THE DESIGN AND IMPLEMENTATION OF MEMORY MANAGEMENT AND INITIALIZATION MODULES FOR A LISP INTERPRETER

by

LEE ROY WHITLEY

B.S., Texas Technological University, 1961

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:

LD
2668
R4
1977
W52
c.2
Document

TABLE OF CONTENTS

Page

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

APPENDIXES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This paper describes a portion of an overall project designed to implement a LISP interpreter on a minicomputer. The programming for the interpreter was done in the FORTRAN IV programming language. The purpose of this portion of the project was:

(1) To design the initialization and memory management routines for a LISP interpreter.

(2) To program these modules in a high-level language.

(3) To implement the modules on the INTERDATA 8/32.

1.1 LISP Programming Language

LISP is a list processing language . It is an efficient language for use with symbolic data manipulation. Symbolic data manipulation requires extensive use of recursive techniques, and LISP is designed to make recursion easy to use. The use of recursion involves temporary storage of many different environments. To allow for the allocation and deallocation of these elements in storage requires the use of a dynamic memory management system.

The dynamic memory management system will be discussed in detail in chapters 2 and 3. The remainder of this section contains a brief discussion of the overall project and the interaction with the INTERDATA 8/32.

1.2 LISP Interpreter System

The interpreter (Figure 1-1.) consists of a driver module and five major subsystem modules. The five major subsystem modules have different functions and were designed so that they could be developed and tested separately. Each major subsystem module is capable of functioning under the direction of a single driver module. The modular concept is not only efficient, but it is also convenient for splitting a large project into smaller projects which can be developed by different individuals.

LISP INTERPRETER SYSTEM DESIGN

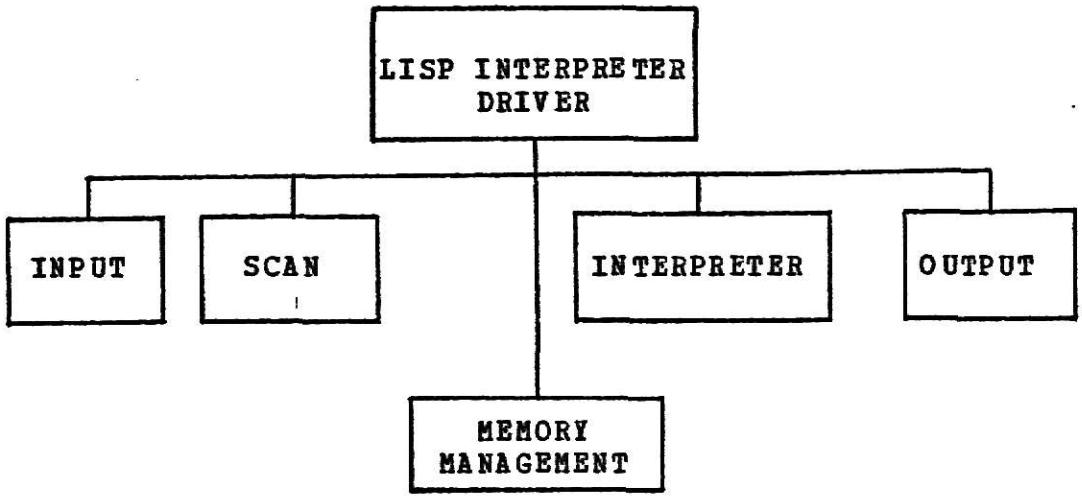


Figure 1-1

1.2.1 Driver

The driver module contains four basic parts: (1) the information describing the use of the module; (2) the common data statements declaring the global data; (3) a dictionary of variables and constants used; and (4) the call statements which invoke the various subsystem modules. The code for

the driver is at Appendix B.

1.2.2 Input

The input module is responsible for: (1) requesting memory space from the memory management module and; (2) reading the user's program into addressable storage locations. The INPUT, SCAN, and a portion of the INTERPRETER modules were prepared by David C. Bosserman and are discussed in depth in his report.

1.2.3 Scan

The SCAN module is a driver routine. It calls a token generator routine and symbol table search routine. It causes the user's program to be scanned and placed into an address array. The address array shows: (1) the location of each symbol/string in the user's program; (2) the length of each symbol/string; (3) its key word/function status; and (4) its location within a key word/function table described in block data.

1.2.4 Interpreter

The INTERPRETER module performs two major functions. It initially creates a general tree representation of the user's program. Once the general tree has been created it is passed to the EXECUTOR portion of the module. The EXECUTOR portion is the subject of another future report.

1.2.5 Memory Management

The memory management module is a driver routine which operates a dynamic memory management system. There are two main submodules which allow for variable sized blocks to be allocated. One system was designed for use by the scanner and parser and the other for the executor. Memory management will be discussed in detail in chapter 2.

1.3 Modular Design

This project was designed from the beginning in a top-down manner. All necessary modules were visualized and written into the initial programs as stubs of routines. The routine stubs were replaced with completed modules only after the modules had been tested. All global and external parameters required by the modules were provided by a test driver.

All modules have been designed with a definite hierarchy. No module calls a module of a higher level, and all parameters are passed to the called routine thru common statements or calling statement parameters. All modules have been written as subroutines or functions containing only one entry and one exit point. The number of lines within each module have been limited to a specific function for ease of reading and debugging, and all primary actions have been documented within the program.

1.4 INTERDATA 8/32

The INTERDATA 8/32 was used during the designing, programming, and testing of this portion of the LISP interpreter. The computer is owned by the Computer Science Department and is located in Fairchild Computer Room. All of the necessary input/output devices are available for the user, and 640K bytes of memory are more than adequate for the normal program. The problems encountered using the INTERDATA are discussed in Chapter 4.

1.5 Summary

This report covers one part of a three part project involving the development of a LISP interpreter written in a high-level language and implemented on a minicomputer. In addition to the initialization, the major portions of the interpreter discussed in this report are: (1) Memory Management; and (2) Output.

The INTERDATA 8/32 was used during the program and test phases, and it proved to be an excellent computer. After the initial hurdles of learning how to keep it running, the remaining problems were insignificant.

This project enabled the author to gain "hands-on" experience with devices which had previously only been read about in textbooks. It also provided valuable experience in working with a minicomputer.

Chapter 2

MEMORY MANAGEMENT DESIGN

2.1 Introduction

This report is concerned with two different types of memory management systems. Both systems use dynamic memory allocation techniques, and both allocate variable sized blocks of memory from a single area called ISPACE. When the allocatable space is full, memory is regenerated by moving all blocks in use to the low address portion of memory. All blocks which remain in the high address portion are available for reuse.

The two memory management modules discussed in this report are named IGTBLK and GETBLK. Both systems use an indirect method of addressing and will be discussed in detail in chapter three. The basic terminology of the two systems, and the fundamental concepts involved will be discussed in this chapter.

2.2 Marking

During memory regeneration all blocks must be identified as current if they are still in use. This process is called marking. Normally a single mark bit is set aside in each block header for this purpose, but a whole word was used for this project. The reason for using a word will be discussed in a chapter on the INTERDATA. The position used

for marking will be referred to as the MARK bit or MARK field. If a block is referenced directly or indirectly by a variable, the block is currently in use and should be retained in memory. The executor is allowed to explicitly free any block no longer needed. A block could also be freed by GETBLK, but so far this has not been required.

2.3 Compaction

Sequential memory allocation is a fast method of memory allocation, but it requires that all useable memory be contained in a contiguous block of memory. During memory regeneration and after marking, the system which moves all useable space into a contiguous block is called memory compaction. During memory compaction all current blocks (all marked blocks) are moved to the low address portion of memory, and all of the pointers are changed to reflect the new locations.

The use of memory compaction makes memory allocation simple; searching for a block that is large enough is not required. Compaction also eliminates the problem of memory fragmentation, which happens when there is adequate available memory, but the blocks are spread all over memory in such small portions that a single request for space cannot be met. Since compaction makes all current blocks contiguous, it causes a reduction in page thrashing.

2.4 System Differences

This report describes two different memory management systems, IGTBLK and GETBLK. GETBLK was designed to be used by the input, scan, and initialization modules of the interpreter. IGTBLK was designed to be used by the executor. GETBLK provides for contiguous storage in large blocks without headers interspersed. This method is simple and fast; however, it does not allow for blocks pointing to other blocks. IGTBLK is more general, but it is slower and uses much more space for pointers and header information. GETBLK requires that all current blocks be referenced by an indirect pointer table, IPT. IGTBLK relaxes this restriction in that it requires current blocks to be referenced by an initial address list, IAL, only if they are not reachable through any sequence of pointers in blocks, starting with a block referenced by IAL.

The executor module should decide which blocks must be referenced by the initial address list. In general only pointers to the beginning of a structure are placed on the initial address list. It was impossible to coordinate with the programmer of the executor since that is the subject of a future report; therefore, it became necessary to implement a patch for test purposes. In the current implementation all blocks are pointed to by the IAL.

2.5 IGTBLK Concept

IGTBLK is a function and can be called by any module, but it was designed to be initiated by the STORE module. The executor would initiate a call to STORE, which would then

obtain the desired space from IGTBLK. STORE would then place the data into the block and return the address of a pointer to the block. The FREED module was designed to allow any block to be freed. Use of the FREED module was necessary during the testing of memory regeneration and maintenance of the initial address list. The executor module can properly save and free blocks by passing a flag in the calling statement to the STORE module. During each memory regeneration, any current block may be moved. In this case all pointers to that block must be adjusted. This creates the following requirements:

(1) The header must contain information about which words in the blocks are to be used as pointers to other blocks.

(2) All required blocks not referenced by internal block pointers must be placed on the IAL.

(3) The marking routine must mark all blocks referenced by the IAL or by internal block pointers of current blocks.

2.5.1 IGTBLK Data Structures

The data structures used by this module are shown in Figures 2-2 thru 2-4. Figure 2-2 shows the fields that are contained in each block. The LENGTH field contains an integer representing the number of words in the block including the header. The POINT field is used during memory regeneration. It is initialized to zero before being assigned for storage. During marking the POINT field is used to indicate which word in the data field is being processed.

Compaction uses the POINT field to indicate the new location of the block after movement. The MARK field is initialized to zero and is set to one if the block is in use at the time of marking. The USE field contains a use bit for each word in the DATA field. The use bit is set to zero or one by the STORE module to indicate a use of data or pointer respectively. In the DATA field each word contains a pointer or data.

Figure 2-3 shows the IGTBLK linkage with all blocks being referenced directly by the IAL or indirectly by the internal block pointers. Figure 2-4 shows the overall system. NLIST is a variable indicating the number of elements on the IAL. ISLIST is a constant indicating the maximum size of the IAL. ISPACE is an array used for allocating memory. IFREE is a pointer to the first unallocated block in memory. ISIZE is the maximum size of allocatable memory.

2.5.2 Marking Algorithm

```
I=1
do while (I ≤ NLIST)
      BEGIN
           Mark the block located at IAL(I)
           Depth first mark all unmarked blocks
             referenced by internal pointers
            I=I+1
      END
```

2.5.3 IGTBLK Marking Module

Figure 2-5 thru 2-10 show the marking procedure used by

blocks and reversing the pointers in the structure while descending (moving deeper into the structure), then restoring the pointers while ascending. Figure 2-5 shows the blocks prior to marking. The first block is of length seven; the MARK field is zero; it has two use bits with value one, indicating the two pointers. Figure 2-6 shows that the block being processed is pointed to by the variable A and the point and MARK field are set to one. The value one in the POINT field indicates that the first word is being processed. Temporary pointers B and C are set to zero.

Figure 2-7 shows the status of the blocks after one descent. Each time that a non-null pointer is found that points to an unmarked block, the following steps apply:

(1) B is changed to point to the same block as A.

(2) A is changed to point to the block referenced by the pointer being processed.

(3) The pointer being processed is changed to point to the block referenced by C.

(4) C is changed to point to the same block as B.

(5) The mark and POINT fields of the new block pointed to by A are set to one, and the processing continues.

All of the above steps can be traced through on Figure 2-7. Notice that the next to last pointer in the first block is now zero. Both B and C were initialized to zero before descent as shown at Figure 2-6. Figure 2-8 shows the status of the blocks after one more descent, and Figure 2-9 shows the status of the blocks after the descent is complete. Notice that the POINT field contains a value of three and there are only two words in the block. This

situation is caused by a loop which is used to increment the POINT field each time a word is processed. When the POINT field exceeds the number of words in the block, the ascent will follow. During the ascent the pointers are set back to their previous values. Figure 2-10 shows the status of the blocks after one ascent.

2.5.4 IGTBLK Compaction Module

The compaction routine for this module is more complicated than the one for GETBLK because of the additional pointers involved. The compaction is done in steps which are:

(1) Process all current blocks, and put the new address of each block in the POINT field of the header.

(2) Update each element on the IAL so that it contains the value of the POINT field of the block that it references.

(3) Process the blocks sequentially and change each pointer in the DATA field to reflect the new location of the block that it points.

(4) Move the blocks to their new locations.

(5) Update IFREE to reflect the first available block to be allocated in the new contiguous block.

IGTBLK BLOCK

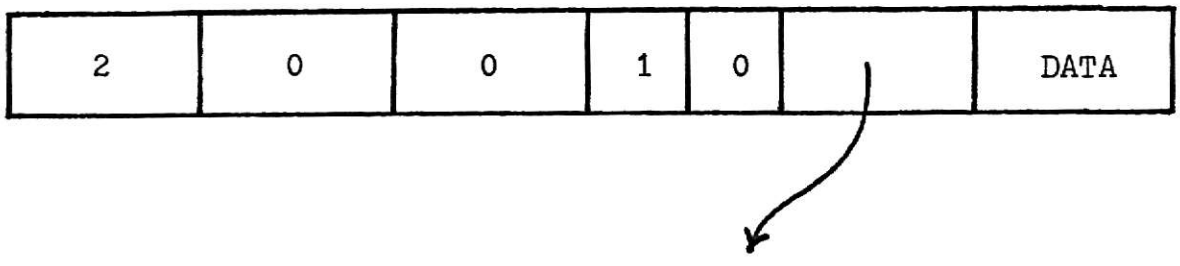


Figure 2-2

IGTBLK LINKAGE

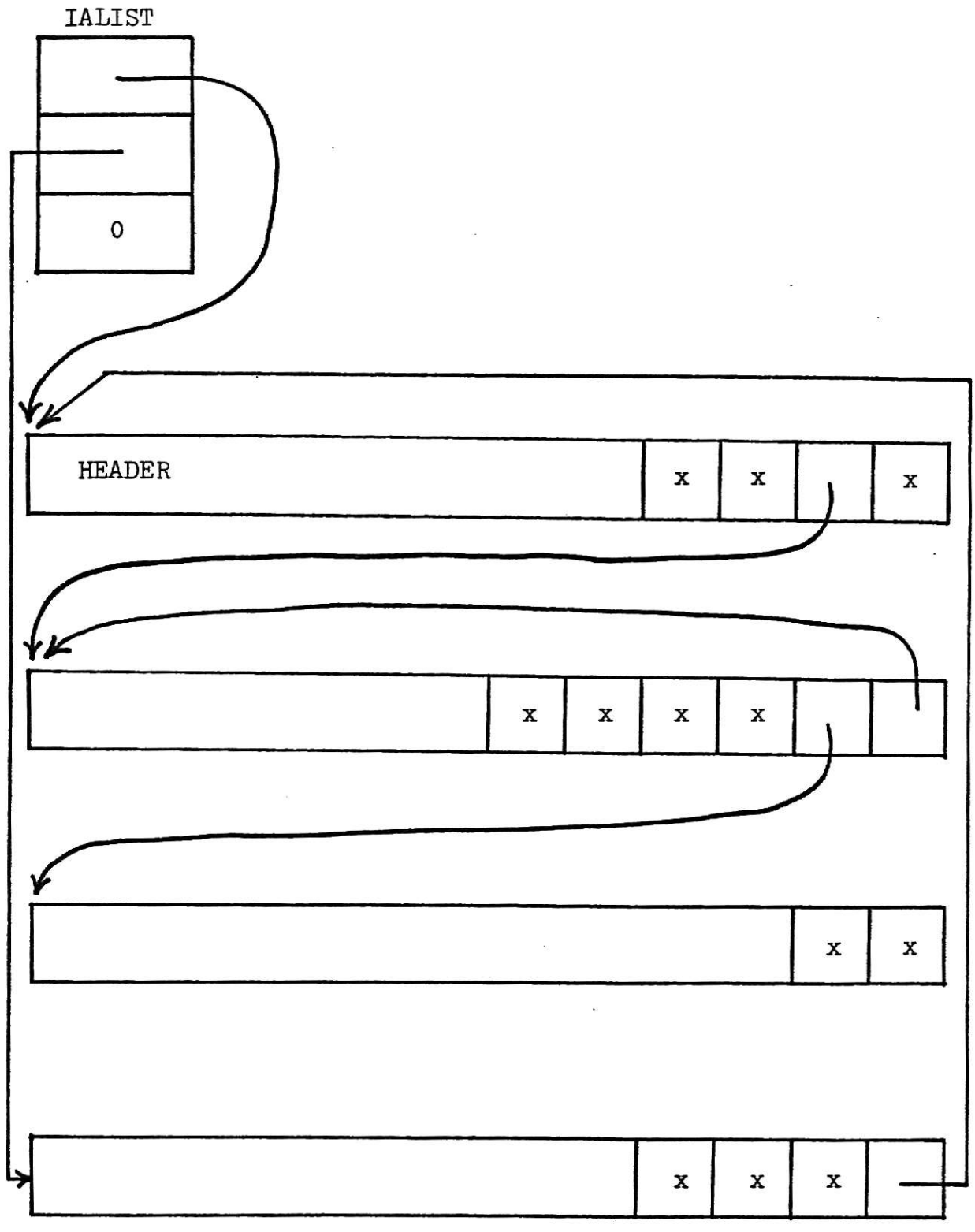


Figure 2-3

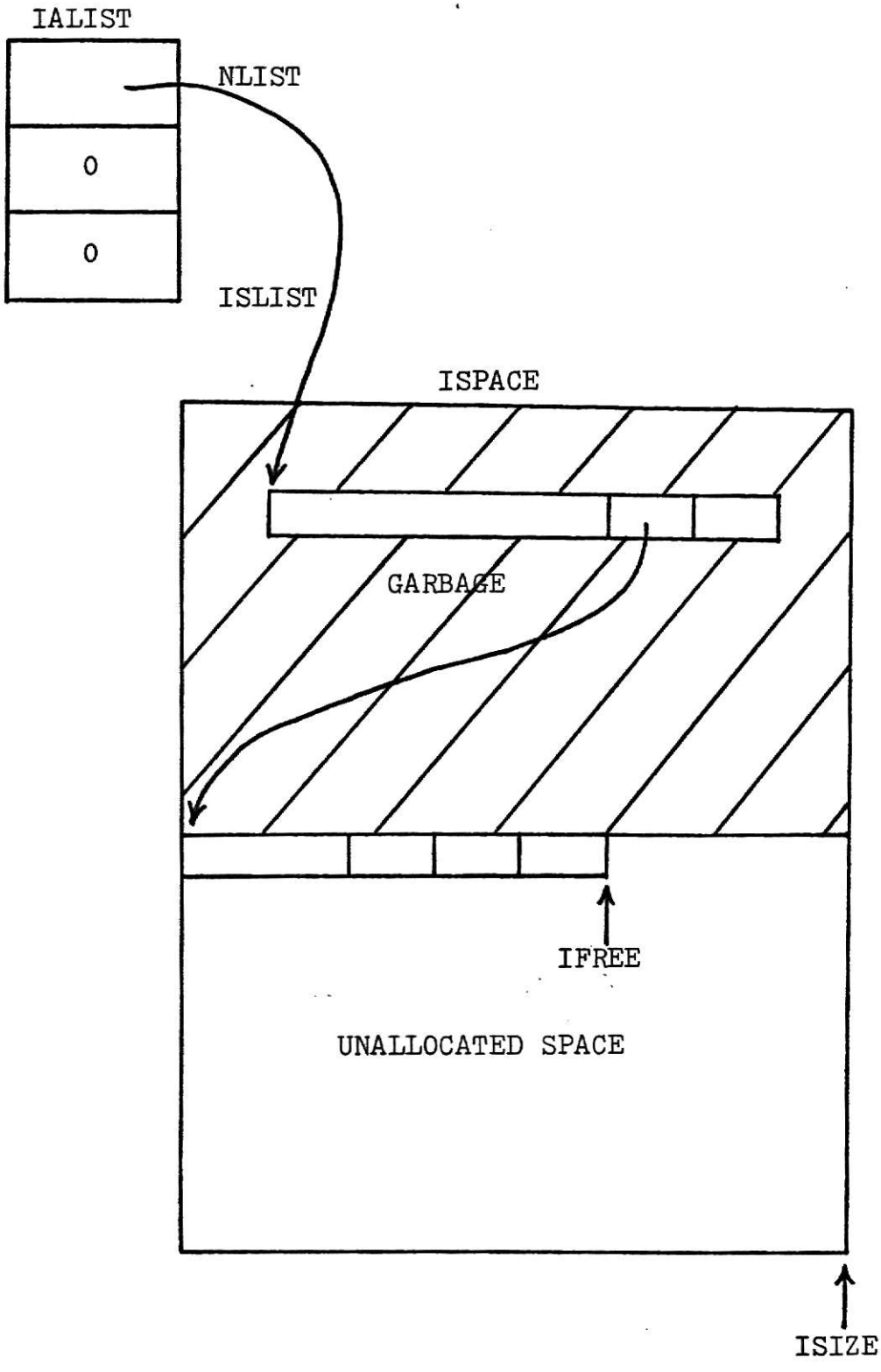

IALIST
NLIST
0
0
ISLIST
ISPACE
GARBAGE
IFREE
UNALLOCATED SPACE
ISIZE

Figure 2-4

BEFORE MARKING

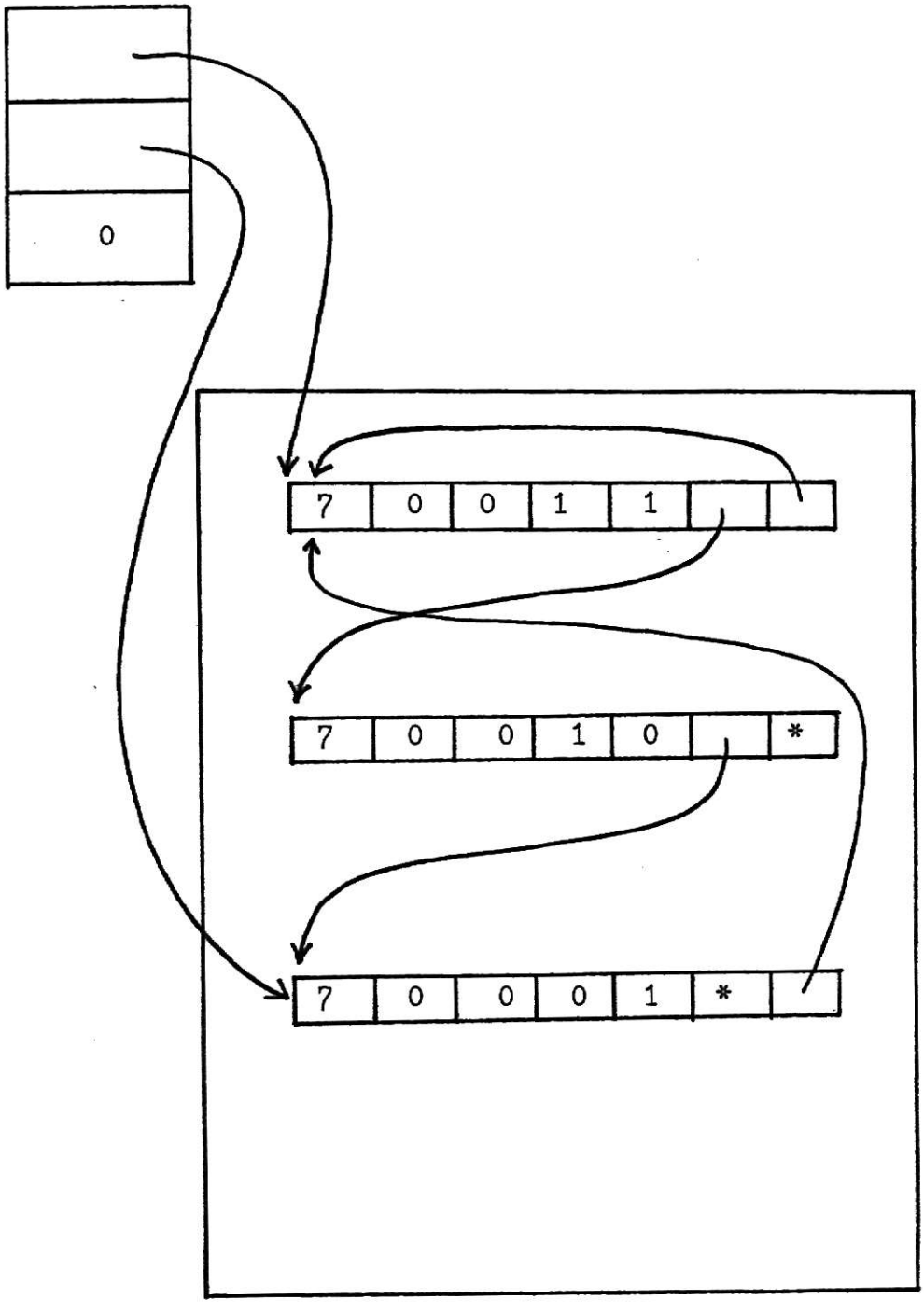


Figure 2-5

BEFORE DESCENT

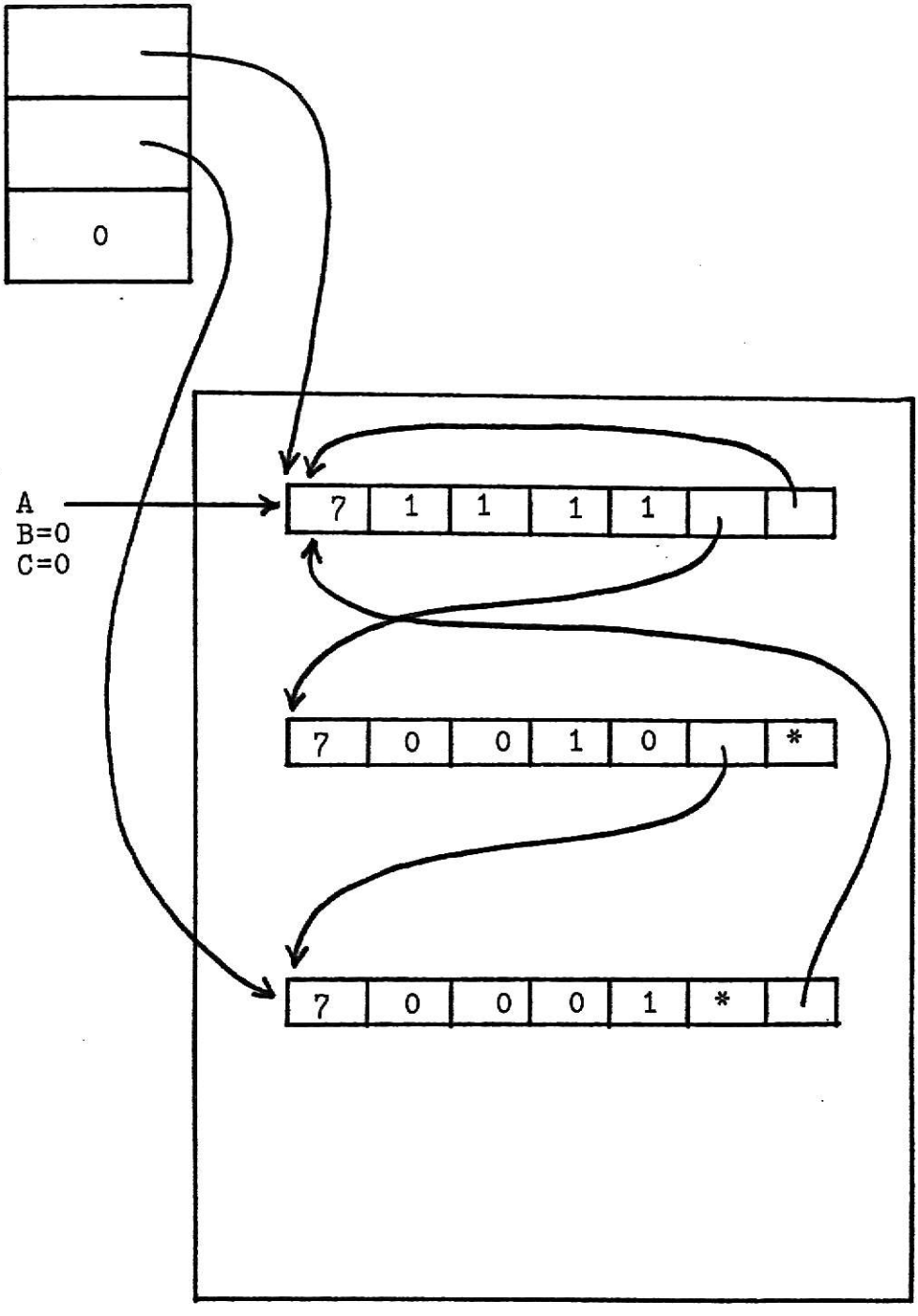


Figure 2-6

ONE LEVEL DESCENT

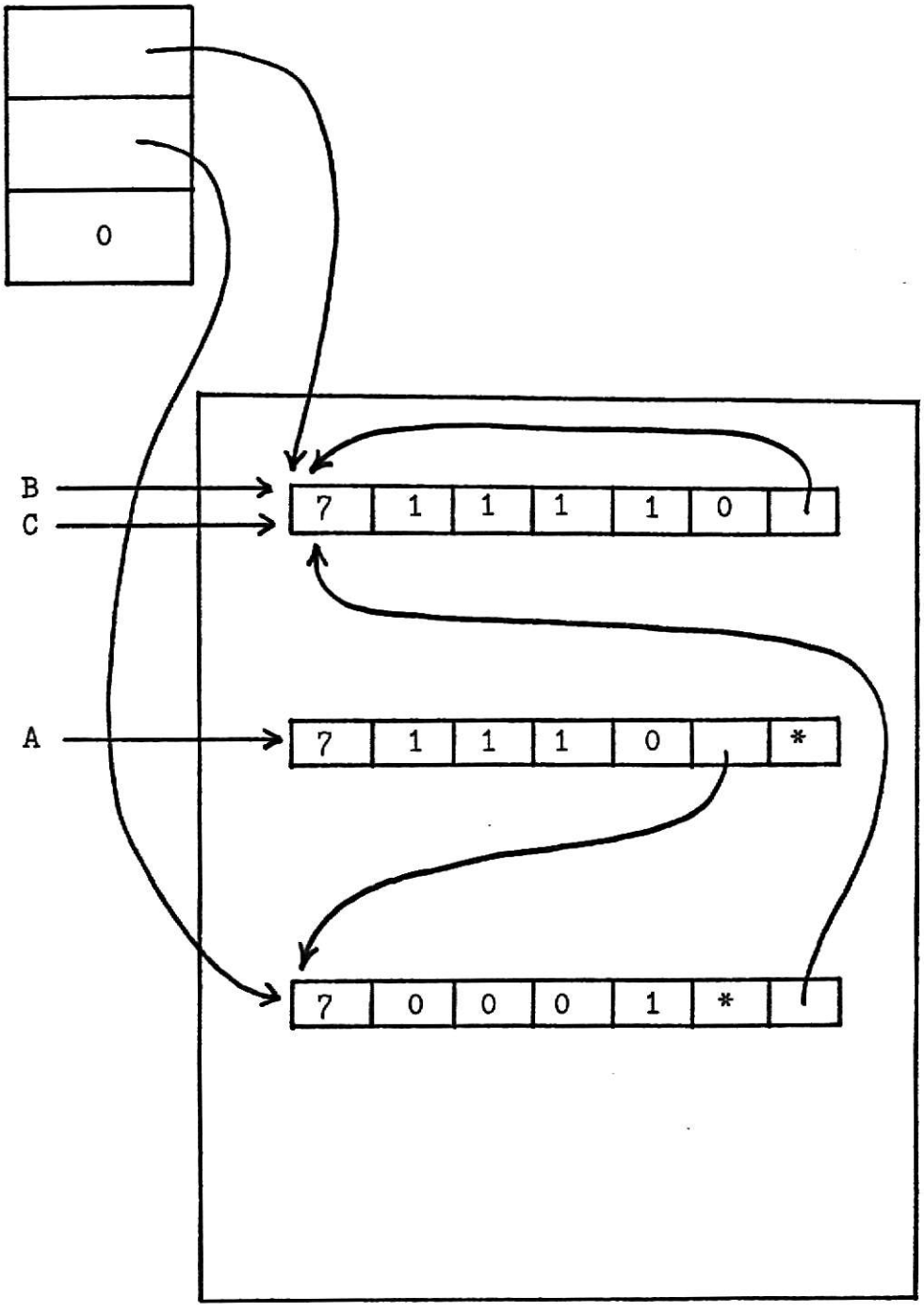


Figure 2-7

SECOND LEVEL DESCENT

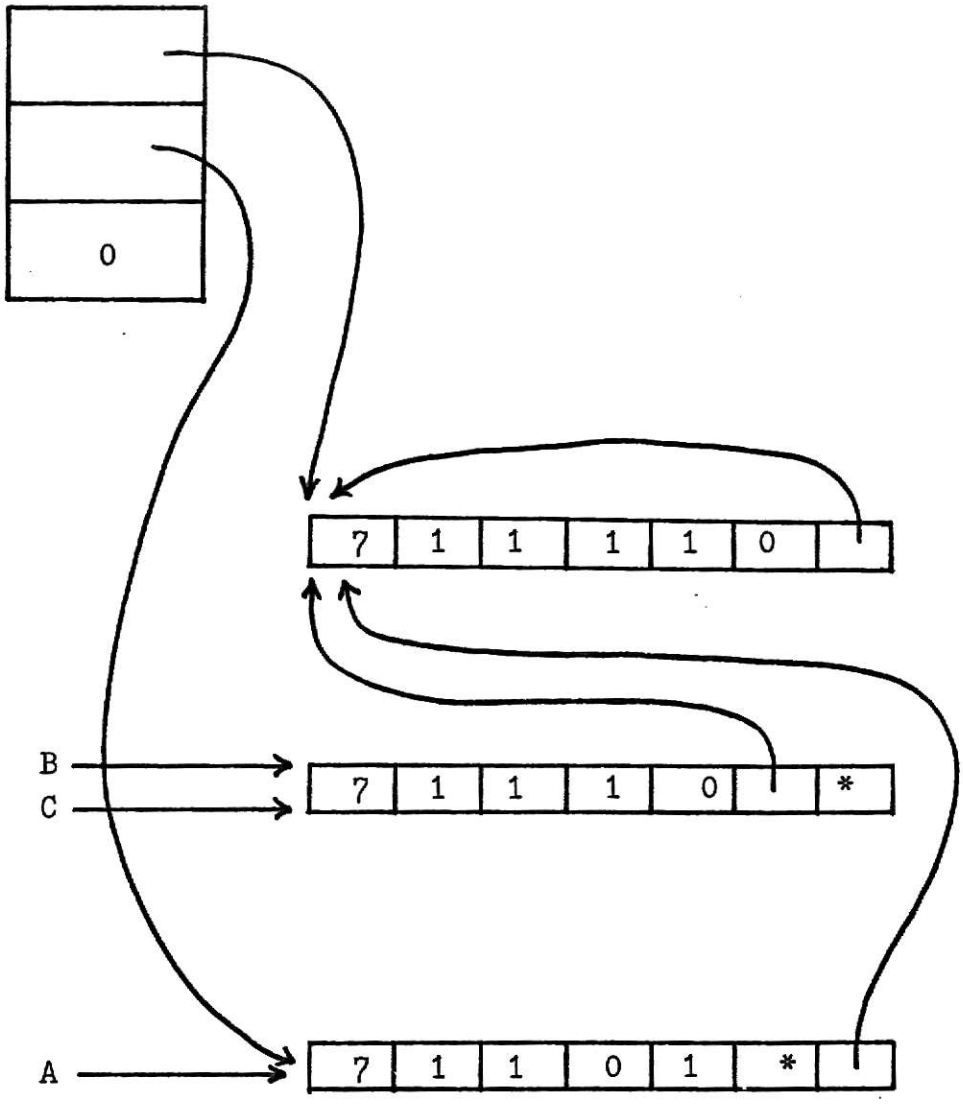


Figure 2-8

PRIOR TO ASCENT

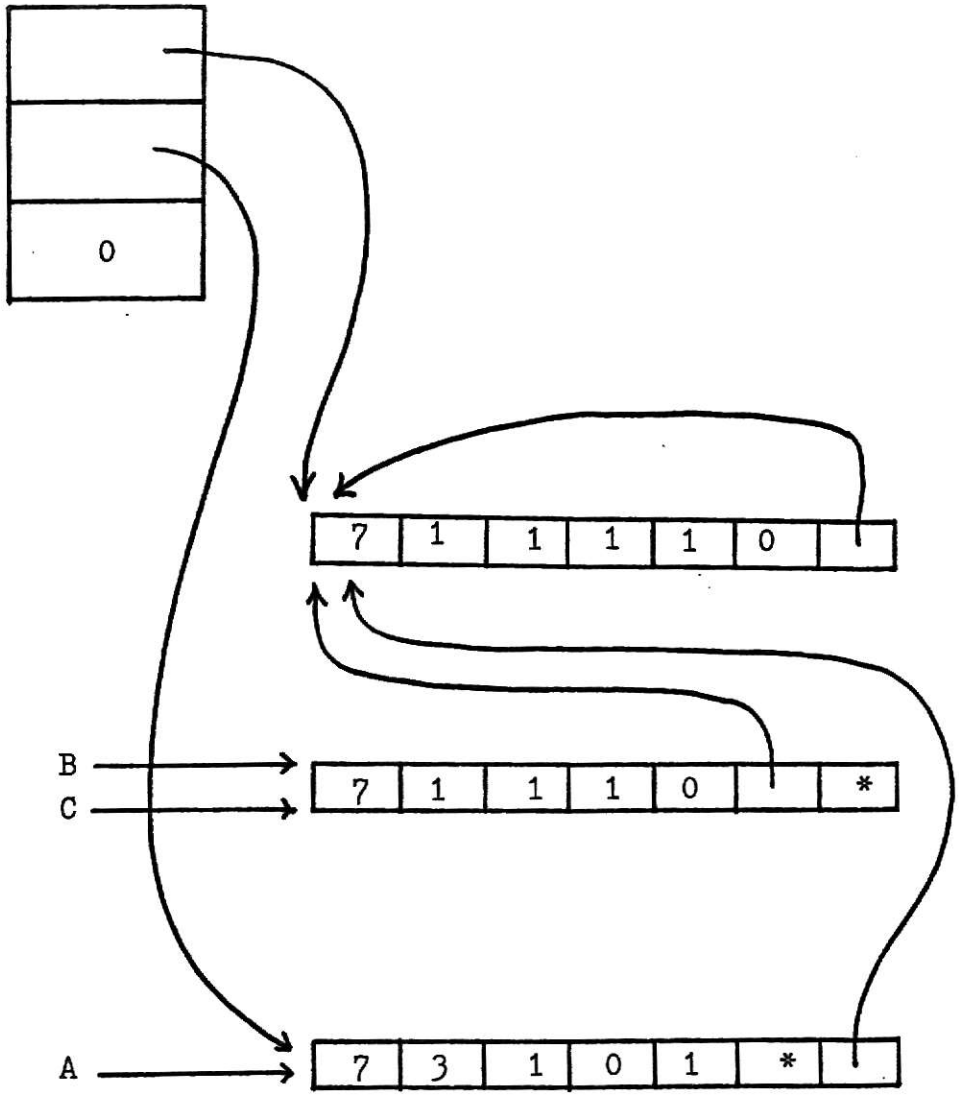


Figure 2-9

AFTER ONE ASCENT

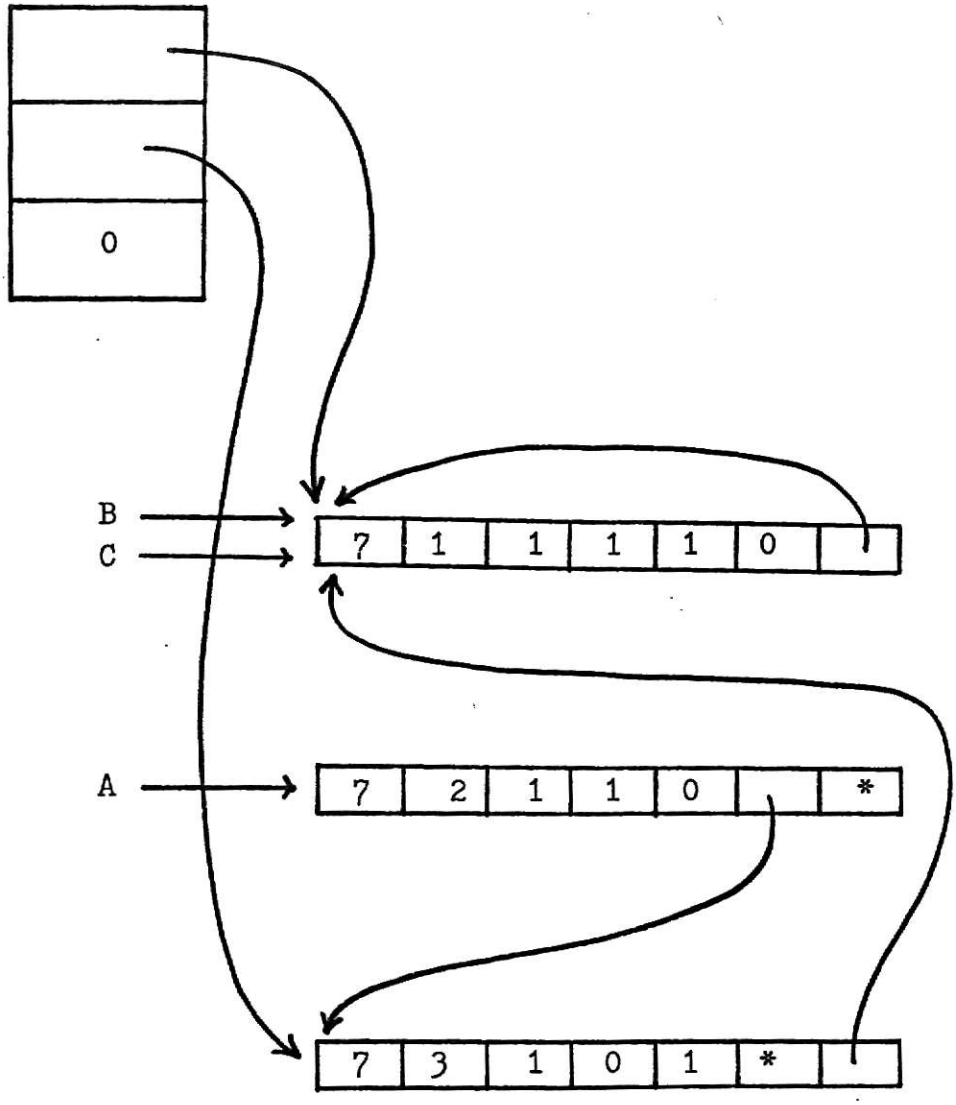
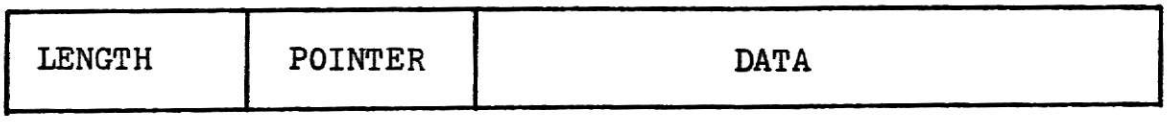


Figure 2-10

GETBLK BLOCK



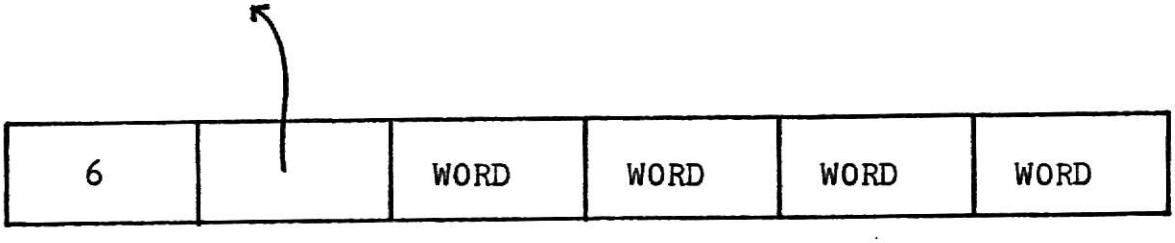


Figure 2-11

GETBLK LINKAGE

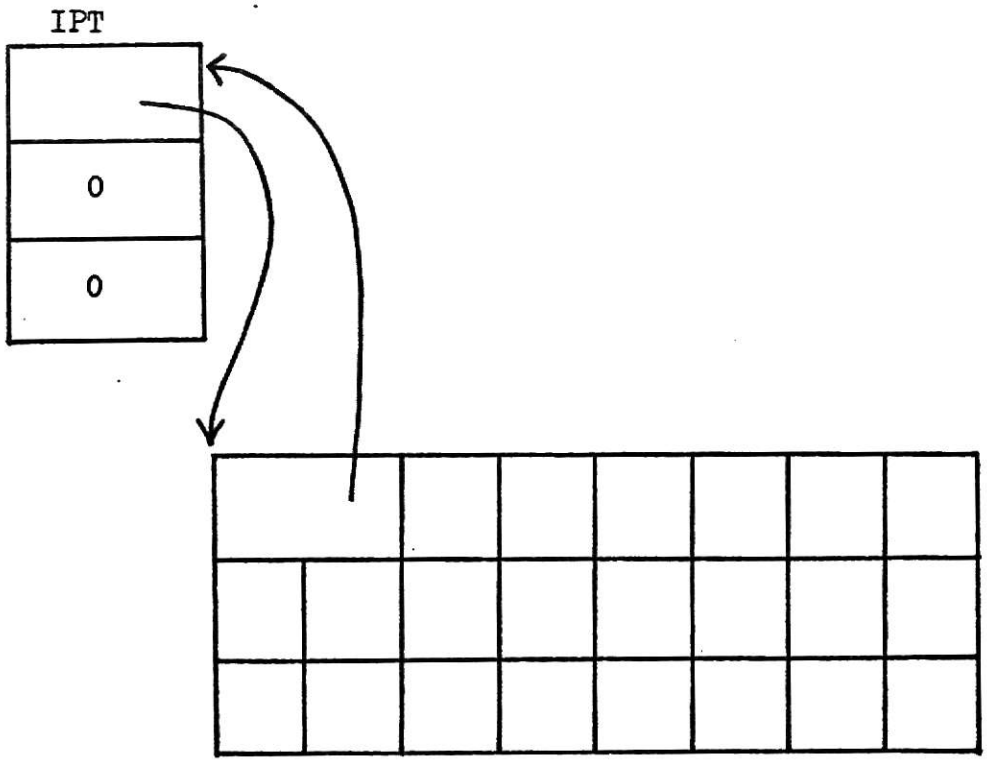


Figure 2-12

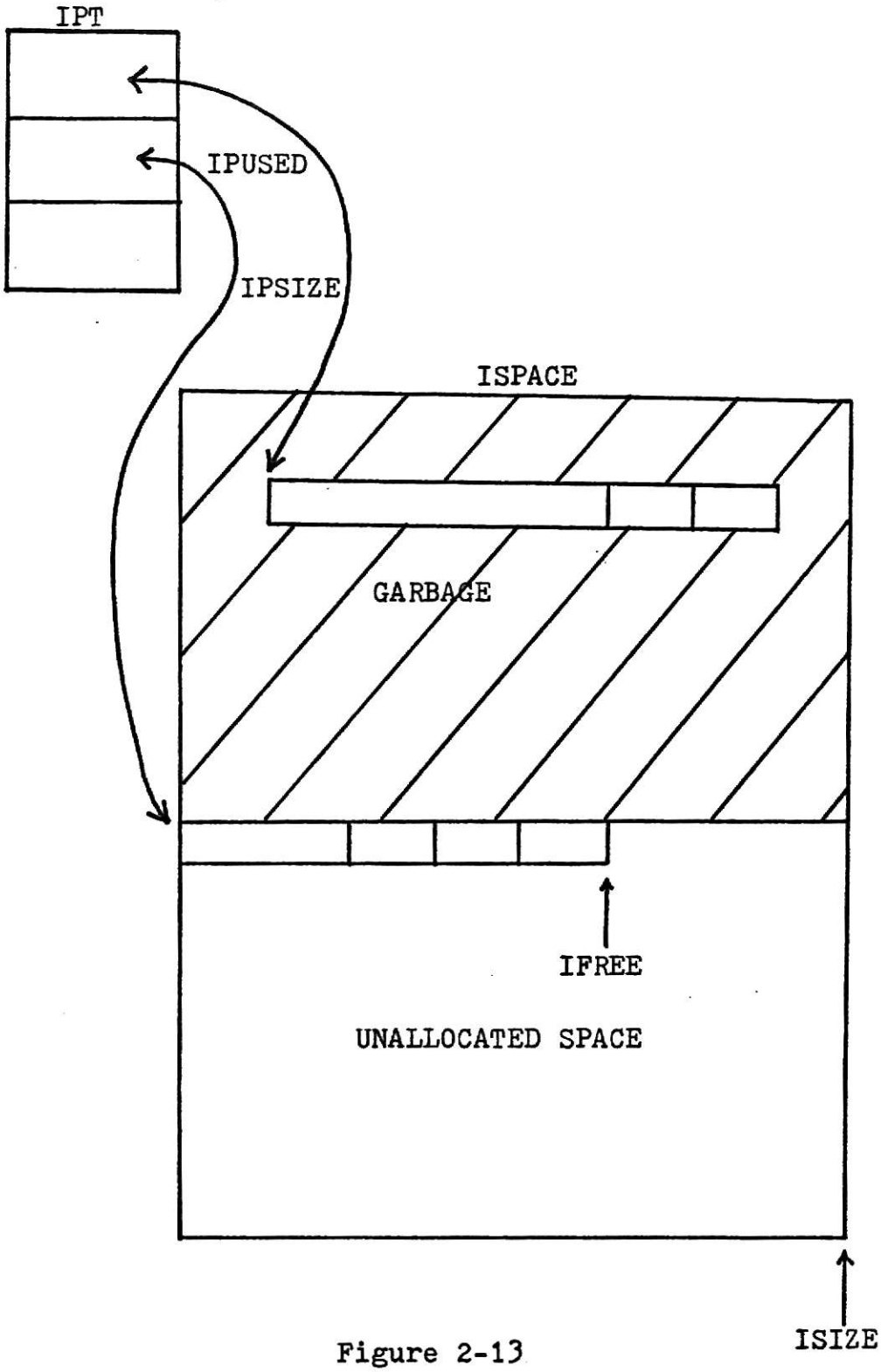

IPT
IPUSED
IPSIZE
ISPACE
GARBAGE
IFREE
UNALLOCATED SPACE
ISIZE

Figure 2-13

2.6 GETBLK Concept

As mentioned before this module was designed for use by the READ module, the SCAN module, and a portion of the INTERPRETER modules. The main difference between this module and IGTBLK is that this module allows a large block of data to be allocated in several different cells without any headers being interspersed. This system was used to great advantage during the testing phase of the READ module. When this module is called a block of words of any size can be requested, normally eighty words are requested which is the length of an input card. A lock is passed if a large contiguous block is desired which keeps a header from being placed on each block. The starting address of the data is returned to the calling routine each time a block is requested. On the last request from the calling module the lock is changed and the header information is updated to reflect the current length of the block. The only requirements that exist for this system are:

(1) Each block must have only one pointer to it which is the indirect pointer table,IPT.

(2) Each block must have at least one reference to it, and the IPT must be changed to zero for each block not in use.

2.6.1 GETBLK Data Structures

The data structures used by this module are shown in Figures 2-11 thru 2-13. Figure 2-11 shows the fields that are contained in each block. The LENGTH field includes the

number of words in the block including the header. The POINT field is used to point back to the indirect pointer table, IPT. The DATA field can contain any type of data and any number of words. There is no MARK field in this block as the length field is used during marking.

Figure 2-12 shows the GETBLK linkage which is quite simple. The IPT contains the pointers to the current blocks. IPUSED is the last element on the IPT. IPUSED is initialized as zero. IPSIZE is the maximum number of elements on the IPT. ISPACE is the same array used by IGTBLK. The space used by GETBLK will always be at the low address portion of memory, and this portion of memory will not be allocated again after the executor starts requesting space from IGTBLK. ISPACE and IFREE are the same as in the other module. The overall GETBLK system is shown in Figure 2-13.

2.6.2 Regeneration Algorithm

```
I=1
comment  Mark each block in use
do while (I≤NLIST)
     begin
          Change the LENGTH field of the block
             referenced by IPT(I) to negative
          I=I+1
     end
comment  Change the IPT to point to the new location
I=1
IFREE1=1
do while (I<IFREE)
     begin
          if LENGTH<0
             Pointer(I)=IFREE1
             IFREE1=IFREE1+LENGTH
          else
             I=I+LENGTH
     end
comment  Move the blocks
     Sequentially move all blocks
     Update IFREE1=IFREE1+LENGTH  after each move
```

```
comment  All blocks have been moved
     IFREE=IFREE1
     end
```

The algorithm shows that there are three stages of operation in GETBLK regeneration (1) mark the blocks; (2) update the IPT; and (3) relocate the blocks.

Chapter 3

MEMORY MANAGEMENT MODULES

3.1 Introduction

This chapter describes the structure of both of the memory management systems. The calling statement, input requirements, output, and sequence of operation will be covered in all modules unless no explanation is needed.

3.2 Main Submodule IGTBLK

The function of this main submodule is to : (1) find the first available block in memory and pass the location of the block to the calling routine; (2) print an overflow message when required and abort the run when recovery is impossible; (3) regenerate memory by using garbage collection and compaction when necessary.

3.2.1 Function IGTBLK

This module is called by the STORE module and it calls REGEN1 when required for memory regeneration. The relationship of the overall main submodule is shown at Figure 3-1. The code for this module is at Appendix C.

IGTBLK MANAGEMENT RELATIONSHIP

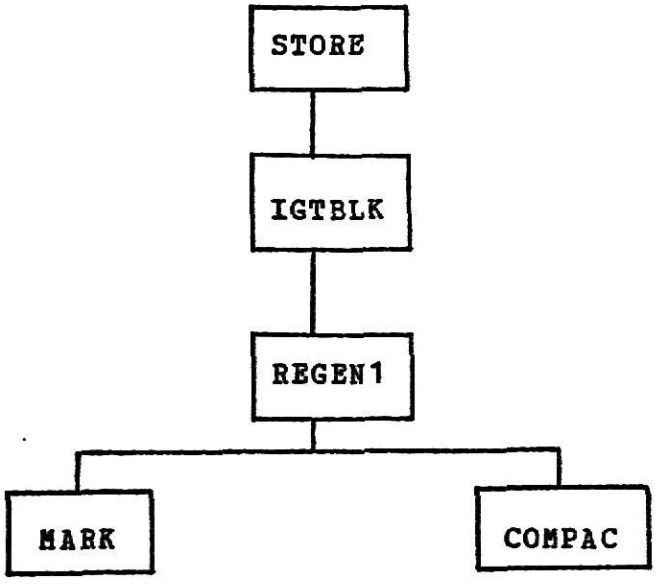


Figure 3-1

3.2.1.1 Calling Statement

IGTBLK(NWORDS) contained in an executable statement

3.2.1.2 Input

NWORDS is an input parameter passed as an argument in the calling statement and represents the number of words to be allocated. The labeled common blocks SPACE and IAL are needed.

```
 COMMON/SPACE/IPTABL(50),ISPACE(2000),IPUSED
1,IFREE,IPSIZE,ISIZE

COMMON/IAL/IALIST(20),NLIST,ISLIST
```

ISPACE,IFREE, and ISIZE are used from the common block SPACE. ISPACE is an array used to store the different

elements in allocatable memory. IFREE is a pointer used to indicate the next free block in memory, and ISIZE is the size of allocatable memory. All of the IAL common block is used. IALIST is an initial address list used to store the pointers to all active blocks. NLIST is the number of pointers on the initial address list, and ISLIST is the maximum number of pointers that may be placed into the IALIST.

3.2.1.3 Output

If an overflow condition is created this module will print a message which indicates the reason for program termination. If an overflow condition does not exist, the address of the present block being assigned is placed in the initial address list and NLIST is updated. The location of the block on the initial address list is assigned to IGTBLK which is passed back to the calling routine. The total number of words including the header is placed in the LENGTH field of the header, and the mark and pointer fields are set to zero. The use bits are set to zero before control is passed back. IFREE which is the pointer to the first cell of the next available block is updated. The address of the present block being assigned is placed in the initial address list.

3.2.1.4 Sequence of Operation

(1) This module selects the first available pointer

location in the initial address list to be used for the address of the block to be assigned. If a pointer location does not exist an error message will be printed and execution will terminate.

(2) If the number of words to be stored will cause an overflow on memory size, REGEN1 is called to regenerate space.

(3) If the number of words to be stored still causes an overflow on memory size an error message is printed and execution stops.

(4) The number of elements on the initial address list, NLIST is updated.

(5) The function name is assigned to the pointer location.

(6) The header is initialized; the use bits are set to zero, and IFREE is updated.

(7) Control is returned.

3.2.2 Subroutine REGEN1

This module is called by IGTBLK to regenerate space. It calls the MARK and COMPAC modules. The relationship of this module within the overall main submodule is shown at Figure 3-1. The code for this module is shown at Appendix C.

3.2.2.1 Calling Statement

```
CALL REGEN1
```

3.2.2.2 Input

The only input parameters needed by this module are the SPACE and IAL common statements. The elements of the initial address list are processed sequentially.

3.2.2.3 Output

All output of this routine is produced by its submodules, MARK and COMPAC.

3.2.2.4 Sequence of Operation

(1) The MARK module is called for each block currently in use.

(2) All blocks in use are compacted.

(3) Control is returned to IGTBLK.

3.2.3 Subroutine MARK

This module is called by REGEN1 and does not call any other module. The relation of this module to other modules is shown at Figure 3-1. The code for this module is at Appendix C.

3.2.3.1 CALLING STATEMENT

CALL MARK(N)

3.2.3.2 Input

N is the address of the current block to be marked. All pointers contained within this block will be processed until all blocks pointed to by this block and the blocks that it points to are marked. Two variables ISPACE and IFREE are used from the common block SPACE.

3.2.3.3 Output

The MARK field, ISPACE(BLOCK + 1), and the POINT field, ISPACE(BLOCK + 2), are changed. The MARK field is set to one and the POINT field is changed to show the word being processed during a traverse of the blocks being marked.

3.2.3.4 Sequence of Operation

(1) If the block is not marked, the mark and POINT fields are set to one.

(2) Temporary blocks, B and C, used in traversing the blocks are set to zero.

(3) The use bits are checked to see if the word contains a pointer or data.

(4) If the word is a pointer it is checked to verify that it is not null.

(5) If the pointer is not null, the block that it points to is checked to verify that it is marked.

(6) If the block is not marked, block B is set to point to the current block A. A is then changed to point to the new block, and the word containing the pointer is changed to

point to block C.

(7) C is changed to point to block B.

(8) Steps 3-7 are repeated until the bottom is reached.

(9) C is changed to point to the block pointed to by the B block DATA field. This is a block previously marked.

(10) B block DATA field is changed to point to block A.

(11) A is changed to point to B, and B to point to C.

(12) This process is continued until block B again is equal to zero.

3.2.4 Subroutine COMPAC

This module is called by REGEN1 and it calls no other modules. The relationship with other modules is shown in Figure 3-1. The code of this module is at Appendix C.

3.2.4.1 Calling Statement

```
CALL COMPAC
```

3.2.4.2 Input

The labeled common blocks SPACE and IAL are needed by this module. The three elements of the header in each block are used to relocate the blocks.

3.2.4.3 Output

The number of elements on the initial address list, NLIST, is updated and the pointers in the initial address

list, IALIST(I), are changed to point to the new location of the blocks indicated by the POINT fields. The MARK fields are reset to zero. IFREE is updated to point to the first available block.

3.2.4.4 Sequence of Operation

(1) If the block is marked it should be relocated. If there are any unused blocks in front of it, Put the new address of the block in the POINT field and change IFREE1 to point to the next block.

(2) If a block is not marked IFREE1 is not changed until another block is relocated.

(3) After the location of the block is put in the POINT field, put its location in the initial address list.

(4) All pointers within the blocks are now changed to point to the location contained in the POINT field of the block that it pointed to previously.

(5) Relocate all marked blocks and update IFREE1.

(6) All blocks having been relocated , update IFREE.

(7) Return control to REGEN1.

3.3 Main Submodule GETBLK

The function of this main submodule is to: (1) find the first available block in memory and pass the location of the block to the calling routine; (2) only assign headers to new blocks as indicated by the calling statement; (3) print an overflow message when required and abort the run when recovery is impossible; (4) regenerate memory by using garbage collection and compaction when necessary.

3.3.1 Subroutine GETBLK

This module is called by INPUT, SCAN, and modules within the INTERPRETER. It calls REGEN when required. The relationship of the overall main submodule is shown at Figure 3-2. The code for the main submodule is at Appendix E.

3.3.1.1 Calling Statement

```
CALL GETBLK(NWORDS,IADDRS,LOCK)
```

3.3.1.2 Input

NWORDS is an input parameter passed as an argument in the calling statement. NWORDS is a variable describing the number of words to be stored. LOCK is an input parameter passed as an argument in the calling statement. It has a value of zero or one. A value of zero indicates that a larger block is desired and the completion of header information must wait until the lock is changed to one. The

labeled common block SPACE is required, and all of the variables in that block are used in this module.

GETBLK MANAGEMENT RELATIONSHIP

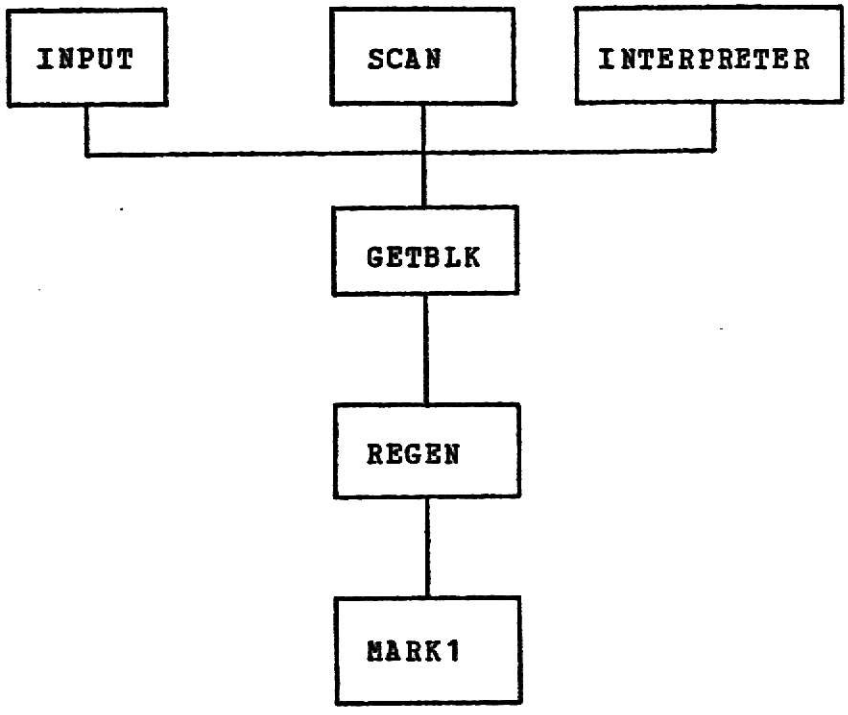


Figure 3-2

3.3.1.3 Output

If a condition exists that causes a space overflow this module will print a message indicating the reason for program termination. IADDRS is an output parameter passed to the calling routine thru the calling statement. IADDRS is the starting address of data to be stored in the new block. This module will cause memory space to be placed into a contiguous block of any desired size with a small header

identifying the length, and pointer location in the indirect pointer table. A header for GETBLK is shown at Figure 2.4. IFREE which is the pointer to the first cell of the next available block is updated.

3.3.1.4 Sequence of Operation

(1) Find the first available space in the indirect pointer table (IPT).

(2) If the table is full call REGEN to regenerate space.

(3) If REGEN was called, try again to find space in the IPT. If the IPT is still full then print an overflow message.

(4) If space was available in the IPT, see if the desired size block will fit in the remaining memory space. If the block is too big call REGEN to regenerate space and then try again after control is returned. ELSE if the block fits into memory, then allocate the block.

(5) If a larger contiguous block is not desired assign the header information, update the IPT, update FREE, and return the starting address IADDRS to the calling routine.

(6) If a larger contiguous block is desired set the flag to indicate that a block is being built and wait until the lock is changed before assigning a value to the LENGTH field in the header.

(7) If the lock changes update the total number of words in the block, ITOTAL, update FREE, update the header, and change the contiguous block indicator, IFLAG, back to

zero.

3.3.2 Subroutine REGEN

This module is called by GETBLK and calls no other module. The relationship of this module is shown at Figure 3-2. The code of the module is at Appendix D. The function of this module is to act as a driver for MARK1, use the indirect pointer table to keep track of the current blocks, and perform compaction on all blocks in use.

3.3.2.1 Calling Statement

```
CALL REGEN
```

3.3.2.2 Input

This module uses all of the elements in the labeled common statement, SPACE.

```
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE
```

IPTABL is an array used to store the indirect pointers. ISPACE is an array used to store the different elements in allocatable memory. IPUSED is a variable indicating the last used space in the IPT. IFREE is a pointer used to indicate the start point of the next free block in memory, ISIZE is the size of allocatable memory,and IPSIZE is a variable indicating the size of the IPT.

3.3.2.3 Output

If regeneration is successful IFREE is updated with the new location of the first available block. All pointers in the indirect pointer table are changed to point to the new location of their previous block. Any block that is not in use has its indirect pointer set to zero. All blocks in use are compacted at the low address portion of memory.

3.3.2.4 Sequence of Operation

(1) Call MARK1 to mark all used blocks.

(2) Using the header information, process each one of the blocks in memory.

(3) If the block is marked then assign the starting address to its indirect block pointer, IBP , and move IFREE1 to the start of the next block; move to the start of the next block.

(4) Perform steps 2 thru 3 until the address of the next block is greater than IFREE.

(5) Starting with the first block relocate all blocks in use.

(6) If the block is marked then relocate the block; update IFREE1; unmark the block.

(7) Move to the next block and perform steps 6 and 7 until the starting address is greater than IFREE.

(8) All blocks have been moved so update IFREE.

(9) Return control to GETBLK.

3.3.3 Subroutine MARK1

This module is called by REGEN and does not call any other modules. Its relationship to other modules is shown at Figure 3-2. The code of this module is at Appendix D. The function of this module is to mark each block referenced by the IPT by changing the LENGTH field to a negative number.

3.3.3.1 Calling Statement

```
CALL MARK1
```

3.3.3.2 Input

Variable IPUSED, and arrays IPTABL and ISPACE from the SPACE labeled common are used. IPUSED is the last used space in the IPT. IPTABL(I) contains the pointer to the block being processed. ISPACE is the array used for storage of elements.

3.3.3.3 Output

All blocks pointed to by the IPT have their length fields marked by changing the field to negative.

3.3.4 Sequence of Operation

(1) Locate each block pointed to by the IPT.

(2) Mark the block by making the LENGTH field negative.

(3) Return control to REGEN.

Chapter 4

AUXILLIARY ROUTINES

4.1 Introduction

This chapter will discuss four different routines which are mainly supporting modules. The modules are SETUP, STORE, FREED, and OUTPUT. The first three routines are used primarily in memory management and the last can be used by any routine. The code for these modules is at Appendix B.

4.2 Subroutine SETUP

This module is called by the driver and does not drive any other module. The relationship of the module is shown in Figure 4-1.

SETUP MODULE RELATIONSHIP

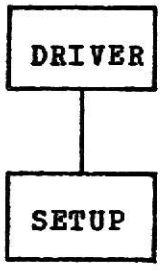


Figure 4-1

4.2.1 Overview

This module pertains to the establishment of the global variables. The only function of the SETUP module is to

establish the size of the variables in common statements. Flexibility is gained by having the common variables initialized in one module. During the testing phase variables can be made small to check the various overflow and recovery mechanisms as well as faster tests. The variables can be easily changed at a future date prior to implementation to take advantage of the available memory. All variables used in memory allocation have been initialized with small values and should only be changed prior to final implementation in order to be fair to the multiple users of the minicomputer.

4.2.2 Calling Statement

```
CALL SETUP
```

4.2.3 Input

The input required by the setup consists of all of the labeled common statements which contain variables to be initialized.

4.2.4 Output

The desired size of all global variables contained in labeled common statements is the only output.

4.3 Subroutine STORE

This module is called by the driver and calls the

IGTBLK module. The relationship of the module is shown in Figure 4-2. Code for the module is at Appendix B.

STORE MODULE RELATIONSHIP

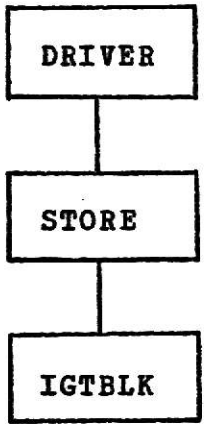


Figure 4-2

4.3.1 Overview

This module is used to store data into the allocatable portion of memory. The module first calls the IGTBLK memory management and obtains the appropriate size block requested by the executor. It then stores the values into the use bits and DATA fields of the block. After the data is stored the address of the pointer contained in the initial address list is passed back to the executor.

4.3.2 Calling Statement

```
CALL STORE(IWORD,IA,NUM)
```

4.3.3 Input

IWORD and NUM are input parameters passed in the

calling statement. IWORD is the array of the data to be stored into the temporary dimension space. NUM is the variable representing the number of words to be stored. Labeled common statements SPACE and IAL are used. All data stored from the executor falls into one of two classes, pointer or data. The type of data is determined by the array IBIT which is passed in the calling statement. IA is an input parameter obtained from the function IGTBLK. IA represents the number of the stored block in the initial address list.

4.3.4 Output

The output produced by the store module is the number of the element on the initial address list (IA). IA is passed back to the calling routine as a parameter in the calling statement. The use bits of the block header are set to the appropriate value of one or zero.

4.4 Subroutine FREED

This module is called by the INTERPRETER, and it does not call any other module. The code for this module is shown at Appendix B.

4.4.1 Overview

The FREED module is designed to free an unused block of memory. It was designed for use with the executor portion of

the INTERPRETER. By freeing a block it will be able to be used again if memory regeneration has ever been called. When a block is freed it is removed from the initial address list, and after garbage collection it can be compacted or placed back on the avail list if links are used.

4.4.2 Calling Statement

```
CALL FREED(I)
```

4.4.3 Input

This module needs the number of the element on the initial address list to be freed, I. The labeled common statement IAL is also used to update the initial address list.

4.4.4 Output

The freed element is taken out of the initial address list if it was a valid element in the first place. The elements of the list are then moved to replace the void space created by the removal. NLIST is then updated.

4.4.5 Sequence of Operation

(1) If the element to be freed is zero then return.

(2) If the number of the element to be freed is greater than the number on the list then return.

(3) If the freed element is not the last one on the

list then move all elements up to fill the void space.

(4) Decrement the number on the list.

(5) Return control to the calling module.

4.5 Subroutine Output

This module is called by the interpreter to print the output of each function call. It does not call any other module. The code for this module is at Appendix B.

4.5.1 Overview

This is a flexible output module which can print any block referenced by the initial address list. The module was designed for the executor portion of the interpreter; however, it could be used by all modules during the testing phase to check what elements had been stored in memory.

4.5.2 Calling Statement

```
CALL OUTPUT(IWORD,IA,NUM)
```

4.5.3 Input

In addition to the SPACE and IAL labeled common blocks, three variables, IWORD, IA, and NUM are needed. IWORD is used for temporary storage for each word coming out of memory. IA is the index of the pointer in the initial address list and is used to identify which block is desired for output. NUM is a variable to be used in the event that an entire block is not desired for output.

4.5.4 Output

This module prints a referenced block from memory. The output is preceded by "THE VALUE IS ".

4.5.5 Sequence of Operation

(1) If ISTART is not equal to zero a specific block which is not necessarily referenced by the initial address list. Print the block.

(2) If ISTART is zero then a block referenced by the initial address list is wanted. Determine the location of the block.

(3) Determine the number of words in the block to be printed.

(4) Check the use bits to see if the data contained in the block is alphanumeric or numeric.

(5) Select the appropriate format statement and print the block.

(6) Return control to the calling routine.

Chapter 5

INTERDATA 8/32

5.1 Introduction

The INTERDATA 8/32 is a very good training vehicle for graduate students and should be used much more than it is. This chapter will discuss some of the problems encountered during this project so that future users might avoid the same pitfalls.

5.2 Documentation

One major problem in using the INTERDATA 8/32 is that there is no central location of publications that a new user can identify that will enable him to use the system. Documentation does exist for the various languages implemented on the INTERDATA. The user should allow sufficient time to become familiar with the procedures which are not implemented in the selected programming language. New versions of compilers have been obtained for COBOL and FORTRAN; however, the documentation that explains the new changes has not been received.

5.3 Hardware

No significant problems were encountered with the INTERDATA 8/32 hardware, but programming was halted or reduced on numerous occasions. Recurring problems have

existed with the memory and with the disk cooling system. About thirty percent of available memory was unuseable for an extended period while a piece of diagnostic equipment was being obtained. The window air conditioners used to cool the computers are inadequate to cool the disks, and several times the disks could not be used due to overheating.

5.4 Software

The software package supplied with the INTERDATA 8/32 consists of: (1) nonproduction INTERDATA software; (2) software taken off the shelf for immediate issue; and (3) software generated by personnel of the Computer Science Department. The greatest hindrance to this project was caused by faulty or inadequate software. The major categories of software delays were; (1) system crashes caused by the software when valid commands were entered at the terminals; and (2) useless error messages which do not explain the errors or errors generated by the software itself.

Some of the specific software problems are discussed below.

5.4.1 Specification Statement

During programming of the block data module an error message of "WRONG PROGRAM" was obtained. All available manuals on FORTRAN and user manuals for INTERDATA were researched to no avail. The program was then run on the IBM

370 sucessfully with no errors. By systematically eliminating portions of the module and numerous compilations it was found that the error was caused by a specification statement "INTEGER*2". BY researching INTERDATA information sheets it was discovered that an error in the FORTRAN compiler does not align the words in memory after "INTEGER*2" is used. If the number of words stored under the specification statement is not a multiple of 2 then the memory is out of alignment and the program will not compile.

5.4.2 Common Data

A problem similar to the one above was obtained after adding another common statement to block data. Approximately fifty hours were spent in isolating the item causing the error. The reason for the lengthy delay was caused by the bad diagnostics of the software in that the only error that it would display was "WRONG PROGRAM". A call to Interdata revealed that they were aware of an error in their compiler and that a patch was needed. Computer Science personnel were able to apply the correction to the pertinent addresses in less than five minutes. The program was then recompiled sucessfully.

5.4.3 Job Control Language

The job control language of the INTERDATA 8/32 has not been developed fully. It is not possible to compile and execute a program in one step. In fact it requires the user

to reply to a query from the terminal five different times just to execute a task that has been previously established. This is a waste of time and presents more opportunities for user typing errors.

Chapter 6

TESTS

6.1 Introduction

This chapter will discuss some of the testing used during the development of the memory management systems. Three different tests that were used are still a part of this program to enable the executor to test storage of data at future times. No tests that were used during the development of GETBLK are shown as these modules were tested earlier with the READ and SCAN modules which use the system.

6.2 Test 1

Test 1 and the other test routines are shown at Appendix E. Test 1 is a simple test designed to check a call to the function IGTBLK, store some data into the cells of the array ISPACE, and then have the values printed to verify if the values were entered correctly.

6.3 Test 2

Test 2 is a routine designed to be implemented along with Test 1. It tests successive calls to IGTBLK. After printing the results of Test 2 the point and data fields can be verified. The variables used for memory management such as IFREE, NLIST and the elements in the IAL were checked with this test.

6.4 Test 3

Test 3 is a routine designed to check the garbage collector. By giving IFREE a value slightly smaller than the size of the array ISPACE and causing another call to IGTBLK for space all functional parts of the system could be checked. If blocks were freed for the test then the compaction routine was checked; otherwise, the overflow mechanism was checked.

6.4 Test 4

Test 4 is a routine designed to check all of the working parts of the main submodule IGTBLK. The test starts by using a read routine for obtaining data. The STORE module is then called to obtain space from IGTBLK on four different requests for blocks. After the data is stored the blocks are printed by the module OUTPUT. A block is freed by a call to the FREED module. The IAL is printed to check the repositioning of the elements on the IAL after the element is freed. IFREE is set to a large value, and another call for a block causes the garbage collector to be exercised.

6.5 Summary

All of the above tests proved to be useful during the testing. All tests were completed sucessfully with different data and separate runs of the program task. These tests should be retained and modified as needed by the executor. All tests should be taken out of the program prior

to final implementation

APPENDIX A

Knuth,Donald E. The Art of Computer Programming. 2d ed. Massachusetts: Addison-Wesley, 1975.

Shapiro,Linda. Data Structures.

Weissman,Clark. LISP 1.5 Primer. Belmont, California: Dickenson, 1967.

APPENDIX B

```
      ****************************************
THIS IS  A DRIVER ROUTINE  USED TO CHECK  AND IMPLEMENT
THE MODULES OF  THE INITIALIZATION PORTION OF  THE LISP
INTERPRETER.  THIS  WOULD  BE  REPLACED  BY  THE  MAIN
PROGRAM.
      ********************************
      INTEGER*2 IARG
      DOUBLE PRECISION FUNCT
      COMMON/TABLE/FUNCT(8,24),IARG(8,24)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      DIMENSION IBIT(80),IWORD(80)
      CALL SETUP
            ****************************
 SETUP IS  USED TO INITIALIZE  THE VARIABLES  IN COMMON
DATA. THIS  ROUTINE SHOULD  BE CHECKED  FOR THE  PROPER
SIZE OF ALL ELEMENTS BEFORE FINAL IMPLEMENTATION.
            ******************************
      LOCK=1
            ******************************
READNO IS A ROUTINE TO READ IN NUMBERS FROM THE SCANNER
OR EXECUTOR.
            ******************************
CALL STORE(IWORD,I,NUM)
          **********************************
STORE  IS A  ROUTINE  TO PUT  DATA  IN THE  ALLOCATABLE
PORTION OF MEMORY.
```

```
            ************************************

            ************************************

OUTPUT IS  A ROUTINE WHICH  CALLS DATA OUTPUT  FROM THE
STORAGE BY USING THE INITIAL ADDRESS LIST NUMBER.

            ************************************

      CALL TEST4
      DO 10 I=1,10
      WRITE(6,11) NLIST,IALIST(NLIST)
11    FORMAT(' ','IALIST(',I2,') IS EQUAL TO ',I3)
10    CONTINUE
99    STOP
      END
```

```
      SUBROUTINE SETUP
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      IPUSED=0
      IFREE=1
      IPSIZE=50
      ISIZE=1000
      NLIST=0
      ISLIST=20
        ****************************************
THE SIZE  OF IPTABL,ISPACE,IALIST SHOULD BE  MADE LARGE
ENOUGH FOR THE LISP PROGRAM  WITHIN THE RESTRICTIONS OF
THE PARTITION ALLOCATED. IF THE PROGRAM STOPS DUE TO AN
OVERFLOW OF MEMORY SIZE, THE  SIZE OF PARTITION FOR THE
PROGRAM WILL HAVE TO BE  ALLOCATED LARGER AND THE ABOVE
VARIABLES WILL HAVE TO BE RESET.
         ****************************************
      RETURN
      END
```

```
      SUBROUTINE STORE(IBIT,IWORD,NUM,IA)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      DIMENSION IWORD(80),IBIT(80)
      IA=IGTBLK(NUM)
      JDATA=IALIST(IA) +2 + NUM
      JUSE=IALIST(IA)+2
           **********************************
CHECK THE  DATA TO SEE IF  IT IS A POINTER.   A POINTER
MUST HAVE A "1" AS ITS LAST DIGIT.
        ****************************************
      DO 10 I=1,NUM
      M=JUSE+I
      N=JDATA+I
      ISPACE(M)=IBIT(I)
      ISPACE(N)=IWORD(I)
      WRITE(6,20) I,IBIT(I)
20    FORMAT(' ','IBIT(',I2,') IS = TO ',I1)
      IF(IBIT(I) .EQ. 1)GO TO 40
      WRITE(6,25) I,IWORD(I)
25    FORMAT(' ','IWORD(',I2,') IS = TO ',A4)
      GO TO 10
40    WRITE(6,45) I,IWORD(I)
45    FORMAT(' ','IWORD(',I2,') IS = TO ',I8)
10    CONTINUE
      RETURN
      END
```

```
      SUBROUTINE OUTPUT(IWORD,IBIT,IA,ISTART,ISTOP)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      DIMENSION IWORD(80),IBIT(80)
      IF(ISTART .NE. 0)GO TO 1
      IBLK=IALIST(IA)
      LENGTH=ISPACE(IBLK)
      NWORDS=(LENGTH-3)/2
      ISTRT=IBLK+3+NWORDS
      ISTP=ISTRT+NWORDS-1
      DO 101 I=1,NWORDS
      IF(IBIT(I) .EQ. 1)GO TO 25
101   CONTINUE
      GO TO 10
1     ISTRT=ISTART
      ISTP=ISTOP
10    WRITE(6,15) (ISPACE(M),M=ISTRT,ISTP)
15    FORMAT(' ','THE VALUE IS ',20A4)
25    WRITE(6,30) (ISPACE(M),M=ISTRT,ISTP)
30    FORMAT(' ','THE VALUE IS ',10I8)
      RETURN
      END
```

```
      **********************************
      ********************** FREED ****
      **********************************
      SUBROUTINE FREED(I)
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      **********************************
```

THIS ROUTINE FREES THE BLOCK IN MEMORY. IF IT IS NOT POINTED TO BY ANOTHER BLOCK IT CAN BE COLLECTED BY THE GARBAGE COLLECTOR OR PUT BACK ON THE AVAIL LIST IF LINKS ARE USED.

```
      **********************************
      IF(IALIST(I) .EQ. 0) RETURN
      IF(I .GT. NLIST)RETURN
      DO 10 M=I,NLIST
      IALIST(I)=IALIST(I+1)
10    CONTINUE
      NLIST=NLIST-1
      RETURN
      END
```

```
      SUBROUTINE READER(IWORD,NUM)
          ************************************
THIS ROUTINE IS DESIGNED TO READ IN SOURCE-CODE OR DATA
AND CODE FROM THE EXECUTOR.
          ************************************
      DIMENSION IWORD(80)
      DO 10 I=1,80
      READ(5,10,END=15)IWORD(I)
      NUM=I
10    FORMAT(A1)
15    RETURN
      END
```

```
      SUBROUTINE READNO(IWORD,NUM)
      DIMENSION IWORD(18)
          ************************************
THIS MODULE READS IN DIGITIZED CODE FROM THE EXECUTOR.
          ************************************
      READ(5,19,END=25)IWORD
      NUM=I
19    FORMAT(6(I8,2X))
25    RETURN
      END
```

APPENDIX C

```
**************************************************
************************** IGTBLK **************
```

IF SPACE IS NOT AVAILABLE AT THE TIME CALLED IT WILL CALL THE REGENERATE MODULE TO SEE IF SPACE CAN BE COLLECTED THAT IS NOT IN USE. IF SPACE IS NOT AVAILABLE AN ERROR MESSAGE WILL BE PRINTED.

DECLARATION OF VARIABLES

NUMTOT -- NUMBER OF WORDS IN THE BLOCK

IGTBLK -- THE FUNCTION WHICH RETURNS THE ADDRESS OF THE BLOCK

NWORDS -- NUMBER OF WORDS REQUESTED BY THE CALLING ROUTINE

IUSE -- WORD TO REPRESENT THE USE - EITHER DATA OR POINTER

```
*********************************************************
      FUNCTION IGTBLK(NWORDS)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
        ***************************************
```

FIND THE FIRST AVAILABLE SPACE ON THE IALIST. IF THE LIST IS FULL PRINT OUT AN OVERFLOW MESSAGE.

```
         ***************************************
      DO 1 I=1,ISLIST
      IF(IALIST(I) .EQ. 0) GO TO 6
1     CONTINUE
      WRITE(6,5)
5     FORMAT(' ','INITIAL ADDRESS LIST IS FULL-UNABLE
```

```
TO
     1 CONTINUE')
      IGTBLK=0
      GO TO 99
       ******************************************
THE TOTAL  NUMBER IN THE BLOCK  IS EQUAL TO THE  SUM OF
THE DATA WORDS, USEBITS,LENGTH,POINT, AND MARK FIELDS.
       ******************************************
6     NUMTOT=NWORDS*2+3
       ******************************************
IF THE  NEW BLOCK  EXCEEDS THE  AVAILABLE SPACE,  START
REGENERATION, OTHERWISE ALLOCATE THE SPACE.
       ******************************************
      IF((NUMTOT+IFREE) .LE. (ISIZE+1)) GO TO 15
      CALL REGEN1
       ******************************************
CHECK TO  SEE IF  SPACE WAS  MADE AVAILABLE  BY GARBAGE
COLLECTION  IF  AVAILABLE  THEN  ALLOCATE,  ELSE  PRINT
OVERFLOW MESSAGE.
       ******************************************
      IF((NUMTOT+IFREE) .LE. (ISIZE+1)) GO TO 15
      WRITE(6,10)
10    FORMAT(' ','SPACE OVERFLOW - UNABLE TO CONTINUE')
     ******************************************
IF SPACE OVERFLOW STOP THE RUN HERE AS TOO MANY MODULES
CAN CALL THIS  MODULE AND ANY FURTHER  PROCESSING WOULD
BE INEFFICIENT.
    ******************************************
      GO TO 99
```

```
      RETURN
     ***********************************************
ALLOCATE THE BLOCK AND INITIALIZE  THE HEADER 1. ASSIGN
A SPACE ON THE IALIST.  2. PUT THE ADDRESS OF THE BLOCK
IN  THE  IALIST.  3.  ASSIGN  THE  LENGTH  FIELD  AND
INITIALIZE THE POINT AND MARK FIELDS TO ZERO.
     ***********************************************
15    NLIST=I
      IGTBLK=NLIST
      IALIST(NLIST)=IFREE
      ISPACE(IFREE)=NUMTOT
      ISPACE(IFREE+1)=0
      ISPACE(IFREE+2)=0
        ***************************************
WORDS HAVE BEEN USED TO INDICATE WHETHER A WORD IS DATA
OR A  POINTER .  FOR EFFICIENCY  THESE WORDS  SHOULD BE
REPLACED  BY  BITS.   EACH  OF  THESE  BITS  SHOULD  BE
INITIALIZED TO ZERO.
     ***********************************************
      DO 20 ICOUNT=1,NWORDS
      IUSE=IFREE+ICOUNT+2
20    ISPACE(IUSE)=0
       *****************************************
SET IFREE TO POINT TO THE NEXT BLOCK.
       *****************************************
      IFREE=IFREE+NUMTOT
      RETURN
99    STOP
      END
```

```
 ****************************** COMPAC **************
```

THIS ROUTINE RELOCATES THE BLOCKS THAT ARE BEING USED IN THE ALLOCATABLE PORTION OF MEMORY. IT CALLS NO MODULES AND IT IS CALLED BY REGEN.

DECLARATION OF VARIABLES

IFREE1 -- LOCAL USE OF IFREE

LENGTH -- LENGTH OF THE BLOCK

IMARK -- THE MARK BIT OF THE BLOCK

IPOINT -- FIELD USED IN MARKING AND RELOCATION

NLIST -- NUMBER OF ELEMENTS ON THE INITIAL ADDRESS LIST

IADRES -- INITIAL ADDRESS OF THE BLOCK

ILEN -- VALUE PLACED IN LENGTH

IDATWD -- VARIABLE USED TO IDENTIFY THE SPECIFIC DATA WORD

```
    ************************************************
      SUBROUTINE COMPAC
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
        ***************************************
```

COMPUTE NEW BLOCK ADDRESSES AND START THE COMPACTION BY POINTING TO THE FIRST BLOCK.

```
     ***********************************************
      IFREE1=1
      I=1
     ***********************************************
```

DEFINE THE LENGTH, POINT, AND MARK FIELD.

```
     ***********************************************
1     LENGTH=ISPACE(I)
```

```
      IPOINT=ISPACE(I+1)
      IMARK=ISPACE(I+2)
      ********************************************
IF THE BLOCK IS MARKED IT  SHOULD BE RELOCATED IF THERE
ARE ANY UNUSED BLOCKS IN FRONT OF IT.
     *********************************************/
      IF(IMARK .NE. 1) GO TO 5
      ********************************************
THE BLOCK IS  IN USE. PUT THE NEW ADDRESS  OF THE BLOCK
IN THE POINT FIELD. CHANGE IFREE1  TO POINT TO THE NEXT
BLOCK.
     *********************************************
      ISPACE(I+1)=IFREE1
      IFREE1=IFREE1+LENGTH
   ************************************************
THE BLOCK WAS NOT IN USE SO IFREE1 REMAINS UNCHANGED AS
IT IS STILL  THE POSITION OF THE NEXT  USED BLOCK. MOVE
TO THE NEXT BLOCK TO SEE IF IT  IS IN USE IF IT IS LESS
THAN IFREE.
   ************************************************
5     I=I+LENGTH
      IF(I .LT. IFREE) GO TO 1
UPDATE BLOCK POINTERS REFERENCED BY THE IAL
      DO 20 I=1,NLIST
    ***********************************************
THE NEW  LOCATION OF THE BLOCK  IS IN THE  POINT FIELD.
PUT ITS LOCATION IN THE IALIST SPACE.
     *********************************************
      IADRES=IALIST(I)
```

```
      IPOINT=ISPACE(IADRES+1)
      IALIST(I)=IPOINT
20    CONTINUE
UPDATE THE BLOCK POINTERS WITHIN THE BLOCK DATA FIELDS
      I=1
30    ILEN=ISPACE(I)
      NWORDS=(ILEN-3)/2
      DO 40 J=I,NWORDS
          ************************************
DEFINE THE USE AND DATA FIELDS
          ************************************
      IUSE=ISPACE(J+3)
      IDATWD=J+3+NWORDS
      IDATA=ISPACE(IDATWD)
      ******************************************
CHECK  TO SEE  IF  THE DATA  FIELD  CONTAINS A  POINTER
(VALUE OF 1)
     ********************************************
      IF(IUSE .NE. 1) GO TO 40
      ******************************************
CHECK TO SEE IF THE POINTER IS NULL.
      ******************************************
      IF(IDATA .EQ. 0) GO TO 40
      NEWBLK=IDATA
      IPOINT=ISPACE(NEWBLK+1)
      ISPACE(IDATWD)=IPOINT
          ************************************
THE DATA FIELD CONTAINED EITHER  DATA OR A NULL POINTER
- MOVE TO THE NEXT FIELD.
```

```
      ******************************************
40    CONTINUE
       ******************************************
THE PREVIOUS  BLOCK HAS BEEN COMPLETELY  PROCESSED MOVE
THE POINTER TO THE NEXT BLOCK.
      ******************************************
      LENGTH=ISPACE(I)
      I=I+LENGTH
       ******************************************
IF  THE NEW  BLOCK RESIDES  IN  THE PORTION  PREVIOUSLY
ALLOCATED, PROCESS THE BLOCK.
      ******************************************
      IF(I .LT. IFREE) GO TO 30
RELOCATE THE BLOCKS
      IFREE1=1
      I=1
50    LENGTH=ISPACE(I)
      ******************************************
IF THE BLOCK  IS NOT MARKED - DO NOT  RELOCATE -MOVE TO
THE NEXT BLOCK.
      ******************************************
      IF(ISPACE(I+2) .NE. 1) GO TO 60
       ******************************************
THE BLOCK WAS MARKED - SET  THE MARK FIELD BACK TO ZERO
AND MOVE THE  IFREE1 POINTER TO THE END  OF THIS BLOCK.
MOVE  THE  BLOCK INDICATOR  TO  THE  NEXT BLOCK  TO  BE
PROCESSED.
      ******************************************
      ISPACE(I+2)=0
```

```
      DO 55 K=I,LENGTH
      ISPACE(IFREE1)=ISPACE(K)
      IFREE1=IFREE1+1
55    CONTINUE
60    I=I+LENGTH
        ****************************************
CONTINUE UNTIL ALL PREVIOUSLY ALLOCATED BLOCKS HAVE
BEEN MOVED OR FREED.
      ********************************************
      IF(I .LT. IFREE) GO TO 50
      ********************************************
UPDATE FREE. ALL BLOCKS THAT ARE IN USE HAVE BEEN MOVED
TO THE LOW END OF STORAGE. THE POINTER "FREE" CAN NOW
BE MOVED BACK TO THE FIRST AVAILABLE BLOCK TO ALLOW
FURTHER MEMORY ALLOCATION.
      ********************************************
UPDATE FREE
      IFREE=IFREE1
      RETURN
      END
```

```
C*****************************************************
C***************************** REGEN1 ***************
C THIS MODULE IS CALLED BY IGTBLK TO REGENERATE SPACE. IT
C CALLS  THE MARK  AND  COMPAC  ROUTINES. IT  STARTS  THE
C MARKING PROCESS BY  TAKING THE ELEMENTS OF  THE INITIAL
C ADDRESS  LIST AND  PASSING THEM  TO  THE MARK  ROUTINE.
C CONTROL IS RETURNED TO IGTBLK AFTER CALLING COMPAC
C*************************************
      SUBROUTINE REGEN1
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
C*************************************
C MARK ALL BLOCKS REFERENCED BY THE IAL
C*****************************************
      DO 10 I=1,NLIST
      IADRES=IALIST(I)
      IF(IADRES .EQ. 0) GO TO 10
      IBLK=IADRES
      CALL MARK(IBLK)
10    CONTINUE
C*****************************************
C ALL BLOCKS HAVE  BEEN MARKED. RELOCATE ALL  USED BLOCKS
C TO THE LOWER PORTION OF ALLOCATABLE MEMORY.
C*********************************************
      CALL COMPAC
      RETURN
      END
```

```
**************************************************
************************** MARK ****************
```

THIS MODULE IS USED TO MARK ALL BLOCKS THAT ARE IN USE IN THE ALLOCATABLE PORTION OF MEMORY. IT IS CALLED BY REGEN1, AND IT DOES NOT CALL ANY OTHER MODULE.

```
**************************************************
```

VARIABLES USED

IA -- THE CURRENT BLOCK BEING PROCESSED

IB -- A TEMPORARY BLOCK USED IN TRAVERSING

IC -- A TEMPORARY BLOCK USED IN TRAVERSING

IPOINT -- FIELD USED FOR RELOCATION AND TRAVERSING

IUSEBT -- VARIABLE TO SHOW THE USE - DATA OR POINTER

IDATWD -- SUBSCRIPT TO IDENTIFY A SPECIFIC A DATA WORD

IMARK -- THE MARK FIELD

IDATAB -- VARIABLE USED WITH THE DATA FIELD OF BLOCK IB

```
*****************************************
      SUBROUTINE MARK(N)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      IA=N
      IPOINT=ISPACE(IA+1)
      IMARK=ISPACE(IA+2)
      LENGTH=ISPACE(IA)
*****************************************
```

IF THE BLOCK IS NOT MARKED, THEN MARK IT AND SET THE POINT FIELD.

```
*****************************************
      NWORDS=(LENGTH-3)/2
      IF(ISPACE(IA+2) .EQ. 1) GO TO 30
```

```
      ISPACE(IA+1)=1
      ISPACE(IA+2)=1
      IPOINT=ISPACE(IA+1)
      IMARK=ISPACE(IA+2)
      LENGTH=ISPACE(IA)
        ****************************************
INITIALIZE THE TEMPORARY BLOCKS - B AND C.
        ****************************************
      IB=0
      IC=0
SEARCH
10    CONTINUE
       ****************************************
CHECK THE USE BIT TO SEE IF  THE WORD IN MEMORY IS DATA
OR A POINTER. DATA IS REPRESENTED BY A 0, AND A POINTER
IS REPRESENTED BY A 1.
       ****************************************
      IUSEBT=IA+2+IPOINT
      IUSE=ISPACE(IUSEBT)
      IF(IUSE .NE. 1) GO TO 20
       ****************************************
THE WORD IS A POINTER. NOW CHECK TO SEE IF IT IS NULL.
       ****************************************
      LENGTH=ISPACE(IA)
      NWORDS=(LENGTH-3)/2
      IDATWD=IA+2+NWORDS+IPOINT
      IDATA=ISPACE(IDATWD)
      IF(IDATA .EQ. 0) GO TO 20
      ****************************************
```

```
THE POINTER IS NOT NULL. CHECK THE BLOCK THAT IT POINTS
AT TO SEE IF IT IS MARKED.
      ******************************************
      IF(ISPACE(IDATA) .EQ. 0)GO TO 19
      IMARK=ISPACE(IDATA+2)
      IF(IMARK .NE. 0) GO TO 20
    ************************************************
DESCEND AS FAR AS POSSIBLE  I.E., TRAVERSE THE USE BITS
IN EACH BLOCK  UNTIL A POINTER IS IDENTIFIED  - THEN GO
TO THE BLOCK POINTED TO AND START THE TRAVERSE ON IT.
   ************************************************
ASSIGN THE  PRESENT ADDRESS  OF THE A  BLOCK TO  B THEN
CHANGE  A  TO POINT  TO  THE  BLOCK IDENTIFIED  BY  THE
POINTER IN THE DATA FIELD.
   ************************************************
DESCEND
      IB=IA
      IA=IDATA
    ************************************************
THE BLOCK IDENTIFIED  BY THE POINTER IS  NOW POINTED TO
BY "A" SO CHANGE THE POINTER TO POINT TO C.
    ************************************************
      IDATWD=IB+2+NWORDS+IPOINT
      ISPACE(IDATWD)=IC
      IDATAB=ISPACE(IDATWD)
      ******************************************
MARK THE BLOCK AND SET THE POINT FIELD.
     ******************************************
      IC=IB
```

```
      ISPACE(IA+2)=1
      IMARK=ISPACE(IA+2)
      ISPACE(IA+1)=1
      IPOINT=ISPACE(IA+1)
      GO TO 30
     ***********************************************
MOVE THE POINTER  TO CHECK THE NEXT USE  BIT.  ASCEND A
LEVEL IF POSSIBLE.
     ***********************************************
19    WRITE(6,18)IDATA
20    IPOINT=ISPACE(IA+1)
18     FORMAT('   ','BLOCK  REFERENCED  BY   ',I8,'  IS
INVALID.')
      IPOINT=IPOINT+1
      ISPACE(IA+1)=IPOINT
      LENGTH=ISPACE(IA)
      NWORDS=(LENGTH-3)/2
30    IF(IPOINT .LE. NWORDS)GO TO 10
ASCEND
      IF(IB .EQ. 0)GO TO 40
      IPOINT=ISPACE(IB+1)
      IDATWD=IB+2+NWORDS+IPOINT
     ***********************************************
CHANGE  C  TO POINT  TO  THE  BLOCK IDENTIFIED  BY  THE
B-BLOCK DATA FIELD. THIS IS A BLOCK THAT WAS PREVIOUSLY
MARKED ON THE DESCENT.
    ************************************************
      IDATAB=ISPACE(IDATWD)
      IC=IDATAB
```

```
    ************************************************
CHANGE THE POINTER  IN THE B-BLOCK DATA  FIELD TO POINT
TO  THE  CURRENT BLOCK  OF  A  WHICH  IS THE  BLOCK  IT
PREVIOUSLY POINTED AT.
  ***************************************************
      IDATAB=IA
     ***********************************************
MOVE BACK UP TO PREVIOUSLY CONSIDERED BLOCKS
    ************************************************
      IA=IB
      IB=IC
    ************************************************
INCREMENT THE  POINTER SO THAT  THE REMAINING  USE BITS
CAN BE CHECKED.
    ************************************************
      IPOINT=ISPACE(IA+1)
      IPOINT=IPOINT+1
      ISPACE(IA+1)=IPOINT
      LENGTH=ISPACE(IA)
      NWORDS=(LENGTH-3)/2
    ************************************************
IF  IPOINT IS  NOT  GREATER THAN  THE  LENGTH THEN  THE
ENTIRE BLOCK HAS NOT  BEEN CHECKED. CONTINUE PROCESSING
THE USE BITS AND DATA FIELDS.
    ************************************************
      IF(IPOINT .LE. NWORDS)GO TO 10
40    RETURN
      END
```

APPENDIX D

```
            SUBROUTINE GETBLK(NWORDS,IADDRS,LOCK)
      ******************************************
THIS  IS A  MEMORY  MANAGEMENT  ROUTINE WHICH  PROVIDES
SEQUENTIAL ALLOCATION OF MEMORY.
      DEFINITION OF VARIABLES
ISPACE -- USED FOR ALLOCATED BLOCKS OF SPACE
IPTABL -- POINTER TABLE FOR INDIRECT BLOCK POINTERS
IFREE -- POINTS TO THE FIRST CHARACTER IN ISPACE WHICH
         IS AVAILABLE FOR ALLOCATION.
ISIZE -- MAXIMUM SIZE OF MEMORY SPACE
IPSIZE -- MAXIMUM SIZE OF THE PTABLE
IPUSED -- POINTER TO THE LAST ALLOCATED INDIRECT BLOCK
          POINTER
        ****************************************
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      WRITE(6,5)IFREE,IPTABL(1)
5     FORMAT(' ','IFREE = ',I2,' IPTABL(1) = ',I3)
       ****************************************
ASSIGN THE FIRST UNUSED INDIRECT BLOCK POINTER TO I.
       ****************************************
      I=PUSED+1
      DO 10 I=1,IPSIZE
      IF(IPTABL(I) .EQ. 0) GO TO 20
10    CONTINUE
20    IF(I .NE. (IPSIZE+1)) GO TO 50
       ****************************************
ALL  INDIRECT BLOCK  POINTERS  ARE  IN USE.  REGENERATE
```

```
MEMORY.
      ****************************************
      CALL REGEN
      I=1
      ****************************************
FIND THE FIRST AVAILABLE INDIRECT BLOCK POINTER.
      ****************************************
      DO 30 K=1,IPSIZE
      IF(IPTABL(K) .EQ. 0) GO TO 50
30    CONTINUE
40    IF(I .NE. (IPSIZE+1)) GO TO 50
      ****************************************
THE PTABLE IS FULL. PRINT AN OVERFLOW MESSAGE.
      ****************************************
      WRITE(6,45)
45    FORMAT(' ','PTABLE IS FULL--UNABLE TO CONTINUE')
      RETURN
50    IPUSED=I
      ****************************************
ALLOCATE A BLOCK OF SPACE.
     ****************************************
      IF((IFREE+2+NWORDS) .LE. (ISIZE+1)) GO TO 60
     ****************************************
SPACE IS FULL. REGENERATE MEMORY.
      ****************************************
      CALL REGEN
      IF((IFREE+2+NWORDS) .LE. (ISIZE+1)) GO TO 60
     ****************************************
ATTEMPT TO REGENERATE WAS  UNSUCCESSFUL. PRINT OVERFLOW
```

```
MESSAGE.
     ********************************************
      WRITE(6,55)
55    FORMAT(' ','SPACE IS FULL--UNABLE TO CONTINUE')
      RETURN
       ******************************************
CREATE THE HEADER AND INDIRECT BLOCK POINTER. THE FIRST
IS USED  TO INDICATE THE LENGTH  OF THE BLOCK,  AND THE
SECOND IS USED TO INDICATE  THE PTABLE POINTER.  IF THE
LOCK IS SET TO ZERO A CONTIGUOUS BLOCK IS DESIRED WHERE
A HEADER IS ONLY CREATED AT THE FIRST OF THE BLOCK.
      ******************************************
60    IF(LOCK .EQ. 0) GO TO 70
      IF(IFLAG .EQ. 0)GO TO 80
      ISPACE(IFREE)=NWORDS
      ISPACE(IFREE+1)=I
      IPTABL(I)=IFREE
      IFREE=IFREE+2+NWORDS
      IADDRS=IFREE-NWORDS
      RETURN
     ********************************************
PASS  THE START  POINT OF  THE AVAILABLE  BLOCK TO  THE
CALLING ROUTINE.
    *********************************************
THE LOCK  IS ZERO  MEANING THAT  A CONTIGUOUS  BLOCK IS
WANTED. ASSIGN THE SPACE AND DO NOT CHANGE THE HEADER.
        ************************************
70    IF(IFLAG .EQ. 0)GO TO 80
        ************************************
```

```
THE FLAG WAS  NOT PREVIOUSLY SET, SO THIS  IS THE FIRST
REQUEST FOR STORAGE WITHIN THIS CONTIGUOUS BLOCK RECORD
THE START POINT  OF THE BLOCK OF THE BLOCK  AND START A
TALLY ON THE NUMBER OF WORDS.
         ************************************
      IFLAG=0
      ISTART=IFREE
      ISPACE(ISTART+1)=I
      IPTABL(I)=ISTART
      IFREE=IFREE+2+NWORDS
      IADDRS=IFREE-NWORDS
      ITOTAL=NWORDS
      RETURN
80    IF(LOCK .EQ. 1)GO TO 90
         ************************************
THE LOCK HAS  NOT CHANGED. ASSIGN MORE  STORAGE IN THIS
BLOCK AND UPDATE THE NUMBER OF WORDS.
         ************************************
      IFREE=IFREE+NWORDS
      IADDRS=IFREE-NWORDS
      ITOTAL=ITOTAL+NWORDS
      RETURN
90    IFLAG=1
         ************************************
THE BLOCK SHOULD NOW BE CLOSED. PUT THE NUMBER OF WORDS
IN THE HEADER.
         ************************************
      ITOTAL=ITOTAL+NWORDS
      IFREE=IFREE+NWORDS
```

```
      ISPACE(ISTART)=ITOTAL
      RETURN
      END
      SUBROUTINE REGEN
         ************************************
```

THIS ROUTINE CHECKS THE INDIRECT POINTER TABLE AND THE AVAILABLE MEMORY TO SEE IF SPACE EXISTS. IF EITHER OF THE TWO ARE FULL, A MESSAGE IS PRINTED STATING WHICH AREA IS FULL ALONG WITH THE FACT THAT THE PROGRAM CANNOT CONTINUE. IF SPACE EXISTS IN THE TWO THE GARBAGE COLLECTOR IS CALLED AND MEMORY IS THEN COMPACTED BY THIS ROUTINE. FOR MORE EFFICIENT OPERATION THE COMPACTION PORTION MAY BE REMOVED BY LINKING THE "GARBAGE CELLS". COMPACTION WAS CHOSEN INITIALLY FOR EASE IN MANIPULATING LARGE CONTIGUOUS BLOCKS.

```
      ********************************************
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      I=1 *********************************************
```

LET FREE1 POINT TO THE FIRST CELL IN THE ALLOCATABLE MEMORY. ASSIGN IFREE2 THE VALUE OF (IFREE - 1) FOR THE PURPOSE OF USING THIS VALUE IN DO-LOOPS.

```
       *****************************************
      IFREE1=1
      IFREE2=IFREE-1
    ***********************************************
```

DO WHILE I IS LESS THAN FREE DO 15 I=1,IFREE2

```
      *********************************************
```

USING THE HEADER INFO ASSIGN THE LENGTH AND INDIRECT

```
BLOCK POINTER TO THE VARIABLES - LEN,AND IBP.
      ******************************************
1     LEN=ISPACE(I)
      IBP=ISPACE(I+1)
      IF(LEN .GE. 0) GO TO 5
     ********************************************
BLOCK IS MARKED.  CHANGE ITS INDIRECT POINTER  TO POINT
TO ITS NEW LOCATION.
      ********************************************
      IPTABL(IBP)=IFREE1
      IFREE1=IFREE1+LEN+2
      ********************************************
BLOCK I IS NOT MARKED. SET ITS INDIRECT POINTER TO 0.
      ********************************************
      GO TO 15
5     IPTABL(IBP)=0
15    I=I+LEN+2
      IF(I .LE. IFREE2) GO TO 1
     ********************************************
MOVE THE BLOCKS  TO A NEW LOCATION IF  COMPACTION IS TO
BE USED.
        ************************************
      I=1
      IFREE1=1
16    LEN=ISPACE(I)
      IF(LEN .GE. 0) GO TO 30
       ***********************************
THE BLOCK  IS MARKED.  RELOCATE  THE BLOCK.  UNMARK THE
HEADER.
```

```
      ****************************************
      ISPACE(I)=-LEN
      LENGTH=ISPACE(I)+2
      DO 20 K=I,LENGTH
      ISPACE(IFREE1)=ISPACE(K)
      IFREE1=IFREE1+1
20    CONTINUE
30    I=I+LEN+2
      IF(I .LE. IFREE2) GO TO 16
      ****************************************
UPDATE FREE
      ****************************************
      IFREE=IFREE1
      RETURN
      END
```

```
      BLOCK DATA
          ************************************
THIS ROUTINE  INITIALIZES THE TABLES  USED BY  THE LISP
INTERPRETER.  THE  LENGTHS OF  THE VARIOUS  ARRAYS HAVE
BEEN SET  SMALL FOR DESIGN  AND TESTING, AND  THEY WILL
HAVE TO BE SET TO THE DESIRED SIZE FOR ACTUAL USE.
          ************************************
        DECLARATION OF VARIABLES
        FUNCT - VARIABLE FOR FUNCTIONS AND SYMBOLS
        IARG  - VARIABLE  FOR THE  NUMBER OF  ARGUMENTS
FOR THE FUNCT
        IPTABL - INDIRECT POINTER TABLE
        ISPACE - ALLOCATABLE SPACE CELLS IN MEMORY
        IPUSED - POINTS TO LAST USED CELL IN IPTABL
        IFREE - POINTS  TO THE FIRST AVAILABLE  CELL IN
MEMORY
        IPSIZE - SIZE OF THE IPTABL
        ISIZE - SIZE OF ALLOCATABLE MEMORY
           ************************************
      INTEGER*2 IARG
      DOUBLE PRECISION FUNCT
      COMMON/TABLE/FUNCT(8,24),IARG(8,24)
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      DATA
FUNCT/'(','GO','CAR','ADD1','EQUAL','ABSVAL','NUMBERP',
     1'UNSPECIA',')','EQ','CDR','ATOM','EVENP','DEFINE-
','REVERSE',
```

```
     2'DIFFEREN','$','OR','NOT','EVAL','FLOAT','ENTIER-
','EVALQUOT',
     3'EVALQUOT','+','<','SIN','FIXP','LABLE','FLOATP'-
,'UNTRACE',
     4'GREATERP','-','<','GET','NULL','MACRO','LENGTH'-
,'MAPLIST',
     5'INTERSEC','.','<','MAX','PRIN','PRIN1','MINUSP'-
,'REMPROP',
     6'LEFTSHIF',',','<','MIN','SQRT','PRINT','RETURN'-
,'<',
     7'QUOTIENT','/','<','SET','SUB1','QUOTE','APPEND'-
,'<',
     8'REMAINDE','
','<','PUT','CONS','RECIP','DELETE','<',
     9'FACTORIA','&','<','AND','CSET','TRACE','DIVIDE'-
,'<',
     1'FUNCTION',']','<','MAX','EXPT','ZEROP','EXPAND'-
,'<','<',
     2'<','<','<','MAPC','CSETQ','MAPCAR','<','<',
     3'<','<','<','PLUS','LESSP','MEMBER','<','<',
     4'<','<','<','PROG','NCONC','RPLACA','<','<',
     5'<','<','<','SETQ','PROG2','RPLACD','<','<',
     6'<','<','<','PROP','TIMES','GENSYM','<','<',
     7'<','<','<','READ','SUBST','TEREAD','<','<',
     8'<','<','<','COND','RATOM','TERPRI','<','<',
     93*'<','LIST','LOGOR','LOGAND',6*'<','PROGN','LOG-
XOR',7*'<',
     1'SELECT',7*'<','LAMBDA',18*'<'/
      DATA
```

```
IARG/24*0,1,2,4,21*0,4*1,5*2,3,4,4,12*0,9*1,7*2,3,0,4,-
4,4*0,
     113*1,7*2,3,0,4,4,6*1,9*2,4*0,3*4,2*0,4*1,2,2,18*-
0,1,7*2,2*1,14*0/
      DATA
IPTABL/50*0/,ISPACE/2000*0/,IPUSED/0/,IFREE/1/,IPSIZE/-
50/,
     1ISIZE/200/
      DATA IALIST/20*0/,NLIST/0/,ISLIST/20/
      END
```

APPENDIX E

```
      SUBROUTINE TEST1
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE,ISIZE
           **********************************
```

THIS MODULE WAS USED AS A TEST TO SEE THAT DATA WAS BEING STORED IN THE USE AND DATA BITS. IT SHOULD BE REMOVED BEFORE ACTUAL IMPLEMENTATION.

```
           **********************************
      I=IGTBLK(70)
      ISPACE(5)=1
      ISPACE(75)=144
      ISPACE(7)=1
      ISPACE(77)=1
      RETURN
      END
```

```
      SUBROUTINE TEST2
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE
         ************************************
```

THIS MODULE WAS USED AS A TEST TO SEE IF SUCCESSIVE CALLS FOR STORAGE WERE PROCESSED CORRECTLY. IT ALSO CHECKED THE POINTER AND DATA FIELDS TO SEE THAT DATA WAS CORRECTLY INSERTED. THIS MODULE SHOULD BE REMOVED BEFORE IMPLEMENTATION

```
          ************************************
      J=IGTBLK(10)
      ISPACE(147)=1
      ISPACE(157)=167
      ISPACE(152)=1
      ISPACE(162)=144
      K=IGTBLK(10)
      ISPACE(174)=1
      ISPACE(184)=1
      RETURN
      END
```

```
      SUBROUTINE TEST3
      COMMON/SPACE/IPTABL(50),ISPACE(2000)
     1,IPUSED,IFREE,IPSIZE
      L=IGTBLK(20)
          ************************************
THIS MODULE  WAS USED TO  CHECK THE  GARBAGE COLLECTOR.
THIS MODULE  SHOULD BE  REMOVED BEFORE  IMPLEMENTATION.
THIS  REQUEST  SHOULD  OVERFLOW MEMORY  AND  REQUIRE  A
GARBAGE COLLECTION.
          ************************************
      RETURN
      END
```

```
      SUBROUTINE TEST4
      COMMON/SPACE/IPTABL(50),ISPACE(2000),
     1IPUSED,IFREE,IPSIZE,ISIZE
      COMMON/IAL/IALIST(20),NLIST,ISLIST
      DIMENSION IWORD(80),IBIT(80)
      DO 50 K=1,4
      NUM=4
      DO 40 I=1,NUM
      READ(5,10,END=15)IBIT(I)
10    FORMAT(I1)
      WRITE(6,12)I,IBIT(I)
12    FORMAT(' ','IBIT(',I2,') IS = TO ',I1)
      IF(IBIT(I) .EQ. 1)GO TO 21
11    READ(5,20,END=15)IWORD(I)
      WRITE(6,24)I,IWORD(I)
24    FORMAT(' ','IWORD(',I2,') IS EQUAL TO ',A4)
25    FORMAT(' ','IWORD(',I2,') IS EQUAL TO ',I8)
20    FORMAT(A4)
      GO TO 40
21    READ(5,30,END=15)IWORD(I)
      WRITE(6,25)I,IWORD(I)
30    FORMAT(I8)
40    CONTINUE
15    CALL STORE(IBIT,IWORD,NUM,IA)
      ISTART=0
      ISTOP=0
      CALL OUTPUT(IWORD,IBIT,IA,ISTART,ISTOP)
50    CONTINUE
      CALL FREED(3)
```

```
      DO 70 I=1,NLIST
      IA=I
      CALL OUTPUT(IWORD,IBIT,IA,ISTART,ISTOP)
70    CONTINUE
      IFREE=1999
      DO 80 I=1,NUM
      READ(5,10,END=95)IBIT(I)
      IB=IBIT(I)
      IF(IBIT(I) .EQ. 1)GO TO 120
      READ(5,20,END=95)IWORD(I)
      WRITE(6,24) I,IWORD(I)
      GO TO 80
120   READ(5,30,END=95)IWORD(I)
      WRITE(6,25) I,IWORD(I)
80    CONTINUE
95    CALL STORE(IBIT,IWORD,NUM,IA)
      CALL OUTPUT(IWORD,IBIT,IA,ISTART,ISTOP)
      RETURN
      END
```

THE DESIGN AND IMPLEMENTATION OF MEMORY MANAGEMENT AND INITIALIZATION MODULES FOR A LISP INTERPRETER

by

LEE ROY WHITLEY

B.S., Texas Technological University, 1961

ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1977

A LISP Interpreter is a program which interprets strings. This paper describes the design and implementation of two parts of LISP Interpreter. The parts included are the memory management and output.

All parts are written in the FORTRAN V programming language and have been tested on the INTERDATA 8/32 computer. The memory management module consists of two different systems. One system provides for large blocks of contiguous storage and uses only small headers for management. The second system provides for recursion with varying size cells. Both systems allocate space dynamically. The output module prints the user's program and result. The code for all modules completed and discussed as a part of this report are included in Appendices to the report.

The output and memory management modules are to be combined with input, scan, interpreter, and executor modules to form an efficient high level language interpreter for use on minicomputers. These other modules are the subjects of other reports by different authors. To insure compatability the modules in this report are designed for maximum portability with minimum adaption.