /*  Typewriter input buffer.  */
declare  changes_occurred       bit (1);
declare  code                   fixed binary (35);
declare  commands               character (count) based (addr (buffer));
                                                         /*  Valid portion of buffer */
declare  count                  fixed binary (21);       /*  Valid length of data in "buffer " */
declare  csize                  fixed binary (21);
declare  edct                   fixed binary;
declare  dir_name               character (168);         /*  Directory containing segment */
declare  entry_name             character (32);
declare  exptr                  pointer;                 /*  Temporary pointer holder.  */
declare  from_ptr               pointer;                 /*  Pointer to current from_seg.  */
declare  from_seg               character (1048576) based (from_ptr);
                                                         /*  Editing is from this segment.  */
declare  globsw                 bit (1);
declare  i                      fixed binary (21);
declare  ij                     fixed binary (21);
declare  indf                   fixed binary (21);
declare  indt                   fixed binary (21);
declare  j                      fixed binary (21);
declare  k                      fixed binary (21);
declare  l                      fixed binary (21);
declare  line                   character (linel) based (addr (line_buffer));
declare  line_buffer            character (210);         /*  Holds line currently being edited.  */
declare  linel                  fixed binary;            /* length of "line" */
declare  located                fixed binary;
declare  m                      fixed binary (21);
declare  n                      fixed binary (21);
declare  sname                  character (sname_lth) based (sname_ptr);   /* Source name */
declare  sname_lth              fixed binary (21);       /*  Length of source segment name.  */
declare  sname_ptr              pointer;                 /*  Pointer to source segment name.  */
declare  source_count           fixed binary (24);       /*  Holds segment bit length.  */
declare  source_ptr             pointer;                 /*  Pointer to source seg.  */
declare  source_seg             character (1048576) based (source_ptr);
                                                         /*  Outside segment for read or write.  */
declare  temp_segs              dimension (2) pointer;
declare  tlin                   character (210);         /*  Buffer to hold output of change.  */
declare  tkn                    character (8);           /*  Holds next item on typed line */
declare  to_seg                 character (1048576) based (to_ptr);
                                                         /*  Editing is to this segment.  */
declare  to_ptr                 pointer;                 /*  Pointer to to_seg.  */

/*  Constants */

declare  NL                     character (1) static options (constant) initial ("
");
declare  WHITESPACE             character (3) static options (constant) initial ("
         "); /* NL SP TAB */
```

```
declare  myname                  character (3) static options (constant) initial ("eds");

/*        external subroutine declarations.  */

declare  com_err_                entry options (variable);
declare  cu_$arg_ptr             entry (fixed binary, pointer, fixed binary (21), fixed binary (35));
declare  cv_dec_                 entry (character (*)) returns (fixed binary(35));
declare  expand_pathname_        entry (character (*), character (*), character (*), fixed binary (35));
declare  get_temp_segments_      entry (character (*), pointer dimension (*), fixed binary (35));
declare  hcs_$initiate_count     entry (character (*), character (*), character (*), fixed binary (24),
                                 fixed binary, pointer, fixed binary (35));
declare  hcs_$make_seg           entry (character (*), character (*), character (*),
                                 fixed bin (5), ptr, fixed binary (35));
declare  hcs_$set_bc_seg         entry (pointer, fixed binary (24), fixed binary(35));
declare  hcs_$terminate_noname   entry (pointer, fixed binary (35));
declare  hcs_$truncate_seg       entry (pointer, fixed binary (19), fixed binary(35));
declare  ioa_                    entry options (variable);
declare  iox_$control            entry (pointer, character (*), pointer, fixed binary (35));
declare  iox_$get_line           entry (pointer, pointer, fixed binary (21), fixed binary (21), fixed binary (35));
declare  iox_$put_chars          entry (pointer, pointer, fixed binary (21), fixed binary (35));
declare  release_temp_segments_  entry (character (*), pointer dimension (*), fixed binary (35));

declare  cleanup condition;
declare  (addr, divide, index, length, null, reverse, search, substr, verify)
                                 builtin;


/*  External data */

declare (iox_$user_output, iox_$user_input)       pointer external static;
declare  error_table_$noentry                     fixed bin (35) external static;

```

```
/*              .          .          .           P R O G R A M          .          .          .          */


/* Check to see if an input argument was given */

          call cu_$arg_ptr (1, sname_ptr, sname_lth, code);
          if code ^= 0 then do;
               call com_err_ (code, myname, "Usage: ^a <PATH>", myname);
               return;
          end;

/* Now get a pointer to the segment to be edited */

          call expand_pathname_ (sname, dir_name, entry_name, code);
          if code ^= 0 then do;                                    /* Bad pathname */
               call com_err_ (code, myname, "^a", sname);
               return;
          end;

/*  Set up a cleanup handler in case the program is aborted */

          source_ptr = null ();
          temp_segs (*) = null ();                                 /* Make sure handler has valid data */
          on condition (cleanup) call clean_up;

/* Initiate the source segment. */

          call hcs_$initiate_count (dir_name, entry_name, "", source_count, 0, source_ptr, code);
                                                                   /* Initiate the segment */
          if source_ptr = null ()
               then if code ^= error_table_$noentry then do;/* Problem or just new seg? */
                    call com_err_ (code, myname, "Cannot access ^a^[>^]^a",
                         dir_name, (dir_name ^= ">"), entry_name);
                    return;
               end;

/*  Set up Buffer segments.  */

          call get_temp_segments_ (myname, temp_segs, code);
          if code ^= 0 then do;
               call com_err_ (code, myname, "Cannot get temporary segments.");
               call clean_up;
               return;
          end;
          from_ptr = temp_segs (1);
          to_ptr = temp_segs (2);

/* Check to see that the segment is there */

          csize, indf, indt = 0;                                   /* Initialize buffer control vars. */
          if source_ptr = null then do;
               call ioa_ ("Segment ^a not found.", entry_name);
               go to pinput;
          end;
          csize = divide (source_count, 9, 21, 0);                 /* change bit count to char count */
          substr (from_seg, 1, csize) = substr (source_seg, 1, csize);
                                                                   /*  Move source segment into buffer.  */

/*  Main editing loop . . . . .  */
```

```
pedit:
        call ioa_ ("Edit.");
next:
        call iox_$get_line (iox_$user_input, addr (buffer), length (buffer), count, code);
        if code ^= 0 then do;
            call com_err_ (code, myname, "Error reading input line");
            go to fifish;
        end;
        if count = 1 then go to next;              /* if null line then get another line, don't print error */
        edct = 1;                                   /*  Set up counter to scan this line.  */
        call get_token;                             /*  Identify next token.  */

        if tkn = "i" then go to insert;
        if tkn = "r" then go to retype;
        if tkn = "l" then go to locate;
        if tkn = "p" then go to print;
        if tkn = "n" then go to nexlin;
        if tkn = "save" then go to file;
        if tkn = "c" then go to change;
        if tkn = "d" then go to dellin;
        if tkn = "w" then go to wsave;
        if tkn = "t" then go to top;
        if tkn = "b" then go to bottom;
        if tkn = "." then go to pinput;

 /* If none of the above then not a request */

        call ioa_ ("'^a' Not an edit Request", substr (commands, 1, length (commands) - 1));
        call resetread;
        go to next;

/* ********* input mode ********* */

pinput:
        call ioa_ ("Input.");                       /* print word input */
input:
        call iox_$get_line (iox_$user_input, addr (buffer), length (buffer), count, code);
        if code ^= 0 then do;
            call com_err_ (code, myname, "Error reading input-mode line.");
            go to fifish;
        end;

        if substr (commands, 1, 1) = "." & count = 2
            then go to pedit;                       /* check for mode change */
        call put;
        linel = length (commands);
        line = commands;                            /* move line inputted into intermediate storage */
        go to input;                                /* repeat 'til "." */


/* ********* delete ********* */

dellin:
        call get_num;
        do i = 1 to n - 1;                          /* do for each line to be deleted */
            call get;
        end;
        linel = 0;                                  /* nullify last line */
```

```
          go to next;

/* ********* insert ********* */

insert:
          call put;                                   /*  Add current line to output segment */
retype:                                               /*  This is also the retype request.  */
          linel = length (commands) - edct;
          line = substr (commands, edct + 1);         /* add replaced line */
          go to next;

/* ********* next ********* */

nexlin:   call get_num;
          if n < 0 then go to backup;
          m, j = indf;                                /* save where you are */
          call put;
          do i = 1 to n;                              /* once for each nl */
               if j >= csize then go to n_eof;        /* check for eof */
               k = index (substr (from_seg, j + 1, csize - j), NL);
                                                      /* locate end of line */
               if k = 0 then do;                      /* no nl (eof) print eof */
n_eof:              if indf >= csize then go to eof;
                    linel = 0;                        /* set to no line */
                    substr (to_seg, indt + 1, csize - m) = substr (from_seg, m + 1, csize - m);
                                                      /* move in top of file */
                    indf = csize;
                    indt = indt + csize - m;          /* set pointers */
                    go to eof;
               end;
               j = j + k;                             /* increment j by length of line */
          end;
          indf = j;                                   /* set pointers and move in top of file */
          linel = k;
          line = substr (from_seg, j - k + 1, linel);        /* put working line in line */
          substr (to_seg, indt + 1, indf - linel - m) = substr (from_seg, m + 1, indf - linel - m);
                                                      /* fill rest of file */
          indt = indt + indf - linel - m;
          go to next;

/* ********* locate ********* */

locate:   if edct = length (commands) then go to bad_syntax;     /* check for plain "l NL" */
          edct = edct + 1;                            /*  Skip delimiter.  */
          j = indt;                                   /* initialize pointers for index type search */
          m = indf;
          n = csize - indf;
          call put;
          if (csize = 0) ¦ (n <= 0) then do;
               call switch;
               if j > 0 then n = j - 1;
               else n = 0;
               m, j = 0;
          end;
          i = index (substr (from_seg, indf + 1, n), substr (commands, edct, length (commands) - edct));
          if i ^= 0 then do;                          /* if found then do */
               k = index (reverse (substr (from_seg, 1, indf + i)), NL);
               if k ^= 0 then k = indf + i - k + 1;          /* k = index of NL */
               j = index (substr (from_seg, k + 1, csize - k), NL); /* fifd end of line */
               if j = 0 then indf = csize;
```

```
                else indf = j + k;
                substr (to_seg, indt + 1, k - m) = substr (from_seg, m + 1, k - m);
                                                        /* move in top of file */

                linel = indf - k;
                indt = indt + k - m;
                line = substr (from_seg, k + 1, linel);      /* put found line in line */
                n = 1;
                go to print1;                                /* print found line if wanted */
           end;
           call copy;
           call switch;
           go to next;                                       /* get next command */

/* ********* print ********* */

print:     call get_num;
           if linel = 0 then do;                             /* print indication of no lines */
                call ioa_ ("No line.");
                go to noline;
           end;

print1:    call iox_$put_chars (iox_$user_output, addr (line), length (line), code);
           if code ^= 0 then do;
                call com_err_ (code, myname, "Problem writing editor output");
                go to fifish;
           end;
                                                             /* write the line */
noline:    n = n - 1;
           if n = 0 then go to next;                         /* any more to be printed? */
           call put;
           call get;
           go to print1;

/* ********* change ********* */

change:    located = 0;
           if count = 2 then do;
bad_syntax:
                count = count - 1;                           /* Strip NL off "commands " */
                call ioa_ ("Improper:  ^a", commands);
                call resetread;
                go to next;
           end;
           brk1 = edct + 2;
           break = substr (commands, edct + 1, 1);           /*  Pick up the delimiting character.  */
           i = index (substr (commands, brk1), break);
           if i = 0 then go to bad_syntax;
           j = index (substr (commands, i + brk1), break);
           if j = 0 then j = length (commands) - i - brk1 + 1;
           edct = edct + i + j + 1;                          /*  Continue scanning edit line.  */
           globsw = "0"b;                                    /*  Assume only one change.  */
           n = 1;                                            /*  Assume only one line changed.  */
nxarg:
           call get_token;
           if tkn ^= " " then do;                            /*  If token there, process it.  */
                if tkn = "g" then globsw = "1"b;             /*  Change all occurrances.  */
                else call cv_num;
                go to nxarg;                                 /*  Try for another argument.  */
           end;
           if linel = 0 then go to skipch;                   /*  Skip changing empty line.  */
```

```
ch1:      changes_occurred = "0"b;
          m, ij, l = 1;                                   /* indexes to strings */
          if i = 1 then do;                               /* add to beginning of line */
               changes_occurred = "1"b;
               located = 1;
               substr (tlin, 1, j - 1) = substr (commands, brk1 + i, j - 1);
                                                          /* copy part to be added */
               substr (tlin, j, length (line)) = line;    /* copy old line */
               ij = j + linel - 1;
               l = j + linel + 1;
               go to cprt;
          end;
ch2:      k = index (substr (line, m), substr (commands, brk1, i - 1));
                                                          /* locate what is to be changed */
          if k ^= 0 then do;
               substr (tlin, ij, k - 1) = substr (line, m, k - 1);
                                                          /* copy line up to change */
               substr (tlin, ij + k - 1, j - 1) = substr (commands, brk1 + i, j - 1);
                                                          /* put in change */
               m = m + k + i - 2;                         /* increment indexes */
               ij = ij + k + j - 2;
               l = l + k + j - 2;
               changes_occurred = "1"b;                   /* indicate that you did someting */
               located = 1;
               if globsw then go to ch2;
          end;
          substr (tlin, ij, length (line) - m + 1) = substr (line, m);
                                                          /* copy rest of line */
          ij = ij + length (line) - m;
          l = l + length (line) - m;
cprt:
          if changes_occurred then do;                    /* Write changes */
               call iox_$put_chars (iox_$user_output, addr (tlin), l, code);
               if code ^= 0 then do;
                    call com_err_ (code, myname, "Error writing change line");
                    go to fifish;
               end;
          end;
          linel = ij;
          line = substr (tlin, 1, ij);

skipch:   if n <= 1 then do;                              /* fifished */
               if located = 0 then do;
                    count = count - 1;                    /* Get rid of NL i "commands" */
                    call ioa_ ("Nothing changed by:  ^a", commands);
                                                          /* if not located */
                    call resetread;
               end;
               go to next;
          end;
          n = n - 1;
          call put;
          call get;
          go to ch1;



/* ********* top ********* */
```

```
top:        call copy;
            call switch;
            go to next;

/* ********* bottom ********* */

bottom:     call copy;                                   /* No line buffer */
            linel = 0;
            go to pinput;

/* ********* backup ********* */
                                                         /* save ptrs */
backup:     i = indt;
            call copy;
            call switch;                                 /* restore ptrs */
            indf = i + 1;                                /*  Note that "n" starts negative.  */
            do n = n to 0;
                 j = index (reverse (substr (from_seg, 1, indf - 1)), NL);
                 if j ^= 0 then indf = indf - j;         /*  First line case */
                 else if n = 0 then indf = 0;
                 else do;                                /* went off top of file */
                      linel = 0;
                      n = 1;
                      indt, indf = 0;
                      go to eof;
                 end;
            end;                                         /* line starts as indt */
            indt = indf;
            substr (to_seg, 1, indt) = substr (from_seg, 1, indt);
                                                         /* move in top of file */
                                                         /* fifd end of line */
            do indf = indt + 1 by 1 to csize;
                 substr (line, indf - indt, 1) = substr (from_seg, indf, 1);
                                                         /* move into line */

                 if substr (from_seg, indf, 1) = NL      /* search for end of line */
                 then go to line_end;
            end;
            indf = csize;
line_end:
            linel = indf - indt;
            n = 1;
            go to print1;

/* ********** "file" request ********** */
                                             /*  Finish copy.  */
file:       call copy;                       /*  Save it.  */
            call save;                       /*  Terminate source and release temp segs */
fifish:     call clean_up;                   /*  Return to command processor */
            return;

/* ********** write save ********** */
                                             /*  Finish copy.  */
wsave:      call copy;                       /*  Save it.  */
            call save;                       /*  Continue accepting requests.  */
            go to next;

/* ********* eof ********* */
                                             /* Remove NL */
eof:        count = count - 1;
            call ioa_ ("End of File reached by:^/^a", commands);
            call resetread;
            go to next;
```

```
/* ********* I N T E R N A L   P R O C E D U R E S ********* */



copy: procedure;                                          /* copy rest of file into to file */
          substr (to_seg, indt + 1, length (line)) = line;
                                                          /*  Copy current line.  */
          indt = indt + length (line);
          linel = 0;                                      /* No more line */
          if csize = 0
          then return;                                    /*  If new input, then no copy needed.  */
          ij = csize - indf;                              /* do rest of file */
          if ij > 0
          then substr (to_seg, indt + 1, ij) = substr (from_seg, indf + 1, ij);
          indt = indt + ij;                               /* set counters */
          indf = csize;
          return;

     end copy;


save: procedure;                                          /*  Procedure to write out all or part of "to" buffer.  */
          if source_ptr = null then do;                   /* Must be a new segment */
               call hcs_$make_seg (dir_name, entry_name, "", 01010b, source_ptr, code);
               if code ^= 0 then do;
                    call com_err_ (code, myname, "Cannot create ^a^[>^]^a.",
                         dir_name, (dir_name ^= ">"), entry_name);
                    return;
               end;
          end;
          substr (source_seg, 1, indt) = substr (to_seg, 1, indt);
          call hcs_$set_bc_seg (source_ptr, indt * 9, code);
          if code = 0
          then call hcs_$truncate_seg (source_ptr, divide (indt + 3, 4, 19, 0), code);
          if code ^= 0 then do;
               call com_err_ (code, myname, "Cannot truncate/set bit count (^d) on ^a^[>^]^a",
                    indt * 9, dir_name, (dir_name ^= ">"), entry_name);
          end;
          return;

     end save;


put:
     procedure;
          substr (to_seg, indt + 1, length (line)) = line;  /* do move */
          indt = indt + length (line);                    /* set counters */
          linel = 0;                                      /*  Discard old line.  */
          return;
     end;


get:
     procedure;
          linel = 0;                                      /*  Reset current line length.  */
          if indf >= csize then go to eof;                                    /*  If no input left, give up.  */
          linel = index (substr (from_seg, indf + 1, csize - indf), NL);
                                                          /*  Find the next new line.  */
          if linel = 0 then linel = csize - indf;         /*  If no nl found, treat end of segment as one.  */
```

```
          line = substr (from_seg, indf + 1, linel);       /*  Return the line to caller.  */
          indf = linel + indf;                             /*  Move the "from" pointer ahead one line.  */
          return;
     end;

switch:
     procedure;                                            /* make from-file to file, and v.v. */
          exptr = from_ptr;
          from_ptr = to_ptr;
          to_ptr = exptr;
          csize = indt;
          indt, indf = 0;
          linel = 0;
          return;

     end switch;

resetread:
     procedure;                                            /*  Call i/o system reset read entry.  */
                                                           /* In one place to centralize error handling */

          call iox_$control (iox_$user_input, "resetread", null (), code);
          if code ^= 0 then call com_err_ (code, myname, "Cannot resetread user_input");
          return;

     end resetread;

get_token:
     procedure;

declare (token_lth, white_lth) fixed binary (21);

          tkn = " ";                                       /* Set for easy failure */
          white_lth = verify (substr (commands, edct), WHITESPACE) - 1;
          if white_lth < 0 then return;                    /* Only whitespace left */
          edct = edct + white_lth;
          token_lth = search (substr (commands, edct), WHITESPACE) - 1;
          if token_lth < 0 then token_lth = length (commands)- edct;
          tkn = substr (commands, edct, token_lth);        /* Extract token */
          edct = edct + token_lth;
          return;

     end get_token;


get_num:
     procedure;                                            /*  Routine to convert token to binary integer.  */
          call get_token;                                  /*  Delimit the token.  */
cv_num:
     entry;                                                /*  Enter here if token already available.  */
          n = cv_dec_ (tkn);                               /*  Convert it.  */
          if n = 0 then n = 1;                             /*  Default count is 1.  */
          return;

     end get_num;

clean_up:
     procedure;

          call release_temp_segments_ (myname, temp_segs, (0));
          if source_ptr ^= null then call hcs_$terminate_noname (source_ptr, (0));
```

```
	end clean_up;

	end eds;
```

SOURCE FILES USED IN THIS COMPILATION.

LINE	NUMBER	DATE MODIFIED	NAME	PATHNAME
	0	05/06/80 1456.1	eds.pl1	>user_dir_dir>Multics>JRDavis>doc>ag90>eds.pl1

NAMES DECLARED IN THIS COMPILATION.

IDENTIFIER	OFFSET	LOC	STORAGE CLASS	DATA TYPE	ATTRIBUTES AND REFERENCES (* indicates a set context)
NAMES DECLARED BY DECLARE STATEMENT.					
NL		003750	constant	char(1)	initial unaligned dcl 51 ref 224 261 263 402 418 504
WHITESPACE		000001	constant	char(3)	initial unaligned dcl 53 ref 539 542
addr				builtin function	dcl 78 ref 149 149 174 174 174 174 182 182 189 192 193 193 211 213 213 239 247 259 259 270 286 286 286 286 286 286 304 309 310 312 313 331 333 333 338 338 341 343 352 352 354 355 358 358 365 370 416 443 453 453 455 493 493 494 507 539 542 543 544
break		000100	automatic	char(1)	unaligned dcl 5 set ref 309* 310 312
brk1		000101	automatic	fixed bin(17,0)	dcl 6 set ref 308* 310 312 313 331 338 343
buffer		000102	automatic	char(210)	unaligned dcl 7 set ref 149 149 149 149 174 174 174 174 182 182 182 182 189 192 193 211 213 247 259 259 304 309 310 312 313 331 338 343 370 443 539 542 543 544
changes_occurred		000167	automatic	bit(1)	unaligned dcl 8 set ref 326* 329* 348* 356
cleanup		000470	stack reference	condition	dcl 77 ref 109
code		000170	automatic	fixed bin(35,0)	dcl 9 set ref 91* 92 93* 99* 100 101* 113* 115 117* 124* 125 126* 149* 151 152* 182* 184 185* 286* 287 288* 358* 359 360* 471* 472 473* 479* 480 480* 482 483* 527* 528 528*
com_err_		000010	constant	entry	external dcl 59 ref 93 101 117 126 152 185 288 360 473 483 528
commands			based	char	unaligned dcl 10 set ref 174 174 174 174 189 192 193 211 213 247 259 259 304* 309 310 312 313 331 338 343 370* 443* 539 542 543 544
count		000171	automatic	fixed bin(21,0)	dcl 12 set ref 149* 155 174 174 174 174 182* 189 189 192 193 211 213 247 259 259 301 302* 302 304 304 309 310 312 313 331 338 343 369* 369 370 370 442* 442 443 443 539 542 543 544
csize		000172	automatic	fixed bin(21,0)	dcl 13 set ref 135* 140* 141 141 223 224 227 229 229 231 232 251 253 263 264 415 421 457 459 463 503 504 506 517*
cu_$arg_ptr		000012	constant	entry	external dcl 60 ref 91
cv_dec_		000014	constant	entry	external dcl 61 ref 556
dir_name		000174	automatic	char(168)	unaligned dcl 15 set ref 99* 113* 117* 117 471* 473* 473 483* 483
divide				builtin function	dcl 78 ref 140 480 480
edct		000173	automatic	fixed bin(17,0)	dcl 14 set ref 156* 211 213 247 248* 248 259 259 308 309 314* 314 539 541* 541 542 543 544 545* 545
entry_name		000246	automatic	char(32)	unaligned dcl 16 set ref 99* 113* 117* 137* 471* 473* 483*
error_table_$noentry		000052	external static	fixed bin(35,0)	dcl 84 ref 115
expand_pathname_		000016	constant	entry	external dcl 62 ref 99
exptr		000256	automatic	pointer	dcl 17 set ref 514* 516
from_ptr		000260	automatic	pointer	dcl 18 set ref 130* 141 224 229 239 240 259 261 263 266 270 402 413 416 418 460 504 507 514 515*
from_seg			based	char(1048576)	unaligned dcl 19 set ref 141* 224 229 239 240 259 261 263 266 270 402 413 416 418 460 504 507
get_temp_segments_		000020	constant	entry	external dcl 63 ref 124
globsw		000262	automatic	bit(1)	unaligned dcl 21 set ref 315* 320* 350
hcs_$initiate_count		000022	constant	entry	external dcl 64 ref 113
hcs_$make_seg		000024	constant	entry	external dcl 66 ref 471
hcs_$set_bc_seg		000026	constant	entry	external dcl 68 ref 479
hcs_$terminate_noname		000030	constant	entry	external dcl 69 ref 566

hcs_$truncate_seg	000032 constant	entry	external dcl 70 ref 480
i	000263 automatic	fixed bin(21,0)	dcl 22 set ref 201* 222* 259* 260 261 262 310* 311 312 313 314 328 331 338 343 345 397* 400
ij	000264 automatic	fixed bin(21,0)	dcl 23 set ref 327* 334* 341 343 346* 346 352 354* 354 364 365 459* 460 460 460 462
index		builtin function	dcl 78 ref 224 259 261 263 310 312 338 402 504
indf	000265 automatic	fixed bin(21,0)	dcl 24 set ref 135* 220 227 231* 237* 240 240 242 250 251 259 261 262 264* 265* 268 400* 402 403* 403 404* 408* 412 415* 416 416 418* 421* 422 459 460 463* 503 504 504 506 507 508* 508 518*
indt	000266 automatic	fixed bin(21,0)	dcl 25 set ref 135* 229 232* 232 240 242* 242 249 266 269* 269 397 408* 412* 413 413 415 416 422 453 455* 455 460 462* 462 478 478 479 480 480 483 493 494* 494 517 518*
ioa_	000034 constant	entry	external dcl 71 ref 137 147 174 180 282 304 370 443
iox_$control	000036 constant	entry	external dcl 72 ref 527
iox_$get_line	000040 constant	entry	external dcl 73 ref 149 182
iox_$put_chars	000042 constant	entry	external dcl 74 ref 286 358
iox_$user_input	000050 external static	pointer	dcl 83 set ref 149* 182* 527*
iox_$user_output	000046 external static	pointer	dcl 83 set ref 286* 358*
j	000267 automatic	fixed bin(21,0)	dcl 26 set ref 220* 223 224 224 235* 235 237 239 249* 255 255 257* 263* 264 265 312* 313 313* 314 331 331 333 334 335 343 343 346 347 402* 403 403
k	000270 automatic	fixed bin(21,0)	dcl 27 set ref 224* 226 235 238 239 261* 262 262* 262 263 263 265 266 266 268 269 270 338* 340 341 341 343 345 346 347
l	000271 automatic	fixed bin(21,0)	dcl 28 set ref 327* 335* 347* 347 355* 355 358*
length		builtin function	dcl 78 ref 149 149 174 174 182 182 192 211 247 259 286 286 313 333 352 354 355 453 455 493 494 543
line	based	char	unaligned dcl 29 set ref 193* 213* 239* 270* 286 286 286 286 333 333 338 341 352 352 354 355 365* 416* 453 453 455 493 493 494 507*
line_buffer	000272 automatic	char(210)	unaligned dcl 30 set ref 193 213 239 270 286 286 286 286 333 333 338 341 352 352 354 355 365 416 453 453 455 493 493 494 507
linel	000357 automatic	fixed bin(17,0)	dcl 31 set ref 192* 193 204* 211* 213 228* 238* 239 239 240 240 242 268* 270 270 281 286 286 286 286 324 333 333 334 335 338 341 352 352 354 355 364* 365 392* 406* 416 422* 453 453 455 456* 493 493 494 495* 502* 504* 506 506* 507 507 508 519*
located	000360 automatic	fixed bin(17,0)	dcl 32 set ref 300* 330* 349* 368
m	000361 automatic	fixed bin(21,0)	dcl 33 set ref 220* 229 229 229 232 240 240 240 242 250* 257* 266 266 266 269 327* 338 341 345* 345 352 352 354 355
myname	000000 constant	char(3)	initial unaligned dcl 55 set ref 93* 93* 101* 117* 124* 126* 152* 185* 288* 360* 473* 483* 528* 565*
n	000362 automatic	fixed bin(21,0)	dcl 34 set ref 201 219 222 251* 253 255* 256* 259 271* 292* 292 293 316* 367 376* 376 401* 401* 404 407* 424* 556* 557 557*
null		builtin function	dcl 78 ref 107 108 115 136 470 527 527 566
release_temp_segments_	000044 constant	entry	external dcl 75 ref 565
reverse		builtin function	dcl 78 ref 261 402
search		builtin function	dcl 78 ref 542
sname	based	char	unaligned dcl 35 set ref 99* 101*
sname_lth	000363 automatic	fixed bin(21,0)	dcl 36 set ref 91* 99 99 101 101
sname_ptr	000364 automatic	pointer	dcl 37 set ref 91* 99 101
source_count	000366 automatic	fixed bin(24,0)	dcl 38 set ref 113* 140
source_ptr	000370 automatic	pointer	dcl 39 set ref 107* 113* 115 136 141 470 471* 478 479* 480* 566 566*
source_seg	based	char(1048576)	unaligned dcl 40 set ref 141 478*

```
substr                                                  builtin function     dcl 78 set ref 141* 141 174 174 189 213 224 229* 229
                                                                               239 240* 240 259 259 261 263 266* 266 270 309 310
                                                                               312 331* 331 333* 338 338 341* 341 343* 343 352*
                                                                               352 365 402 413* 413 416* 416 418 453* 460* 460
                                                                               478* 478 493* 504 507 539 542 544
temp_segs                     000372 automatic          pointer              array dcl 42 set ref 108* 124* 130 131 565*
tkn                           000464 automatic          char(8)              unaligned dcl 44 set ref 159 160 161 162 163 164 165
                                                                               166 167 168 169 170 319 320 538* 544* 556*
tlin                          000376 automatic          char(210)            unaligned dcl 43 set ref 331* 333* 341* 343* 352*
                                                                               358 358 365
to_ptr                        000466 automatic          pointer              dcl 47 set ref 131* 229 240 266 413 453 460 478 493
                                                                               515 516*
to_seg                               based              char(1048576)        unaligned dcl 45 set ref 229* 240* 266* 413* 453*
                                                                               460* 478 493*
token_lth                     000554 automatic          fixed bin(21,0)      dcl 536 set ref 542* 543 543* 544 545
verify                                                  builtin function     dcl 78 ref 539
white_lth                     000555 automatic          fixed bin(21,0)      dcl 536 set ref 539* 540 541

NAMES DECLARED BY EXPLICIT CONTEXT.
backup                        002353 constant           label                dcl 397 ref 219
bad_syntax                    001656 constant           label                dcl 302 ref 247 311
bottom                        002350 constant           label                dcl 391 ref 169
ch1                           002017 constant           label                dcl 326 ref 379
ch2                           002064 constant           label                dcl 338 ref 350
change                        001652 constant           label                dcl 300 ref 165
clean_up                      003301 constant           entry                internal dcl 562 ref 109 127 431
copy                          002527 constant           entry                internal dcl 452 ref 274 385 391 398 429 436
cprt                          002227 constant           label                dcl 356 ref 336
cv_num                        003254 constant           entry                internal dcl 554 ref 321
dellin                        001221 constant           label                dcl 199 ref 166
eds                           000231 constant           entry                external dcl 1
eof                           002500 constant           label                dcl 442 ref 227 233 409 503
file                          002466 constant           label                dcl 429 ref 164
fifish                        002470 constant           label                dcl 431 ref 153 186 289 361
get                           003040 constant           entry                internal dcl 500 ref 202 295 378
get_num                       003251 constant           entry                internal dcl 551 ref 199 218 280
get_token                     003172 constant           entry                internal dcl 533 ref 157 317 553
input                         001131 constant           label                dcl 182 ref 194
insert                        001237 constant           label                dcl 209 ref 159
line_end                      002460 constant           label                dcl 422 ref 418
locate                        001401 constant           label                dcl 247 ref 161
n_eof                         001312 constant           label                dcl 227 ref 223
nexlin                        001253 constant           label                dcl 218 ref 163
next                          000706 constant           label                dcl 149 ref 155 176 205 214 243 276 293 306 374 387
                                                                               438 445
noline                        001643 constant           label                dcl 292 ref 283
nxarg                         001775 constant           label                dcl 317 ref 322
pedit                         000673 constant           label                dcl 147 ref 189
pinput                        001116 constant           label                dcl 180 ref 138 170 393
print                         001554 constant           label                dcl 280 ref 162
print1                        001573 constant           label                dcl 286 ref 272 296 425
put                           003025 constant           entry                internal dcl 491 ref 191 209 221 252 294 377
resetread                     003110 constant           entry                internal dcl 524 ref 175 305 372 444
retype                        001240 constant           label                dcl 211 ref 160
save                          002565 constant           entry                internal dcl 469 ref 430 437
skipch                        002304 constant           label                dcl 367 ref 324
switch                        003073 constant           entry                internal dcl 512 ref 254 275 386 399
top                           002345 constant           label                dcl 385 ref 168
wsave                         002475 constant           label                dcl 436 set ref 167
```

THERE WERE NO NAMES DECLARED BY CONTEXT OR IMPLICATION.

STORAGE REQUIREMENTS FOR THIS PROGRAM.

	Object	Text	Link	Symbol	Defs	Static
Start	0	0	4204	4260	3752	4214
Length	4474	3752	54	200	231	0

BLOCK NAME	STACK SIZE	TYPE	WHY NONQUICK/WHO SHARES STACK FRAME
eds	574	external procedure	is an external procedure.
on unit on line 109	64	on unit	
copy		internal procedure	shares stack frame of external procedure eds.
save		internal procedure	shares stack frame of external procedure eds.
put		internal procedure	shares stack frame of external procedure eds.
get		internal procedure	shares stack frame of external procedure eds.
switch		internal procedure	shares stack frame of external procedure eds.
resetread		internal procedure	shares stack frame of external procedure eds.
get_token		internal procedure	shares stack frame of external procedure eds.
get_num		internal procedure	shares stack frame of external procedure eds.
clean_up	80	internal procedure	is called by several nonquick procedures.

STORAGE FOR AUTOMATIC VARIABLES.

STACK FRAME	LOC	IDENTIFIER	BLOCK NAME
eds	000100	break	eds
	000101	brk1	eds
	000102	buffer	eds
	000167	changes_occurred	eds
	000170	code	eds
	000171	count	eds
	000172	csize	eds
	000173	edct	eds
	000174	dir_name	eds
	000246	entry_name	eds
	000256	exptr	eds
	000260	from_ptr	eds
	000262	globsw	eds
	000263	i	eds
	000264	ij	eds
	000265	indf	eds
	000266	indt	eds
	000267	j	eds
	000270	k	eds
	000271	l	eds
	000272	line_buffer	eds
	000357	linel	eds
	000360	located	eds
	000361	m	eds
	000362	n	eds
	000363	sname_lth	eds
	000364	sname_ptr	eds
	000366	source_count	eds
	000370	source_ptr	eds
	000372	temp_segs	eds
	000376	tlin	eds
	000464	tkn	eds
	000466	to_ptr	eds
	000554	token_lth	get_token
	000555	white_lth	get_token

THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM.

```
r_ne_as          alloc_cs     call_ext_out_desc   call_ext_out   call_int_this   call_int_other
return           enable       shorten_stack       ext_entry      int_entry       set_cs_eis
index_cs_eis
```

THE FOLLOWING EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM.

```
com_err_                  cu_$arg_ptr              cv_dec_                     expand_pathname_
get_temp_segments_        hcs_$initiate_count      hcs_$make_seg               hcs_$set_bc_seg
hcs_$terminate_noname     hcs_$truncate_seg        ioa_                        iox_$control
iox_$get_line             iox_$put_chars           release_temp_segments_
```

THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM.

```
error_table_$noentry      iox_$user_input          iox_$user_output
```

LINE	LOC	LINE	LOC	LINE	LOC	LINE	LOC	LINE	LOC	LINE	LOC	LINE	LOC
1	000230	91	000236	92	000254	93	000256	94	000310	99	000311	100	000341
101	000343	102	000375	107	000376	108	000400	109	000413	113	000435	115	000477
117	000507	119	000553	124	000554	125	000575	126	000577	127	000623	128	000627
130	000630	131	000632	135	000634	136	000637	137	000643	138	000663	140	000664
141	000667	147	000673	149	000706	151	000731	152	000733	153	000757	155	000760
156	000763	157	000765	159	000766	160	000773	161	001000	162	001005	163	001012
164	001017	165	001024	166	001031	167	001036	168	001043	169	001050	170	001055
174	001062	175	001113	176	001115	180	001116	182	001131	184	001154	185	001156
186	001202	189	001203	191	001212	192	001213	193	001215	194	001220	199	001221
201	001222	202	001232	203	001233	204	001235	205	001236	209	001237	211	001240
213	001243	214	001252	218	001253	219	001254	220	001256	221	001261	222	001262
223	001271	224	001274	226	001311	227	001312	228	001315	229	001316	231	001333
232	001335	233	001340	235	001341	236	001342	237	001344	238	001346	239	001350
240	001357	242	001374	243	001400	247	001401	248	001404	249	001405	250	001407
251	001411	252	001414	253	001415	254	001421	255	001422	256	001427	257	001430
259	001432	260	001451	261	001453	262	001467	263	001474	264	001510	265	001514
266	001516	268	001532	269	001535	270	001541	271	001546	272	001550	274	001551
275	001552	276	001553	280	001554	281	001555	282	001557	283	001572	286	001573
287	001614	288	001616	289	001642	292	001643	293	001645	294	001647	295	001650
296	001651	300	001652	301	001653	302	001656	304	001660	305	001706	306	001707
308	001710	309	001713	310	001720	311	001736	312	001737	313	001757	314	001765
315	001772	316	001773	317	001775	319	001776	320	002003	321	002013	322	002014
324	002015	326	002017	327	002020	328	002024	329	002027	330	002031	331	002033
333	002047	334	002054	335	002060	336	002063	338	002064	340	002110	341	002112
343	002130	345	002147	346	002154	347	002161	348	002166	349	002170	350	002172
352	002174	354	002217	355	002223	356	002227	358	002231	359	002250	360	002252
361	002276	364	002277	365	002301	367	002304	368	002307	369	002311	370	002313
372	002336	374	002337	376	002340	377	002342	378	002343	379	002344	385	002345
386	002346	387	002347	391	002350	392	002351	393	002352	397	002353	398	002355
399	002356	400	002357	401	002362	402	002366	403	002403	404	002407	406	002413

407 002414	408 002416	409 002420	411 002421	412 002423	413 002425	415 002432
416 002442	418 002450	420 002454	421 002456	422 002460	424 002463	425 002465
429 002466	430 002467	431 002470	432 002474	436 002475	437 002476	438 002477
442 002500	443 002502	444 002525	445 002526	452 002527	453 002530	455 002536
456 002540	457 002541	459 002544	460 002546	462 002560	463 002562	464 002564
469 002565	470 002566	471 002572	472 002631	473 002633	475 002700	478 002701
479 002707	480 002725	482 002746	483 002750	486 003024	491 003025	493 003026
494 003034	495 003036	496 003037	500 003040	502 003041	503 003042	504 003045
506 003062	507 003066	508 003071	509 003072	512 003073	514 003074	515 003076
516 003100	517 003102	518 003104	519 003106	520 003107	524 003110	527 003111
528 003143	529 003171	533 003172	538 003173	539 003175	540 003214	541 003220
542 003221	543 003240	544 003244	545 003247	546 003250	551 003251	553 003252
554 003253	556 003255	557 003274	558 003277	562 003300	565 003306	566 003330
568 003347						

SECTION 7

ABSENTEE FACILITY

A common programming pattern is to develop a program online, using debugging tools and the ability to interactively try a variety of test cases to check on a program's correctness. After the program is working, one may wish to do a large "production" run. Since the production run may produce much output or take much time, the programmer does not wish to wait at his terminal for the results. Production runs on Multics are best done using "absentee" jobs.

An absentee job uses Multics in the same way that a person does, except that instead of being associated with a terminal, its input comes from a file, and its output goes to a file. It is like "batch" jobs provided by other systems. The language used in absentee jobs is the same as the interactive command language. No special knowledge is required to write absentee job control files. At its simplest, an absentee job is just a collection of commands to be executed.

An absentee job runs in an environment similar to that of an interactively logged-in user. The job logs in in the user's home directory, and runs the user's start_up.ec, if any. This must be kept in mind when writing a start_up.ec, and when submitting absentee jobs: beginning users often err in falsely assuming that absentee jobs log in to the directory from which they were submitted.

An absentee control file has the suffix "absin." An absentee job is submitted by supplying the name of the absin file to the enter_abs_request command. The absentee job is placed in a queue and run as background to the normal interactive work of the system. This technique allows the system to utilize its resources most effectively, by keeping a queue of jobs that can always be run, and preempted for serving interactive users. For these reasons, the charging rate for absentee jobs is normally substantially lower than for interactive work.

Output from the absentee job goes into a file whose name is the same as the absin segment with the suffix "absout" instead of "absin". When the job completes, this segment may be printed by the user.

For example, suppose that the prime program used in the section on performance is to used to check the prime-ness of the first five integers.

```
!
primep ([index_set 5])
   1 is  a prime
   2 is  a prime
   3 is  a prime
   4 is not a prime
   5 is  a prime
 r 16:33 0.119 17
```

[The correct operation of the primep command is shown by brief testing, using the index_set active function, which returns the numbers from 1 to 5. The primep command is invoked with each of these values, and seems to work.

Next, an absin file is created using the qedx editor.]

```
! qedx
! a
! primep ([index_set 5])
! \f
! w t5.absin
! q
 r 1640.4 0.218 39
```

[Now that the absin has been created, it is submitted for execution.]

```
! enter_abs_request t5
 ID: 210805.1; 5 already requested
 r 1641.3 0.450 63
```

[Multics confirms the submission, giving the request id and the number of previously submitted jobs in the absentee queue. Often, many of these jobs may be "deferred," which is to say, they will not be run until a later time. Thus, "5 already requested" does not necessarily mean that five jobs must be run and completed before the newly-submitted job will run.]

```
! who -absentee

 Absentee users 3/9
 Franklin.Mint*
 Gibson.YORMA*
 Grant.States*
 r 1642.1 0.272 22
```

[The who command is used to print a list of all absentee jobs. It shows that there are three running, and a total of nine can run at the time. Absentee users are identified by the asterisk after their project.

When the job is done, the user prints the output file.]

```
! print t5.absout
                    t5.absout 04/20/80  1643.6 est Sun

Absentee user Grant States logged in: 04/20/80  1641.4 mst Sun
r 16:41 2.364 55

primep ([index_set 5])
   1 is  a prime
   2 is  a prime
   3 is  a prime
   4 is not a prime
   5 is  a prime
r 16:42 0.198 20

abs_io_: Input stream exhausted.

Absentee user Grant States logged out 04/20/80  1643.1 mst Sun
CPU usage 3 sec, memory usage 1.0 units
```

With more advanced use of the absentee facility, the user can also supply arguments to be substituted inside the absentee control segment, make absentee job steps conditional, delay absentee work until a chosen time, and develop a periodic absentee job which is run, say, once every two days. This is possible because the absin segment is interpreted like an exec_com segment. All the power of the Multics command interpreter is available. The user can verify the correctness of the absentee job by running it as an exec_com.

The next example shows how absentee jobs can accept arguments.

```
! print p.absin

                    p.absin    04/20/80   1655.7 est Sun

primep ([index_set &1])

r 16:55 .110 19
```

[This absentee segment accepts one argument. The character string "&1" is replaced by the argument wherever it occurs. To test this absin segment, the user invokes it as an exec_com. In order to use the segment as an exec_com, it must have a name with suffix "ec" added to it.]

```
! add_name p.absin p.ec
 r 1656.3 0.100 5

! exec_com p 2
 primep ([index_set 2])
    1 is  a prime
    2 is  a prime
r 1700.1 0.210 30
```

[The exec_com is invoked with the argument 2. As it runs, it prints the commands in the file. The argument mechanism seems to work, so the user submits an absentee job.]

```
! enter_abs_request p -arguments 100
 ID: 221023.4; 6 already requested.
 r 17:05 0.273 50
```

[Here, the argument 100 is passed to the absentee job. The user goes about other business while the request runs.]

For further information, see the MPM Commands manual description of the enter_abs_request and exec_com commands. The exec_com command is also discussed in Part II of the New User's Introduction to Multics.

SECTION 8

LARGE FILES IN MULTICS

A frequent point of confusion about Multics concerns the handling of large data files within the segmented virtual memory environment. A file, in Multics terminology, is a (usually structured) collection of data of arbitrary size. A file which happens to require less than 256K words of storage is usually stored in a single segment of the Multics storage system, and is addressed by mapping the segment containing the entire file into the current address space. Source and object programs, and small, linear ASCII text files are examples of files handled this way. A file which is larger than 256K words (or which is smaller but may someday grow that large) is usually stored in several segments in a single directory in the Multics storage system, and is addressed by mapping relevant parts (records) of the file into the current address space. The directory contains, in addition to the raw data of the file, any maps or indexes needed to maintain its internal organization. Three file management facilities (sometimes called Access Methods on IBM systems) are available to handle the details of setting up, indexing, and searching of files. These are:

1. Multisegment files (MSF's): There is a system-wide standard format for ASCII text files which require more than 256K words of storage. Most translators, for example, are prepared to produce very long output listings for the printer using this format; the high speed line printer facilities also recognize the format. Other system facilities use multisegment files for objects other than ASCII text files. See the description of the msf_manager_ subroutine in the MPM Subsystem Writer's Guide, Order No. AK92.

2. vfile_: A general purpose file manipulation system that provides sequential record files, indexed (keyed) record files, and stream (unstructured) files. vfile_ is an "I/O module" (see the MPM Reference Guide) and is not called directly, but rather through the Multics I/O system, and its interface, the iox_ subroutine.

   The size of files managed by vfile_ is practically limitless. The files are accessed using the virtual memory: one calls the I/O system giving the index or key of the record desired; vfile_ (via the I/O system) can either return the contents of the record into a buffer, or return a pointer to the location of that record in the address space, and the program then can manipulate the contents of the record using, for example, a PL/I based structure. vfile_ provides interlocking facilities for multiple users, and also guarantees integrity of a file in the case where a system failure occurs while the user is updating the file. For further information, see the descriptions in the MPM Reference Guide and the vfile_ I/O module in the MPM Subroutines Guide.

3. PL/I record-oriented I/O: The full ANSI standard PL/I I/O system is implemented on Multics, allowing construction of a data manipulation system which is in principle system independent. Since the PL/I I/O system uses vfile_ (2, above) very large files can be efficiently set up, updated, and searched using only the PL/I language. For further information, one should consult the Multics PL/I language specification, Order No. AG94.

In addition, users with unusually sophisticated needs such as completely inverted files, files with indexes on different elements, etc., will find that appropriate facilities can easily be developed using the virtual memory combined with techniques similar to those used by vfile_. It is important to realize that vfile_, while organized as a subsystem, is written in PL/I, using only Multics facilities which are also available to the user. Thus, a user could construct his own file management facility, providing facilities not offered by vfile_ without recourse to special privileges or need to modify the Multics supervisor.

Finally, the Multics I/O system, which is organized to allow attachment of arbitrary source-sink I/O devices, may be used to read and write magnetic tape in any of several formats, or detachable disk packs, for applications in which permanent on-line storage is not appropriate. See the "Multics Peripheral I/O" manual, order No. AX49, for further details on these matters.

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

CUT ALONG LINE

TITLE: LEVEL 68 MULTICS PROGRAMMER'S MANUAL

ORDER NO. AG90-02

DATED MAY 1980

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ______________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell