

FILE COPY

ESD PRODUCTION LIST

DRI Call No. 88497

Page No. 1 of 2 cys.

ESD-TR-77-259, Vol. III

MTR-3294, Vol. III

DESIGN AND ABSTRACT SPECIFICATION
OF A MULTICS SECURITY KERNEL

BY J. P. L. WOODWARD

MARCH 1978

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts



Approved for public release;
distribution unlimited.

Project No. 522N

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-77-C-0001

ADAD53149

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

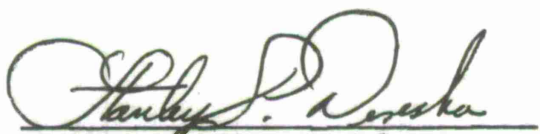
Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.


SYLVIA R. MAYER
Acting Chief, Techniques
Engineering Division


WILLIAM R. PRICE, Capt, USAF
Techniques Engineering Division


STANLEY B. DERESKA, Col, USAF
Director, Computer Systems Engrg.
Deputy for Command & Management Systems

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-259, Vol. III	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN AND ABSTRACT SPECIFICATION OF A MULTICS SECURITY KERNEL		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MTR-3294, Vol. III
7. AUTHOR(s) J. P. L. Woodward		8. CONTRACT OR GRANT NUMBER(s) F19628-77-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522N
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, MA 01731		12. REPORT DATE MARCH 1978
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES 49
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) COMPUTER SECURITY FORMAL SOFTWARE SPECIFICATION MULTICS SECURITY KERNEL		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) On the basis of the recommendations of the Electronic Systems Division Computer Security Technology Panel (1972), The MITRE Corporation developed techniques for the design, implementation, and formal mathematical verification of a security kernel: a hardware and software mechanism to control access to information within a computer system.		

(over)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. Abstract (continued)

This three-volume report describes the design of a security kernel for Honeywell Information System's Multics computer system. This third volume gives a formal, top-level specification of the secondary subsystems of the kernel, including the System Security Officer, reconfiguration, and initialization. It is sufficiently detailed to allow its security, compatibility, and efficiency to be determined.

The first volume gave a methodology and design overview. The second volume dealt with the primary subsystems of the kernel.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ACKNOWLEDGMENT

This report has been prepared by The MITRE Corporation under Project No. 522N. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

This report describes a design that has been evolving since September 1974. A number of individuals have contributed, including W. L. Schiller, who was responsible for the major part of the storage management design; S. R. Ames, Jr., K. J. Biba, E. L. Burke, M. Gasser, S. B. Lipner, and P. T. Withington of The MITRE Corporation; and Lt. Col. R. R. Schell, Capt. W. R. Price, and Capt. P. A. Karger of the U. S. Air Force. The design has also been influenced by discussions with personnel from other Air Force contractors and subcontractors: Honeywell Information Systems, the Computer Systems Research group of M. I. T's Laboratory for Computer Science, and Stanford Research Institute's Computer Science group.

TABLE OF CONTENTS

	<u>Page</u>
LIST OF ILLUSTRATIONS	4
SECTION I INTRODUCTION	5
SECTION II THE SYSTEM SECURITY OFFICER (SSO) INTERFACE	6
TRUSTED SUBJECTS	6
THE SSO INTERFACE DESIGN	7
COMPATIBILITY WITH THE CURRENT MULTICS	8
SPECIFICATION	8
SSO Types, Parameters, Constants, and Definitions	9
SSO V-function Macros	11
SSO V-functions	11
SSO OV-function	14
SSO O-functions	14
Design Considerations	18
SSO REVIEW	19
SECTION III RECONFIGURATION	20
THE CURRENT DESIGN	20
Major Hardware Modules	20
Hardware Module Connections	21
Memory Reconfiguration	22
Processor Reconfiguration	24
Bulk Store Reconfiguration	24
THE KERNEL DESIGN	25
COMPATIBILITY WITH THE CURRENT MULTICS	26
SPECIFICATION	26
Data Types, Parameters, and Constants	26
Reconfiguration V-functions	28
Memory Reconfiguration O-functions	30
Paging Device Reconfiguration O-functions	32
CPU Reconfiguration O-functions	32
RECONFIGURATION REVIEW	35

TABLE OF CONTENTS (Concluded)

SECTION IV	INITIALIZATION	36
	THE CURRENT DESIGN	36
	THE KERNEL DESIGN	37
	V-function Initialization	38
	Initialization Reconfiguration Functions	38
	Lower Level Initialization	38
	COMPATIBILITY WITH THE CURRENT MULTICS SPECIFICATION	39
	Initialization Parameters and Constants	39
	Initialize_Top_Level O-function	41
	Initialization Configuration O-functions	45
	INITIALIZATION REVIEW	45
APPENDIX I	INDEX TO SPECIFICATIONS	47
REFERENCES		48

LIST OF ILLUSTRATIONS

<u>Figure Number</u>		<u>Page</u>
1	SSO Types, Parameters, Constants, and Definitions	10
2	SSO V-function Macros	12
3	SSO V-functions	13
4	SSO OV-function	15
5	SSO O-functions	16
6	Hardware Module Connections	23
7	Reconfiguration Data Types, Parameters, and Constants	27
8	Reconfiguration V-functions	29
9	Memory Reconfiguration O-functions	31
10	Paging Device Reconfiguration O-functions	33
11	CPU Reconfiguration O-functions	34
12	Initialization Parameters and Constants	40
13	Initialize_top_level O-function	42
14	Initialization Configuration O-functions	46

SECTION I

INTRODUCTION

This report is Volume 3 of a three-volume report on the design of a security kernel for Multics. This volume presents the top-level specification of some of the secondary subsystems of some of the Multics kernel. The design methodology is discussed in Volume 1 of this report, and the details of the specification language used are given in Volume 2.

Three secondary subsystems of the Multics kernel are discussed in this volume. The functions presented in this paper are different from those presented in Volume 2 because they are not intended for use by normal Multics users.

The first subsection discussed is the System Security Officer (SSO). The SSO is the direct interface between the kernel and a user responsible for system security. The SSO interface is important because it provides some functions that cannot be performed by normal users because they do not follow all the security rules.

Next, hardware reconfiguration is discussed. The reconfiguration functions provided are to be used only by system high processes and allow the user to dynamically alter the hardware configuration of the system. It is intended that the supervisor running on the kernel restrict the use of these functions to the operator.

Finally, the initialization of the Multics kernel is discussed. Initialization provides the kernel with an initial secure state that will be maintained as secure by the kernel O-functions.

Other security-related subsystems, such as the Salvager and the Emergency Shutdown subsystems, are not discussed in this volume.

SECTION II

THE SYSTEM SECURITY OFFICER (SSO) INTERFACE

TRUSTED SUBJECTS

A secure computer system is not a closed system. External inputs are required to complete and maintain security. A mechanism is required to associate the computer system elements correctly with their counterparts in the people/paper world. Unverified software cannot be trusted to make these bindings.

All software on which the security of the computer system depends must be verified to perform its required function correctly if the system is to be certified. The kernel software, the software that implements the reference monitor, represents part of the verified software of the system. The remainder of the verified software comprises the "trusted subjects": the active system entities that perform the security-related binding of computer system elements to the external environment.

There are two areas in the Multics system where trusted subjects are needed. One area is initialization, which is treated in the last section. The other area is the interface between the kernel and the System Security Officer (SSO). This section describes the nature of trusted subjects and the SSO Interface specification.

The verification technique requires that all software to be certified have a top-level specification. Because of the distinctly different duties of trusted subjects, however, the requirements of their interface specification are unlike those of the kernel.

The arguments (external information) required by trusted subjects are supplied by users who are trusted to perform correctly, or by hardware. An example of a hardware-supplied argument is a unique identifier of a terminal, supplied by the hardware of the terminal or its controller. The trusted subject functions must be performed completely by verified software, since security depends on their correct operation. The trusted subject functions must interact directly with the trusted users.

Since the users of trusted functions are themselves trusted, the access control performed at the kernel interface is not required at the trusted subject interface. Nevertheless, the requirement of direct interaction with users implies that an additional mechanism must be supplied to ensure trusted subject functions are only

available to trusted users. Since this mechanism is implementation dependent, a narrative description is provided, rather than a high-level specification.

THE SSO INTERFACE DESIGN

The most significant part of the design of the SSO interface, how to ensure direct interaction with trusted software, cannot be specified. The enforcement of availability of the functions is an implementation dependent mechanism. One possible implementation is to provide the SSO functions in a verified process invoked by the power-on interrupt of a terminal. A verified process is required since it must handle terminal communications completely. No uncertified software may be interposed between the SSO function and the SSO -- even for menial tasks like I/O -- as it would then be possible to spoof the SSO or the SSO function and cause a breach of security.

The process providing the interface functions must be protected such that software may not invoke it. By implementing the process as an interrupt service routine for the terminal power-on interrupt, it can be ensured that only terminals and no software may execute the process. Of course, only certain kinds of terminals, with the proper hardware, may be used.

The SSO completes the computer security mechanism by providing necessary external inputs and performing operations that cannot be performed within the constraints of the kernel interface.

The functions required by the SSO that have been identified are: a function to inform the kernel of current device access levels, a function to remove quota from an upgraded directory, a function to downgrade segments, and functions to allow the SSO to process logical volume mount requests. The first function provides external information to the kernel. The last two functions provide kernel information to the outside environment, and external information to the kernel, as explained below. The others perform needed functions that cannot be performed under the restrictions of the kernel interface. The requirements for verification of the functions and relaxation of security controls imply that the functions must be performed by trusted subjects.

The SSO must be able to inform the kernel of device access levels because the kernel has no way to ascertain this information by itself. For example, when SECRET paper is mounted in a printer, the kernel must be informed of the access level of the printer. The levels of disk and tape drives must also be made known to the kernel.

Moving quota from an upgraded directory back to its parent is reading information from a higher level (the upgraded directory) and writing it at a lower level (of the parent). Therefore, this function violates the *-property and is not allowed at the kernel interface. The SSO must perform this function, since the SSO is permitted to violate the *-property in a controlled manner.

Downgrading segments is another function that violates some properties of the security model and hence cannot be performed at the kernel interface. The SSO has a function to set the access level of segments. This function is used to downgrade segments.

When a process calls the "Mount" function to mount a logical volume, and that logical volume is not already mounted, "Mount" enters a request to the SSO to mount the volume. The list of requests is a first-in-first-out stack, so the SSO has a function to read the earliest mount request. After he has read the request, the SSO mounts the appropriate volume(s), and then uses other SSO functions to inform the kernel that the logical volume is mounted for the requesting process, and which drive(s) it is mounted on.

As with the kernel, a trusted subject has no way of guaranteeing the identity of a user, so it must rely on its ability to identify terminals and on physical security to ensure that only the SSO executes the functions provided by the SSO trusted subjects.

COMPATIBILITY WITH THE CURRENT MULTICS

Most of the SSO functions have no analogue in the current Multics since the security policy they are concerned with is not a part of the current Multics. Compatibility is not a problem because all of the SSO functions will have a restricted number of users.

SPECIFICATION

Whereas the security kernel is concerned with the process as the specification representation of a subject, in the specification of the trusted software, the terminal embodies the active subject. The terminal acts as proxy for the user, since we lack the hardware capability to identify different users on a terminal.

Corresponding to the process control function "process", which embodies all process information, the trusted subject module uses a hardware-supplied argument "terminal" to determine if the user of the function is using the SSO terminal. This hardware-supplied argument appears in angle brackets (<>) in function argument lists.

The sections that follow provide descriptions of the SSO specification itself, broken down by parts of the specification.

SSO Types, Parameters, Constants, and Definitions

Figure 1 illustrates the SSO type definitions, parameters, constants, and definitions. There are three data types defined in the specification. The first two are related to user-specified segment pathnames, and the last is concerned with requests to mount logical volumes.

The SSO is the only part of the kernel specification that identifies segments by their pathname. The SSO interface is an interface to a person, not a program, so it is more convenient and less error-prone to identify segments by their pathname rather than their segment number. The abstract form of a Multics pathname is defined by "path_name_type". According to this definition, a pathname is a vector of entrynames. Thus, "path_name_type" is the abstract data type of a pathname as entered by a user.

Each entryname in a pathname (except the last one) is the name of a directory (the parent directory) that contains information that describes the next entryname. When entered by a user, a Multics pathname begins with a ">", which is shorthand for "root>". In our secure Multics system, the root has a defining entry which is stored in the root directory. Therefore, the parent directory of the root is the root itself. Therefore, if we wish to precisely specify a Multics pathname, we must start the pathname with "rootroot>". We need to represent a pathname with this precision in the specification, so we have another data type that describes this "full" pathname. This data type is called "full_path_name_type". In the rest of this document we will refer to this latter type of pathname as a "full" pathname, and to the pathname the user enters as a "regular" pathname.

The last type definition is of a mount request, which specifies that the given process wants the given volume to be mounted.

The parameter "path_name" is the regular pathname of a segment. Similarly, "full_path" is the full pathname of a segment. The parameter "terminal", as mentioned before, is the hardware-supplied uid of the terminal calling the SSO function. The constant "max_path_length" is the maximum number of entrynames allowed in a regular pathname, and the constant "SSO" is the uid of the SSO terminal.

In the definition section, "path_name_length" is the number of entries in a regular (user-supplied) pathname, "full_path_name" is a full pathname (as created by the "Make_full_path_name" V-function

```

/* SSO type definitions */

type
  path_name_type = vector(1 to max_path_length) of entry_type
  full_path_name_type = vector(1 to max_path_length+2) of entry_type
  mount_request_type = structure
    (process: uid_type,
     vol_id: uid_type)

/* SSO parameters */

parameter
  path_name: path_name_type
  full_path: full_path_name_type
  terminal: uid_type

/* SSO constants */

constant
  max_path_length: integer
  SSO: uid_type

/* SSO definitions */

define
  path_name_length = cardinality{i | path_name[i] ≠ "undefined"};
  full_path_name = Make_full_path_name(path_name);
  full_path_name_length = cardinality
    {i | full_path_name[i] ≠ "undefined"};
  full_path_length = cardinality
    {i | full_path[i] ≠ "undefined"};
  Branch_path = Directory
    (Path_to_uid(Parent_path(full_path_name)),
     full_path_name[full_path_name_length]);
  Dir_branch_path = Directory
    (Path_to_uid(Parent_path(Parent_path(full_path_name))),
     full_path_name[full_path_name_length-1]);

```

Figure 1. SSO Types, Parameters, Constants, and Definitions

macro), and "full_path_name_length" is the number of entries in a full pathname. The definitions "full_path_name" and "full_path_name_length" are used only by O-functions and in definitions used by O-functions.

The number of entries in a "full_path" is given by "full_path_length". Remember that "full_path" is a parameter to a V-function, so "full_path_length" is used only by some V-functions (those which are passed full pathnames).

The last two definitions are used only by O-functions or V-functions with regular pathnames as arguments. "Branch_path" is the directory branch associated with a "path_name". "Dir_branch_path" is the directory branch associated with the parent (directory) of a "path_name".

SSO V-function Macros

The SSO V-function macros are given in Figure 2. All of these V-function macros are associated with pathnames.

"Make_full_path_name" takes as its argument a regular pathname and returns a full pathname. A full pathname is a regular pathname with "root>root" concatenated at the beginning. This macro is used only in the definition of "full_path_name".

The remaining three macros take full pathnames as arguments. "Parent_path" returns the full pathname of the parent of the specified full pathname. "Path_accessible" is a boolean-valued function that tells whether the specified full pathname exists and can be accessed by the SSO at its current access level. "Path_to_uid" converts the given full pathname to the uid of the segment.

SSO V-functions

Figure 3 illustrates the SSO V-functions, both hidden and non-hidden.

The hidden V-function "Mount_request" is the list of mount requests compiled by the "Mount" O-function and processed by the SSO with the "Read_mount_request" OV-function.

The non-hidden V-function "SSO_access_level" represents the current access level of the SSO and is derived from the access level associated with the SSO terminal. The exception for this function assures that the user is at the SSO terminal.

```

/* SSO V_function_macros */

V_function_macro Make_full_path_name(path_name): full_path_name_type

derivation
  Make_full_path_name[1] = "root";
  Make_full_path_name[2] = "root";
  (∀ i ∈ {1,2, ... path_name_length})
    (Make_full_path_name[i+2] = path_name[i]);
  (∀ i ≥ path_name_length+2) (Make_full_path_name[i] = "undefined");

V_function_macro Parent_path(full_path): full_path_name_type

derivation
  if full_path_length = 0
    then Parent_path = "undefined";
  else (∀ i ∈ {1, 2, ... , full_path_length-1})
    (Parent_path[i] = full_path[i]);
    (∀ i ≥ full_path_length) (Parent_path[i] = "undefined");
  end

V_function_macro Path_accessible(full_path): boolean

derivation
  if full_path_length = 0
    then Path_accessible = true;
  else Path_accessible =
    if Path_accessible(Parent_path(full_path))
    then Entry_defined(Path_to_uid(Parent_path(full_path)),
      full_path[full_path_length]) &
      Dominates(Device(SSO).access_level,
        Directory(Path_to_uid(Parent_path(full_path)),
          full_path[full_path_length]).access_level);
    end
  end

V_function_macro Path_to_uid(full_path): uid_type

derivation
  if full_path_length = 0
    then Path_to_uid = root_uid;
  else Path_to_uid = Directory(Path_to_uid(Parent_path(full_path)),
    full_path[full_path_length]).uid;
  end

```

Figure 2. SSO V-function Macros


```

/* SSO Hidden_V_function */
Hidden_V_function Mount_request(time): mount_request_type

/* SSO V-function */
V_function SSO_access_level(<terminal>): access_level_type
exception
    ^terminal = SSO;
derivation
    Device(SSO).access_level;

```

Figure 3. SSO V-functions

SSO OV-function

The SSO has one OV-function, "Read_mount_request," as shown in Figure 4. This function has as its value the earliest mount request, and deletes that request from the list once it has been returned. The exceptions check that the caller is at the SSO terminal, that the SSO is executing at "system_high," and that there is a mount request to return.

SSO O-functions

The SSO O-functions appear in Figure 5. These O-functions manipulate kernel information but are trusted to do so securely; exceptions check the consistency of the arguments.

"Set_segment_al" is an SSO function used to declassify information by changing the level of the segment that holds it. Although in the people/paper world individuals are allowed to do a form of declassifying by extracting paragraphs from documents, the analogous mechanism cannot be supported here due to the granularity of the information protection unit.

The exceptions insure: that the caller is at the SSO terminal; that the segment specified is accessible by the SSO at its current access level; that the segment specified is not the root; that the new access level remains increasing as you move down the hierarchy; that the segment is not a non-empty directory; and that segment does have terminal quota.

The effect is simply to change the access level of the segment, and to revoke any current access each process has to the segment.

"Set_device_al" is used by the SSO to inform the kernel of changes in device usage. An example is changing of paper in a printer to allow it to print information at a different security level; the kernel must be given this external information by the SSO.

The exceptions check that the function is being invoked by the SSO terminal, that the new access level is within the bounds allowed for the device, and that no process is using the device.

This function can also be used to change the level of the SSO terminal, since this terminal is considered a device in the specification. The SSO terminal has a device_id of "SSO", so "Device(SSO)" is the SSO terminal.

The "Remove_upgraded_quota" function is used by the SSO to remove quota from an upgraded segment and return it to its parent. This

```

/* SSO OV-function */

OV_function Read_mount_request(<terminal>): mount_request_type

exception
    ^terminal = SSO;
    Device(SSO).access_level ≠ "system_high";
    Mount_request = "undefined";

effect
    Mount_request(min{(itime)('Mount_request(itime)^="undefined")}) =
        "undefined";

derivation
    'Mount_request(min{(itime)('Mount_request(itime)^="undefined")});

```

Figure 4. SSO OV-function

```

/* SSO 0-functions */

0_function Set_segment_al(path_name, access_level, <terminal>)

exception
  ^terminal = SSO;
  ^Path_accessible(full_path_name);
  Path_to_uid(full_path_name) = root_uid;
  ^Dominates(access_level, Dir_branch_path.access_level);
  Branch_path.type = "directory" &
    (!entry)(Entry_defined(Branch_path.uid, !entry));
  Branch_path.quota_given ≤ 0;

effect
  Branch_path.access_level = access_level;
  (∀iprocess_id, iseg)
    if 'Process(iprocess_id).KST(iseg).uid = Path_to_uid(full_path_name)
      then Process(iprocess_id).KST(iseg) = "undefined";
    end

0_function Set_device_al(device_id, access_level, <terminal>)

exception
  ^terminal = SSO;
  ^Dominates(Device(device_id).max_al, access_level);
  ^Dominates(access_level, Device(device_id).min_al);
  ^Dominates(Device(SSO).access_level, Device(device_id).access_level);
  Device(device_id).owner ≠ "undefined";

effect
  Device(device_id).access_level=access_level;

```

Figure 5. SSO 0-functions

```

O_function Remove_upgraded_quota(path_name, quota, <terminal>)

exception
    ^terminal = SS0;
    ^Path_accessible(full_path_name);
    Path_to_uid(full_path_name) = root_uid;
    ^quota > 0;
    (Branch_path.type = "directory") &
        (Dir_branch_path.sons_vol_id ≠ Branch_path.sons_vol_id);
    Branch_path.quota = 0;
    Branch_path.quota - quota < Branch_path.quota_used;
    Branch_path.quota_given - quota ≤ 0;

effect
    Branch_path.quota_given = 'Branch_path.quota_given - quota;
    Branch_path.quota = 'Branch_path.quota - quota;
    Dir_branch_path.quota = 'Dir_branch_path.quota + quota;

O_function Set_Drive(drive_no, vol_id, <terminal>)

exception
    ^terminal = SS0;
    Device(SS0).access_level ≠ "system_high";
    (‡ iprocess)(Process(iprocess).mount_list(Drive(drive_no)));

effect
    Drive(drive_no) = vol_id;

O_function Set_mount_list(process_id, vol_id, <terminal>)

exception
    ^terminal = SS0;
    Device(SS0).access_level ≠ "system_high";
    ^Process(process_id) ≠ "undefined";

effect
    Process(process_id).mount_list(vol_id) = "true";

```

Figure 5. SS0 O-functions (Concluded)

function cannot be provided at the kernel interface because it violates the *-property. Although it would be reasonable to provide this function to all users, in the interest of maintaining a simple specification and implementation of trusted subjects only the SSO is allowed to execute it.

In addition to checking that the SSO invoked the function, the exceptions ensure: that the specified segment is accessible to the SSO at its current access level; that the segment is not the root; that a positive amount of quota is to be removed; that the segment is not a master directory (quota cannot be removed from a master directory); and that if the quota will remain non-zero it will cover the quota used.

The effect of "Remove_upgraded_quota" is to reduce the quota and the quota_given in the segment's parent by the specified amount, and to increase the quota field of the segment's parent's parent.

The last two O-functions are used by the SSO in processing mount requests. After the SSO has used the OV-function "Read_mount_request" to find what process wants what logical volume mounted, he mounts the appropriate disk(s). Once he has done the mounting, he uses the function "Set_drive" to tell the kernel what drive(s) the logical volume is mounted on. He then uses the function "Set_mount_list" to notify the requesting process that the volume has been mounted.

"Set_drive" has as arguments a drive number and a logical volume id. The exceptions insure that the caller is the SSO and that there is no process that currently has a volume mounted on the specified drive. The effect is to save the name of the volume mounted on the specified drive.

"Set_mount_list" has as arguments the name of the process that want the volume mounted and the name of the volume. The exceptions check that the caller is the SSO and that the specified process exists. The effect is to set the mount list of the specified process "true" for the specified volume.

Design Considerations

As has been noted, the users of the functions provided by this interface are trusted to preserve the integrity of the system and, therefore, none of the usual kernel access control checks are made. These functions are not required to correspond to the mathematical model of security because they represent an implementation requirement that is not addressed by the model.

The existence of an SFEP to handle terminal I/O is not reflected by the top-level specification of trusted subjects, since the trusted subject specification describes the interface provided by the cooperative effort of the SFEP and host. Trusted subjects will be implemented as unified functions through appropriate communication between the certified software on both machines. Since the software implementing trusted subjects, in both machines together, must be verified to implement the single top-level specification given, it is inappropriate at the top level to attempt to specify separately the responsibilities allocated to each machine.

SSO REVIEW

The SSO fulfills an important requirement: interfacing the security kernel with the external environment. Because such a requirement does not exist in the current Multics, the SSO interface does not represent an incompatibility with current Multics. It is instead, an addition to the functionality of the current Multics system.

SECTION III

RECONFIGURATION

A Multics system is made up of a number of different types of hardware modules, such as CPUs, memory modules, and I/O modules. These modules must be interconnected in a very precise manner, and once connected, all the modules can be used in running Multics. However, modules can be connected but not used in running Multics. The operator is allowed to reconfigure the system by specifying which of the connected modules should be used. One major use for reconfiguration is to allow modules that need service to be removed from system use without having to stop Multics.

This section deals with the reconfiguration of certain hardware modules of the Multics computer system. In this section we review the current Multics reconfiguration design. Next, we present the kernel design and consider compatibility issues, and finally we give a detailed specification of the reconfiguration top-level interface functions.

THE CURRENT DESIGN

The current Multics reconfiguration design provides operator functions to handle the reconfiguration of the major hardware modules. Each of the modules will be identified and described. Then, the permissible physical and logical connections of the modules will be discussed, and the operator reconfiguration functions dealing with each type of module will be described.

Major Hardware Modules

The hardware modules that are handled by reconfiguration are described briefly below. A more detailed description is given in the Multics Reconfiguration Program Logic Manual [1].

A processor, or CPU, is a major processing unit, and is one of three types of modules called active modules. The other two types of active modules, the IOM and the bulk store, are defined below.

An IOM is an input/output controller. It is another of the active modules.

A bulk store is the third kind of active module. It provides auxiliary memory for paging, and is sometimes referred to as the paging device.

A system controller is a non-active hardware module that interfaces an active module to the memory of the configuration. The system controller also manages system interrupts and contains the system calendar clock. The system controller is often referred to as the controller. Since the system controller interfaces the active modules to the memory and hence provides memory functions to its users, the system controller is also often referred to as a memory. Since this report was researched a new system controller was announced and is being offered by Honeywell. This new controller, often referred to as the "four megaword controller", is not considered in this report.

Hardware Module Connections

A port is a connection point for two hardware modules. A controller port, or memory port, is a port on a system controller for connection to an active module. A processor port, or CPU port, is a port on a processor for connection to a system controller.

All active modules (CPU, IOM, bulk store) are connected to the system via the system controllers. Each active module is connected to every system controller at the same port.

In other words, if a given CPU is connected to port 1 on one controller, then it must be connected to port 1 on the other controllers as well. This restriction is not due to hardware, but to software convention. The relationships between connected memories and CPUs are best described in terms of the notion of control.

A control processor is a processor that is allowed to change the port control and interrupt masking values of a system controller. A processor that can change these values has some degree of control over the controller. Each system controller has one and only one control processor, although a processor can be a control processor for more than one system controller. There is a switch on each system controller (the Execute Interrupt Mask Assignment, or EIMA, switch) that defines the control processor for that controller.

A CPU is usually connected to several controllers, each of which connect the CPU to some memory. The CPU uses a controller to access the memory the controller interfaces. When a CPU needs some other controller function, such as sending interrupts, there is, by software convention, a specific controller that the CPU always uses. This controller is called the control memory for the CPU. Each memory with its EIMA switch selecting a particular CPU is potentially a control

memory for that CPU, but only one of these memories is actually chosen as the control memory. Each CPU must have a control memory, and a memory cannot be a control memory for more than one CPU. Therefore, there must be at least as many memories in the configuration as there are CPUs.

Figure 6 illustrates a Multics system with two processors and three memories/system controllers, showing the control processors and control memories. CPU A is connected to each memory on memory port 7, and CPU B is connected on memory port 6. Memory A is connected to each CPU on CPU port 0, Memory B on CPU port 1, and Memory C on CPU port 2. The EIMA switch on memory A is set to port 7 and selects CPU A as control processor. The EIMA switch on memory B is set to port 7 and also selects CPU A as control processor. The EIMA switch on memory C is set to port 6 and selects CPU B as control processor. CPU A is control processor for two memories, so each of these two memories is a potential control memory. By convention, only one of these memories is used as control memory, and memory A has been chosen in this example. CPU B is control processor for only one memory, so that memory is control memory for CPU B.

Now that we have presented the relevant terminology, we can describe the operator reconfiguration functions for memories, CPUs, and bulk store.

Memory Reconfiguration

Memory Reconfiguration involves adding or deleting a system controller (memory) from the current configuration. The names of all the memories in the system must be specified during initialization, but they are not necessarily configured at that time. The memories that exist and are specified at initialization time may be added or deleted after initialization using operator commands. At least one memory must be configured at initialization time. This memory is called the bootload memory.

The operator command for adding a memory specifies the name of the memory to be added, and optionally specifies the name of a particular CPU to be the control processor for the new memory. The operator is prompted to perform certain actions in the course of the execution of this command.

The operator command for deleting a memory specifies the name of the memory to be deleted. The bootload memory cannot be deleted. If the memory deleted was the control memory for some CPU, the operator will be requested to make switch settings on the memory that is to be the new control memory for that CPU.

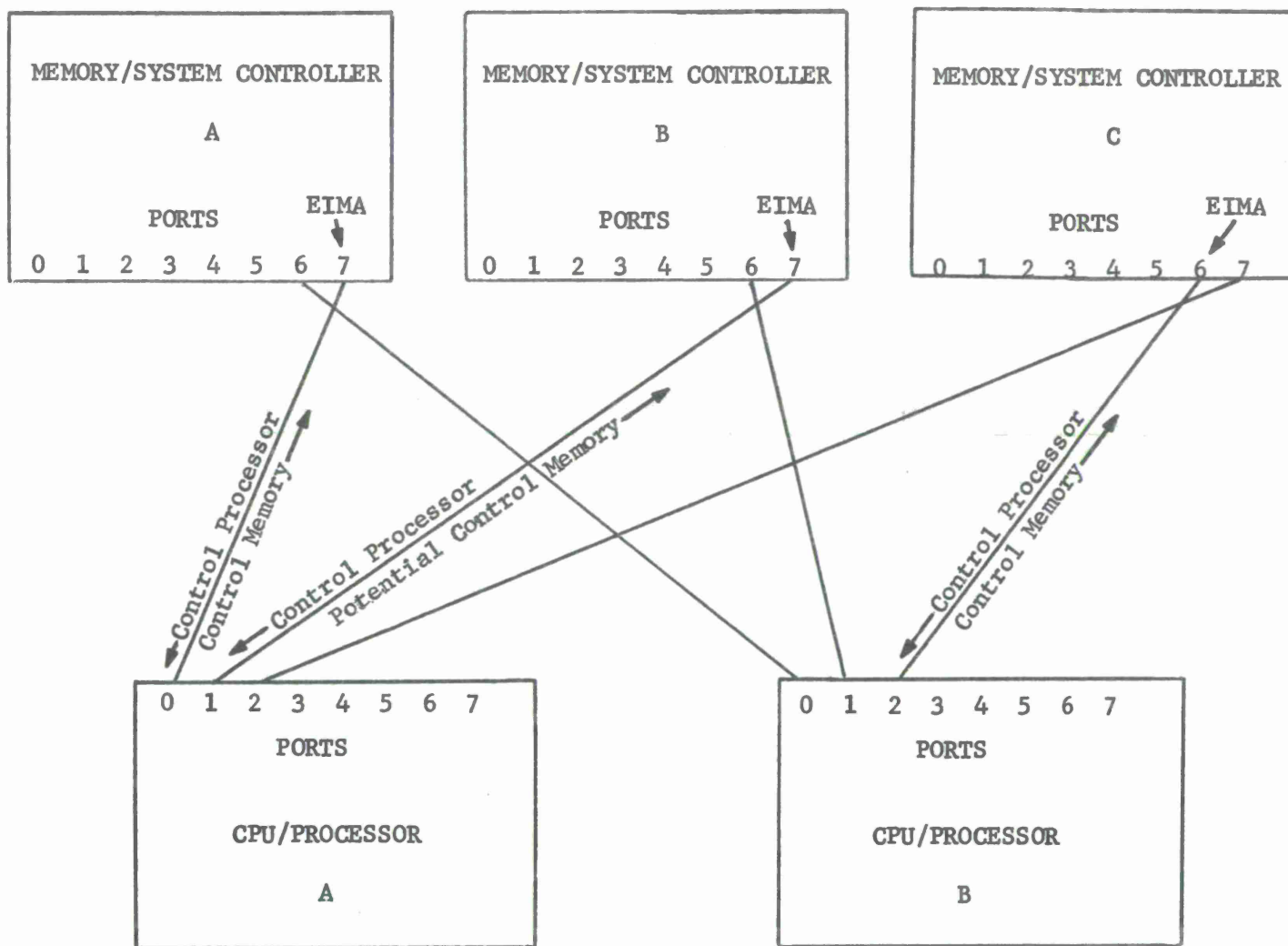


Figure 6. Hardware Module Connections

Processor Reconfiguration

The Multics system can run with more than one CPU, but when it is initialized, only one CPU (called the bootload CPU) is running. Any other CPU's desired must be added by using the reconfiguration functions. Similarly, system shutdown occurs with only one CPU running, so the processor reconfiguration functions are a regular part of multiple-CPU system operation.

The operator function for adding a CPU specifies the name of the CPU, the system controller port to which it is connected, and the name of its control memory. Certain operator actions are necessary during this command, and the operator is prompted to perform them.

The operator command for deleting a CPU specifies the name of the CPU to be deleted. The operator is prompted to change some switches on all memories that are controlled by the CPU being deleted.

Bulk Store Reconfiguration

A bulk store is used as a paging device in the Multics system. Bulk store reconfiguration is different from other types of reconfigurations in that it deals with bulk store records rather than with the bulk store as a whole. There is only one bulk store in a system, and the records of the bulk store each hold one page of memory. The records are added or deleted from the current configuration by operator commands.

The operator command for adding bulk store records specifies the first record to be added and the number of records to be added. The command is allowed if the specified range of records exists in the bulk store. Thus, it is legal to specify the addition of a record that is already configured, making it easy to add a large block of records without knowing exactly which records in the block are already configured.

The operator command for deleting bulk store records specifies the first record to be deleted and the number of records to be deleted. For convenience, it is legal to call for deletion of records that are already deleted. If all records in the paging device are deleted, then the device may be disconnected from the system for repairs.

It should be noted that paging device record deletion is also done automatically by system software if it is determined that a certain record is bad (i.e., causes read errors).

THE KERNEL DESIGN

The kernel provides reconfiguration O-functions that support all the functions currently available to the operator to perform reconfiguration. Hardware reconfiguration must be performed by the kernel because the hardware "belongs" to the kernel - the kernel uses the hardware to create objects available at the kernel interface. The data manipulated by the kernel in performing reconfigurations has been assigned an access level of "system_high," so only processes with access levels of "system_high" may use the reconfiguration functions.

A major consideration in the design of the reconfiguration functions is the handling of operator interactions. In the current design, the operator is prompted to perform switch settings on controllers and processors in the course of a reconfiguration. This prompting of the operator is undesirable for two reasons. First, since kernel functions are indivisible, prompting in the middle of a function causes specification problems. Second, having the kernel rely on an operator's actions to work properly is not reliable. The procedure of performing a reconfiguration with operator actions is error-prone. Therefore, in an effort to simplify the kernel and provide for more error free operation, we have attempted to remove the necessities of operator prompting.

The major cause of operator prompting and switch settings is the Execute Interrupt Mask Assignment (EIMA) switch. As mentioned above, this switch is used to specify the control processor for each controller. Each controller has four EIMA switches, each of which may be enabled by software. In the current design, only one of the switches is enabled, so the setting on the enabled switch determines the control processor for the controller. During some reconfigurations, the setting of the switch must be changed to select a different control processor.

An example of the need for changing an EIMA switch is the delete CPU operator function. When a CPU that is the control processor for some system controller is deleted, then the EIMA switch on that system controller must be changed to specify some other CPU as a control processor. In the current design, the operator is told which controller to set the switch on, and what the switch setting should be. Referring to our previous example shown in Figure 6, if CPU B is deleted, the EIMA switch on memory C (which was controlled by CPU B), must be changed to select port 7, and hence CPU A, as control processor.

The kernel design removes the requirement for operator prompting and switch settings in two ways. First, a non-hidden V-function that contains information about the current configuration is provided. The

information in this V-function can be used by the operating system to prompt the operator before the reconfiguration O-function is called. Second, a new convention for the use of EIMA switches is employed by the kernel. Since there are four EIMA switches on each controller, each switch can be set to select a different processor before the system is initialized. The EIMA switch settings are the same for each controller, i.e., if EIMA switch 1 on controller A selects processor B, then EIMA switch 1 on all other controllers also selects processor B. Once the switches are set on all controllers, and the kernel is informed of the setting of the switches, the kernel can select a control processor for each controller by enabling only the appropriate EIMA switch in software. The kernel is informed at initialization time of the EIMA switch settings. Only processors that have EIMA switches selecting them may be configured, so this convention imposes the restriction that only four CPUs may be configured.¹

Since we have avoided operator interaction problems, the kernel O-functions can perform the reconfigurations present in the current design with no interruptions.

COMPATIBILITY WITH THE CURRENT MULTICS

Since reconfiguration functions are performed only by operators, compatibility is not as great an issue as it is with more user oriented subsystems. The reconfiguration functions provided by the kernel roughly correspond to the operator functions provided in the current design. The main incompatibilities are in the area of operator interaction, and have been discussed in the previous section.

SPECIFICATION

This section presents a detailed description of the V-functions and O-functions that define the top-level specification of the kernel reconfiguration design.

Data Types, Parameters, and Constants

Figure 7 shows the data types, parameters, and constants used in the reconfiguration top-level specification.

There are five non-standard data types used in this specification. `Processor_port_number` is the port number of a CPU port, to

¹This restriction does not seem to be too great, since newer versions of the Multics reconfiguration software impose this same restriction.

type

```
processor_port_number = integer (0 to max_processor_port);
memory_port_number = integer(0 to max_controller_port);
processor_index = integer(0 to max_processor_index);
EIMA_switch_number: integer (1 to 4);
controller_data_type = structure
    (exists: boolean /* true if memory exists */
    configured: boolean /* true if mem currently configured */
    abs_wired: boolean /* true if segment can contain abs_wired
        segments and hence cannot be deconfigured */
    control_processor: cpu_id /* control processor for this mem */
    controlled_proc: cpu_id); /* processor for which this
        controller is control memory; "undefined" if this
        memory is not a control memory */
processor_data_type = structure
    (configured: boolean /* true if cpu is currently configured */
    control_memory: processor_port_number /* processor port
        number of the control memory for this cpu. */
    memory_port: memory_port_number /* this field tells to which
        memory port the processor is attached. It is attached to
        the same memory port on each system controller. */
    EIMA_switch: EIMA_switch_number); /* This field tells which
        EIMA (if any) points to this cpu. */
```

parameter

```
controller: processor_port_number;
record: integer (0 to 2word_length-1);
count: integer (1 to 2word_length-1);
cpu_id: processor_index;
mem_port: memory_port_number;
control_mem: processor_port_number;
control_proc: processor_index;

icpu: processor_index;
imem: processor_port_number;
```

constant

```
max_processor_port: integer;
max_controller_port: integer;
max_processor_index: integer;
```

Figure 7. Reconfiguration Data Types, Parameters, and Constants

which a system controller (memory) is attached. `Memory_port_number` is the system controller (memory) port number to which a CPU, IOM, or bulk store controller is attached. `Processor_index` is a name for a CPU.

`Controller_data_type` is a structure that contains data about all the possible controllers or memories in the system. `Exists` is true if the memory exists, as specified at initialization time. `Configured` is true if the memory is configured. `Abs_wired` is true if the memory contains `abs_wired` segments.² `Control_processor` specifies the control CPU for this memory, and `controlled_proc` specifies the CPU for which this is control memory.

`Processor_data_type` is a structure that contains data about all the possible CPUs in the system. `Configured` is true if the CPU is configured. `Control_memory` specifies the control memory for this CPU. `Memory_port` specifies the memory port (on all memories) to which this CPU is attached. `EIMA_switch` is undefined if this CPU is not specified on any EIMA switches; otherwise, it is the number of the EIMA switch on which the CPU is selected.

The parameter section defines the data types of the arguments used in the specification. `Controller` is a `processor_port_number`, and serves to identify a specific system controller (memory). `Record` and `count` are used to specify a paging device record number and number of records, respectively, for the paging device reconfiguration functions. `Cpu_id` is a `processor_index`, and serves to identify a specific CPU. `Mem_port` is a `memory_port_number` for some CPU. `Control_mem` is the `processor_port_number` of some control memory, and `control_proc` is the name of some control processor. `Icpu` and `imem` are used as quantified variables in the specification.

The constant section lists certain constant arguments whose actual value is irrelevant to the security of the top level. The constants listed are the maximum number of processor ports, controller ports, and CPU's, respectively.

Reconfiguration V-functions

Figure 8 contains the hidden and non-hidden V-functions in the specification. The hidden V-functions, listed first, serve as data bases internal to the kernel and not available at the kernel interface. The first four hidden V-functions contain data about the major

²`Abs_wired` segments are segments that are permanently core resident. If a memory contains any `abs_wired` segments then it cannot be deleted. Normally only the bootload memory contains `abs_wired` segments.


```

Hidden_V_function Controller_data(controller): controller_data_type;

Hidden_V_function Processor_data(cpu_id): processor_data_type;

Hidden_V_function IOM_memory_port: memory_port_number;
    /* This function tells to which memory port the IOM is attached.
       It is attached to the same memory port on each controller */

Hidden_V_function Pd_size: integer; /* # of pages on paging device */
    size of paging device */

Hidden_V_function Nprocessors: integer(1 to max_processor_index+1);

Hidden_V_function Nmemories: integer(1 to max_processor_port+1);

/* Interface (non-hidden, derived) V-function */

V_function Configuration: structure
    (controllers(controller): controller_data_type
     processors(cpu_id): processor_data_type);

exception
    Cur.access_level ^= "system_high";

derivation
    controllers = Controller_data;
    processors = Processor_data;

```

Figure 8. Reconfiguration V-functions

hardware modules. `Controller_data` contains data about all possible system controllers (memories) in the system, as defined by `controller_data_type`. Similarly, `Processor_data` contains data about the CPUs in the system. `IOM_memory_port` is the memory (system controller) port to which the IOM is attached. This value is set at initialization time and does not change. `Pd_size` is the size, in pages, of the paging device. This value is set at initialization time and does not change.

The last two hidden V-functions, `Nprocessors` and `Nmemories`, are counts of the number of CPUs and memories currently in the configuration, respectively.

There is one non-hidden interface V-function in this specification, which provides information about the current configuration for use by uncertified software executing at the operator's request. This V-function, called `Configuration`, is derived from `Controller_data` and `Processor_data`.

Memory Reconfiguration O-functions

Figure 9 shows the functions `Add_memory` and `Delete_memory`. `Add_memory` takes as arguments the name (`processor_port_number`) of the controller (memory) to be added, and an argument to specify the CPU to be used as control processor for this memory. The function requests that the specified memory be added with the specified CPU as control processor.

The exceptions for `Add_memory` make sure that the current access level is "system_high", that the specified controller exists and is not already configured, and that the control processor is configured.

The effects of the function include the following. `Configured` is set for the specified memory. `Abs_wired` is set to false for the specified memory, because it will contain not `abs_wired` segments. `Control_processor` is set from the specified second argument, and the count of configured memories is incremented.

`Delete_memory` takes two arguments also. The first argument is the name of the memory (controller) to be deleted. The second argument is the name of another memory to be used as control memory for any CPUs for which the deleted memory was a control memory. Each CPU must have a control memory. The second argument can be left undefined if this deleted memory was not a control memory. Uncertified software can determine from the V-function `Configuration` controllers whether or not the memory to be deleted was a control memory. If it was, the second argument must be supplied.


```

/* Memory (system controller) Reconfiguration O-functions */

O_function Add_memory(controller, control_proc)

exception
  Cur.access_level ^= "system_high";
  ^Controller_data(controller).exists;
  Controller_data(controller).configured;
  ^Processor_data(control_proc).configured;

effect
  Controller_data(controller).configured = "true";
  Controller_data(controller).abs_wired = "false";
  Controller_data(controller).control_processor = control_proc;
  Controller_data(controller).controlled_proc = "undefined";
  Nmemories = 'Nmemories + 1;

O_function Delete_memory(controller, control_mem)

let
  controlled_proc = Controller_data(controller).controlled_proc;
  control_memory = Processor_data(controlled_proc).control_memory;

exception
  Cur.access_level ^= "system_high";
  Nmemories = Nprocessors; /* Can't delete any more memories */
  ^Controller_data(controller).configured;
  Controller_data(controller).abs_wired;
  controlled_proc ^= "undefined" &
    (^Controller_data(control_mem).configured |
     Controller_data(control_mem).controlled_proc ^= "undefined");

effect
  Controller_data(controller).configured = "false";
  Nmemories = 'Nmemories -1;
  if controlled_proc ^= "undefined"
    then control_memory = control_mem;
      Controller_data(control_mem).controlled_proc =
        controlled_proc;
    end;
end;

```

Figure 9. Memory Reconfiguration O-functions

The exceptions for Delete_memory check that the current access level is "system_high," and that the number of memories configured is not equal to the number of CPUs configured. If these counts are equal, there will not be enough memories for control memories if a memory is deleted. There must always be at least as many memories as there are CPUs in the configuration.

The exceptions also check that the memory is not abs_wired. Finally, if the control_mem argument is needed, then control_mem must be configured and must not be a control memory already.

The effects of Delete_memory are as follows. The configured flag is set to false for the indicated memory. The count of the number of memories is decremented. If the deleted memory was controlling some CPU, then control_mem (the second argument) is made the control memory for that CPU.

Paging Device Reconfiguration O-functions

Figure 10 contains the two paging device (bulk store) reconfiguration O-functions. As mentioned earlier, bulk store reconfiguration deals with bulk store records rather than with the bulk store itself.

The two functions each take as arguments the number of the first record to be added or deleted, and the number of records to be added or deleted. The exceptions for both functions are the same: to check that the current access level is "system_high", and that the range of records specified lies within the bulk store.

These functions have no visible effects at the interface level. In other words, a call to these functions has no effect on the results of any future calls, because it is legal to add already configured records and to delete already deleted records. The only possible top-level effect is on the speed of the system, but time is not observable at the top level.

CPU Reconfiguration O-functions

Figure 11 contains the CPU reconfiguration O-functions, Add_cpu and Delete_cpu.

Add_cpu requires three arguments: the name of the CPU to be added, the control memory for that CPU, and the memory port to which that CPU is attached. The exceptions for the function make sure that: the current access level is "system_high", the specified CPU is not already configured, the specified control memory is configured and is not already a control memory, and that the specified memory port is not already being used for some processor or for the IOM.

```
/* Paging Device Record Reconfiguration O-functions */
```

```
O_function Add_pd_records(record, count)
```

```
exception
```

```
Cur.access_level ^= "system_high";
```

```
record+count-1 > Pd_size;
```

```
O_function Delete_pd_records(record, count)
```

```
exception
```

```
Cur.access_level ^= "system_high";
```

```
record+count-1 > Pd_size;
```

Figure 10. Paging Device Reconfiguration O-functions

```

/* Processor (CPU) Reconfiguration O-functions */

O_function Add_cpu(cpu_id, control_mem, mem_port)

exception
  Cur.access_level ^= "system_high";
  Processor_data(cpu_id).configured;
  Processor_data(cpu_id).EIMA_switch = "undefined";
  ^Controller_data(control_mem).configured;
  Controller_data(control_mem).controlled_proc ^= "undefined";
  (†icpu) (Processor_data(icpu).memory_port = mem_port);
  IOM_memory_port = mem_port;
effect
  Processor_data(cpu_id).configured = "true";
  Processor_data(cpu_id).control_memory = control_mem;
  Processor_data(cpu_id).memory_port = mem_port;
  Controller_data(control_mem).controlled_proc = cpu_id;
  Nprocessors = 'Nprocessors + 1;

O_function Delete_cpu(cpu_id, control_proc)

let
  control_memory = Processor_data(cpu_id).control_memory;

exception
  Cur.access_level ^= "system_high";
  Nprocessors = 1; /* can't delete last cpu */
  ^Processor_data(cpu_id).configured;
  ^Processor_data(control_proc).configured;

effect
  Processor_data(cpu_id).configured = "false";
  Nprocessors = 'Nprocessors - 1;
  Controller_data(control_memory).controlled_proc = "undefined";
  (‡imem) if Controller_data(imem).control_processor = cpu_id
    then Controller_data(imem).control_processor = control_proc;
  end;

```

Figure 11. CPU Reconfiguration O-functions

The effects of `Add_cpu` are as follows. The configured flag is set for the specified CPU, and the `control_memory` and `memory_port` fields are filled in. Also, the `controlled_proc` field on the specified control memory is set to the CPU being added. Finally, the number of processors configured is incremented.

The O-function `Delete_cpu` takes two arguments. The first is the name of the CPU to be deleted. The second argument is the name of another CPU to be used as control processor for any memories for which the deleted CPU was control processor. This argument can be "undefined" if the deleted CPU was not a control processor.

The exceptions for `Delete_cpu` insure that: the current access level is "system_high", there is more than one CPU configured, the specified CPU is configured, and that if needed, the specified `control_proc` is configured.

The effect of `Delete_cpu` is as follows. Configured is set to false for the specified CPU. The number of processors is decremented. The `controlled_proc` field for the memory that was controlling the CPU is set to "undefined". Each memory that was controlled by the specified CPU is set to be controlled by `control_proc`.

RECONFIGURATION REVIEW

In this section we have described the kernel interface functions dealing with the reconfiguration of the various hardware modules of the Multics system. The current design of the hardware reconfiguration operator functions has been described, and we have seen that the kernel functions are basically compatible. The main area of incompatibility, that of operator interaction during reconfigurations, has been pointed out. We have also described the detailed specification of the kernel functions for memory (system controller) reconfiguration, paging device (bulk store) reconfiguration, and CPU (processor) reconfiguration.

SECTION IV

INITIALIZATION

Initialization is the process through which Multics is loaded into the computer and started running in its normal mode. This section discusses the problems involved with Multics system initialization. Initialization, very important to the security of the kernel, puts the kernel into a secure state, and the kernel O-functions map the kernel from one secure state into another. Once the system is in a secure state, it will stay secure.

In this section we shall discuss briefly how initialization is accomplished in the current Multics, then we will discuss initialization of the security kernel. Compatibility issues with the current Multics will be considered, and the specifications dealing with initialization will be presented.

THE CURRENT DESIGN

As it exists today, initialization of a Multics system is very complex and is difficult to verify. The initialization process is not well modularized and is extremely hard to understand and modify. The complexity of the design and coding makes verification infeasible.

The information necessary to initialize the Multics system resides in two places, the Multics System Tape (MST), and the CONFIG deck. The MST contains the programs and data that must be loaded to start Multics in any configuration, at any installation. The information on the MST is installation independent. The CONFIG deck contains the installation dependent data that is read by Multics to initialize itself in a particular configuration.

To initialize the system, the operator issues commands to the Bootload Operating System (BOS) [2]. At this time the operator specifies whether this initialization is to be a warm start or a cold start. A warm start (the most common), means the initial storage hierarchy is to be the one present at the last system shutdown. A cold start means the storage hierarchy is to be recreated.

At the command of the operator, BOS puts the CONFIG deck in a fixed place in memory and starts reading the MST. The segments on the MST are organized into three collections of data. Each collection, when loaded, initializes itself to provide a richer environment for the next collection to run in, and then reads in the next collection.

Through this reading and initializing of collections, the standard ring 0 environment is eventually achieved. A more detailed description of the initialization procedure can be found in [3].

One of the problems with this initialization scheme is that it is never really clear what environment a given initialization program runs in. The number of different environments makes verification infeasible.

THE KERNEL DESIGN

The initialization of a kernel based Multics system is slightly different from that of the current Multics system, and consists of two parts: initialization of the kernel, and initialization of the rest of the operating system. From the standpoint of security, we are concerned only with the initialization of the kernel. Once the kernel is securely initialized, the remainder of initialization may run with unverified code.

The scenario for initialization of a kernel based Multics system is as follows. First, the Bootload Operating System (BOS) will be retained. It is not known at this time how much of BOS can remain unverified, but it is certain that part of BOS must be verified, specifically, the portion that handles the CONFIG deck and the portion that loads the kernel tape.

To start initialization, BOS loads the kernel from a protected tape³ and initializes the kernel. This loading, initialization, and starting of the kernel may be accomplished in several ways. One might be to save a core image of the kernel after it has initialized itself, and simply load and start this (already initialized) core image. Once loaded and started at the appropriate point, the kernel would then proceed to load and initialize the rest of the operating system from another tape, similar to the MST in the current design. From the standpoint of security, we are mainly concerned with the loading and initializing of the kernel. The operating system resides on two tapes because one tape, which contains the kernel, must be protected and handled with special procedures. The same restrictions do not apply to the tape which contains the rest of the operating system.

The kernel representation will be placed on the protected tape so that when loaded into core and started in a specific place it will be secure. Thus, the kernel must be completely initialized before it is

³While tape is currently used as a storage medium for system software, we do not imply the kernel storage medium is restricted to tape.

placed on the protected tape. Once loaded, the kernel will be configured to the hardware using initialization configuration functions and information from the CONFIG deck. The CONFIG deck in the kernel design will be similar to the CONFIG deck in the current design, but will be expanded to contain some security-sensitive information. Processing the CONFIG deck and calling the initialization configuration functions must be done by verified code.

To verify that the system is initialized securely, we must verify that the kernel representation on the protected tape is correct. This verification may be performed in two ways. First, we may inspect the contents of the tape. This need be done only once for each copy of the system. Second, we can generate the tape with verified software. It is unclear which method is preferable.

V-function Initialization

Since the initial state of the kernel must be secure, we must have some way to specify the initial secure state in terms of the V-functions that define the kernel. We must therefore specify the initial values of all non-derived V-functions.

Specifying the initial values of the V-functions will define, for example, the initial hardware configuration and the initial state of the system.

Initialization Reconfiguration Functions

The initial secure kernel, as loaded from the protected tape, is a functional kernel, but will not reflect the hardware and software on which the system is to run. We must provide O-functions to modify the state of the kernel in a secure manner. The functions dealing with the hardware are described in the section on reconfiguration. There are other functions, however, that will be desirable.

Additional functions may be required to deal with changing the size and/or number of kernel data bases and to assimilate the data found on the CONFIG deck into the kernel configuration. Not all of the desirable functions have been identified yet, but some are included in the specification.

Lower Level Initialization

The functions shown in this specification are concerned only with the initialization of top level V-functions. There is some information that must be supplied at initialization time that is supplied to lower levels of the kernel, such as physical device addresses. This information is not treated in this specification.

COMPATIBILITY WITH THE CURRENT MULTICS

The kernel initialization scheme is not compatible with the current scheme because only part of the total initialization process will be verified. Multics initialization as a whole is different, because of the existence of two system tapes, one protected, and one not. Compatibility is not as much of an issue with initialization as it is with other, more user-oriented subsystems. An incompatible initialization interface effects only the operators, not the users.

The real compatibility issues, however, are concerned with the generality and installation independence of the information on the two tapes. The current MST is installation independent. The current initialization process is unstructured and ad hoc. The new initialization technique provides more structuring, and provides a standard (kernel) environment in which most of initialization can run. The standard environment is achieved by binding together most (if not all) of the kernel at the protected tape generation time, instead of at run time. It is possible that pre-binding may cause a slight loss of generality of the kernel tape, the extent of which is not known at this time.

SPECIFICATION

We will now describe in detail the specification of the top level of initialization. The initialization specification consists of parameter and constant definitions, an O-function to initialize the V-functions in the top level, and some initialization configuration O-functions.

The initialization O-functions are trusted functions (trusted subjects), because the security of their operation depends on the validity of their arguments. In other words, for secure operation, the arguments must be specified correctly. This restriction should not be a problem, however, because the arguments required by these functions are supplied by users who are trusted to perform correctly. The interface between these functions and their users is like the interface described in the previous section on the SSO.

Initialization Parameters and Constants

Figure 12 shows the parameters and constants defined for initialization. The first group of parameters defines the types of the arguments to the O-function "Initialize_top_level." The second group of parameters defines the types of the arguments for the initialization configuration O-functions.


```

/* Initialization parameters */

parameter

cold_start: boolean;
IOM_port: memory_port_number;
bootload_cpu: processor_index;
bootload_cpu_memory_port: memory_port_number;
bootload_memory: processor_port_number;
idevice_id: uid_type;
ientry: entry_type;
iprocess_id: uid_type;
iuid: uid_type;
paging_device_size: integer;

device_max_al, device_min_al, device_access_level: access_level_type;
vol_id: uid_type;
vol_access_level: access_level_type;
vol_#_of_packs, vol_quota: integer;

/* Initialization constants */

constant

SSO: uid_type;
initializer: uid_type;
initializer_process_data: process_type;
initial_interpreter_data: interpreter_data_type;
root_branch: branch_type;

```

Figure 12. Initialization Parameters and Constants

Five constants are defined for this specification. "SSO" is the unique identification of the System Security Officer (SSO) terminal. "initializer" is the unique ID of the initializer process, and "initializer_process_data" is the initial data about the initializer process. Data about the initial state of the interpreter is contained in "initial_interpreter_data". The constant "root_branch" is the branch that describes the root, and is stored in the root.

Initialize Top Level O-function

Figure 13 illustrates the O-function that initializes all non-derived V-functions in the top level specification. Initializing the non-derived V-functions also initializes the derived V-functions, since they are derived from non-derived V-functions. The effect section for this function lists the effects pertaining to each subsystem in the specification. The arguments to Initialize_top_level will be discussed along with the subsystem that uses them.

The effects pertaining to reconfiguration use the first nine arguments. These arguments must be specified correctly for the system to run correctly and for the reconfiguration functions to work securely and correctly. "IOM_port" is the memory port number of the IOM. "Bootload_CPU" is the processor index of the running CPU. "Bootload_CPU_memory_port" is the memory port to which this CPU is attached. "Bootload_memory" is the number of the processor port to which the bootload memory is attached. "EIMA_switch_" 1 to 4 are the processor indices of the CPUs selected by the four EIMA switches. "Paging_device_size" is the number of records in the paging device. The V-functions in the reconfiguration specification are initialized to reflect the hardware base of the system as specified by these five arguments.

"IOM_memory_port" saves the memory port number of the IOM for future error checking. "Processor_data" is initialized to reflect the current CPU configuration. Only the bootload CPU is configured. The control memory for the bootload CPU is the bootload memory, as specified by the O-function argument. The memory port to which the bootload CPU is attached is set from "bootload_CPU_memory_port". The "EIMA_switch" field for each processor is set to the number of the EIMA switch selecting that processor. If no EIMA switch is selecting the processor, then the "EIMA_switch" field is undefined.

"Controller_data" is set to reflect the initial set of memories. During this phase of initialization, only the bootload memory is considered to exist. The existence of other memories is made known with the "Define_system_controller" O-function (described below). The bootload memory is configured and contains abs_wired segments. The

```

/* Initialization O-functions */

O_function Initialize_top_level (IOM_port, bootload_cpu,
    bootload_cpu_memory_port, bootload_memory, EIMA_switch_1,
    EIMA_switch_2, EIMA_switch_3, EIMA_switch_4, paging_device_size,
    cold_start)

effect

/* Reconfiguration Initialization effects */

IOM_memory_port = IOM_port;
Processor_data(bootload_cpu).configured = "true";
Processor(bootload_cpu).control_memory = bootload_memory;
Processor_data(bootload_cpu).memory_port = bootload_cpu_memory_port;
(∀icpu≠bootload_cpu) Processor_data(icpu).configured = "false";
(∀icpu) if icpu=EIMA_switch_1 then Processor_data(icpu).EIMA_switch=1;
    else if icpu=EIMA_switch_2 then Processor_data(icpu).EIMA_switch=2;
    else if icpu=EIMA_switch_3 then Processor_data(icpu).EIMA_switch=3;
    else if icpu=EIMA_switch_4 then Processor_data(icpu).EIMA_switch=4;
    else Processor_data(icpu).EIMA_switch = "undefined";
end;
Controller_data(bootload_memory).exists = "true";
Controller_data(bootload_memory).configured = "true";
Controller_data(bootload_memory).abs_wired = "true";
Controller_data(bootload_memory).controlled_proc = bootload_cpu;
Controller_data(bootload_memory).control_processor = bootload_cpu;
(∀imem≠bootload_memory) Controller_data(imem).exists = "false";
Nprocessors = 1;
Nmemories = 1;
Pd_size = paging_device_size;

/* Interpreter Initialization effects */

Interpreter_data = initial_interpreter_data;

/* SSO Initialization effects */

Mount_request = "undefined";

```

Figure 13. Initialize_top_level O-function

```

/* External I/O initialization effects */

Device(SS0).max_al = "system_high";
Device(SS0).min_al = "system_low";
Device(SS0).access_level = "system_high";
Device(SS0).owner = "undefined";
Device(SS0).status = "undefined";
Device(SS0).buffer = "undefined";
(∀device_id ≠ SS0)(Device(idevice_id) = "undefined");

/* Process control initialization effects */

Cur_process = initializer;
Process(initializer) = initializer_process_data;
(∀iprocess_id ≠ initializer)(Process(iprocess_id) = "undefined");

/* Common initialization effects */

Audit_log = "undefined";

/* Storage Control initialization effects */

Drive = "undefined";
if cold_start then
  Directory(root_uid, "root") = root_branch;
  (∀ientry ≠ "root")(Directory(root_uid, ientry) = "undefined");
  (∀iuid ≠ root_uid) (∀ientry)
    (Directory(iuid, ientry) = "undefined");
  LVRF = "undefined";
end;

/* If this is a cold start, we must initialize with an empty tree.
otherwise, the tree is already there, having been initialized when
created. */

```

Figure 13. Initialize_top_level 0-function (Concluded)

control processor for the bootload memory is the bootload CPU, and the CPU controlled by the bootload memory is the bootload CPU.

The remainder of reconfiguration initialization sets the current number of processors to one, the current number of memories to one, and the size of the paging device to "paging_device_size".

The initialization of the interpreter involves setting the initial value of "Interpreter_data" to , "initial_interpreter_data."

The SSO is initialized by setting the list of mount requests to "undefined."

The initialization of external I/O defines the initial device configuration. "Device" contains information about one terminal, the SSO terminal. Information about other terminals is not stored in "Device", because they are considered SFEP devices, whereas the SSO terminal is considered a kernel device. The access level in the SSO terminal is system high. The SSO is the only device defined at initialization time. Other devices are defined using the "Define_device" configuration function described below.

Whether or not Multics needs separate operator's and SSO terminals is still an open question.

The initialization of process control defines the initial process, the initializer process. There are no other processes defined initially.

Initialization of the common V-functions is completed by setting "audit_log" to undefined.

The initialization of storage control starts by setting the information about the mounted volumes ("Drive") to "undefined". The remainder of storage control initialization depends on whether or not this is a cold start, as indicated by the argument "cold_start". If this is not a cold start, then the segment hierarchy and the Logical Volume Record File (LVRF) are assumed to be unchanged since the last shutdown, i.e., the storage system is intact on disk.

If this is a cold start, however, no hierarchy exists on disk, and a new one must be created by the kernel and the (unspecified) restore subsystem (which is trusted). For a cold start, the root is created with only one entry, which is a description of the root itself. No other segments or directories exist. They are created by the restore subsystem from a previously created backup. Also, the LVRF is set to undefined. Entries in the LVRF are defined by the "Define_LVRF" configuration function. Note that entries in the LVRF

cannot be removed. The LVRF can shrink in size only as the result of a cold start.

Initialization Configuration O-functions

Figure 14 illustrates configuration O-functions that are used for initialization. It is expected that most of these functions will not be called directly as the result of an operator command, but will be called by the verified software processing the CONFIG deck. For example, one of the initialization configuration functions already mentioned is "Define_system_controller". This function is called by verified software that is processing the CONFIG deck, to specify which system controllers exist, even though they may not yet be configured. The exceptions for all of these functions check that they are being called by the SSO terminal with a "system_high" access level.

The O-function "Define_device" provides a way to define what devices are available. An exception for this function checks that the maximum access level being assigned the device dominates the minimum for the device. The effect of this function is to set the minimum and maximum access levels for the device. The current "access_level", "owner", "status", and "buffer" of the device are set to "undefined".

"Define_LVRF" is used to specify an entry in the Logical Volume Record File (LVRF), which defines the logical volumes in the system. The effect of the function is to mark that the specified LVRF entry exists, but that the volume is not mounted. Also, the access level, number of packs, and quota of the logical volume are set.

The O-function "Define_system_controller" is used to make the existence of a system controller known to the system. System controllers not made known at initialization time cannot be configured later. The effect is to mark that the controller exists, but is not configured. The controller can eventually be configured using the hardware reconfiguration functions.

INITIALIZATION REVIEW

We have described, in this section, how Multics is currently initialized, and how the initialization process must be reorganized to accommodate a kernel based Multics system. The security kernel must be securely initialized, but the rest of initialization can be performed by unverified code. The V-functions that comprise the top-level specification are initialized by the O-function Initialize_top_level. Once the kernel is initialized, it is installation independent and must be reconfigured to the hardware using the initialization configuration functions.


```

/* Initialization Configuration O-functions */

O_function Define_device(device_id, device_max_al, device_min_al,
    <terminal>)

exception
    terminal ≠ SSO;
    Device(SSO).access_level ≠ "system_high";
    ~Dominates(device_max_al, device_min_al);

effect
    Device(device_id).max_al = device_max_al;
    Device(device_id).min_al = device_min_al;
    Device(device_id).access_level = device_min_al;
    Device(device_id).owner = "undefined";
    Device(device_id).status = "undefined";
    Device(device_id).buffer = "undefined";

O_function Define_LVRF (vol_id, vol_access_level,
    vol_quota, <terminal>);

exception
    terminal ≠ SSO;
    Device(SSO).access_level ≠ "system_high";
    LV_defined(vol_id);

effect
    LVRF(vol_id).mounted = "false";
    LVRF(vol_id).access_level = vol_access_level;
    LVRF(vol_id).quota = vol_quota;

O_function Define_system_controller(controller, <terminal>)

exception
    terminal ≠ SSO;
    Device(SSO).access_level ≠ "system_high";

effect
    Controller_data(controller).exists = "true";
    Controller_data(controller).configured = "false";

```

Figure 14. Initialization Configuration O-functions

APPENDIX I

INDEX TO SPECIFICATIONS

Functions

Add_cpu 34
Add_memory 31
Add_pd_records 33
Branch_path 10
Configuration 29
Controller_data 29
Define_LVRF 46
Define_device 46
Define_system_controller 46
Delete_cpu 34
Delete_memory 31
Delete_pd_records 33
Dir_branch_path 10
EIMA_switch_number 27
IOM_memory_port 29
IOM_port 40
Initialize_top_level 42
Make_full_path_name 12
Mount_request 13
Nmemories 29
Nprocessors 29
Parent_path 12
Path_accessible 12
Path_to_uid 12
Pd_size 29
Processor_data 29
Read_mount_request 15
Remove_upgraded_quota 17
SSO_access_level 13
Set_Drive 17
Set_device_al 16
Set_mount_list 17
Set_segment_al 16

Basic Definitions

bootstrap_cpu 40
bootstrap_cpu_memory_port 40
bootstrap_memory 40

cold_start 40
control_mem 27
control_proc 27
controller 27
controller_data_type 27
count 27
cpu_id 27
device_access_level 40
device_max_al 40
device_min_al 40
full_path 10
full_path_length 10
full_path_name 10
full_path_name_length 10
full_path_name_type 10
icpu 27
idevice_id 40
ientry 40
imem 27
iprocess_id 40
iuid 40
max_controller_port 27
max_processor_index 27
max_processor_port 27
mem_port 27
memory_port_number 27
mount_request_type 10
paging_device_size 40
path_name 10
path_name_length 10
path_name_type 10
processor_data_type 27
processor_index 27
processor_port_number 27
record 27
terminal 10
vol_#_of_packs 40
vol_access_level 40
vol_id 40
vol_quota 40

REFERENCES

1. Honeywell Information Systems, Reconfiguration Program Logic Manual, AN71, June 1974.
2. Honeywell Information Systems, Bootload Operating System (BOS) Program Logic Manual, AN74.
3. Honeywell Information Systems, System Initialization Program Logic Manual, AN70, February 1975.