SEGMENTED-VIRTUAL MEMORY DESIGN FOR AN ALGOL 68 COMPILER

By

MARK GOTO

Bachelor of University Studies

Oklahoma State University

Stillwater, Oklahoma

1975

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1978

SEGMENTED-VIRTUAL MEMORY DESIGN FOR

AN ALGOL 68 COMPILER

THESIS APPROVED:

G. E. Hedrick
Thesis Adviser

J P Chandler

James R. Van Doren

Norman N Durham
Dean of Graduate College

1011892

PREFACE

This thesis is a description of a design for an ALGOL 68 run-time organization. The design relies heavily on a segmented-virtual memory scheme for simulating a large memory store and handling memory management requests. The author would like to thank each of the members of the Computing and Information Sciences Department who have made his study at O.S.U. enjoyable, and especially Dr. G. E. Hedrick who has been more than his advisor. The author would also like to acknowledge the support of the National Science Foundation for sponsoring this research under grant NSF-MCS576-06090.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objectives

Since 1973 a project has been underway at Oklahoma State University to write a portable compiler for the ALGOL 68 language (1) (2) (3) (4). This project was started by an implementation of a subset translator and an interpretive executor with the original intent of providing a scientific subset compiler (1). Since that time, many modifications and additions have been incorporated with the long range goal of providing full support of the ALGOL 68 language. As a result of this work, it has been recognized by people concerned with this project that the run-time environment provided by the current interpretive executor is inadequate for the expanding implementation and on-going work. Therefore, it was decided to revise the present executor so as to increase the flexibility of the storage management functions.

Prior to 1977, the Oklahoma State University ALGOL 68 compiler had only limited storage management beyond the classical stack environment. The main objective of this thesis is to present a run-time environment design that would simplify implementation of non-LIFO storage allocation such as ALGOL 68 heap generators, flexible multiple values and transput-file information. Other objectives of this design are to handle large ALGOL 68 user storage demands on small computers and to ease implementation of ALGOL 68 parallel processing.

History of the Oklahoma State University ALGOL 68 compiler

In 1973, John Jensen implemented a scientific subset compiler for ALGOL 68 on an IBM 1130 with 8K 16-bit words of storage (1)[1]. This original version (referred to here as Version I) of the Oklahoma State University ALGOL 68 compiler has developed into an implementation that supports a sizable subset of the language.

Major contributions have come from thesis work by Roger Berry (2), Alan Eyler (3), Walter Seay (4) and Alan Robertson (5). This includes development of a subset ALGOL 68 transput package, addition of procedures to the Version I compiler and the enhancement of mode processing. Other contributions have come from several students who volunteered time to work on this project, most notably Larry Hanes, Charles Hanes and Alan Robertson.

----

[1]"1K" is equivalent to 1024.

Berry's (2) implementation of formatted transput resulted in an independent package which supports a subset of transput as defined by the "Report on the Algorithmic Language ALGOL 68" (6). Incorporating this package with the Version I compiler on an IBM 360/65 resulted in the Version II compiler in late 1975.

TABLE I

VERSIONS OF THE OSU ALGOL 68 COMPILER

Version	Description
I	John Jensen's original implementation on the IBM 1130 in 1973
II	Version I with Roger Berry's Transput package incorporated on the IBM 360 in 1975
III	Version I enhanced to support procedures implemented on the IBM 1130 in 1975
IV	Re-integration of Berry's Transput package, Eyler's implementation of procedures and Jensen's original version on the IBM 360 in 1976

Alan Eyler completed his work (3) on implementation of procedures during late 1975, but on the same machine as John Jensen's original work. This version (Version III) was later recombined with Version II and satisfactorily completed in early 1976 (Version IV).

In mid 1976, Walter Seay completed his work (4) on mode processing. It is mainly due to his modifications and the work on integrating the Version II and III compilers that the need for improving the run-time environment became crucial.

In 1978, Alan Robertson completed research on Transformational grammars (5), and proposed a system by which ALGOL 68 format denotations may be parsed and interpretively executed at run-time. In designing this addition to the compiler, he required some heap storage mechanizms that are easily provided by the design presented in this thesis. In fact, the consideration of Alan Robertson's design for processing formatted transput and the problems encountered by Walter Seay were the inspiration for this thesis.

The latest work on the Oklahoma State University ALGOL 68 compiler includes re-integration of Walter Seay's work with the Version IV compiler, updating the formatted transput package, testing the compiler on various different machines (TI ASC, IBM 370/158, CDC CYBER 175), and implementation of the design presented here.

Review of Current Work

The ALGOL 68 language, as defined in the "Revised Report on the Algorithmic language ALGOL 68" (7), is a very powerful language. Several implementation efforts on the language are currently in progress. Three of the most notable efforts which have come to the attention of the author are: 1) the Cambridge ALGOL 68 compiler, 2) the ALGOL 68 compiler of Paris-Sud University (Orsay) and 3) the Manchester ALGOL 68 compiler.

The Cambridge ALGOL 68 compiler has been implemented and used at O.S.U. and has proven to be a very efficient and fast compiler. Its drawbacks are the lack of numerical facilities, the lack of formatted transput and the difficulty involved in transporting the compiler onto non-370-like machines. Despite these drawbacks, it has excellent capabilities for use in writing compilers (ie. almost full implementation of modes and structures). Performance tests on an IBM 370/158 show that equivalent programs execute somewhere between one and a half to three times faster when using the Cambridge compiler as compared to the IBM PL/I optimizer.

The ALGOL 68 compiler of Paris-Sud University (Orsay) is not available at O.S.U. but its implementors claim (8) to have implemented almost all of the language as defined by the "Revised Report" (7). This compiler was implemented in FORTRAN on a Univac 1100-series computer and is a true compiler in the sense that it produces relocatable object code. The use of FORTRAN suggests that portability and readability were objectives in the initial choice of the implementation language. However, the authors admit that certain machine dependent features of the Univac FORTRAN V compiler were used and that the code generation phase is strongly dependent on the run-time computer.

The Manchester ALGOL 68 compiler was of interest due to the run-time environment design (9) (10), specifically, the use of segments. This presentation consists of a design that uses segments of a slightly different form, with the same goal of flexible storage management. This design uses segments as a means for handling storage management of flexible rowed values and heap generators in a similar manner to the way that the Manchester ALGOL 68 handles problems with these modes.

The above descriptions are a brief overview of current compilers whose design have points of similarity or have influenced this proposal. Other notable implementations of ALGOL 68 include ALGOL 68-R (11) and the Control Data Corporation ALGOL 68 compiler for the 6600 and similar computers.

Besides the current implementation efforts mentioned above, literature on ALGOL 68 and compiler writing topics have contributed to the design presented here. An excellent survey of the ALGOL 68 language is presented in the revised edition of An Informal Introduction to ALGOL 68 (12). Much of the information about the Manchester ALGOL 68 compiler appears with discussions of other compiler writing topics in the "Proceedings of the Vth Annual III Conference on Implementation and Design of Algorithmic Languages" (13). In addition, the author has used material from Gries (14) as a basis for designing parts of the run-time storage organization. Ideas for the remaining portion of the storage management scheme in this thesis came from discussions of operating systems storage management in Madnick and Donovan (15), especially the descriptions of MULTICS[2].

Notes on Terminology

The author's background in compiler writing and operating systems has resulted in the use of terminology from both fields. Terms used in this thesis such as pseudo-code, stack environment and display vector are borrowed from compiler writing, while terms such as layered design, page fault and virtual memory come from an operating systems background[3]. While Gries (14) and Madnick and Donovan (15) provide adequate definitions of several such terms, presentation of a few descriptions here are worthwhile.

Gries (14) describes a stack environment for a block structured language, as a large table of contiguous locations from which storage areas are allocated. These areas,

---

[2]The Multiplexed Information and Computing System (MULTICS) is described in a case study by Madnick and Donovan (15).

[3]In particular, the term "layer" as used in this thesis, is not the same as the ALGOL 68 term "layer" in the language definition.

which are called stack-frames, are allocated when a program block is entered and are de-allocated when the block is exited. Because the stack-frames are allocated in a last-in-first-out manner, the run-time storage organization is called a stack environment. In addition to the allocation of memory, a vector of table indices or addresses is maintained that indicate the beginning of each stack-frame allocation. This vector of addresses is called a display vector. Because this vector is updated and saved each time a stack-frame is created, a display vector exists for each stack-frame. For the purposes of this thesis, the storage allocated upon entering a block is called a stack-frame, the saved list of addresses that maps the allocations is called a display and a list of addresses that includes the most recent allocation (ie. the active stack-frame) is the display vector.

Gries (14) also described an interpreter as a program that performs two functions: 1) translates a source program into an internal form, and 2) executes (interprets or simulates) the program in this internal form. For the purposes of this thesis, the internal form of the source program is called pseudo-code (especially because it resembles machine code for a hypothetical ALGOL 68 machine) and the portion of the interpreter that performs the second function mentioned above is called an interpretive executor.

Madnick and Donovan (15) describe several operating systems besides MULTICS. MULTICS, however, has special significance in that it uses a segmented-virtual memory management scheme and had a design approach used known as a layered design. This thesis adopts the layered design approach and uses a form of segmented-virtual memory management that resembles that of MULTICS. In MULTICS, a section of high speed primary storage is used as a cache memory to which memory references are made. A virtual memory space is maintained by translating virtual addresses into cache memory addresses. Since the virtual memory size is much larger that the cache memory size, secondary storage is used to store subsections of the cache memory until needed. These subsections of cache memory are called pages in MULTICS and in this thesis. Additionally, a reference to a page that is not currently in cache memory causes a paging operation known as a page swap to occur (a page of cache memory is copied to secondary storage and the desired page of virtual memory is copied into cache memory). All memory that is addressable by a virtual address is considered to be pagable, as opposed to hardware registers or cache memory pages for which no address translation occurs.

Two ALGOL 68 terms of special significance are used throughout this thesis: 1) flexible rowed values and 2) heap values or storage. Rowed values in ALGOL 68 are values which may be accessed via a name and a subscript; ie., they correspond to Fortran or PL/I arrays. Flexible rowed values are values that may change their bounds after the declaration of the values has been executed. This is somewhat similar to the way PL/I varying strings may change in length. The consequences of this feature are that the storage requirements of a flexible rowed value may change after allocation of the rowed value is complete, which makes this type of storage allocation a special problem that cannot be handled by the stack environment described above.

Heap values also cannot be handled by a stack environment, because by definition, they must remain allocated as long as any references exists to the allocated storage, even after the block where the heap values were allocated has been exited[4]. Heap values may be of any valid ALGOL 68 mode which creates further problems in that data-handling and allocation techniques must exist for both heap and non-heap values.

For the most part, this thesis uses the same terminology as can be found is Gries (14) and Madnick and Donovan (15) unless otherwise stated. The above descriptions hopefully provide some added insight into the terms most frequently used by this thesis which are not commonly known.

---

[4]Note that "Heap" refers to specifically ALGOL 68 heap storage which is storage that remains allocated even after a program block is exited and which is garbage collected as the need is perceived.

CHAPTER II

PRESENT RUN-TIME ENVIRONMENT

Introduction

The OSU ALGOL 68 compiler generates pseudo-code which is interpretively executed. The interpretive execution is performed in a simulated memory with an extended stack model run-time environment. An excellent description of the overall compiler was delivered at the 1975 International Conference on ALGOL 68 (16).

This presentation is concerned with three aspects of the current run-time environment: 1) the run-time symbol table and run-time addressing, 2) the stack organization of memory, and 3) the simulated memory itself. The design presented here should either improve the performance of and/or increase the flexibility of each aspect.

Run-time Symbol Table and Addressing

The run-time symbol table is represented by a vector of active entries, linked lists of inactive entries and a table of character representations. The symbol table entries consist of two items: an absolute address of the storage associated with the variable and an indicator of the mode type. When a variable is accessed, a compiler-generated identifier number is retrieved from the pseudo-code instruction and used to index the active entry vector of the symbol table. The address of the variable location in memory is then obtained from the symbol table and the value of the variable is either fetched from or stored into the indicated location. If a pseudo-code instruction accesses more than one variable, then the above process is repeated for each variable.

Due to limitations imposed by the IBM 1130, the run-time symbol table was statically limited to 120 entry positions. Although later versions of the compiler were run on an IBM 360/65, expansion of the symbol table proved to be very difficult due to restrictions in the implementation language, namely FORTRAN. At that time, it was decided not to alter the symbol table structure because the enormous amount of re-design effort required was not available.

The design presented in this thesis eliminates the run-time symbol table by using display offset addressing. Each variable is addressed by an offset from a stack display address which is kept in a display vector. This scheme will remove the restriction of 120 active unique variables and can maintain the speed of address by keeping a copy of the active display vector in primary storage.

Run-time Stack Organization of Memory

The interpretive executor has a very primitive memory management scheme. The allocation of memory is performed according to a single stack organization. Corresponding to each new range of the ALGOL 68 program, a new stack frame is initialized by copying and updating the standard stack model display vector. At the end of each range, the current stack frame is released along with the storage associated with it. This stack organization plus a very simple heap allocation mechanism (with no storage reclamation) constitutes all of the storage management of the interpretive executor.

In order to increase the flexibility of storage management, a segmented-virtual memory scheme is used. Segments are used to allocate memory for two types of requests: 1) requests for memory that cannot be allocated and de-allocated in a stack environment, and 2) requests for memory that may have later requests to increase or decrease the size of the original requests. Examples of these types of requests are ALGOL 68 flexible rowed values, ALGOL 68 heap values, transput-file information and the memory area for sub-allocating a stack environment.

Run-time Simulated Memory

The simulated memory used by the interpretive executor consists of a disk file of 8000 words with two 80 word pages kept in core memory. One page is used to fetch pseudo-code instructions while the other page is used for fetching and storing data values. Since the original implementation machine was an IBM 1130 with 8k 16-bit words of memory, the above limitations were considered reasonable choices. However, due to the many additions and modifications since the original implementation, the power of the implemented language has increased significantly. The natural result of this growth is that users are attempting to write programs of greater complexity which require more user storage.

The proposed design uses a virtual memory space to allow larger memory requests while using segmentation to control paging. A much larger address space and increased flexibility can be provided without excessive overhead costs.

Summary

The major problem points are: 1) the limitations of a fixed and static size addressing scheme, 2) the lack of flexibility in the storage management features, and 3) the limited size of simulated memory. These problems have caused implementation efforts in extended mode processing and transput-file processing to be extremely complex. In addition, user programs of appreciable size simply are not accepted by the interpretive executor.

CHAPTER III

PROPOSED RUN-TIME ENVIRONMENT

Design Description

The proposed run-time environment can be broken down into a layered design consisting of four environments as follows: 1) an inner-most paging virtual memory environment, 2) a segmented memory allocation environment, 3) an

```
 ________________________________________________________
|        USER'S ENVIRONMENT                              |
|   .______________________________________________.     |
|   |        ALGOL 68 ENVIRONMENT                  |     |
|   |   .______________________________________.   |     |
|   |   |        SEGMENTED ENVIRONMENT          |   |     |
|   |   |   .______________________________.   |   |     |
|   |   |   |        VIRUTAL-PAGING         |   |   |     |
|   |   |   |        ENVIRONMENT            |   |   |     |
|   |   |   |_______________________________|   |   |     |
|   |   |                                       |   |     |
|   |   |_______________________________________|   |     |
|   |                                               |     |
|   |_______________________________________________|     |
|                                                         |
|_________________________________________________________|
```

Figure 1. The Layered Environments

ALGOL 68 "machine" environment, and 4) the outer-most layer which is the ALGOL 68 user environment. These environments are briefly diagrammed in Figure 1.

The Paging Environment

The paging environment uses a contiguous page mapping table that maps virtual pages into physical pages that may be stored on disk (or any other appropriate direct-access medium.) The Page Mapping Table (hereafter abbreviated to PMT) is stored at the beginning of the simulated virtual memory such that direct indexing can be used to map a virtual page into a physical page. Figure 2 diagrams the virtual memory to physical memory mapping. Note that the PMT (as shown in the exploded center block) is fixed such that the virtual addresses of the PMT are the same as the physical addresses of the PMT.

The paging environment is simulated using a memory block of 1024-words and a Page Fault Table (PFT). The memory block is divided into eight 128-word page slots (numbered 0 to 7) which are used as a cache memory for paging purposes (for the purposes of this thesis, the term "cache" is used to refer to the primary storage block used by this design). In contrast to the PMT which is used to indicate the mappings between virtual and physical storage, the PFT maps virtual memory into cache memory and is used in a manner similar to the usage of hardware associative registers.

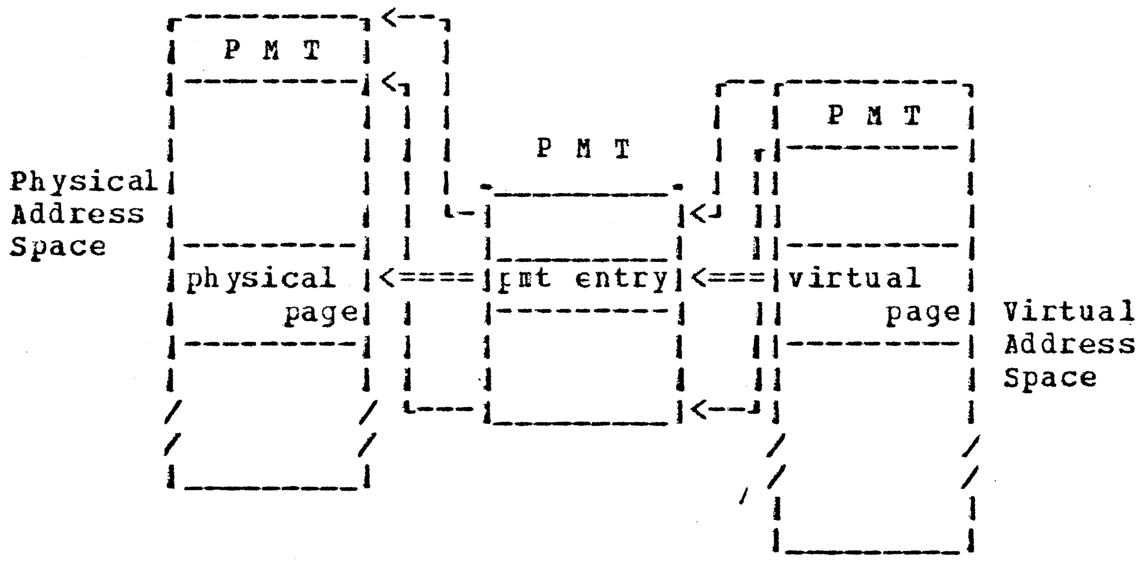

Figure 2. Mapping the Virtual Space into the Physical Space

Of the eight cache memory page slots, slot zero always contains physical page zero which always corresponds to virtual page zero; slots one through seven contain physical pages and the corresponding virtual pages as indicated by entries one through seven of the PFT.

Each entry of the PFT contains five logical fields of information: 1) a virtual page number, 2) a page slot address, 3) the Least-Recently-Used (LRU) reference count, 4) a modification flag, and 5) a physical page number. The virtual page number, page slot address, and physical page number are used to maintain the correspondance between a page slot and a virtual address. The "LRU reference count" field indicates approximately how long ago the page slot was referenced while the modification bit indicates whether or not the contents of that page slot was modified.

A description of the addressing process, shows how the various PFT entry fields are used and how a virtual address is mapped into a physical disk page. The virtual address is de-composed into a virtual page number and a page offset value so that a search of the Page Fault Table may be made to hopefully locate the page in memory. At the same time that the PFT is searched, two other operations are performed: 1) locating the least-recently-used page and 2) updating the page slot reference counts. If the desired virtual page is in memory, then the reference to the appropriate page slot or slots is performed.

If a desired page is not in the cache memory, then a virtual memory reference is made to the appropriate PMT entry. Since the PMT is also a part of pagable memory the physical page numbers of the PMT area are fixed such that they correspond to the virtual page numbers. This fixed correspondence allows a virtual page of the PMT to be paged-in without address translation. The inquiry into the PMT produces the physical page number of the page to be fetched. Using the location of the least-recently-used page found during the PFT search, a page swap is performed and the virtual memory reference is completed. Examining Figure 3, one can see that the PMT occupies the first several pages

```
             Physical                         Virtual
             Address                          Address
             Space                            Space
Page  No .  ________          Page  No .  ________
       0    |PMT page|<----------  0 --|PMT page|
            |--------|                 |--------|
       1    |PMT page|<----------  1 --|PMT page|
            |--------|                 |--------|
       2    |PMT page|<----------  2 --|PMT page|
            |--------|                 |--------|
       3    |PMT page|<----------  2 --|PMT page|
       .    |--------|             .   |--------|
       .    |   .    |             .   |   .    |
       .    |   .                  .   |   .
                .                          .
```

Figure 3. Location of the PMT

of both the virtual address space and the physical address space.

The paging environment level contains four main modules as shown in Table II[1]. These modules perform the operations involved in virtual memory addressing. Routine VMREF is the external linkage to the paging environment in that all virtual memory references are performed via a call to this routine. Routine VMPFX determines whether a reference can be satisfied or if a "page-fault" occurs. Routine VMPFT performs the actual PFT search and routine VMSWP performs page swapping.

---

[1]An informal PDL description of this module(VMPFT) and the other virtual-paging modules (VMREF, VMPFT, VMSWP) is presented in Appendix B.

TABLE II

PAGING ENVIRONMENT ROUTINES

Routine	Description
VMREF	Virtual memory fetch and store module
VMPFX	Virtual memory page fix module
VMPFT	Page Fault Table search module
VMSWP	Virtual memory page swap module

The Segmented Environment

The Segmented environment uses segments as a means of representing memory allocations that are dynamic but are not allocated in any predictable order. A segment allocation creates an entry (or entries) in the Segment Mapping Table (SMT) that reserve a group of PMT entries. The SMT entry consists of four fields: 1) a virtual address origin of the segment, 2) a segment length, 3) an allocated segment list pointer, and 4) a pointer to the next SMT entry (SMTE) node in the segment. The virtual address origin and the segment length indicate the allocated virtual memory, while the allocated segment list pointer and the SMTE node list pointer are used in memory reclamation. Note that a segment may consist of several SMTE nodes but each SMTE refers to a contiguous section of virtual memory[2] (these contiguous sections of virtual memory are refered to as "blocks" in the following discussions). Figure 4 shows how a SMTE corresponds to the PMT entries.

```
                                PMT
    SMT                      _________
 _________                  |         |
|         |                 |         |
|         |                 |         |
|---------| segment origin  |---------|
|SMT ENTRY|---------------->|PMT ENTRY|
|---------|            s  l||---------|
|         |            e  e||PMT ENTRY|
|         |            g  n||---------|
|         |            m  gV|PMT ENTRY|
/         /            e  t |---------|
/         /            n  h /         /
|         |            t    /         /
|---------| segment origin  |---------|
|SMT ENTRY|---------------->|PMT ENTRY|
|---------|            s  l||---------|
|         |            e  eV|PMT ENTRY|
|         |            g  n |---------|
|         |            m  g |         |
|_________|            e  t |         |
                       n  h |_________|
                       t
```

Figure 4. Correspondence between SMT entries and PMT entries

---

[2]A contiguous section of virtual memory means a section of virtual memory with contiguous virtual addresses; ie., the physical addresses may be non-contiguous.

When the Segment Mapping Table is initially created, a page of virtual memory is reserved and entries are created as necessary to fill the page. Two entries are set to non-zero values such that segment 0 (represented by entry 0) describes the origin and length of the SMT itself while segment 1 describes the remainder of unallocated virtual memory. The remaining SMT entries are set to zero and linked together (via the SMTE node list pointer) to form a list of unused SMTE nodes.

The allocated segment list pointer is used to link together the SMTE nodes that are the primary entry of each segment. As shown in Figure 5, the active segment list pointer of entry 0 and entry 1 are used for the purpose of indicating the beginning of the unused node list and the allocated segment list. Each segment of allocated storage is represented by a single SMTE or by a list of SMTE nodes.

The length field of an SMTE indicates the contiguous virtual memory size described by the entry node. Therefore segment 1 represents free or unallocated blocks of memory which can be used to satisfy allocation or expansion requests. An SMT entry node with a zero-value length field does not represent any mencry but may be used in creation of a new segment.

The creation of a new segment and allocation of memory requires that searches of the unused SMTE node list and the free memory segment be made looking for an empty node and

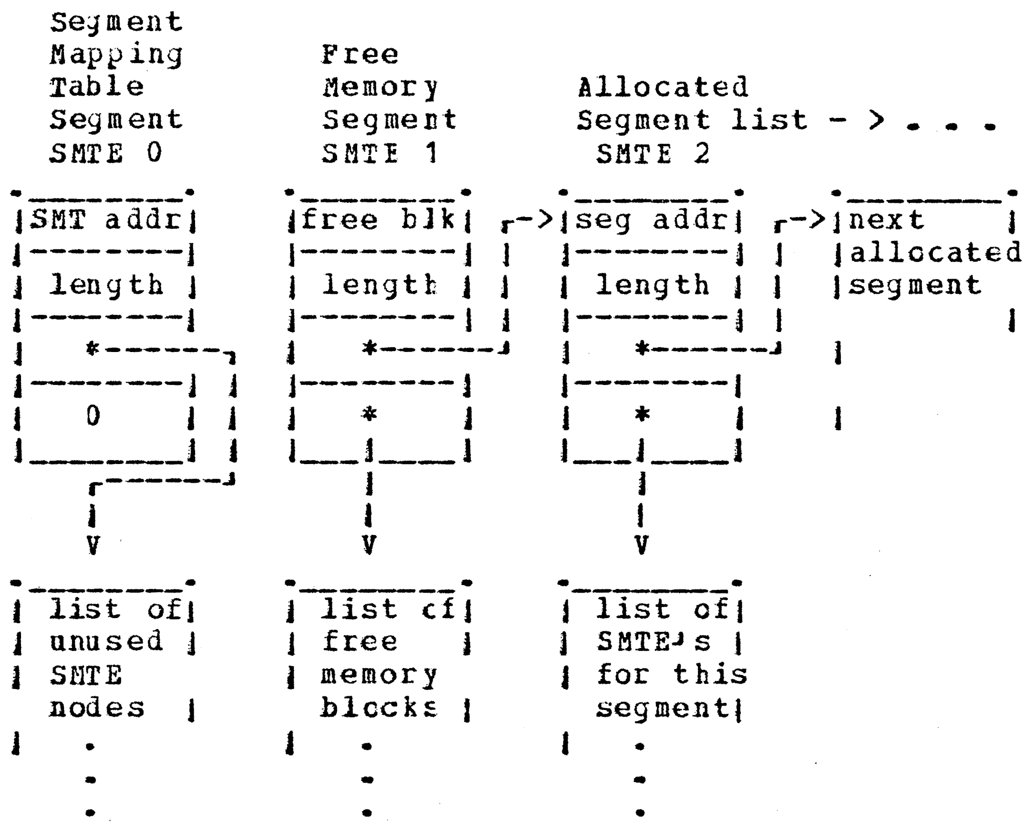
Segment Mapping Table Segment SMTE 0
Free Memory Segment SMTE 1
Allocated Segment list - > . . .
SMTE 2
SMT addr
length
0
free blk
length
seg addr
length
next allocated segment
list of unused SMTE nodes
list of free memory blocks
list of SMTE's for this segment

Figure 5. SMT structures

for a free space block large enough to satisfy the allocation request. The new SMTE node is appropriately filled-in, removed from the unused entry node list and inserted into the allocated segment list.

Another feature of segmented memory management is the ability of segments to expand or contract in size. To expand a segment, an additional SMT entry (representing additional virtual memory) is chained to the primary SMTE such that the desired total segment size is allocated. To contract a segment, the SMTE length field is reduced and a new free SMTE is created to represent the freed virtual memory.

The four functions of segmented memory management: 1) memory allocation, 2) memory de-allocation, 3) expansion of a memory allocation, and 4) contraction of a memory allocation, are provided by four of the routines shown in table III[3]. The only other externally called routine is SMREF which performs segmented-memory addressing. The remaining segmented environment routines perform internal house-keeping on the Segment Mapping Table and Page Mapping Table.

Because of the address mapping from virtual to physical pages, allocated segments may be re-arranged into single blocks by re-ordering the PMT. As a part of the memory management facilities, the routine SMSMT re-compresses segments into single blocks. In order to avoid memory fragmentation, this routine is invoked whenever an allocation request can not be satisfied using one block (in other words, the virtual memory described by a single SMTE node). Although the added memory area is forced to be a single block in processing a segment expansion request, it need not be virtually contiguous to the original segment memory area except when the segment being expanded is the SMT itself (ie. entry 0 of the table).

---

[3]See Appendix B for an informal PDL description of the segmented environment routines.

TABLE III

SEGMENTED ENVIRONMENT ROUTINES

Routine	Function Performed
SMREF	Segmented memory access
SMALC	Allocate a new segment
SMFRE	De-allocate a segment
SMADD	Expand the size of a segment
SMSUB	Contract the size of a segment
SMSMT	Re-compress segments into single blocks
SMGET	Get a block of free memory
SMPUT	Put a block back into the free list
SMTAL	Allocate a SMT entry node
SMTFR	Un-allocate a SMT entry node
SMPMT	Set PMT entries segment numbers

The procedure of addressing segmented memory is performed by the routine SMREF. This routine accepts segmented addresses which consist of a segment number and a segment offset. The segment number is used to locate a segment block list in the SMT and the segment offset is used to locate the appropriate SMTE. By combining the offset value with the SMTE origin value, a virtual address is obtained and the segmented address translation is complete.

The ALGOL 68 Environment

The ALGOL 68 Environment is the "machine-level" of a hypothetical ALGOL 68 machine. For the OSU ALGOL 68 Compiler, this is the level or environment which interpretively executes pseudo-code generated from ALGOL 68 programs. In order to maintain compatibility with earlier versions of the OSU ALGOL 68 compiler, a translation phase must be incorporated which will convert pseudo-code generated by the code-emitter phase of Version IV of the compiler (listed in Appendix D) to pseudo-code suitable for the proposed run-time executor. This translation of the pseudo-code is concerned with two aspects: 1) replacing the addressing with stack-display-offset addressing and 2) modifying the instruction codes to handle explicit stack operations.

ALGOL 68 Level Addressing and Heap Storage Management

The ALGOL 68 Environment level uses an addressing scheme partially based on a stack environment. All instruction references to memory consist of the pair: stack-frame number, stack-frame offset. The stack-frame number is used to index a vector of stack-frame addresses which is added to the offset value to yield the effective address. In order for references to other instructions to be represented by this same address format, an extra outer display is artificially created (display number 0) that maps the storage area of where the program instruction codes are kept.

All non-instruction references to memory (any address not a part of an instruction) take the form of valid segmented addresses. Figure 6 shows how segments are used to allocate memory for ALGOL 68 program stack areas and heap

```
     Segmented
      Memory
 .---------------.
 | _____________ |<----- Segment 2
 ||program area/||
 ||stack frame 0|<----- program and main
 ||-------------||      stack area
 ||storage  for ||
 ||outer-most   ||
 ||user program ||
 ||block /      ||
 ||stack frame 1||
 ||-------------||
 |/remainder of /|
 |/program stack/|
 ||frame areas  ||
 ||_____________||
 |_______________|
 |               |
 | _____________ |<----- segment 3
 ||an indirectly||
 ||referenced   |<----- some ALGOL 68
 ||heap storage ||      heap storage
 ||area         ||
 ||_____________||
 |_______________|
 | remainder of  |
 /  segmented    /
 /   memory      /
 |_______________|
```

Figure 6. Stack and Heap Storage in Segmented Memory

storage areas. The active display vector consists of segmented addresses that refer to the beginning of each stack frame (all stack frames in Figure 6 are contained in segment 2). As the stack grows, the segment containing the stack is expanded, which is accomplished through use of the segmented environment.

TABLE IV

ITEMS ALLOCATED ON THE HEAP

Description of Item	Description of Why
All variables explicitly declared to be "HEAP" variables.	To maintain the heap variables even after closing the block in which the declaration appeared.
The storage allocated to a flexible rowed value (excluding the array descriptor).	To allow for later expansion of a flexible rowed value (if subscript checking is performed, then the segmented address could be easily stored with the array descriptor).
Buffers and internal work areas for transput.	To maintain global storage for transient I/O status, information and data.

TABLE V

ITEMS ALLOCATED ON THE STACK

Description of Item	Description of Why
All local variables of primitive modes such as INT, REAL, BOOL, CHAR, etc.	The storage management for these items conforms to the requirements of a stack-model environment.
All local reference-to variables such as REF INT, REF REF INT, REF REAL, or even REF REF amode.	The storage management for these items conforms to the requirements of a stack-model environment and are used often.
All local structures which do not contain items to be put onto the heap (example - STRUCT(REAL re, REAL im) goes on the stack whereas STRUCT(STRING s) causes the storage of the flexible rowed value "s" to be allocated in a heap area.	The storage of the flexible rowed value must be allowed to "flex" while its descriptor may be allocated on the stack.
All local rowed values that are not flexible rowed values.	The storage management for these items conforms to the requirements of a stack-model environment.

As can be seen in Figure 6, heap storage is maintained in separate memory areas, thereby avoiding allocation conflicts with the stack area. This allows memory management of the stack at the ALGOL 68 level to be straight-forward but requires that references to the heap be made indirectly through a segmented address stored in the stack. There are two conclusions to be drawn from this: 1) it is easy to manipulate the storage of items allocated in heap areas but there is an overhead incurred for referencing them, and 2) the storage of an item allocated in a stack area may be manipulated only under very rigid conditions but such manipulation does not require indirect addressing as for heap items. These conclusions were carefully considered before deciding what items of an ALGOL 68 program should be allocated in the stack area and what items should go on the heap. Tables IV and V show the results of several decisions as to where an item should be allocated.

ALGOL 68 Level Local Storage Management

The ALGOL 68 level local storage management consists of a stack environment maintained within a segment. Figure 7 shows some snapshots of the run-time stack for a sample program. For each "BEGIN" in an ALGOL 68 program, a stack-frame is created. As stack-frames are created, a list of addresses are maintained and copied into the beginning of the storage allocated for each stack-frame. The first snap shot of Figure 7 diagrams the contents of the stack after the first stack-frame has been created. Snapshot 2 of Figure 7 shows the state of the stack after the second stack-frame has been created but before the storage for the rowed value "m" is allocated. Note that the first portion of the

```
BEGIN
  INT i,j,k;                  <--  Snapshot  1
    ...
    BEGIN
      REAL x;
      [1:k] INT m;            <--  Snapshot  2
        ...                   <--  Snapchot  3
    END
END
```

```
Snapshot  1     Snapshot  2     Snapshot  3
┌---------┐<┐ ┌>┌---------┐<┐ ┌>┌---------┐<┐
│    *----+-┘ │ │    *----+-┘ │ │    *----+-┘
│---------│   │ │---------│   │ │---------│
│____i____│   │ │____i____│   │ │____i____│
│____j____│   │ │____j____│   │ │____j____│
│____k____│   │ │____k____│   │ │____k____│
│    .    │   └-+----*    │<┐ └-+----*    │<┐
│    .          │---------│ │   │---------│ │
     .          │    *----+-┘   │    *----+-┘
                │---------│     │---------│
                │____x____│     │____x____│
                │ static  │     │ static  │
                │ portion │     │ portion │
                │_of_m____│     │_of_m____│
                │    .    │     │ dynamic │
                │    .          │ portion │
                     .          │_of_m____│
                                │    .    │
                                │    .
                                     .
```

Figure 7. Stack Display Layout

stack is an address that indicates the beginning of the first stack-frame and that further down in Snapshot 2 are two addresses which indicate the beginnings of the first and second stack-frames. As each new stack-frame is created, another address is added to the display maintained in the display vector and an updated copy of this list is stored onto the stack. The lccal storage management can be summarized as the creaticn and destruction of stack-frames and parallels the techniques described in Gries (14).

CHAPTER IV

SUMMARY, CONCLUSIONS, AND FUTURE WORK

Summary and Conclusions

In keeping with the goals of the Oklahoma State University ALGOL 68 Compiler project, this design adds flexibility with a limited expense of execution time. This design removes the major problem points of earlier versions of the compiler in three ways:

1. by adding flexible storage management facilities;
2. by replacing the addressing scheme of earlier versions;
3. by expanding the storage capacity of the interpretive executor.

The Oklahoma State University ALGOL 68 Compiler is not only enhanced by the above capabilities, but the design of the run-time system should prove easier to modify for varying machine configurations than earlier versions thus enhancing the portability of the compiler. This can be attributed to the layered-design approach which applies very nicely to the segmented virtual memory features described here.

The O.S.U. ALGOL 68 user can benefit greatly from the added heap storage facilities and expanded storage capacities while the layered design approach should reduce the effort required of future implementors to modify or extend the capabilities of the interpretive executor.

Future Work

There are several suggested modifications which are based on the capabilities of the implementation machine. On a machine where primary storage is plentiful, two options may be exercised: 1) the size of the page fault table and the cache memory may be increased, or 2) the virtual memory level may be replaced altogether.

The modification of the size of the page fault table and the cache memory may be performed by adjusting the the initial value of the global variable indicating the page fault table size as shown in Appendix A, and by changing the appropriate table sizes used as the page fault table and the cache memory.

To replace the virtual memory level, the routine VMREF should be modified, the page mapping table kept and all other virtual memory level items discarded. Rather than consulting the page fault table, the routine VMREF should directly access a table of contiguous locations as if it were the simulated memory. The page mapping table translation of virtual addresses must be kept so that segmented memory level compression of free memory blocks can be performed, even though the result of the translation is only an index of the simulated memory in primary storage.

Future extensions to this work include the design of run-time facilities for simulated parallel processing. In ALGOL 68, parallel processing generally takes the form of a set of ALGOL 68 procedures which are to be executed as if they were executing simultaneously. The major problem

```
                Stack
                Memory
               ┌---------┐
               |Memory   |<----   segment 2
               |allocated|
               |before   |
               |parallel |
               |processes|
               |were     |
               /invoked  /
              /         /
       _______|_________|____________________
       |Stack    |<┐  |Stack    |<┐  |Stack    |<┐
       |memory   | |  |memory   | |  |memory   | |
       |for a    | |  |for a    | |  |for a    | |
       |parallel | |  |parallel | |  |parallel | |
       |process  | |  |process  | |  |process  | |
       |_________| |  |         | |  |         | |
                   |  |         | |  |_________| |
                   |  |_________| |             |
                   |              |        segment 5
             segment 3      segment 4
```

Figure 8. Stack Environment for Parallel Processes

arises because while executing in parallel, different and distinct additions may be made to the stack environment. In fact, as shown in Figure 8, the portion of the stack allocated prior to the invocation of the parallel procedures must be shared while distinct portions of the stack must be created for each parallel routine. This problem can be solved by allocating a new segment for the continuation of the stack environment of each parallel process. The address mapping of the stack environment is normally performed by the active display vector. In the case of parallel processing, multiple display vectors are maintained such that each parallel process may access the shared portion of the stack environment and may access its own extension of the stack. Each new display vector will contain a copy of the active display vector up to the point where a parallel procedure is invoked with added stack-frame addresses pointing to its extension of the active stack.

REFERENCES

(1) Jensen, J. C. "Implementation of a Scientific Subset of ALGOL 68." (Unpub. M.S. thesis, Oklahoma State University, 1973.)

(2) Berry, R. "A Practical Implementation of Formatted Transput in ALGOL 68." (Unpub. M.S. thesis, Oklahoma State University, 1973.)

(3) Eyler, A. D. "The Implementation of a Subset of Procedures in an ALGOL 68 Compiler." (Unpub. M.S. thesis, Oklahoma State University, 1975.)

(4) Seay, W. M. "Implementation of a Subset of Modes in an ALGOL 68 Compiler." (Unpub. M.S. thesis, Oklahoma State University, 1976.)

(5) Robertson, A. L. "Transformational Grammars: Their Applications and Implementation" (Unpub. M.S. thesis, Oklahoma State University, 1978.)

(6) van Wijngaarden, A. (Ed.), B. J. Mailloux, J. E. L. Peck and C. H. Koster. "Report on the Algorithmic Language ALGOL 68." _Numerische Mathematik_, Vol. 14 (1969), pp. 70-218.

(7) van Wijngarrden, A. (Ed.), B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. "Revised Report on the Algorithmic Language ALGOL 68." Berlin-Heidelberg: Springer-Verlag, 1976.

(8) Taupin, D. "The ALGOL 68 Compiler of Paris-Sud University." _Proceedings of the 1975 International Conference on ALGOL 68_, Stillwater, Oklahoma: (10-12 June 1975), pp. 16-22.

(9) Barringer, H., and C. H. Lindsey, "The Manchester ALGOL 68 Compiler." _Proceedings of the Vth Annual III Conference on Implementation and Design of Algorithmic Languages_, Guidel, France: (16-18 May 1977), pp. 145-182.

(10) Pierce, R. H., "An ALGOL 68 Run-Time Organization." (Unpub. M.S. thesis, Victoria University of Manchester, 1971.)

(11) Currie, I. F., S. G. Bond, and J. D. Morison. "ALGOL 68-R." ALGOL 68 Implementation. J. E. L. Peck (ed). Amsterdam: North Halland Publishing Co., 1971, pp 21-34.

(12) Lindsey, C. H, and S. G. van der Meulen Informal Introduction to ALGOL 68, Revised Edition, Amsterdam: North-Holland Publishing Co., 1977.

(13) Andre, J., and J. Banatre (Ed.) Proceedings of the Vth Annual III Conference on Implementation and Design of Algorithmic Languages, Guidel, France: (16-18 May 1977).

(14) Gries, D. Compiler Construction for Digital Computers, New York: John Wiley & Sons, Inc., 1971, pp. 171-211, 328-335.

(15) Madnick, S. E., and J. J. Donovan Operating Systems New York: McGraw-Hill Book Co., 1974, pp. 105-208, 534-548.

(16) Robertson, A., and G. E. Hedrick, "A Portable Compiler For An ALGOL 68 Subset." Proceedings of the 1975 International Conference on ALGOL 68, Stillwater, Oklahoma: (10-12 June 1975), pp. 59-63.

(17) Van Doren, J. R. "Notes on Software Design Methods (Flowcharts and PDL's)." Presented as course material for Computer Structure & Programming, a graduate level course at Oklahoma State University.

APPENDIX A

DESCRIPTIONS OF RUN-TIME DATA STRUCTURES

The following descriptions are data structures used throughout the design. Additionally, some descriptions of the variables used in PDL descriptions of the presented design are included. For purposes of the design presentation, global "constants" have been chosen that meet all design requirements (these constants may or may not be optimal for performance considerations).

GLOBAL "Constants"

page size	(PSIZE)	- page size value (128 words)
PMT size	(NPMTE)	- no. of PMT entries
PFT size	(NPFTE)	- no. of PFT entries
PMTE size	(PELEN)	- size of PMTE entry (2 words)
SMTE size	(SELEN)	- size of SMTE entry (4 words)
	(IORD)	- read op-code
	(IOWR)	- write op-code
	(PFILE)	- paging disk file (disk record length = memory page size)

PAGING "Hardware"

memory(1024)	(MEMRY)	- cache paging memory
pft(7,4)	(PFT)	- Page Fault Table
active pgno	(APNUM)	- active page number
active pftn	(APPOS)	- active pft entry

PFT entries (page fault table)
1) virtual page no.
2) cache-memory page slot address
3) LRU reference count
4) Modified bit
   (sign position: >0 -- on, <0 -- off),
   and physical page no.

PMT entries (page mapping table)
1) physical page no.
2) no. of the segment possessing this page

SEGMENTED ADDRESS
1) segment mapping table entry number
2) segment offset address

SMT entries (segment mapping table)
1) Segment origin
2) Segment length
3) Allocated segment list pointer
4) Segment block list pointer

SEGMENT--A68 level "Hardware"

ERROR	termination code
SMTAR	SMT address register
ASNUM	active segment number
ASORG	active segment origin address
ASLEN	active segment length
ASLNK	active allocated segment list pointer
ASPTR	active segment block list pointer
DSPLN	display vector length

DSPVT - display vector of active stack frames
(maximum of 20 active stack frames)

APPENDIX B

PDL DESCRIPTIONS OF RUN-TIME ROUTINES

The following figures are PDL descriptions of the Run-time routines. These descriptions are intended as a rough guide for implementation and therefore omit detailed or error-checking code in the interest of clarity.

Paging Environment Routines

The four routines VMREF, VMPFX, VMPFT and VMSWP form a core of modules that deal directly with the virtual-memory/real-memory interface. With the exception of a few restricted segmentation level routines, all accesses to the paging level environment are performed indirectly through the routine VMREF. The few exceptions to this mechanism are the segmentation level routines that modify the Page Mapping Table for the purposes of garbage collection. This limited access allows the entire paging environment to be removed with the exception of the Page Mapping Table.

Reference to virtual memory routine

```
vmref:
PROC (virtual address, buffer, start, stop,
     I/O flag);
  v := virtual address;
  i := start - 1;
  DO UNTIL i > stop;
    CALL vmpfx (v, c, pfte );
      ¢ c is the returned cache memory address or
        zero if the desired page is not in cache
        memory ¢
      ¢ pfte is the page slot number of
        least-recently used page ¢
    IF c = 0 THEN
      vpage := v / page size ¢ PSIZE ¢;
      voffset := v - (vpage * page size);
      CALL vmpfx ((2 * vpage), c, pn );
        ¢ if c is returned as zero, then there is
          no PMT entry for the desired page, ie.
          the reference is outside the virtual
          address space ¢
      IF c = 0 THEN
        signal address error and quit;
      FI;
      ppage := memory(c);
      CALL vmswp (vpage, ppage, pfte );
      c := pft(pfte, 2) + voffset;
    FI;
    i := i + 1;
    v := v + 1;
    IF write operation THEN
      IF pfte > 0 THEN
        ¢ if pfte = 0 then virtual page := physical
          page := page slot 0 which is always
          paged-in ¢
        set modified-flag of pft(pfte);
      FI;
      memory(c) := buffer(i);
    ELSE
      buffer(i) := memory(c);
    FI;
  END;
  RETURN;
END vmref;
```

Page Fixing routine

```
  vmpfx:
  PROC (vaddr, c, p );
    page :=  vaddr / page size;
    offset := vaddr - (page * page size);
    IF page = 0 THEN
      ¢ virtual page needed is page 0 ¢
      c := offset;
      p := 0;
      RETURN;
    FI;
    IF page = active pgno ¢ APNUM ¢ THEN
      ¢ page needed was the last page accessed ¢
      c := cache address of active pft entry;
      p := active pftn ¢ APPOS ¢;
      RETURN;
    FI;
    CALL vmpft (page, pfte, p );
      ¢ pfte is the returned pft entry position of
        page in cache-memory or zero if desired
        page is not in memory ¢
    IF pfte > 0 THEN
      c := pft(pfte,2) + offset;      .
     RETURN;
    FI;
    IF page ≤ (2 * PMT size / page size) THEN
      ¢ if page requested is a PMT page, then the
        physical page number is known without
        consulting the PMT ¢
      CALL vmswp (page, page, p);
      c := pft(p,2) + offset;
      RETURN;
    ELSE
      c := 0;
      RETURN;
    FI;
  END vmpfx;
```

Page Fault Table search routine

```
vmpft:
PROC (page, pfte, plru );
  pfte := 0;
  plru := 1;
  max ref cnt := pft(plru,3);
  DO i := 1 TO 7 BY 1;
    IF pft(i,3) < 127 THEN
      ¢ increment reference count up to a
        limit of 127 ¢
      pft(i,3) := pft(i,3) + 1
    FI;
    IF pft(i,1) = page THEN
      ¢ page has been found; return pft position ¢
      active pgno ¢ APNUM ¢ := page;
      active pftn ¢ APPOS ¢ := i;
      pfte := i;
      pft(i,3) := 0;
    FI;
    IF pft(i,3) > max ref cnt THEN
      ¢ return position of candidate for page-out ¢
      plru := i;
      max ref cnt := pft(i,3);
    FI;
  END;
  RETURN;
END vmpft;
```

Page Swap routine

```
vmswp:
PROC (vpage, ppage, p );
  IF pft(p,4) > 0 THEN
    perform page-out operation
  FI;
  perform page-in operation;
    ¢ set virtual page number, reference count,
      and physical page number ¢
  pft(p,1) := vpage;
  pft(p,3) := 0;
  pft(p,4) := -ppage ¢ set modified bit off ¢;
  active pgno ¢ APNUM ¢ := vpage;
  active pftn ¢ APPOS ¢ := p;
  RETURN;
END vmswp;
```

Segmented Environment Routines

The segmentation level routines provide all the memory management functions and map all segment-type addresses into virtual addresses. In keeping with the goal of modular design, the segmented environment presents the appearance of being a collection of memory management primitives to all external environment levels. Thus the routines SMALC and SMFRE are used for memory allocation and un-allocation respectively, and the routine SMREF is used for all segmented-level memory accesses. For expansion or contraction of an allocated memory area, the respective routines SMADD and SMSUB would be called.

Reference to segmented memory routine

```
smref:
PROC (segment number, segment offset, buffer,
     start, stop, I/O flag );
  snum := segment number;
  sofst := segment offset;
  len := stop - start + 1;
  i := start;
  j := i - 1;
  IF snum ≠ active segment no. ¢ ASNUM ¢ THEN
    sptr := SMT address ¢ SMTAR ¢ +
      (SMTE node size ¢ SELEN ¢ * snum);
    CALL vmref (sptr, smte, 1, SMTE node size,
               IORD);
    active segment no. ¢ ASNUM ¢ := snum;
    active segment origin addr. ¢ ASORG ¢ :=  smte(1);
    active segment length ¢ ASLEN ¢ := smte(2);
    active alloc. seg. list ptr. ¢ ASLNK ¢ := smte(3);
    active seg. blk. list ptr. ¢ ASPTR ¢ := smte(4);
  FI;
  sorg := active segment origin addr. ¢ ASORG ¢;
  slen := active segment length ¢ ASLEN ¢;
  sptr := active seg. blk. list ptr. ¢ ASPTR ¢;
  DO WHILE len > 0;
    DO WHILE sptr ≠ 0 & sofst ≥ slen;
```

```
          ¢ follow segment chain pointer until entry
            is found that contains desired offset
            address ¢
          sofst := sofst - slen;
          CALL vmref (sptr, smte, 1, SMTE node size,
                      IORD);
          sorg := smte(1);
          slen := smte(2);
          sptr := smte(4);
        END;
        l := slen - sofst + 1;
          ¢ compute remaining length of seg. block ¢
        IF l > len THEN
          ¢ length of desired request is totally
            contained in current SMT entry ¢
          l := len;
        FI;
        addr := sorg + sofst;
        sofst :=  sofst + l;
        len := len - l;
        j := j + l;
        CALL vmref (addr, buffer, i, j, I/O flag);
        i := i + l;
      END;
      RETURN;
    END smref;
```

Segment allocation routine

```
smalc:
PROC (segment length, segment number, return code );
  CALL smtal (SMTE address) ¢ allocate a new SMTE
    node ¢;
  sglen := segment length ¢ rounded up to the
    nearest integer multiple of page size ¢;
  CALL smget (sglen, SMTE, error code) ¢ search
    free segment for needed space, fill-in fields
    of SMTE node to reflect allocated storage
    area and set error code (on of three possible
    conditions:  a) a free block of sufficient
    size was found,  b) no free block was adequate
    but compression of segments could produce the
    necessary free block, and  c) insufficient
    total memory to perform allocation). ¢;
  IF error code is above condition "c" THEN
    set return code to indicate allocation failure;
    RETURN;
  ELSE
    set return code to no error condition;
  FI;
  IF error code is above condition "b" THEN
    CALL smsmt ¢ compress free memory segments ¢;
    CALL smget (sglen, SMTE, error code);
      ¢ search free segment list again for needed
        segment of free memory ¢;
    IF error code is not condition "a" THEN
      set return code to indicate allocation
        failure;
      RETURN;
    FI;
  FI;
  sptr := SMT address ¢ SMTAR ¢ + SMTE node size
    ¢ SELEN ¢ + 2 ¢ compute address of primary
      allocated segment list pointer ¢;
  CALL vmref (sptr, SMTE(3), 1, 1, IORD);
  CALL vmref (sptr, SMTE address, 1, 1, IOWR)
    ¢ insert new segment into allocated segment
      list ¢;
  segment number := (SMTE address - SMT address) /
    SMTE node size;
  CALL vmref(SMTE address, SMTE, 1, SMTE node size,
    IOWR) ¢ update smt entry ¢;
  CALL smpmt (SMTE(1) ¢ segment origin ¢, SMTE(2)
    ¢ segment length ¢, segment number) ¢ set the
    segment number fields of the PMT entries that
    are in the new segment ¢;
  RETURN;
END smalc;
```

Segment de-allocation routine

```
smfre:
PROC (segment number );
  SMTE address := (segment number * SMTE node
    size) + SMT address ¢ SMTAR ¢;
  sptr := SMTE address;
  DO WHILE sptr ≠ 0;
    CALL vmref (sptr, SMTE, 1, SMTE node size,
      IORD) ¢ fetch each SMTE for segment ¢;
    sptr := SMTE(4) ¢ save ptr to next SMTE ¢;
    CALL smpmt(SMTE(1), SMTE(2), 1) ¢ set the
      segment number fields of the PMT entries
      that are in the current segment ¢;
    CALL smput (SMTE address) ¢ return memory
      block to free list ¢;
    SMTE address := sptr;
  END;
  RETURN;
END smfre;
```

Memory block allocation routine

```
smget:
PROC (segment length, new SMTE, error code );
  total free size := 0;
  sptr := SMT address ¢ SMTAR ¢ +
    SMTE node size ¢ SELEN ¢;
  last := sptr;
  DO UNTIL sptr = 0;
    addr := sptr;
    CALL vmref (sptr, free SMTE, 1, SMTE node
      size, IORD) ¢ fetch each SMTE of free
      memory segment (segment 1) ¢;
    IF free SMTE(2) < segment length THEN
      total free size :=  total free size +
        free SMTE(2) ¢ total the amount of
        free memory space ¢;
      last := sptr;
      sptr := free SMTE(4) ¢ get pointer to
        next SMTE ¢;
    ELSE
      sptr := 0;
    FI;
  END;
  IF free SMTE(2) < segment length THEN
    IF total free size < segment length THEN
      error code := ¢ insufficient total space ¢;
    ELSE
      error code := ¢ insufficient contiguous
        space ¢;
    FI;
  ELSE
    IF free SMTE(2) > segment length THEN
      new SMTE(1) := free SMTE(1) ¢ copy
        segment origin ¢;
      new SMTE(2) := segment length;
      free SMTE(1) := free SMTE(1) +
        segment length ¢ update origin of
        free memory block ¢;
      free SMTE(2) := free SMTE(2) -
        segment length ¢ update length of
        free memory block ¢;
      new SMTE(3), new SMTE(4) := 0;
      CALL vmref (last, free SMTE, 1, SMTE
        node size, IOWR);
    ELSE
      new SMTE(1) := free SMTE(1) ¢ copy
        segment origin ¢;
      new SMTE(2) := free SMTE(2) ¢ copy
        segment length ¢;
      new SMTE(3), new SMTE(4) := 0;
      IF last = addr THEN
```

```
          free SMTE(1), free SMTE(2) := 0
            ¢ if the SMTE found is entry 1 in
              the SMT, then reset its origin
              and length fields to zero ¢;
          CALL vmref (addr, free SMTE, 1, SMTE
            node size, IOWR);
        ELSE
          last := last + 3 ¢ update pointer to
            indicate block list ptr field of
            previous SMTE in free memory block
            list ¢;
          CALL vmref (last, free SMTE(4), 1, 1,
            IOWR) ¢ delete current SMTE from
            free memory block list ¢;
          CALL SMTFR (addr) ¢ un-allocate
            unused SMTE ¢;
        FI;
      FI;
    FI;
    RETURN;
  END smget;
```

Memory block de-allocation routine

```
smput:
PROC (SMTE address );
  addr := SMTE address;
  CALL vmref (addr, old SMTE, 1, SMTE node size,
    IORD) ¢ fetch old SMTE ¢;
  last := SMT address ¢ SMTAR ¢ +
    SMTE node size;
  DO UNTIL last = 0;
    CALL vmref (last + 2, sptr, 1, 1, IORD)
      ¢ fetch each SMTE of allocated segment list ¢;
    IF sptr = addr THEN
      sptr := old SMTE(3);
      CALL vmref (last + 2, sptr, 1, 1, IOWR)
        ¢ update allocated segment list ¢;
      last, old SMTE(3) := 0;
    ELSE
      last := sptr;
    FI;
  END;
  sptr := SMT address + SMTE node size + 3;
  CALL vmref (sptr, old SMTE(4), 1, 1, IORD)
    ¢ fetch pointer to free memory block
      list ¢;
  CALL vmref (sptr, addr, 1, 1, IOWR) ¢ insert
    old SMTE into list ¢;
  CALL vmref (addr, old SMTE, 1, SMTE node size,
    IOWR) ¢ update free memory block list ¢;
  RETURN;
END smput;
```

PMTE segment number update routine

```
smpmt:
PROC (origin, length, segment number );
  addr := PMTE node size ¢ PELEN ¢ * (origin /
    page size ¢ PSIZE ¢) - 1;
  npgs := (length + page size - 1) / page size;
  DO i := 1 TO npgs BY 1;
    addr := addr + PMTE node size;
    CALL vmref (addr, segment number, 1, 1, IOWR)
      ¢ set the segment number field of PMT
        entries mapped by the input segment
        origin/length ¢;
  END;
  RETURN;
END smpmt;
```

APPENDIX C

PROGRAM DESIGN LANGUAGE

The Program Design Language descriptions of the Run-time routines use an informal PDL similar to that used by Oklahoma State University Computing and Information Sciences Department. Specifically the introductory notes shown here are based on notes by Dr. J. R. Van Doren describing an informal PDL used as Computer Science course material (17).

Modules or Procedures format

```
Module name:
PROC optional parameter list;
  .
  .
  Sequence of PDL and/or English language statements
  .
  .
  RETURN
END module name;
```

Module Invocation

```
CALL module name(optional parameter list);
```

Elementary Decision logic

```
IF condition THEN
  Sequence of PDL and/or English language statements
ELSE
  Sequence of PDL and/or English language statements
FI;

            or

IF condition THEN
  Sequence of PDL and/or English language statements
FI;
```

Looping Constructs

```
DO WHILE condition;
  Sequence of PDL and/or English language statements;
END;

DO UNTIL condition;
  Sequence of PDL and/or English language statements;
END;

DO index = initial value TO final value BY increment;
  Sequence of PDL and/or English language statements;
END;
```

Comments or Remarks

```
¢ Comment or Remark statement ¢
```

APPENDIX D

OPERATION CODES OF THE VERSION IV OSU ALGOL 68 COMPILER

The Version IV OSU ALGOL 68 Compiler interpretively executes 4-tuple pseudo-code. The meanings of the various 4-tuples are listed below.

BASIC OPERATION CODES

010	00, 00, 00	BLOCK ENTRY
020	R2, R3	BLOCK EXIT
		R2 IS THE ELEMENTAL MODE OF THE RETURNED VALUE
		R3 IS THE NUMBER OF ROWS FOR R2
030	01, 00, R4	UNCONDITIONAL JUMP/BRANCH
		R4 IS THE BRANCH ADDRESS
030	02, R3, R4	CONDITIONAL JUMP/BRANCH
		R3 IS THE ID OF THE CONDITIONAL VALUE
		R4 IS THE BRANCH ADDRESS
030	03, R3, R4	LOAD ADDRESS
		R3 IS THE DISPLACEMENT TO BE ADDED TO THE RESOLVED ADDRESS
		R4 IS THE ID OF THE ADDRESS TO BE PUT ONTO THE STACK TOP
030	04, R3, R4	BRANCH WITH INDEX
		R3 IS THE ID OF THE INDEX VALUE
		R4 IS THE ADDRESS OF THE BRANCH TABLE
C30	05, R3, R4	SET FLAG ON DATA SWITCH
		R3 IS THE FLAG NUMBER
		R4 IS THE DATA SWITCH NUMBER
040	R2, 00, R4	ALLOCATE SYMBOL
		R2 IS THE MODE OF THE SYMBOL
		R4 IS THE IDENTIFIER NUMBER
050	R2, R3, R4	SET STATEMENT NUMBER
		R2 IS THE STATEMENT NUMBER
		R3 IS THE ELEMENTAL MODE OF THE STACK TOP VALUE TO BE VOIDED

		R4 IS THE NUMBER OF ROWS
061	R2, R3, R4	UPDATE SYMBOL TABLE R2 IS THE MODE OF SYMBOL TABLE ENTRY R3 IS THE ADDRESS R4 IS THE IDENTIFIER NUMBER
062	R2, R3, R4	PRINT UNFORMATTED R2 IS THE ELEMENTAL MODE OF THE VALUE TO BE PRINTED R3 IS THE NUMBER OF ROWS R4 IS THE ID OF THE VALUE TO BE PRINTED
070	R2, R3, R4	BECOMES R2 IS THE MODE OF THE VALUE TO BE ASSIGNED R3 IS THE SOURCE ID R4 IS THE DESTINATION ID
080	R2, R3, R4	READ UNFORMATTED R2 IS THE ELEMENTAL MODE OF THE VALUE TO BE READ R3 IS THE NUMBER OF ROWS R4 IS THE ID OF THE VALUE TO BE READ
090	R2, R3, R4	DEFINE LABEL R2 IS THE ADDRESS OF THE LABEL R3 IS THE NEGATIVE OF THE BLOCK NUMBER R4 IS THE ID OF THE LABEL
50N	R2, R3, R4	ALLOCATE DESCRIPTOR FOR ARRAYS N IS THE ELEMENTAL MODE R2 IS THE ID OF THE ARRAY R3 IS THE NUMBER OF ROWS IN THE ARRAY R4 IS THE ADDRESS OF THE SKELETON DESCRIPTOR(S)
510	R2, R3, R4	LOAD SUBSCRIPTED R2 IS THE NUMBER OF ROWS IN THE ARRAY R3 IS THE ID OF THE ARRAY R4 IS THE SYMBOL TABLE POINTER OF THE TEMPORARY SYMBOL TABLE ENTRY GENERATED CONTAINING THE CALCULATED ADDRESS
52N	R2, R3, R4	MOVE ROW OF OPERANDS N IS THE ELEMENTAL MODE R2 IS THE NUMBER OF ROWS R3 IS THE ID OF THE SOURCE OPERAND R4 IS THE ID OF THE DESTINATION OPERAND
530	R2, R3, R4	ALLOCATE SLICING DESCRIPTOR R2 IS THE NUMBER OF ROWS R3 IS THE IS OF THE ARRAY TO BE SLICED R4 IS THE ADDRESS OF THE SLICING TEMPLATE
541	R2, R3, R4	LOWER BOUND R2 IS THE NUMBER OF ROWS IN THE ARRAY

```
                        R3 IS THE ID OF THE ARRAY OPERAND
                        R4 IS THE ID OF THE ROW NUMBER
542  R2, R3, R4         UPPER BOUND
                        R2 IS THE NUMBER OF ROWS IN THE ARRAY
                        R3 IS THE ID OF THE ARRAY OPERAND
                        R4 IS THE ID OF THE ROW NUMBER
61N  R2, R3  00         INTERNALLY GENERATED COERCION
                        N IS THE MODE OF THE STACK TOP
                             ELEMENT TO BE SAVED DURING THE
                             CURRENT COERCION
                        R2 IS THE MODE TO BE WIDENED FROM
                        R3 IS THE MODE TO BE WIDENED TO
                             COERCED RESULT IS PUT ON THE
                             STACK TOP
71N  R2, R3, R4         FORMATTED INPUT
                        N IS THE ELEMENTAL MODE
                        R2 IS THE ID OF THE INPUT FILE
                        R3 IS THE NUMBER OF ROWS IN THE INPUT
                             ITEM
                        R4 IS THE ID OF THE INPUT ITEM
72N  R2, R3, R4         FORMATTED OUTPUT
                        N IS THE ELEMENTAL MODE
                        R2 IS THE ID OF THE OUTPUT FILE
                        R3 IS THE NUMBER OF ROWS IN THE
                             OUTPUT ITEM
                        R4 IS THE ID OF THE OUTPUT ITEM
730  R2, R3, R4         OPEN FILE
                        R2 IS THE ID OF THE FILE
                        R3 IS THE CHANNEL NUMBER FOR CURRENT
                             FILE OPEN OPERATION
                        R4 IS THE ID OF THE IDENTIFICATION
                             STRING (NOT YET IMPLEMENTED)
800  R2, R3, 00         RETRIEVE PARAMETER
                        R2 IS THE IDENTIFIER NUMBER
                        R3 IS THE MODE INCLUDING REF CODE
                        IF R2=0 THEN RETRIEVE THE PARAMETER
                             FLAG
801  R2, R3, R4         COMPLETE PROC DESCRIPTOR
                        R2 IS THE IDENTIFIER NUMBER FOR THE
                             PROCEDURE
                        R3=1 FOR COMPLETING THE STATIC
                             INFORMATION FIELDS, 2 FOR
                             COMPLETING THE ENTRY POINT
                             FIELD
                        R4 IS THE ENTRY POINT IF APPROPRIATE
810  00, 00, 00         PROC ENTRY
815  R2, R3, 00         LOAD PARAMETER
                        R2 IS THE IDENTIFIER NUMBER OR 0 FOR A
                             TEMPORARY
                        R3 IS THE MODE
820  R2, R3, 00         PROC EXIT
                        R2 IS THE MODE OF THE RETURNED VALUE
                        R3 IS THE NUMBER OF ROWS
830  00, 00, 00         LOAD RETURN INFORMATION
```

		CAUSES VALUES NEEDED TO RETURN FROM A PROCEDURE TO BE LOADED ONTO THE RUNTIME STACK
835	R2, 00, 00	LOAD PROC DESCRIPTOR & INVOKE PROC R2 IS THE IDENTIFIER NUMBER OF THE PROCEDURE
840	00, 00, 00	SAVE SYMBOL TABLE

DYADIC OPERATION CODES OF THE FORM:
OPCD,OPRND1,OPRND2,OPRND3

ALL OPERATIONS ARE PERFORMED OPRND1 OP OPRND2 => OPRND3

THE VALUES OF THE OPERANDS HAVE THE FOLLOWING MEANINGS:

OPRND < 0	RUN TIME SYMBOL TABLE REFERENCE
OPRND = 0	RUN TIME STACK TOP REFERENCE
OPRND > 0	RUN TIME VIRTUAL MEMORY ADDRESS

OP-CODE(N IS THE MODE INDICATOR)

10N	+	ADD	VALID FOR N = 1, 2, 3
11N	-	SUBTRACT	VALID FOR N = 1, 2, 3
12N	/	DIVIDE	VALID FOR N = 1, 2, 3
13N	*	MULTIPLY	VALID FOR N = 1, 2, 3
14N	**	RAISE (UP)	VALID FOR N = 1, 2
15N	//:	MODULO	VALID FOR N = 1
16N	+:=	PLUSAB	VALID FOR N = 1, 2, 3
17N	+=:	PRUS	VALID FOR N = 1, 2, 3
18N	-:=	MINUSAB	VALID FOR N = 1, 2, 3
19N	/:=	DIVIDEAB	VALID FOR N = 1, 2, 3
20N	*:=	TIMESAB	VALID FOR N = 1, 2, 3
21N	//::=	MODULOAB	VALID FOR N = 1
22N	¬=	NOT EQUAL	VALID FOR N = 1, 2, 3, 4, 5
23N	<	LESS THAN	VALID FOR N = 1, 2, 5
24N	<=	LESS THAN/EQ.	VALID FOR N = 1, 2, 5
25N	>=	GRTR. THAN/EQ.	VALID FOR N = 1, 2, 5
26N	>	GREATER THAN	VALID FOR N = 1, 2, 5
27N	=	EQUAL	VALID FOR N = 1, 2, 3, 4, 5
284	& (AND)	LOGICAL AND	
294	OR	LOGICAL OR	
402	? (OR I)	PLUS I TIMES	REAL -> COMPLEX

MONADIC OPERATION CODES OF THE FORM:
OPCD1,OPCD2,OPRND2,OPRND3

ALL OPERATIONS ARE PERFORMED OP OPRND1 => OPRND2

THE VALUES OF THE OPERANDS HAVE THE SAME MEANINGS AS FOR THOSE OPERAND VALUES USED IN DYADIC OPERATIONS

IR1	IR2(N IS THE MODE INDICATOR)			
30N	01	+	MONADIC PLUS (ALSO A LOAD)	VALID FOR N = 1, 2, 3
30N	02	-	MONADIC MINUS	VALID FOR N = 1, 2, 3
30N	03	ABS	ABSOLUTE VALUE	VALID FOR N = 1, 2, 3, 4, 5
30N	04	SQRT	SQUARE ROOT	VALID FOR N = 1, 2, 3
30N	05	EXP	E ** X	VALID FOR N = 1, 2
30N	06	LN	NATURAL LOG.	VALID FOR N = 1, 2
30N	07	LOG2	LOG BASE 2	VALID FOR N = 1, 2
30N	08	LOG10	LOG BASE 10	VALID FOR N = 1, 2
30N	09	SIN	SINE	VALID FOR N = 1, 2
30N	10	COS	COSINE	VALID FOR N = 1, 2
30N	11	TAN	TANGENT	VALID FOR N = 1, 2
30N	12	ARCSIN	ARCSINE	VALID FOR N = 1, 2
30N	13	ARCCOS	ARCCOSINE	VALID FOR N = 1, 2
30N	14	ARCTAN	ARCTANGENT	VALID FOR N = 1, 2
303	15	CONJ	COMPLEX CONJUGATE	
303	16	CMPLXSQR	COMPLEX SQUARE ROOT	
313	02	ARG	COMPLEX ARCTAN	COMPLEX -> REAL
313	03	RE	REAL PART	COMPLEX -> REAL
313	04	IM	IMAGINARY PART	COMPLEX -> REAL
322	01	ENTIER	FLOOR FUNCTION	REAL -> INTEGRAL
322	02	LWB	FLOOR FUNCTION	REAL -> INTEGRAL
322	03	ROUND	ROUND FUNCTION	REAL -> INTEGRAL
322	04	SIGN	SIGN TRANSFER	REAL -> INTEGRAL
322	05	UPB	CEIL FUNCTION	REAL -> INTEGRAL
331	01	ODD	ODD FUNCTION	INTEGRAL -> BOOLEAN
334	02	¬ (NOT)	LOGICAL NOT	BOOLEAN -> BOOLEAN
342	01	RANDOM	RANDOM GEN.	-> REAL
351	01	REPR	CHAR. GEN.	INTEGRAL -> CHARACTER

VITA 2

MARK GOTO

Candidate for the Degree of

MASTER OF SCIENCE

Thesis: Segmented-Virtual Memory Design for an ALGOL 68 Compiler

Major Field: Computing and Information Sciences

Biographical:

Personal data: Born in Oklahoma City, Oklahoma, on July 16, 1953.

Education: Graduated from Putnam City High School, Oklahoma City, Oklahoma, in May, 1971; received Bachelor of University Studies from Oklahoma State University, Stillwater, Oklahoma, in July, 1975; completed requirements for Master of Science degree at Oklahoma State University, Stillwater, Oklahoma, in July, 1978.

Professional Experience: Computer Systems Programmer for the Oklahoma State University Computer Center, August, 1977-July, 1978; graduate research assistant at Oklahoma State University under Dr. G. E. Hedrick, Computing and Information Sciences Department, Summer 1976-Spring 1977; graduate teaching assistant, Oklahoma State University, Computing and Information Sciences Department, Fall 1975-Spring 1976.