IMPLEMENTATION OF A CLASS MODEL
INFORMATION STRUCTURING SUBSYSTEM

by

Gary Edward Vining

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1969

Signature of Author . . . .
Department of Electrical Engineering

Certified by . . . .
Thesis Supervisor

Accepted by . . . .
Chairman, Departmental Undergraduate Thesis Committee

ABSTRACT

This paper reports on the flowcharting and coding of a part of a model computing system. The particular portion of the system chosen for the project was the top half of the storage management subsystem. Coding is done in a high-level programming language which resembles PL-1. Both the flowcharts and the complete programs are included in the appendices.

ACKNOWLEDGMENT

A considerable debt of gratitude for understanding and advice is owed to my thesis advisor, Prof. J. H. Saltzer. Explaining and answering an endless stream of questions, often two or three times each, requires a remarkable gift of patience, and I'm extremely fortunate that he had it.

Thanks are also owed to Mike Schroeder and Dave Clark for help with their manual. Despite his being bludgeoned by the same tedious questions that Prof. Saltzer was made to endure, Mike was always available and ready to supply answers.

TABLE OF CONTENTS

INTRODUCTION

The curious goal of this project is to produce system programs for a computing system that does not physically exist, in a language whose compiler has not been written. The system in question is a model computing system, which is used as a pedagogical tool to help students take a large step along the way toward understanding real computing systems.

In the study of such an idealized or model computing system, one is tempted to ask the question, "Why consider a model system that neglects problems which must be dealt with in real life? Why not investigate a real, existing system so that one will be aware of the associated problems from the start?" There are many answers to these questions, two of which have some bearing on this paper.

First of all, the model computing system can be presented, and its implications studied, to a depth not possible with a real system. The modern computing system is so complex that one cannot hope to underststnd its operation at the end of one semester's or even one year's study. The undergraduate who wants to get a fairly subtle understanding of how a computing system works, but who cannot afford to spend years in the study, is clearly in need of the model system.

Secondly, the model is useful for teaching concepts as opposed to teaching facts. General principles are easily isolated in the model system or are easily applied to it. We are familiar with details like minor convenience features or those routines which do not logically belong to the system, but which a real system must have to get off the ground. Both of these problems, which only cloud the important issues, are reduced or eliminated in the model.

Prompted by the above discussion and the title of this paper, another question might be generated: "Why implement a model computing system? Won't the process of adding flowcharts and code to the written description of the system contaminate the model with the same kind of detail we sought to eliminate by creating the model?" If these questions cannot be answered, this thesis is of little value. Therefore the next four paragraphs present what seem to be reasonable answers.

The addition of flowcharts to the model system is easily justified. Rather than cloud the issues, flowcharts add a logical organization to the written description which can only make it more understandable. The value of generating code for a model system is less obvious but not less in magnitude.

Coding the model system shows students that, while it is idealized or unreal in terms of practicality, the model is real in the sense that it can be simulated. If the student has the time or the interest, he can actually look at the programs to see how long they are, what algorithms are used, etc. As detailed a study of the model as desired can be obtained if the code is included in its presentation.

A second purpose of coding is to explore the model and its consequences fully -- to prove that it actually works. Without coding one cannot be sure that the formulation of the model does not have inconspicuous logical flaws which might make the model unfeasible. Thus the coding is not only helpful for detailed study of the model, but necessary for confidence in its validity.

Finally, the addition of code to the model serves a very important pedagogical function. Once the students have been presented with the programs, they may be asked to make modifications in them in order to implement small changes in the model. Tasks of this nature would test the students for a thorough understanding of the subsystem and the language.

STATEMENT OF THE PROBLEM

The principal goal of this project was to produce the flowcharts and actual code for the top half of the Storage Management Subsystem of the CLICS model computing system. The system programs were written in CIMPL, a high-level programming language developed for use with the model system.

This process was to generate two useful outcomes. First, it would have completed a large step toward the actual implementation of the total system. If the system was never implemented, the project would still facilitate a more detailed study of a part of the model than would otherwise be possible. Secondly, it provided the programmer with the opportunity to actually experience the problems associated with coding a part of the system.

A related goal was the examination of the subsystem design for feasibility. The model had not been tested or considered in the detail associated with flowcharting and coding. Scrutiny of the subsystem at this level would uncover any problems that might have been overlooked in the higher-level written specifications. The clarity of the written specifications themselves was to be considered, and appropriate changes in the text would be suggested.

The third topic for consideration was the usefulness of the model at a detailed level as a pedagogical tool. This would involve examining the flowcharts produced to see whether or not they followed directly from the written text. It also would involve deciding whether or not the programs themselves were understandable and clear to the student, given that he had previously studied both the written description of the subsystem and the associated flowcharts.

A fourth and final outcome of the project was to be the determination of whether or not the coding of the subsystem was straightforward in the CIMPL language as it was specified. The project would reveal the adequacy of the language in writing the system programs of the model system. If the programs were long or clumsy due to the lack of certain features in the language, the coding process would make this obvious. Changes in the language might, therefore, be suggested. Inconsistencies in the language might also be revealed, and the appropriate revisions made.

INTRODUCTION TO THE SYSTEM AND THE LANGUAGE

THE CLICS SYSTEM

The CLICS system, a small portion of which is the concern of this thesis, is a model computing system used to teach students at M.I.T. about complex information systems. It is a rather completely specified, but simplified version of MULTICS, a community-utility type of computing system being developed by Project MAC at M.I.T. The specification of the model system (see reference 1) consists of a description of the hardware, a description of the operating system, and a description of the language used to write the system programs. This paper relies quite heavily on that written specification, and derives most of the information it presents about CLICS from those specifications.

A list of the simplifications and changes in MULTICS that the model includes would be long and confusing, but some obvious examples might be helpful. While the MULTICS system uses core, drum, and disc memory with hardware-controlled paging in and out of core, the CLICS system simply uses a very large core

(10^9 words), again with hardware paging. In the software, the directories of the CLICS file system contain much less information than those of MULTICS and are fixed-length rather than variable-length.

A familiarization with the CLICS operating system is useful in understanding this project, so a description of its subsystems is in order. The Storage Management Subsystem, the subsystem with which this thesis is concerned, is one of the major subsystems. It controls the allocation of physical storage and provides the various protection mechanisms needed by the system itself and its users. The Processor Management Subsystem, another major subsystem, has several functions. It performs the multiplexing of processors among processes, provides intercommunication among processes, serves as an interface between processes and the hardware fault-interrupt mechanism, and assigns and releases processes to and from system users. The Command Subsystem provides an interface between the user and the system itself. System loading and initialization is performed by the Operations Subsystem. Input/output between users seated at typewriter terminals and processes in the system is made possible by the Input/output Subsystem. Finally, the Clock Subsystem constitutes the software control of the timing operations needed for determining charges to users.

THE CIMPL LANGUAGE

In order that one may appreciate the programs generated by this project, a few comments about the language used in writing them are necessary. CIMPL is the high-level programming language of the CLICS system. It serves as the language in which most of the system programs are written, and it is also available to CLICS users, although this is not its primary purpose. This language is a simplification of PL-1 (most of the system programs of MULTICS are written in a subset of PL-1) with special built-in functions added. (A nearly complete specification of the language can be found in Section B of reference 1.)

The simplifications in the PL-1 language were introduced to make both the language and its compiler easier to understand, and to make the compiler easier to write. Although too numerous to list, a few examples of these simplifications might be helpful in getting a feel for the language. One is the lack of automatic data conversion in CIMPL, a convenient feature of complete PL-1. Another example is the inability to equate or assign structures as in PL-1. Finally, complex arithmetic operations like exponentiation are not available in CIMPL.

Special functions in CIMPL serve two purposes. First, they make up, to some extent, for CIMPL's lack of certain powerful features. Data conversion functions belong to this category. A second use of CIMPL special functions is to provide the necessary tools for system programming. The lock function which can be used to lock a directory, thereby preventing other procedures from altering it while it is being referenced, or the referencing of program-accessible registers like the interval timer register both provide necessary tools to the system programmer.

THE STORAGE MANAGEMENT SUBSYSTEM

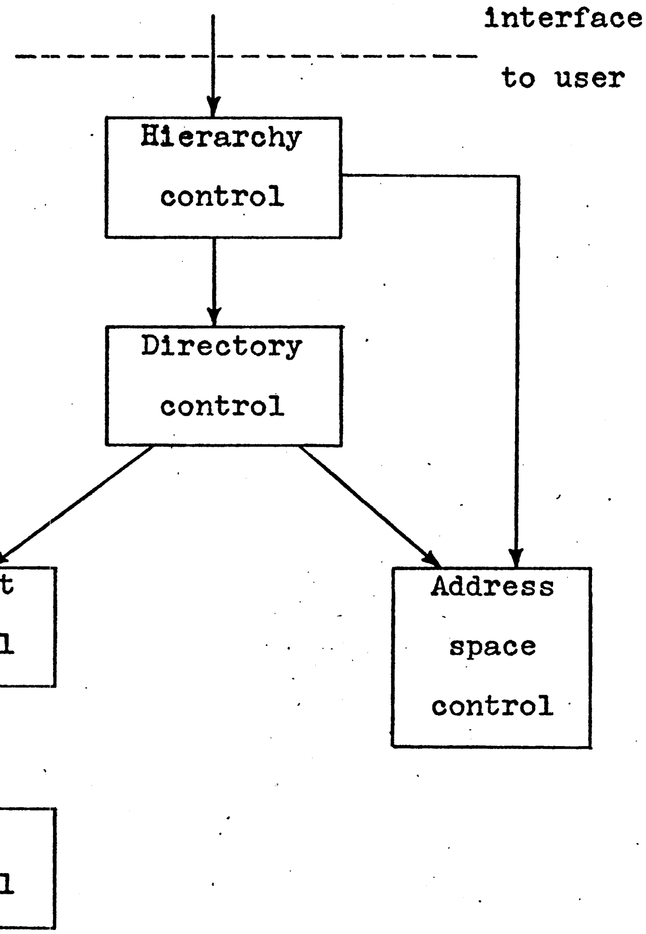
Because this project is specifically concerned with a part of the Storage Management Subsystem, a more detailed description of that subsystem, especially the portion being programmed, is required for a reasonable understanding of the project. The subsystem provides a structure in which the system programs, system data bases, user programs, and user data are all stored. The list of unused memory blocks is the only piece of information that can exist in the system without being a part of this structure. All information is dealt with in blocks called segments (which can be either the procedure or data type). Each user (including the system) can create, name, manipulate, reference, and destroy his own segments and can share these segments with other users in a controlled way.

The Storage Management Subsystem is divided into five modules, as shown in the diagram on page 15: memory control, segment control, address space control, directory control, and hierarchy control. Memory control provides an interface to the hardware for obtaining or releasing blocks of memory. It can create, change the length of, and destroy segments.

Segment control keeps a table of all segments in the system and calls memory control to perform physical manipulation of the segments. The table entry for each segment contains a unique identifier for that segment, an access control list (a list of processes which may reference the segment and their associated access mode indicators, which specify read, write, or call permission), and a list of the processes currently using the segment so that they may be informed if that segment is altered or deleted.

Address space control keeps a list of the segments in the address space of each process. It assigns the segments numbers within each process so that a process may refer to a segment within its address space by that number. This module, when called, can supply a (segment number, word number) pointer to a segment, retrieve an access mode indicator from segment control, change the address space of a process, or inform other processes that a segment has been altered.

Directory control and hierarchy control are the



Storage Management Subsystem[1]

[1]David Clark and Michael Schroeder, CLICS System Specification Notebook (Preliminary version), (unpublished, 1969), Section D.0.00, p.2.

two modules which have been programmed in this project. Directory control organizes all the segments of the system into directory segments. Each directory segment contains its own segment identifier, a list of its branches, the name and segment identifier of each branch, an indication of whether that branch is itself a directory segment or just an ordinary segment, and a directory control list for each branch that is a directory. The directory control list contains the name of each user that may manipulate that directory, and a directory access indicator which specifies what kind of manipulation he is allowed to perform. Directory control can extract information from a directory or make changes in the directory. It can add or delete a branch or alter a directory control list.

The interface between the subsystem and the outside world (the CLICS system and its users) is provided by hierarchy control. It is also the only module in the subsystem that is aware of the hierarchical structure implicit in the construction of directory segments. The entire collection of directories and non-directory segments is linked together to form a single large tree. A segment is specified to the module by the path which must be followed through the tree to reach the segment. Requests to obtain pointers to segments or to manipulate them or to check access to them are received from the outside world. Hierarchy control validates the requests against appropriate directory control list entries, completes tree names if necessary, and calls on directory control or address space control to perform the required manipulation or to retrieve the appropriate information.

One last element of detail is needed. A description of the procedure blocks within directory control and hierarchy control must be given so that the functions described by the flow charts and programs of this project will be familiar. The interconnection of these procedure blocks is shown in the diagram on page 24.

The directory manipulator is the only procedure block in directory control. Calls from the hierarchy access validator specify directory manipulations to be performed. The indicated change within a specified directory is made, and address space control is called to perform further manipulations at lower levels. Calls from the segment locator are to obtain pointers to branches within specific directory segments. The directory manipulator first finds the branch's segment identifier in the directory and then calls segment control for a pointer to the segment whose identifier has been found.

Finding a pointer to a directory or non-directory segment, given its tree name, is the job of the segment locator. It traces the path specified by the tree name through the hierarchy of directories (using the directory manipulator) until the desired segment is found.

The hierarchy access validator receives calls from the user interface manager to manipulate directories, given their tree names. First the tree name is converted to a pointer by calling the segment locator. Then the request is validated against the proper directory control list. If the validation succeeds, the directory manipulator is called to complete the processing of the request.

Pointers can also be obtained for segments whose tree names are only partially specified. The search director accomplishes this task using the search rules, a per-process data base that lists the possible directory paths one might follow to find the beginning of the partially specified path. This procedure calls the segment locator to try possible paths and to get the required pointer if the correct path can be found.

The only entry points in the Storage Management Subsystem which may be called by CLICS users are found in the user interface manager. The module must validate the user-provided arguments and convert them to a form acceptable within the subsystem. It must also complete tree names in calls intended for the hierarchy access validator by using the process working directory found in the search rules. The calls are then passed on to the intended entry points in other subsystem procedure blocks.

DESCRIPTION OF THE WORK

The actual work done on this project consisted of flowcharting and coding the two subsystem modules. A detailed study of both the subsystem and the language was naturally involved in this process. Both flowcharting and coding are mechanical processes, and details like the kind of programming tricks used, etc. will add little to this discussion. However, a few general comments will give the reader an overall picture of what was happening.

One useful comment involves the order in which the work was done. Both flowcharting and coding were started at the lowest level, the directory manipulator, and proceeded upward to the highest level, the user interface manager. This order was chosen for two reasons. First, it gives the programmer immediate contact with the procedures at lower levels in the subsystem. Thus any interfacing problems would be immediately discovered and dealt with. Secondly, this order gives the programmer the clearest idea of how his programs build on one another and fit together.

Another aspect of the order is the fact that the code for each procedure block was written directly after the flowcharts for that block were constructed. This method was used because flowcharts and code are somewhat interdependent. A given flowchart often generated blocks of excess or duplicate code, and appropriate restructuring of such flowcharts was necessary. This order was also used so that conceptual flaws discoverable only by the coding process would appear before the flowcharts of higher levels were written. A good deal of rewriting might be saved this way.

A note about debugging is also appropriate, since there was very little. At the time this project was finished, a compiler for the CIMPL language had not been written. Therefore, no machine debugging was possible. The language itself had not been specified very completely. And finally, the processor management subsystem, which these programs must call on several occasions, had not been specified in any reasonable detail. The programs were examined as closely as possible by hand, but machine debugging would still be necessary before they could become operational.

CONCLUSIONS

The project was successful in that it completed its major goal, to flowchart and program the hierarchy control and directory control modules of the Storage Management Subsystem. The associated analysis of the subsystem design and the language proved them both to be quite successful.

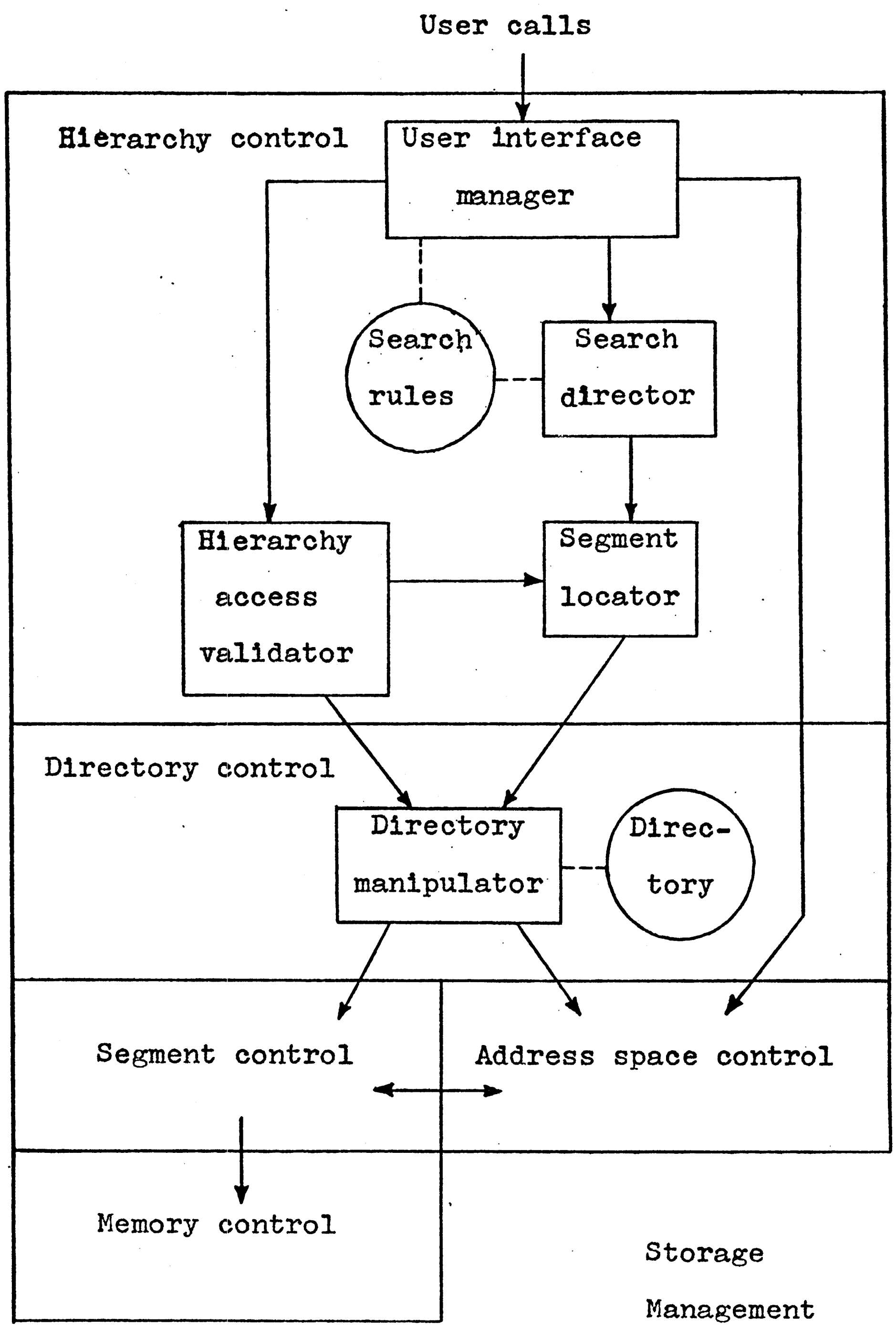
First of all, it has been shown that the subsystem design can be implemented (coded). No major conceptual errors in the design were discovered, and only a few minor programming problems were encountered. The written description of the subsystem is cloudy on some minor points but is generally clear and easy to understand.

Secondly, both the flowcharts and the programs are fairly straightforward and should present no difficulty to the student wishing to study them. The flowcharts follow directly from the written description of the system. Although the programs seem quite lengthy, they are reasonably concise, given the job they have to perform. They should be transparent for anyone who knows the language and has read the written description of the subsystem and looked at the flowcharts. (The complete flowcharts are given in Appendix I, and the corresponding code resides in Appendex II.)

Finally, the CIMPL language has survived the test of extensive use surprisingly well. The programs could all be written transparently without any changes or additions to the language. Some automatic data conversion would have been helpful, but not necessary. String manipulation was often a tedious process, and Post-system-like capabilities would be handy. One thing that did get gruesome was copying structures one piece of information at a time. Often programming tricks were used to make the process less cumbersome. But as a whole, the language in its present form is usable, and the resulting simplifications in the compiler may make its inadequacies bearable.

APPENDIX I

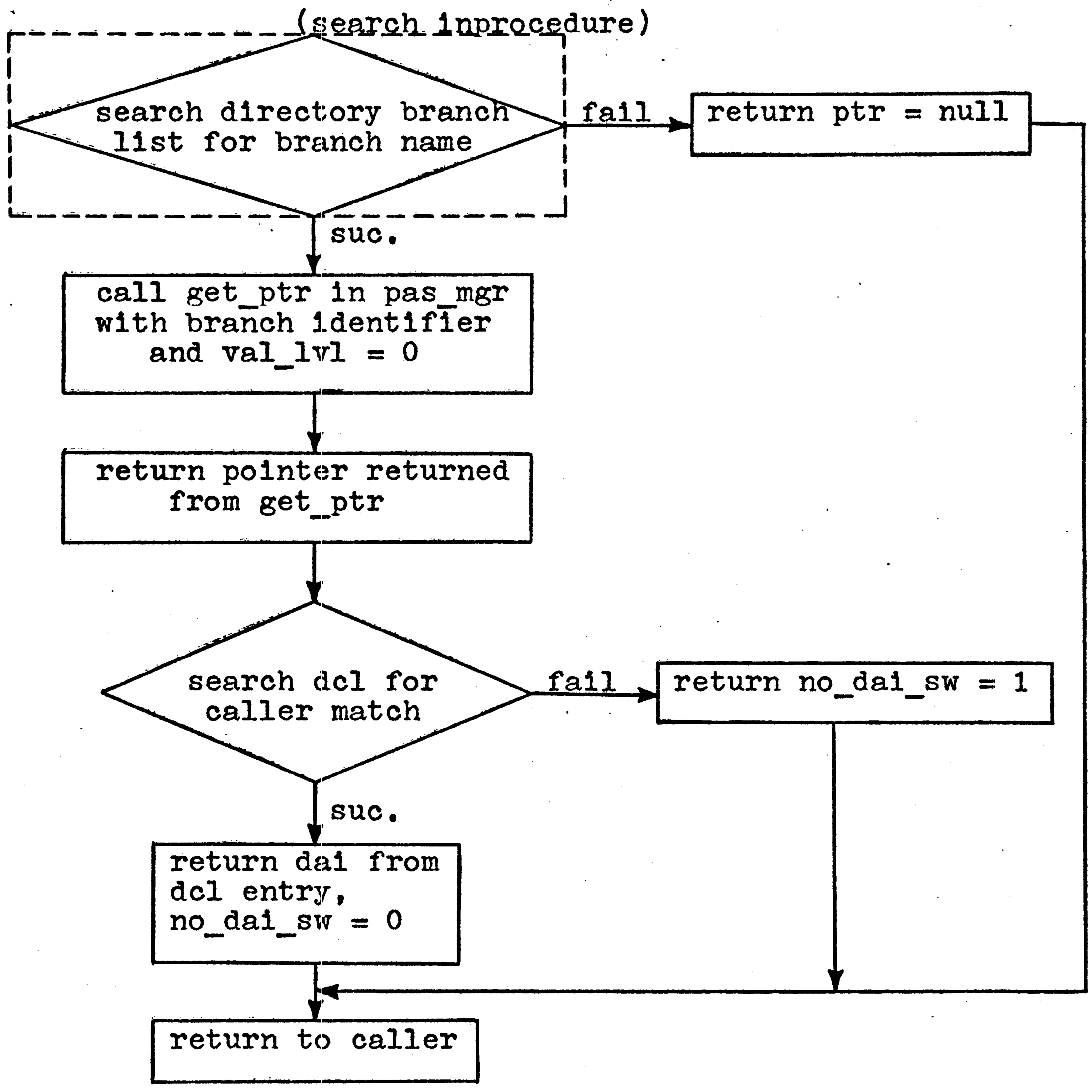
Overall flowchart of the
Storage Management Subsystem
and individual flowcharts
for the entries of the
procedure segments within
the directory control and
hierarchy control modules.



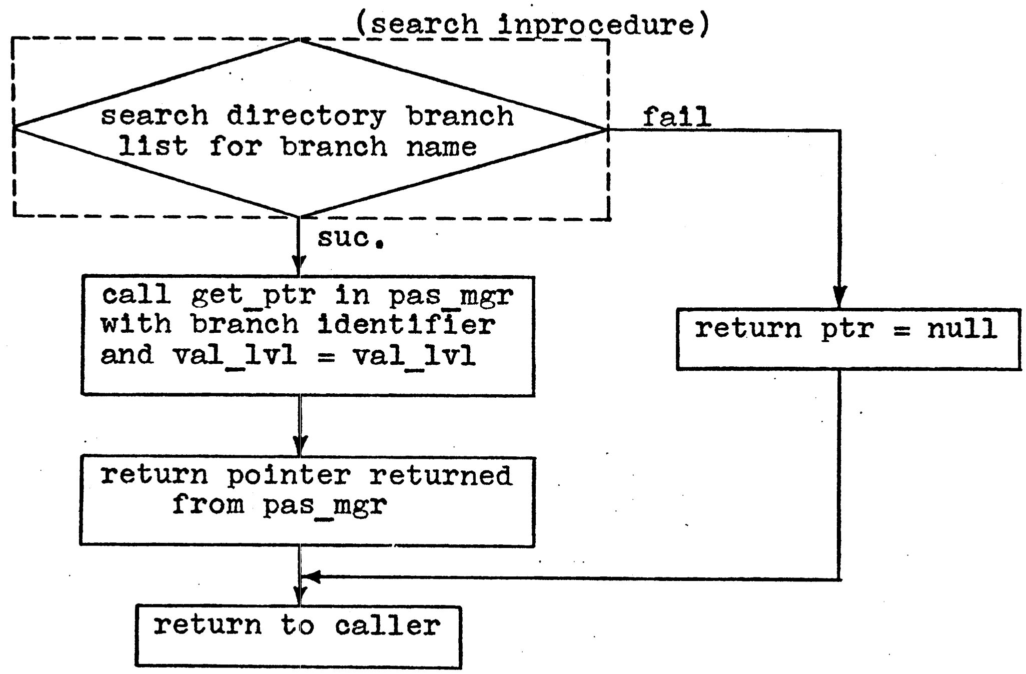
Storage Management Subsystem[2]

[2]*Ibid.*, Section D.0.00, p.9.

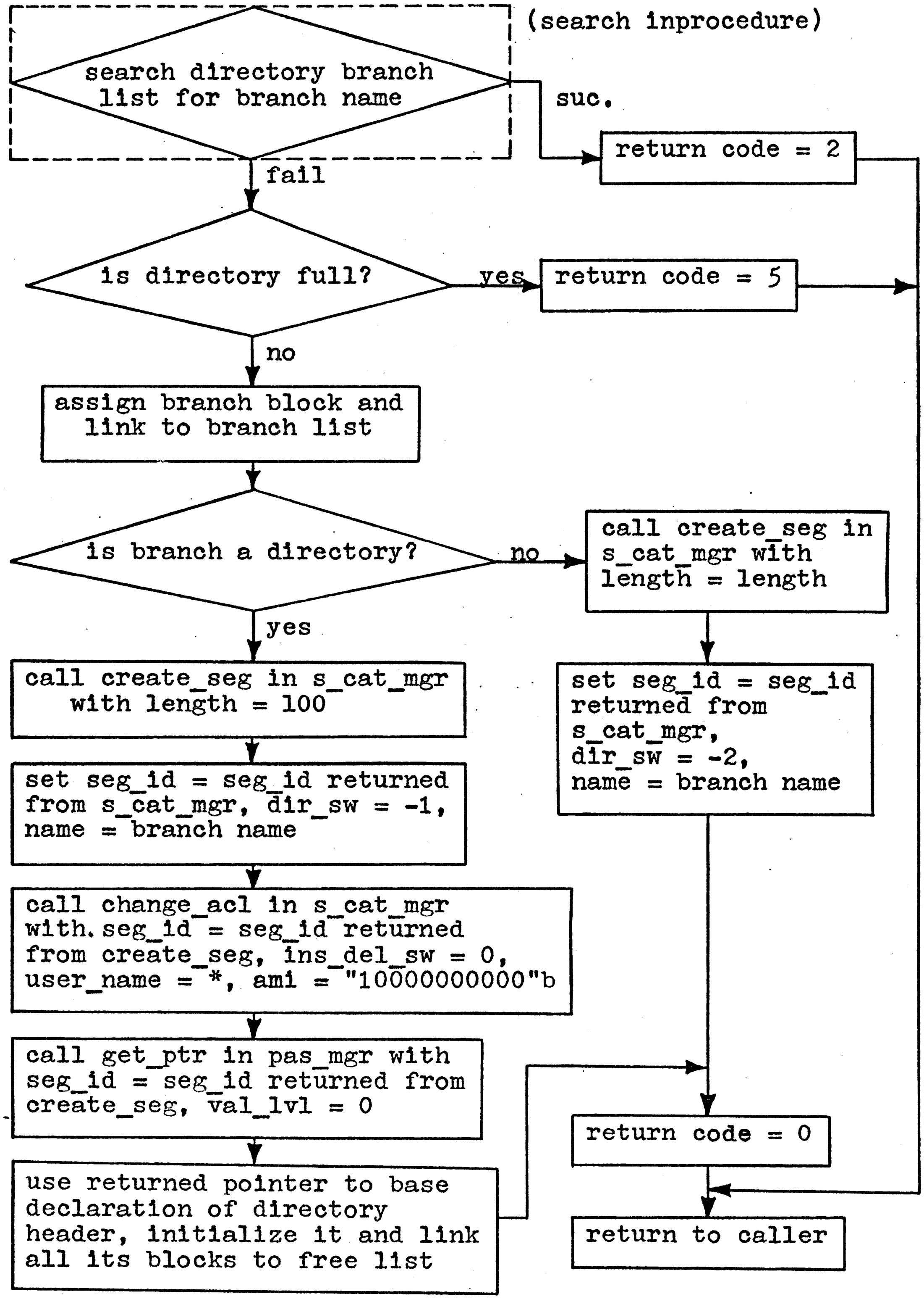
Directory manipulator: get_dir_ptr entry



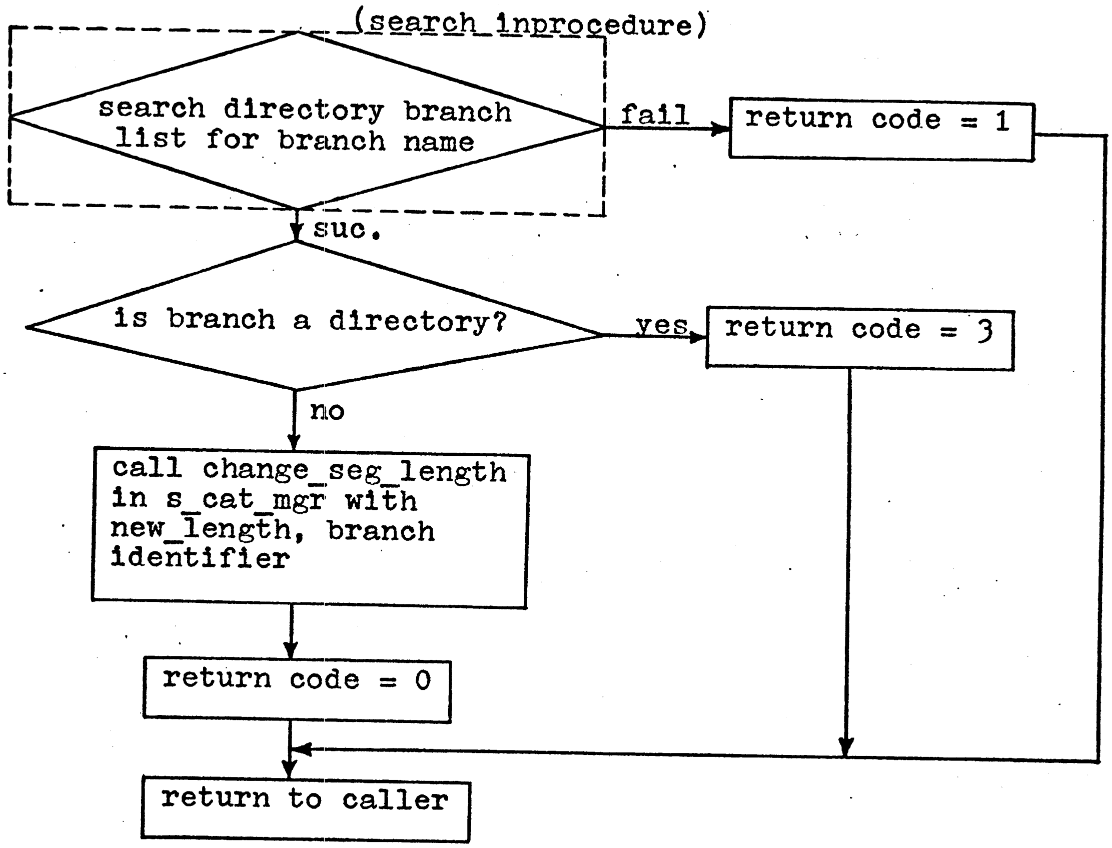
Directory manipulator: get_nondir_ptr entry



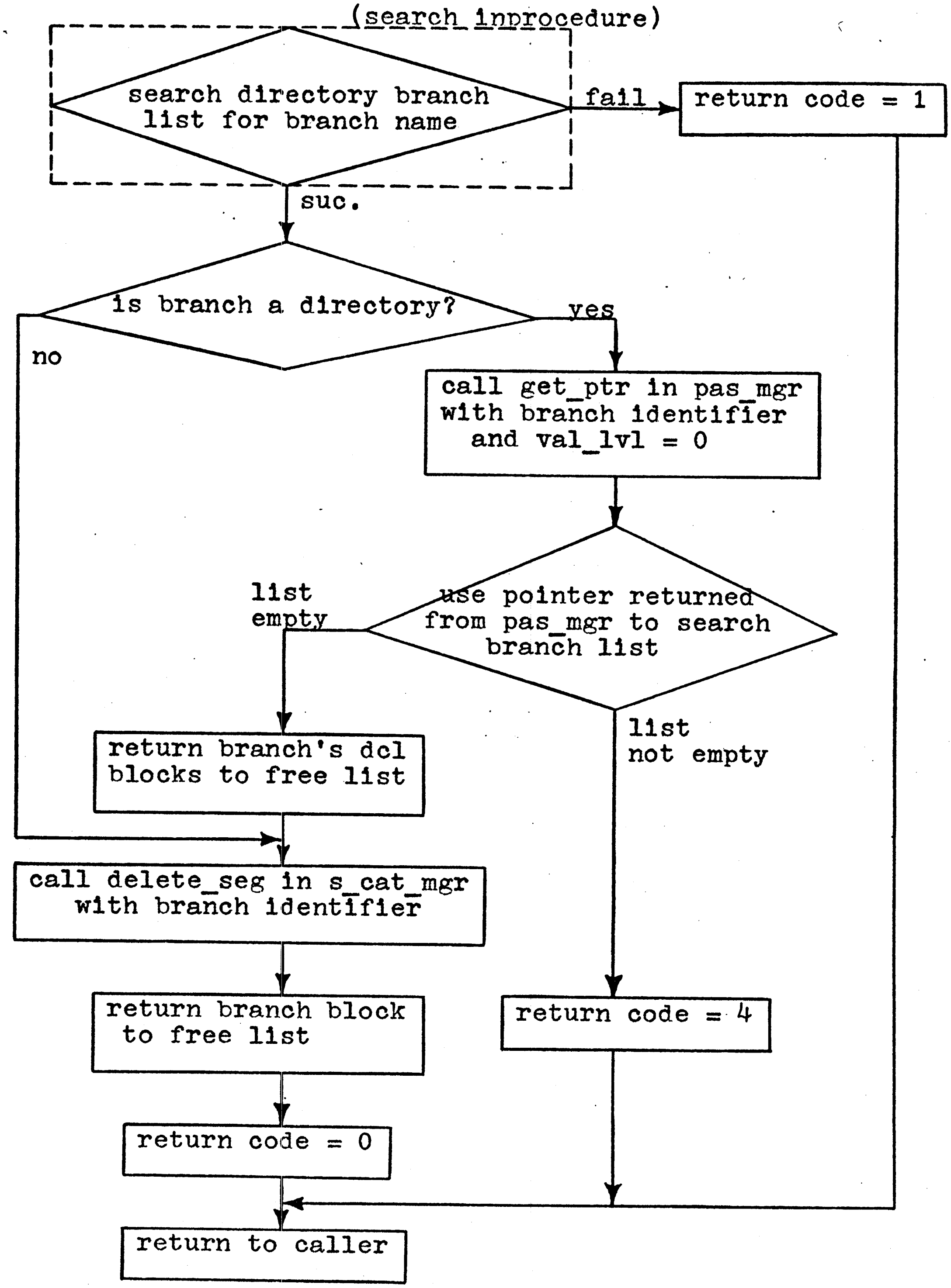
Directory manipulator: create_seg entry



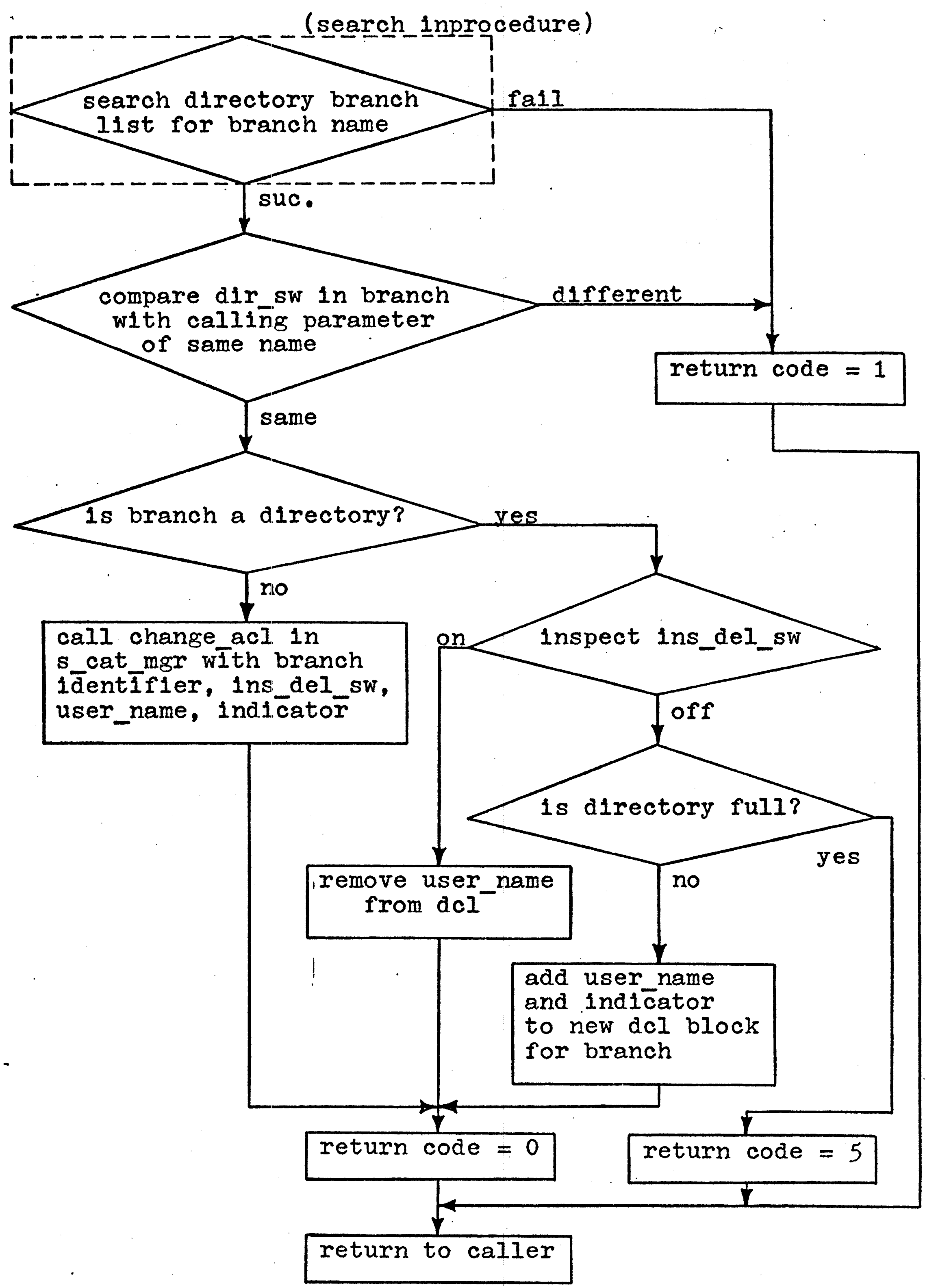
Directory manipulator: change_seg_length entry



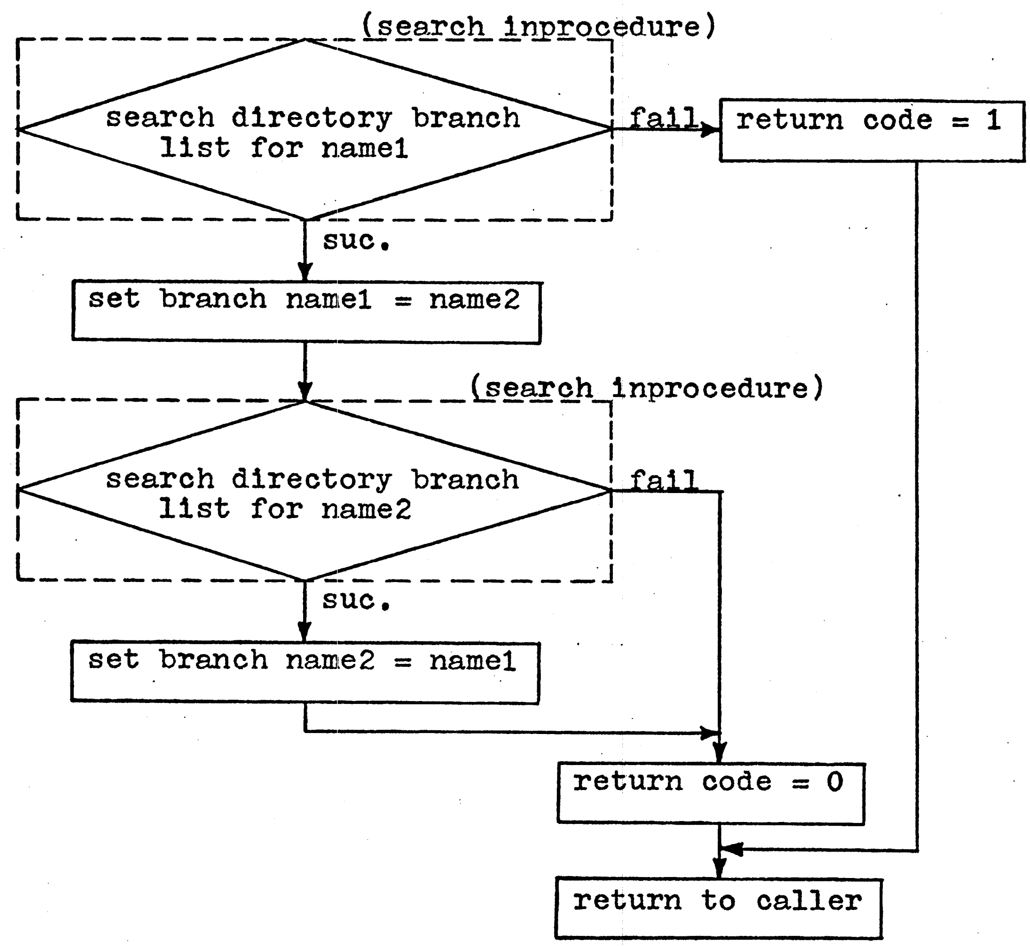
Directory manipulator: delete_seg entry



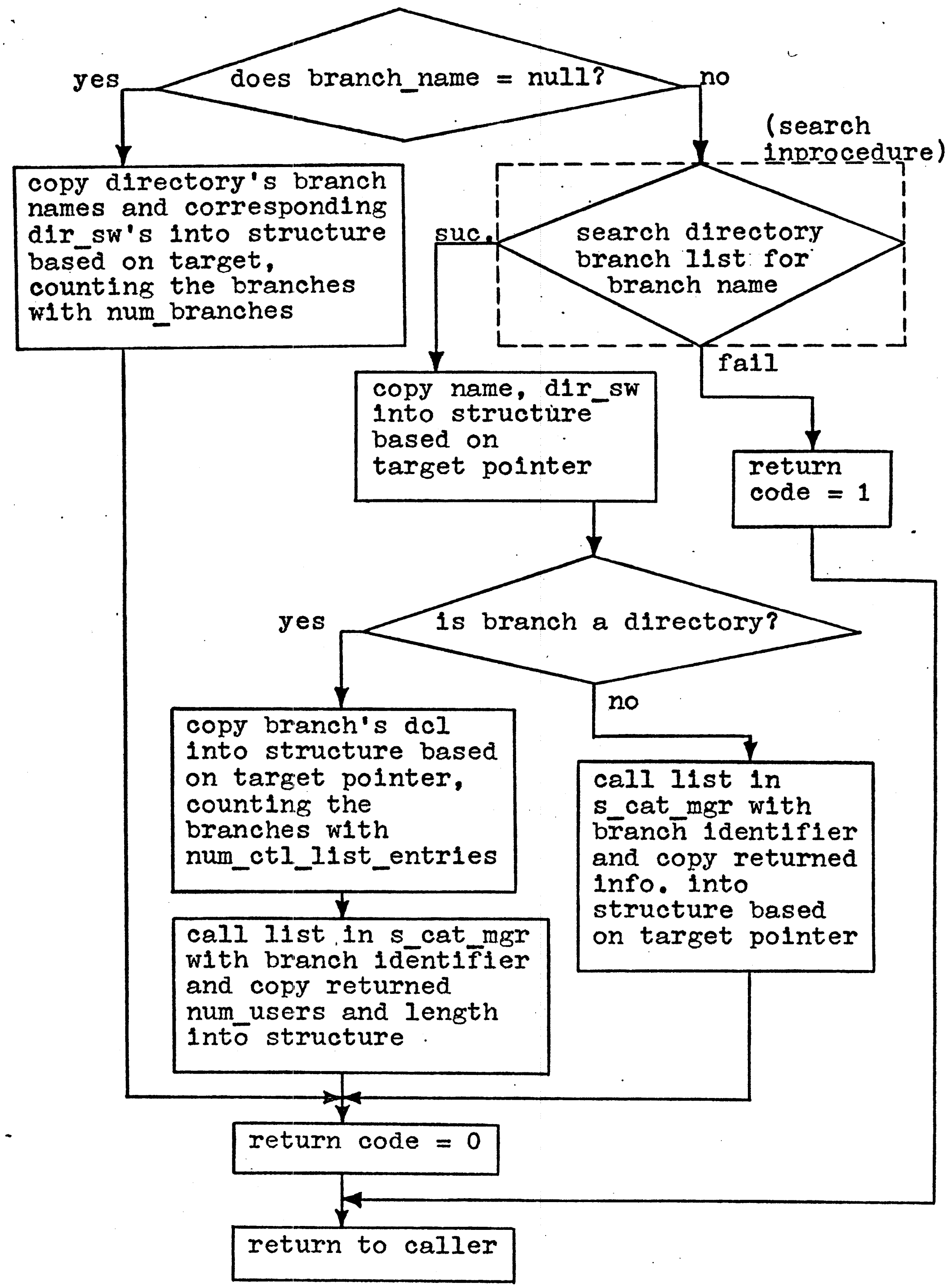
Directory manipulator: change_ctl_list entry



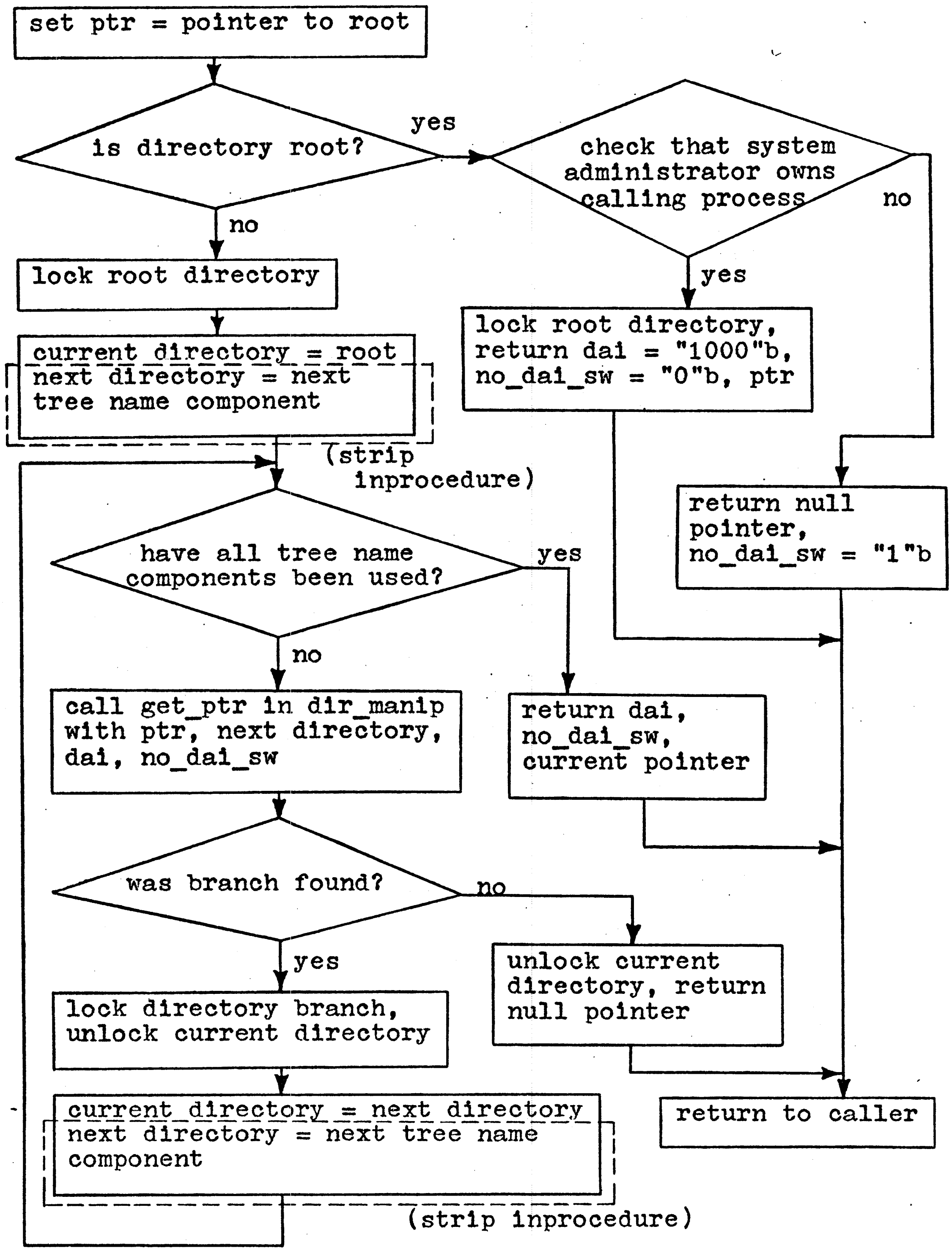
Directory manipulator: rename entry



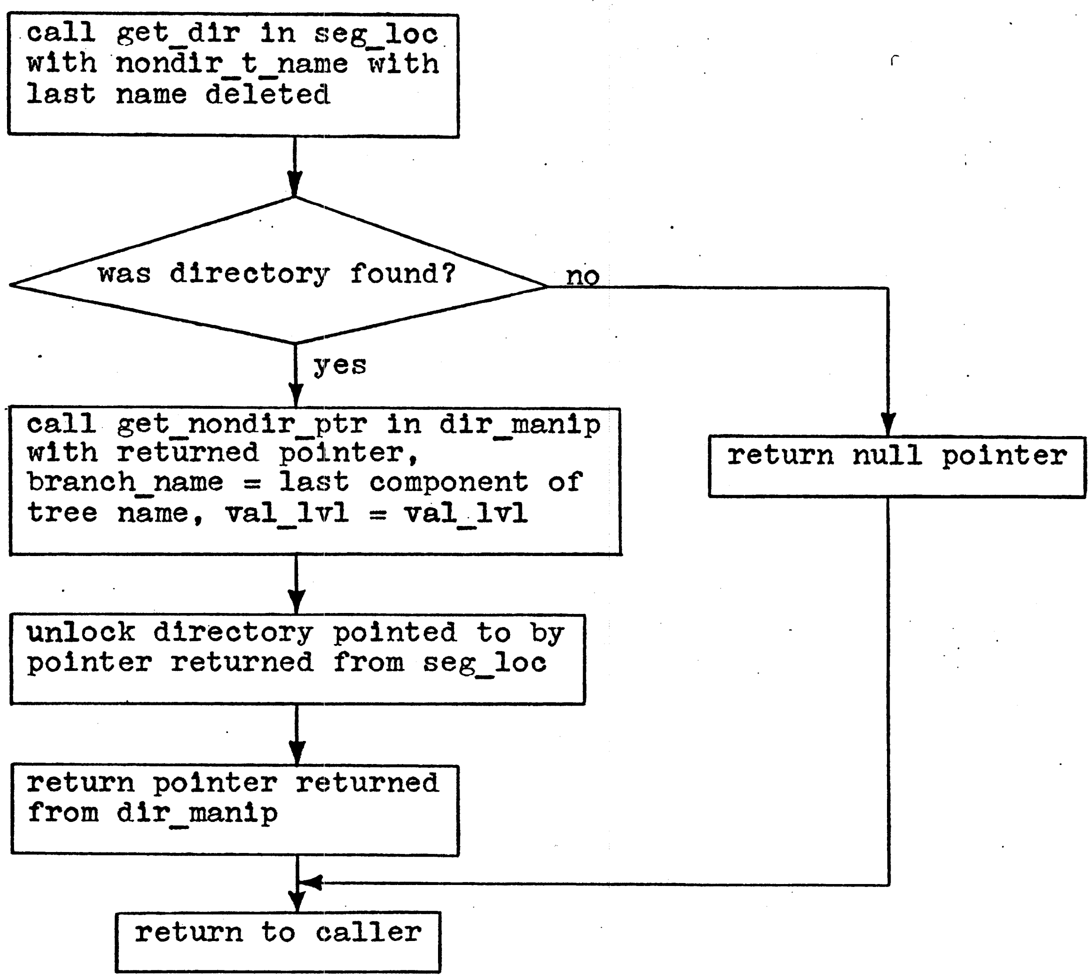
Directory manipulator: list entry



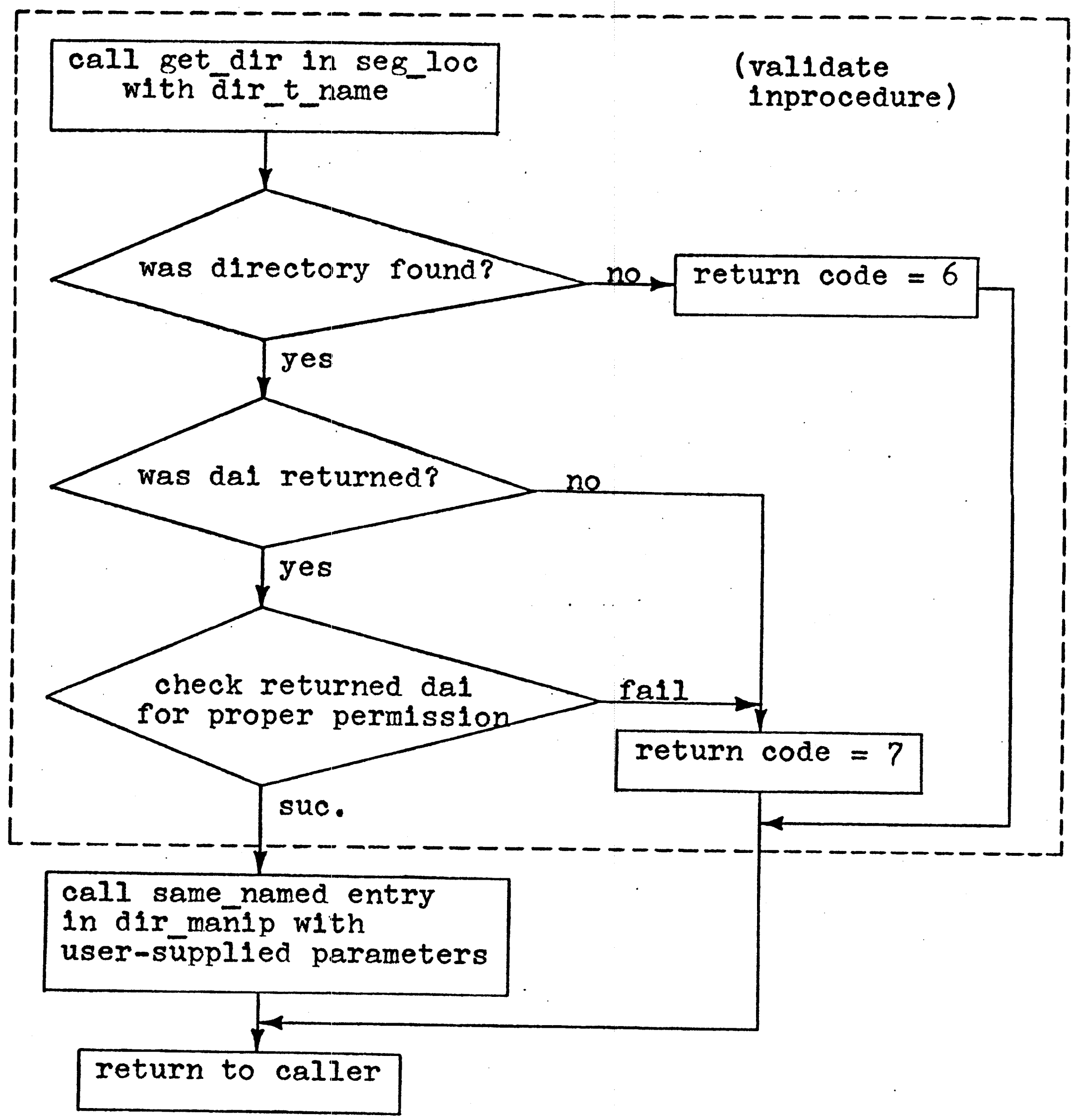
Segment locator: get_dir entry



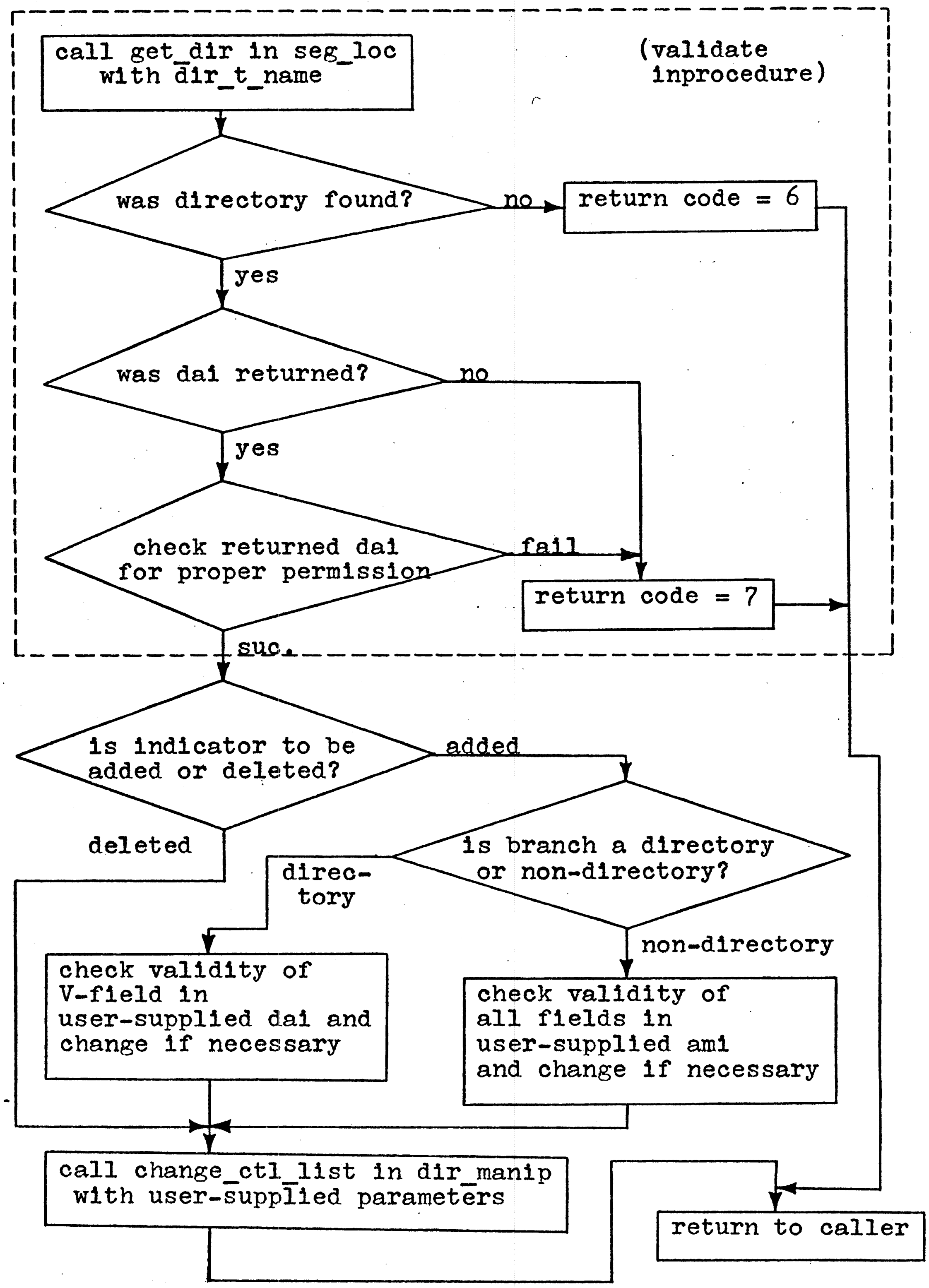
Segment locator: get_nondir entry



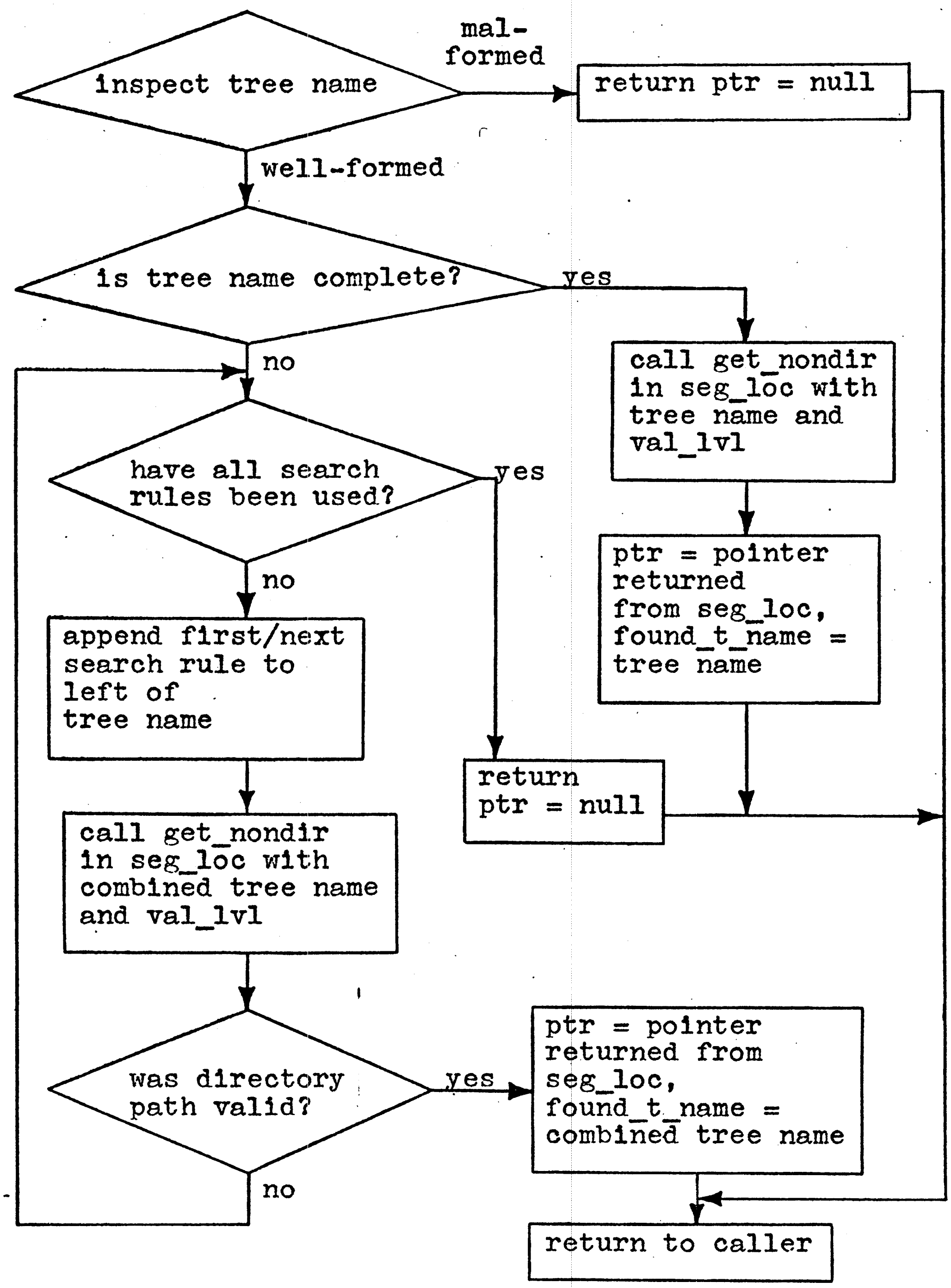
Hierarchy access validator: create_seg, change_seg_length
delete_seg, rename, and list entries



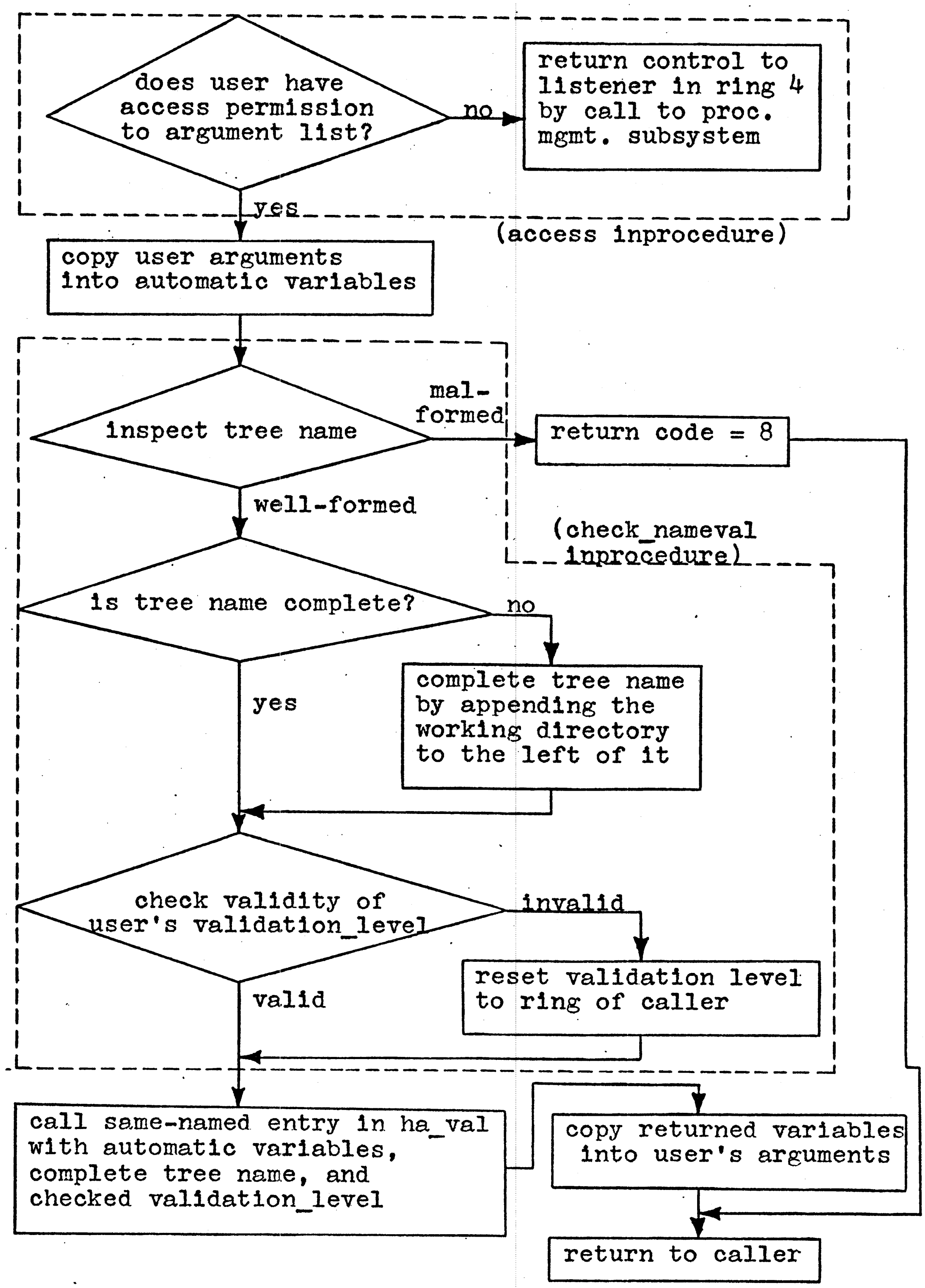
Heierarchy access validator: change_ctl_list entry



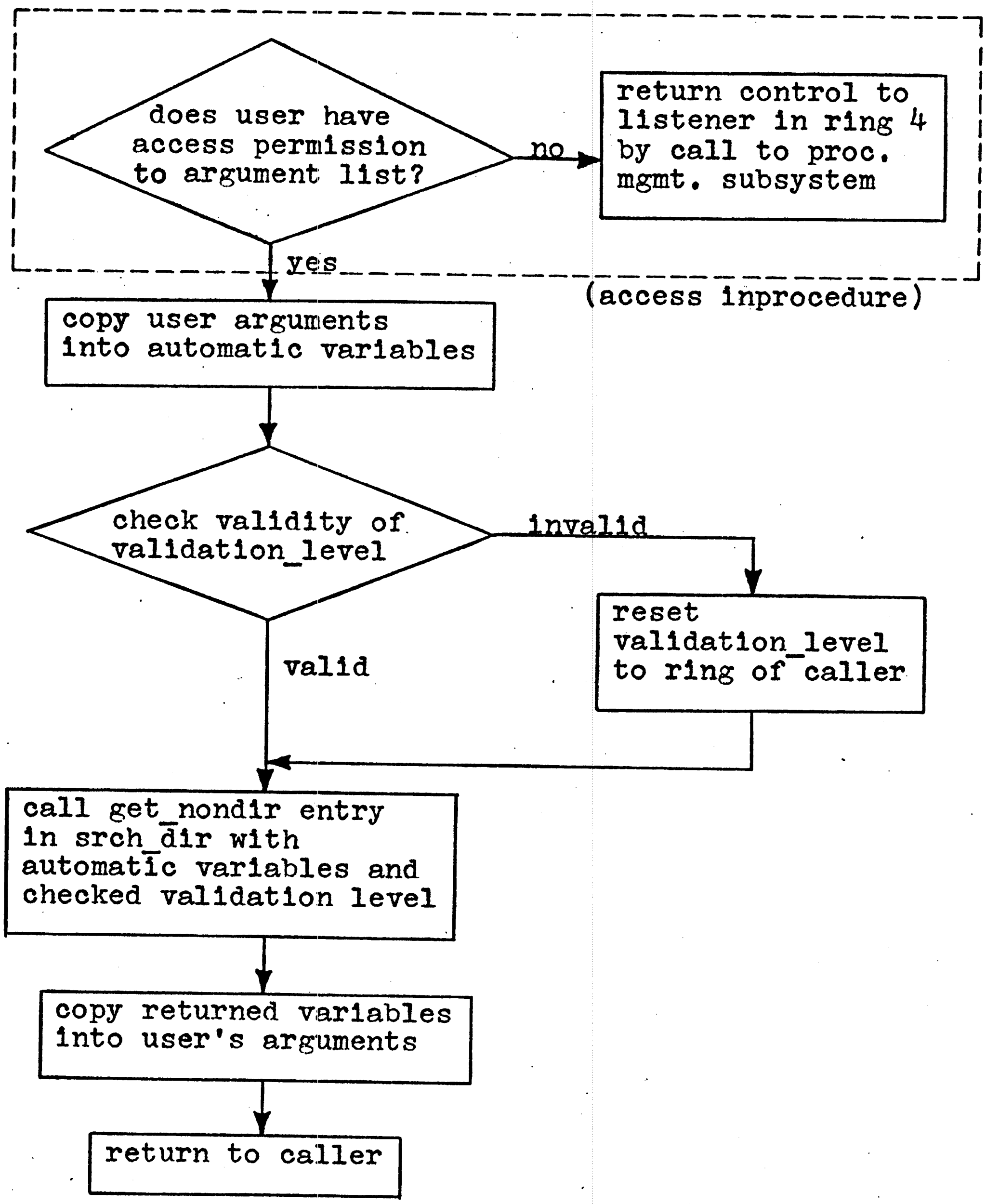
Search director: get_nondir entry



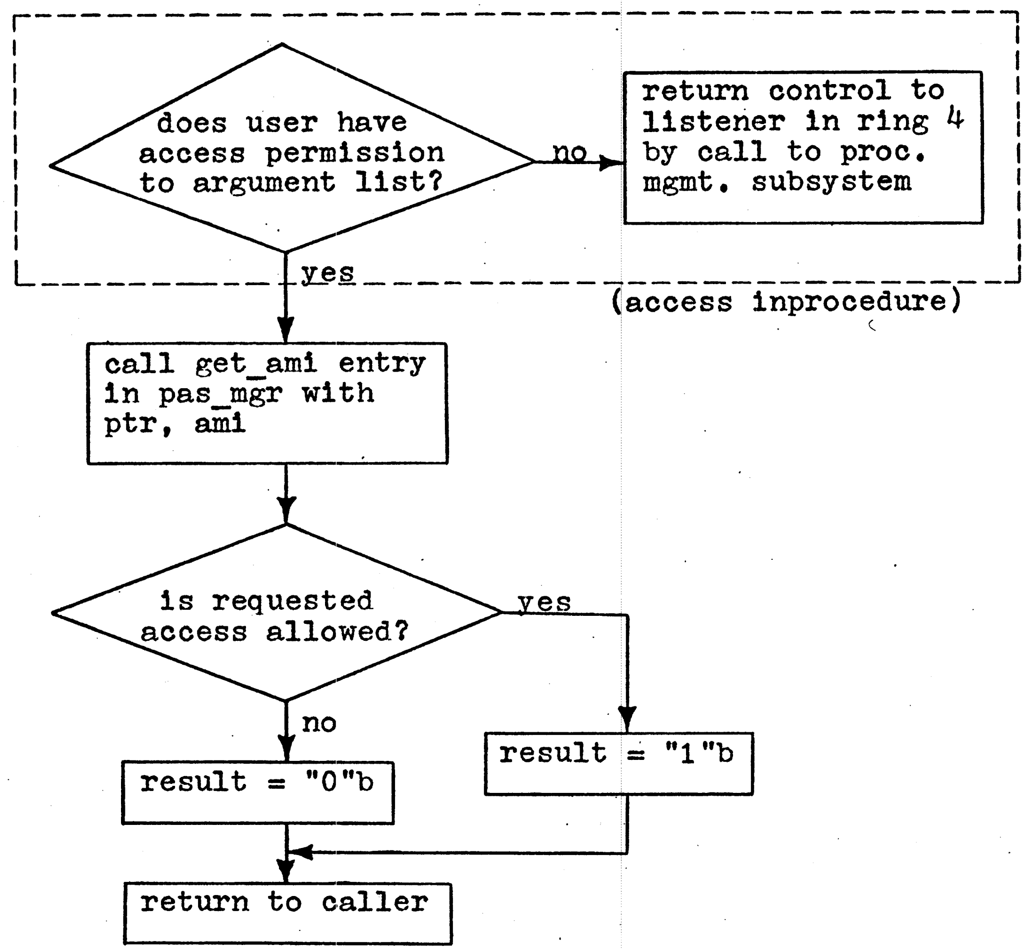
. User interface manager: create_seg, change_seg_length, change_ctl_list, rename, and list entries



User interface manager: get_nondir entry



User interface manager: check_access entry



APPENDIX II

Code of the procedure segments within the directory control and hierarchy control modules of the Storage Management Subsystem

```
/* the name of this segment will be dir_manip */
procedure;
declare 1 dir based,
          2 lock integer,
          2 capacity integer,
          2 free_list integer,
          2 branch_list integer,
          2 dir_id integer,
          2 block (100),
            3 link integer,
            3 name (32) character,
            3 seg_id integer,
            3 dir_sw integer;
declare 1 dcl_entry based,
          2 link integer,
          2 user_name (32) character,
          2 dai (4) bit;
declare 1 branch_cnts based,
          2 name (32) character,
          2 dir_sw bit,
          2 length integer,
          2 num_users integer,
          2 num_ctl_list_entries integer,
          2 ctl_list (500),
            3 user_name (32) character,
            3 indicator (11) bit;
```

```
declare 1 info based,
          2 length integer,
          2 num_users integer;
declare calr (32) character, index1 integer, index2 integer,
        id integer, point pointer, x integer, bpt51
        pointer, ptr61 pointer, ptr62 pointer, ptr63 pointer,
        countr1 integer, targ pointer, countr2 integer,
        dclpt8 pointer, temptrg pointer;
declare subr entry, suc_sw bit, dsw integer, ident integer,
        backlink integer, index integer;
declare block (size(info)) integer;

inprocedure;
search:  entry(dpoint pointer, bname (32) character);
     /* this internal procedure searches a directory for
        a branch of a given name.  if found, the index of
        the branch is returned in index, its segment
        identifier is returned in ident, and the branch's
        directory switch is returned in dsw.  the index
        of the preceding branch in the chain is returned
        in backlink.  if not found, then suc_sw is set
        equal to zero */
index = dpoint->dir.branch_list;
backlink = -1;
test:  if index = -1 then do;
          suc_sw = "0"b;
          return;
```

```
            end;
if dpoint-> dir.block(index).name = bname then do;
            suc_sw = "1"b;
            ident = dpoint-> dir.block(index).seg_id;
            dsw = dpoint-> dir.block(index).dir_sw;
            return;
            end;
      else do;
            backlink = index;
            index = dpoint-> dir.block(index).link;
            go to test;
            end;
return;
end search;


get_dir_ptr:  entry(dirptr pointer, branch_name (32)
      character, dai (4) bit, no_dai_sw bit, ptr pointer);
call search (dirptr, branch_name);
      /* if branch cannot be found, return null pointer */
if suc_sw = "0"b then do;
            ptr = ""p;
            return;
            end;
call "pas_mgr.get_ptr"p-> subr(ident, 0, ptr);
      /* set calr equal to the present user by call to
         processor management subsystem (call not given
         here) */
```

```
index1 = dirptr-> dir.dir_id;
     /* search branch's dcl for name of caller and return
        appropriate indicator */
do while index1 ∧= -1;
     if dirptr-> dir.block(index1).dcl_entry.user_name =
          calr | dirptr->
          dir.block(index1).dcl_entry.user_name = *
          then do;
          dai = dirptr-> dir.block(index1).dcl_entry.dai;
          no_dai_sw = "0"b;
          return;
          end;
     else index1 = dirptr-> dir.block(index1).dcl_entry.link;
     end;
no_dai_sw = "1"b;
return;


get_nondir_ptr:  entry(dirptr pointer, branch_name (32)
     character, val_lvl integer, ptr2 pointer);
call search (dirptr, branch_name);
     /* if branch cannot be found, return null pointer */
if suc_sw = "0"b then do;
          ptr = ""p;
          return;
          end;
call "pas_mgr.get_ptr"p-> subr(ident, val_lvl, ptr2);
return;
```

```
create_seg: entry(dirptr pointer, branch_name (32)
     character, length integer, code3 integer);
call search (dirptr, branch_name);
     /* if branch cannot be found, return null pointer */
if suc_sw = "1"b then do;
          code3 = 2;
          return;
          end;
     /* if directory is full, return appropriate code */
dirptr-> dir.branch_list = dirptr-> dir.free_list;
index1 = dirptr-> dir.branch_list;
dirptr-> dir.free_list = dirptr-> dir.block(index1).link;
     /* for a non-directory segment, create the segment
        and fill in its branch block */
if dsw ∧= -2 then do;
          call "s_cat_mgr.create_seg"p-> subr(length, id);
          dirptr-> dir.block(index1).seg_id = id;
          dirptr-> dir.block(index1).dir_sw = -2;
          dirptr-> dir.block(index1).name = branch_name;
          code3 = 0;
          return;
          end;
     /* for a directory, create the directory segment and
        fill in its branch block */
call "s_cat_mgr.create_seg"p-> subr(size(dir), id);
dirptr-> dir.block(index1).seg_id = id;
dirptr-> dir.block(index1).dir_sw = -1;
```

```
dirptr-> dir.block(index1).name = branch_name;
     /* make appropriate entries in acl of new directory */
call "s_cat_mgr.change_acl"p-> subr(id, 0, *,
     "10000000000"b);
     /* initialize new directory's header and link its
        blocks to its branch list */
call "pas_mgr.get_ptr"p-> subr(id, 0, point);
point-> dir.lock = 0;
point-> dir.capacity = 100;
point-> dir.free_list = 99;
point-> dir.branch_list = -1;
point-> dir.dir_id = id;
do x = 0 by 1 while x < 100;
          point-> dir.block(x).link = x-1;
          end;
code3 = 0;
return;

change_seg_length:  entry(dirptr pointer, branch_name (32)
     character, new_length integer, code4 integer);
call search (dirptr, branch_name);
     /* if branch cannot be found, return null pointer */
if suc_sw = "0"b then do;
          code4 = 1;
          return
          end;
```

```
     /* if branch is a directory, return to caller with
        appropriate code */
if dsw∧= -2 then do;
          code4 = 3;
          return;
          end;
call "s_cat_mgr.change_seg_length"p-> subr(ident, new_length);
code4 = 0;
return;


delete_seg:  entry(dirptr pointer, branch_name (32)
     character, code5 integer);
call search (dirptr, branch_name);
     /* if branch cannot be found, return null pointer */
if suc_sw = "0"b then do;
          code5 = 1;
          return;
          end;
     /* branch to non_dir for a non-directory segment */
if dsw = -2 then go to non_dir;
call "pas_mgr.get_ptr"p-> subr(ident, 0, point);
     /* if directory-branch's branch list not empty, return
        to caller with appropriate code */
if point-> dir.branch_list ∧= -1 then do;
          code5 = 4;
          return;
          end;
```

```
    /* return branch's dcl blocks to directory's free
       list */
index1 = dirptr-> dir.block(index).dir_sw;
index2 = dirptr-> dir.free_list;
dirptr-> dir.free_list = index1;
testr:  bpt51 = addr(dirptr-> dir.block(index1));
if bpt51-> dcl_entry.link = -1 then go to next;
index1 = bpt51-> dcl_entry.link;
go to testr;
next:  bpt51-> block.link = index2;
non_dir:  call "s_cat_mgr.delete_seg"p-> subr(ident);
    /* return branch's block to directory's free list */
if backlink = -1 then do;
          dirptr-> dir.branch_list =
               dirptr-> dir.block(index).link;
          go to then;
          end;
dirptr-> dir.block(backlink).link =
     dirptr-> dir.block(index).link;
then:  dirptr-> dir.block(index).link =
          dirptr-> dir.free_list;
dirptr-> dir.free_list = index;
code5 = 0;
return;
```

```
change_ctl_list:  entry(dirptr pointer, branch_name (32)
     character, dir_sw bit, ins_del_sw bit, user_name (32)
     character, indicator (*) bit code6 integer);
call search (dirptr, branch_name);
     /* if branch cannot be found, return appropriate code */
if suc_sw = "0"b then do;
          code6 = 1;
          return;
          end;
     /* if supplied dir_sw doesn't match dir_sw of branch,
        return to caller with appropriate code */
if dsw = -2 then if dir_sw = "1"b then do;
               code6 = 1;
               return;
               end;
     else do;
          call "s_cat_mgr.change_acl"p-> subr(ident,
               ins_del_sw, user_name, indicator);
          go to setcode;
          end;
     /* if supplied dir_sw doesn't match dir_sw of branch,
        return to caller with appropriate code */
if dir_sw = "0"b then do;
          code6 = 1;
          return;
          end;
```

```
        /* if caller attempts to add a dcl entry to a full
           directory, return the appropriate code */
if ins_del_sw = "0"b then if dirptr-> dir.free_list = -1
           then do;
           code6 = 5;
           return;
           end;
        else do;
        /* add new entry to dcl list, obtaining needed
           block from free list */
           index1 = dirptr-> dir.free_list;
           dirptr-> dir.free_list =
                dirptr-> dir.block(index1).link;
           index2 = dirptr-> dir.block(index).dir_sw;
           dirptr-> dir.block(index).dir_sw = index1;
           ptr61 = addr(dirptr-> dir.block(index1);
           ptr61-> dcl_entry.link = index2;
           ptr61-> dcl_entry.user_name = user_name;
           ptr61-> dcl_entry.dai = indicator;
           go to setcode;
           end;
index2 = dirptr-> dir.block(index).dir_sw;
if index2 = -1 then go to setcode;
ptr63 = ""p;
remove:  ptr62 = addr(dirptr-> dir.block(index2));
if ptr62-> dcl_entry.user_name = user_name then do;
```

```
          if ptr63 ^= ""p then ptr63-> dcl_entry.link =
               ptr62-> dcl_entry.link;
          dirptr-> dir.block(index2).link =
               dirptr-> dir.free_list;
          dirptr-> dir.free_list = index2;
          go to setcode;
          end;
index2 = ptr62-> dcl_entry.link;
ptr63 = ptr62;
go to remove;
setcode:  code6 = 0;
return;

rename:  entry(dirptr pointer, name1 (32) character,
     name1 (32) character, code7 integer);
call search (dirptr, name1);
     /* if branch cannot be found, return null pointer */
if suc_sw = "0"b then do;
          code7 = 1;
          return;
          end;
     /* save index of branch */
index1 = index;
     /* rename branch with new name and rename second
        branch (if found) with first name */
call search (dirptr, name2);
dirptr-> dir.block(index1).name = name2;
```

```
code7 = 0;
if suc_sw = "0"b then dirptr-> dir.block(index).name =
     name1;
return;

list:  entry(dirptr pointer, branch_name (32) character,
     target pointer, code8 integer);
if branch_name = "" then do;
     /* copy directory's branch names and corresponding
        dir_sw's into user-supplied structure */
          countr1 = 0;
          index1 = dirptr-> dir.branch_list;
          loop81:  if index1 = -1 | countr1 = 500 then do;
     /* copy number of branches into user's structure */
               target-> branch_names_list.num_branches =
                    countr1;
               go to coder;
               end;
          countr1 = countr1+1;
          target-> branch_names_list.branch(countr1).name =
               dirptr-> dir.branch(index1).name;
          if dirptr-> dir.branch(index1).dir_sw = -2 then
               target-> branch_names_list.branch(countr1).
                    dir_sw = "0"b;
          else target-> branch_names_list.branch(countr1).
                    dir_sw = "1"b;
          index1 = dirptr-> dir.branch(index1).link;
```

```
            go to loop81;
            end;
call search (dirptr, branch_name);
     /* if branch cannot be found, return appropriate code */
if suc_sw = "0"b then do;
            code8 = 1;
            return;
            end;
target-> branch_cnts.name = branch_name;
     /* insert appropriate dir_sw into user's structure */
if dsw = -2 then target-> branch_cnts.dir_sw = "0"b;
     else target-> branch_cnts.dir_sw = "1"b;
if dsw = -2 then do;
     /* for non-directory branch have s_cat_mgr copy
        required information into user's structure */
            targ = addr(target-> branch_cnts.length);
            call "s_cat_mgr.list"p-> subr(ident, targ);
            return;
            end;
countr2 = 0;
index2 = dirptr-> dir.block(index).dir_sw;
     /* copy directory-branch's dcl into user's structure */
loop82:  if index2 = -1 | countr2 = 99 then do;
            target-> branch_cnts.num_ctl_list_entries =
                 countr2;
            go to next8;
            end;
```

```
countr2 = countr2+1;
dclpt8 = addr(dirptr-> dir.block(index2));
target-> branch_cnts.ctl_list(countr2).indicator(7:4) =
     dclpt8-> dcl_entry.dai;
target-> branch_cnts.ctl_list(countr2).user_name =
     dclpt8-> dcl_entry.user_name;
index2 = dclpt8-> dcl_entry.link;
go to loop82;
     /* use s_cat_mgr to get number of users and length
        and copy into user's structure */
next8:  temptrg = addr(block);
call "s_cat_mgr.list"p-> subr(ident,temptrg);
target-> branch_cnts.length = temptrg-> info.length;
target-> branch_cnts.num_users = temptrg-> info.num_users;
coder:  code8 = 0;
return;

end dir_manip;
```

```
/* the name of this segment will be seg_loc */
procedure;
declare x1 integer, x2 integer, dirname (320) character,
     bname (32) character, dptr pointer, nextdir (32)
     character, pname (320) character, tpoint pointer,
     subr entry, i integer, lock integer based,
     root_segno parameter (8), user (32) character;
inprocedure;
strip:  entry;
     /* remove next component from tree name and set
        nextdir equal to it */
if pname(0) = "" then do;
        nextdir(0) = "";
        return;
        end;
do i = 1 by 1 while i < 33;
        if pname(i) = "." | pname(i) = "" then do;
            nextdir = pname(0:i);
            pname(0:320-i-1) = pname(i+1:320-i-1);
            return;
            end;
        end;
end strip;

get_dir:  entry(dir_t_name (320) character, dai (4) bit,
     no_dai_sw bit, ptr pointer);
```

```
ptr = make_ptr(root_segno, 0);
     /* if the root directory itself is to be manipulated,
        a check must be made to insure that the system
        administrator (ADMINISTRATOR) owns the calling
        process.  if the check succeeds, the appropriate
        dai must be constructed */
if dir_t_name(0:4) = "root" & dir_t_name(4) = "" then do;
     /* set user = owner of the calling process by call to
        processor management subsystem.  (call not
        shown here) */
          if owner = "ADMINISTRATOR" then do;
               dai = "1000"b;
               no_dai_sw = "0"b;
               call "locker.lock"p-> subr(ptr-> lock, 1);
               return;
               end;
          else do;
               ptr = ""p;
               no_dai_sw = "1"b;
               return;
               end;
          end;
pname = dir_t_name(5:315);
call strip;
call "locker.lock"p-> subr(ptr-> lock, 1);
```

```
     /* if tree name components have been exhausted,
        we're done */
loop:  if nextdir(0) = "" then return;
call "dir_manip.get_dir_ptr"p-> subr(ptr, nextdir, dai,
     no_dai_sw, tpoint);
     /* if directory can't be found, return null pointer */
if tpoint = ""p then do;
          call "locker.unlock"p-> subr(ptr-> lock);
          ptr = ""p;
          return;
          end;
call "locker.lock"p-> subr(tpoint-> lock, 1);
call "locker.unlock"p-> subr(ptr-> lock);
ptr = tpoint;
call strip;
go to loop;
return;

get_nondir:  entry(nondir_t_name (320) character, val_lvl
     integer, ptr1 pointer);
     /* if tree name doesn't begin with a directory name
        return null pointer to caller */
if nondir_t_name(4) ^= "." then do;
          ptr1 = ""p;
          return;
          end;
x2 = 0;
```

```
     /* isolate last component of tree name */
do i = 5 by 1 while i < 320;
          if nondir_t_name(i) = "." then do;
               x1 = x2;
               x2 = i;
               end;
          if nondir_t_name(i) = "" then do;
               x1 = x2;
               x2 = i;
               go to out;
               end;
          if i = 319 then x2 = 320;
          end;
     /* set dirname equal to tree name with last component
        removed */
out:  dirname = nondir_t_name(0:x1);
     /* set bname equal to last component of tree name */
bname = nondir_t_name(x1+1: x2-x1-1);
call "seg_loc.get_dir"p-> subr(dirname, dai, no_dai_sw,
     dptr);
     /* if directory not found, return null pointer */
if dptr = ""p then do;
          ptr1 = ""p;
          return;
          end;
```

```
call "dir_manip.get_nondir_ptr"p-> (dptr, bname, val_lvl,
     ptr1);
call "locker.unlock"p-> subr(dptr-> lock);
return;

end seg_loc;
```

```
/* the name of this segment will be ha_val */
procedure;
declare ptr pointer, x integer, dai (4) bit, ins_del_sw bit,
     subr entry, r_val bit, no_dai_sw bit;
inprocedure;
validate:  entry(code integer, t_val bit);
call "seg_loc.get_dir"p-> subr(dir_t_name, dai, no_dai_sw,
     ptr);
     /* if directory not found, return appropriate code */
if ptr = ""p then do;
          code = 6;
          r_val = "1"b;
          return;
          end;
     /* if dai not found, return appropriate code */
if no_dai_sw = "1"b then go to fail;
     /* check validation level against dai */
if val_lvl > bit_to_int(3, dai(1:3)) then go to fail;
     /* check for appropriate manipulation permission */
if dai(0) = "1"b | dai(0) = t_val then do;
          r_val = "0"b;
          return;
          end;
fail:  code = 7;
r_val = "1"b;
return;
end validate;
```

```
create_seg: entry(dir_t_name (320) character, branch_name
     (32) character, val_lvl integer, length integer,
     code1 integer);
call validate (code1, "1"b);
     /* if validation fails, return to caller */
if r_val = "1"b then return;
call "dir_manip.create_seg"p-> subr(ptr, branch_name,
     length, code1);
return;

change_seg_length: entry(dir_t_name (320) character,
     branch_name (32) character, val_lvl integer,
     new_length integer, code2 integer);
call validate (code2, "1"b);
     /* if validation fails, return to caller */
if r_val = "1"b then return;
call "dir_manip.change_seg_length"p-> subr(ptr, branch_name,
     new_length, code2);
return;

delete_seg: entry(dir_t_name (320) character, branch_name
     (32) character, val_lvl integer, code3 integer);
call validate (code3, "1"b);
     /* if validation fails, return to caller */
if r_val = "1"b then return;
call "dir_manip.delete_seg"p-> subr(ptr, branch_name,
     code3);
```

```
return;

rename:  entry(dir_t_name (320) character, name1 (32)
     character, val_lvl integer, name2 (32) character,
     code5 integer);
call validate (code5, "1"b);
     /* if validation fails, return to caller */
if r_val = "1"b then return;
call "dir_manip.rename"p-> subr(ptr, name1, name2, code5);
return;

list:  entry(dir_t_name (320) character, branch_name (32)
     character, val_lvl integer, target pointer, code6
     integer);
call validate (code6, "0"b);
     /* if validation fails, return to caller */
if r_val = "1"b then return;
call "dir_manip.list"p-> subr(ptr, branch_name, target,
     code6);
return;

change_ctl_list:  entry(dir_t_name (320) character,
     branch_name (32) character, val_lvl integer, dir_sw
     bit, ins_del_sw bit, user_name (32) character,
     indicator (*) bit, code4 integer);
call validate (code4, "1"b);
     /* if validation fails, return to caller */
```

```
if r_val = "1"b then return;
if ins_del_sw = "1"b then go to change;
if dir_sw = "1"b then do;
    /* check directory's dai and change if necessary */
        if bit_to_int(3, indicator(1:3)) < val_lvl then
            indicator(1:3) = int_to_bit(3, val_lvl);
        end;
    else do;
    /* check segment's ami fields and change if necessary */
        if bit_to_int(3, indicator(2:3)) < val_lvl then
            indicator(2:3) = int_to_bit(3, val_lvl);
        if bit_to_int(3, indicator(5:3)) < val_lvl then
            indicator(5:3) = int_to_bit(3, val_lvl);
        if bit_to_int(3, indicator(8:3)) < val_lvl then
            indicator(8:3) = int_to_bit(3, val_lvl);
        end;
change:  call "dir_manip.change_ctl_list"p-> subr(ptr,
    branch_name, dir_sw, ins_del_sw, user_name, indicator,
    code4);
return;

end ha_val;
```

```
/* the name of this segment will be srch_dir */
procedure;
declare 1 srch_rules based,
          2 wrk_dir (320) character,
          2 num_rules integer,
          2 rules (*),
            3 dir_t_name (320) character;
declare x1 integer, x2 integer, n integer, countr integer,
     compname (320) character, subr entry;

get_nondir:  entry(t_name (320) character, val_lvl integer,
     ptr pointer, found_t_name (320) character);
     /* inspect tree name for proper form */
x2 = -1;
do n = 0 by 1 while n < 320;
          if t_name(n) = "." | t_name(n) = "" then do;
                    x1 = x2;
                    x2 = n;
                    if x2-x1 = 1 | x2-x1 > 33 then do;
     /* if tree name is mal-formed return null pointer */
                              ptr = ""p;
                              return;
                              end;
                    end;
          if t_name(n) = "" then go to out;
          if n = 319 then if t_name(n) = "." then do;
                    ptr = ""p;
```

```
                    return;
                    end;
                else x2 = 320;
            end;
        /* if tree name is complete, call seg_loc to get
           the required pointer */
out:  if t_name(0:5) = "root." | t_name(0:4) = "root" &
        t_name(4) = "" then do;
            call "seg_loc.get_nondir"p-> subr(t_name,
                val_lvl, ptr);
            found_t_name = t_name;
            return;
            end;
countr = 0;
        /* this loop tries the search rules one at a time */
loop:  if countr > "srch_rules"p-> srch_rules.num_rules
        then do;
            ptr = ""p;
            return;
            end;
do n = 0 by 1 while n+x2+1 < 320;
            if "srch_rules"p-> srch_rules.rules(countr).
                dir_t_name(n) = "" then do;
        /* append current rule to left of tree name and set
           compname equal to the compound name */
                compname = "srch_rules"p->
                    srch_rules.rules(countr).dir_t_name(0:n);
```

```
                compname(n) = ".";
                compname(n+1:x2) = t_name(0:x2);
                go to next;
                end;
            end;
next:  call"seg_loc.get_nondir"p-> subr(compname,
     val_lvl, ptr);
countr = countr+1;
     /* if valid directory was not constructed using
        current rule, try next rule */
if ptr = ""p then go to loop;
found_t_name(0:x2+n+1) = compname(0:x2+n+1);
return;

end srch_dir;
```

```
/* the name of this segment will be uface_mgr */
procedure;
declare dir_tree_name9 (320) character automatic,
     branch_name9 (32) character automatic, validation_level9
     integer automatic, length9 integer automatic,
     code9 integer automatic;
declare new_length9 integer automatic;
declare dir_sw9 bit automatic, insert_delete_sw9 bit
     automatic, user_name9 (32) character automatic,
     indicator9 (*) bit automatic;
declare branch_name19 (32) character automatic, branch_name29
     (32) character automatic, target9 pointer automatic;
declare t_name (320) character automatic, ptr9 pointer
     automatic, found_t_name9 (320) character automatic;
declare subr entry, ami (11) bit, no_ami_sw bit;
declare x1 integer, x2 integer, malform bit, rulept
     pointer, code integer, found_t_name (320) character,
     i integer, z integer, res (3) bit, ring integer;
declare names_block (size(branch_names_list9)) integer;
declare cnts_block (size (branch_cnts9)) integer;
declare 1 blockr (*) based,
          2 dummy integer;
declare 1 prr based,
          2 dummy (61) bit,
          2 val (3) bit;
declare 1 srch_rules based,
          2 wrk_dir (320) character;
```

```
declare 1 branch_names_list9 based,
          2 num_branches integer,
          2 branch (100),
            3 name (32) character,
            3 dir_sw bit;
declare 1 branch_cnts9 based,
          2 name (32) character,
          2 dir_sw bit,
          2 length integer,
          2 num_users integer,
          2 num_ctl_list_entries integer,
          2 ctl_list (500),
            3 user_name (32) character,
            3 indicator (11) bit;


inprocedure;
xcess:  entry;
     /* this internal procedure checks the caller's
        access to the argument list and returns control
        to the listener of the Command Subsystem if the
        check fails */
call "pas_mgr.get_ami"p->subr("ap"p, ami, no_ami_sw);
if no_ami_sw = "1"b then go to listen:
if bit_to_int(3, ami(5:3)) >= bit_to_int(3, "sp"p->
     prr.val) then return;
listen: /* return control to listener by call to processor
           management subsystem (call not shown here) */
```

```
return;
end xcess;


inprocedure;
check_nameval:  entry;
     /* check for manipulation of working directory only */
if dir_tree_name(0) = "" then do;
          x2 = 0;
          go to out;
          end;
     /* check tree name for proper form */
x2 = -1;
do i = 0 by 1 while i < 320;
          if dir_tree_name(i) = "." | dir_tree_name(i) = ""
               then do;
               x1 = x2;
               x2 = i;
               if x2-x1 = 1 | x2-x1 > 33 then do;
     /* if tree name is malformed, return appropriate
        indicator to uface_mgr */
                    malform = "1"b;
                    return;
                    end;
               end;
          if dir_tree_name(i) = "" then go to out;
          if i = 319 then if dir_tree_name(i) = "." then do;
               malform = "1"b;
```

```
                return;
                end;
                else x2 = 320;
            end;
out:  malform = "0"b;
        /* if tree name is complete, branch to comp */
if dir_tree_name(0:5) = "root." | dir_tree_name(0:4) =
        "root" & dir_tree_name(4) = "" then do;
            dir_tree_name9 = dir_tree_name;
            go to comp;
            end;
        /* set dir_tree_name9 equal to given tree name with
           working directory appended to left of it */
do i = 1 by 1 while i < 320;
        if "srch_rules"p-> srch_rules.wrk_dir(i) = "" then do;
            dir_tree_name9 = "srch_rules"p->
                srch_rules.wrk_dir(0:i);
            dir_tree_name9(i) = ".";
            dir_tree_name9(i+1:x2) = dir_tree_name(0:x2);
            go to comp;
            end;
        end;
        /* check validation level supplied by caller against
           his ring number and change if necessary */
comp:  if validation_level9 < bit_to_int(3, "sp"p-> prr.val)
        then validation_level9 = bit_to_int(3, "sp"p-> prr.val);
if validation_level9 > 7 then validation_level9 = 7;
```

```
return;
end check_nameval;


create_seg:  entry(dir_tree_name (320) character,
     branch_name (32) character, validation_level integer,
     length integer, code1 integer);
call xcess;
     /* copy user's arguments into automatic variables for
        ring zero */
branch_name9 = branch_name;
validation_level9 = validation_level;
length9 = length;
code9 = code1;
call check_nameval;
     /* if tree name is malformed, return appropriate code
        to caller */
if malform = "1"b then do;
          code1 = 8;
          return;
          end;
call "ha_val.create_seg"p-> subr(dir_tree_name9,
     branch_name9, validation_level9, length9, code9);
code1 = code9;
return;
```

```
change_seg_length:  entry(dir_tree_name (320) character,
     branch_name (32) character, validation_level integer,
     new_length integer, code2 integer);
call xcess;
     /* copy user-supplied arguments into automatic
        variables for ring zero */
branch_name9 = branch_name;
validation_level9 = validation_level;
new_length9 = new_length;
code9 = code2;
call check_nameval;
     /* if tree name is malformed, return appropriate
        code to caller */
if malform = "1"b then do;
          code2 = 8;
          return;
          end;
call "ha_val.change_seg_length"p-> subr(dir_tree_name9,
     branch_name9, validation_level9, new_length9, code9);
code2 = code9;
return;

delete_seg:  entry(dir_tree_name (320) character,
     branch_name (32) character, validation_level integer,
     code3 integer);
call xcess;
```

```
     /* copy user-supplied arguments into automatic
        variables for ring zero */
branch_name9 = branch_name;
validation_level9 = validation_level;
code9 = code3;
call check_nameval;
     /* if tree name is malformed, return appropriate code
        to caller */
if malform = "1"b then do;
        code3 = 8;
        return;
        end;
call "ha_val.delete_seg"p-> subr(dir_tree_name9,
     branch_name9, validation_level9, code9);
code3 = code9;
return;

change_ctl_list:  entry(dir_tree_name (320) character,
     branch_name (32) character, validation_level integer,
     dir_sw bit, ins_del_sw bit, user_name (32) character,
     indicator (*) bit, code4 integer);
call xcess;
     /* copy user-supplied arguments into automatic
        valiables for ring zero */
branch_name9 = branch_name;
validation_level9 = validation_level;
dir_sw9 = dir_sw;
```

```
ins_del_sw9 = ins_del_sw;
user_name9 = user_name;
indicator9 = indicator;
code9 = code4;
call check_nameval;
     /* if tree name is malformed, return appropriate
        code to caller */
if malform = "1"b then do;
          code4 = 8;
          return;
          end;
call "ha_val.change_ctl_list"p-> subr(dir_tree_name9,
     branch_name9, validation_level9, dir_sw9, ins_del_sw9,
     user_name9, indicator9, code9);
code4 = code9;
return;

rename:  entry(dir_tree_name (320) character, branch_name_1
     (32) character, validation_level integer,
     branch_name_2 (32) character, code5 integer);
call xcess;
     /* copy user-supplied arguments into automatic
        variables for ring zero */
branch_name_19 = branch_name_1;
validation_level9 = validation_level;
branch_name_29 = branch_name_2;
code9 = code5;
```

```
call check_nameval;
     /* if tree name is malformed, return appropriate
        code to caller */
if malform = "1"b then do;
          code5 = 8;
          return;
          end;
call "ha_val.rename"p-> subr(dir_tree_name9, branch_name_19,
     validation_level9, branch_name_29, code9);
code5 = code9;
return;


list:  entry(dir_tree_name (320) character, branch_name
     (32) character, validation_level integer, target
     pointer, code6 integer);
call xcess;
     /* copy user-supplied arguments into automatic
        variables for ring zero */
branch_name9 = branch_name;
validation_level9 = validation_level;
code9 = code6;
call check_nameval;
     /* if tree name is malformed, return appropriate
        code to caller */
if malform = "1"b then do;
          code6 = 8;
```

```
            return;
            end;
        /* make target9 point to ring zero storage of
           appropriate size */
if branch_name9 = "" then target9 = addr(names_block);
        else target9 = addr(cnts_block);
call "ha_val.list"p-> subr(dir_tree_name9, branch_name9,
        validation_level9, target9, code9);
code6 = code9;
if branch_name9 = "" then z = size(names_block);
        else z = size(cnts_block);
        /* use structure-referencing trick to copy data
           stored in ring zero area back into user-supplied
           area */
do i = 0 by 1 while i < z;
            target-> blockr(i).dummy = target9->
                blockr(i).dummy;
            end;
return;

get_nondir:  entry(t_name (320) character, validation_level1
        integer, ptr1 pointer, found_t_name (320) character);
call xcess;
        /* copy user-supplied arguments into automatic
           variables for ring zero */
t_name9 = t_name;
validation_level9 = validation_level1;
```

```
ptr9 = ptr1;
found_t_name9 = found_t_name;
     /* check validation level supplied by caller against
        his ring number and change if necessary */
if validation_level9 < bit_to_int(3, "sp"p-> prr.val) then
     validation_level9 = bit_to_int(3, "sp"p-> prr.val);
if validation_level9 > 7 then validation_level9 = 7;
call "srch_dir.get_nondir"p-> subr(t_name9, validation_level9,
     ptr9, found_t_name9);
ptr1 = ptr9;
found_t_name = found_t_name9;
return;


check_access:  entry(ptr2 pointer, access (3) bit, result
     bit);
call xcess;
call "pas_mgr.get_ami"p-> subr(ptr2, ami, no_ami_sw);
if no_ami_sw = "1"b then do;
          result = "0"b;
          return;
          end;
ring = bit_to_int(3, "sp"p-> prr.val);
res = "000"b;
     /* if read permission desired, check R2 field of
        segment's ami against caller's ring and indicate
        result of check in res */
```

```
if access(0) = "1"b then do;
     if ring <= bit_to_int(3, ami(5:3)) then res(0) = "1"b;
          else res(0) = "0"b;
     end;
     /* if write permission desired, check R1 and T
        fields of ami */
if access(1) = "1"b then do;
     if ring <= bit_to_int(3, ami(2:3)) & ami(0) = "1"b
          then res(1) = "1"b;
          else res(1) = "0"b;
          end;
     /* if call permission desired, check R3 field of ami */
if access(2) = "1"b then do;
     if ring <= bit_to_int(3, ami(8:3)) then res(2) = "1"b;
          else res(2) = "0"b;
          end;
     /* compare res to access to determine whether
        requested permission is allowed */
if bit_to_int(3, res) = bit_to_int(3, access) then
     result = "1"b;
     else result = "0"b;
return;
end uface_mgr;
```

BIBLIOGRAPHY

1. Clark, D., and Schroeder, M., CLICS System Specification Notebook (Preliminary version), unpublished, 1969.

2. Corbato, F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System," Proceedings of the Fall Joint Computer Conference, 27, Spartan Books, Washington, D.C., 1965, pp. 185-196.