Honeywell

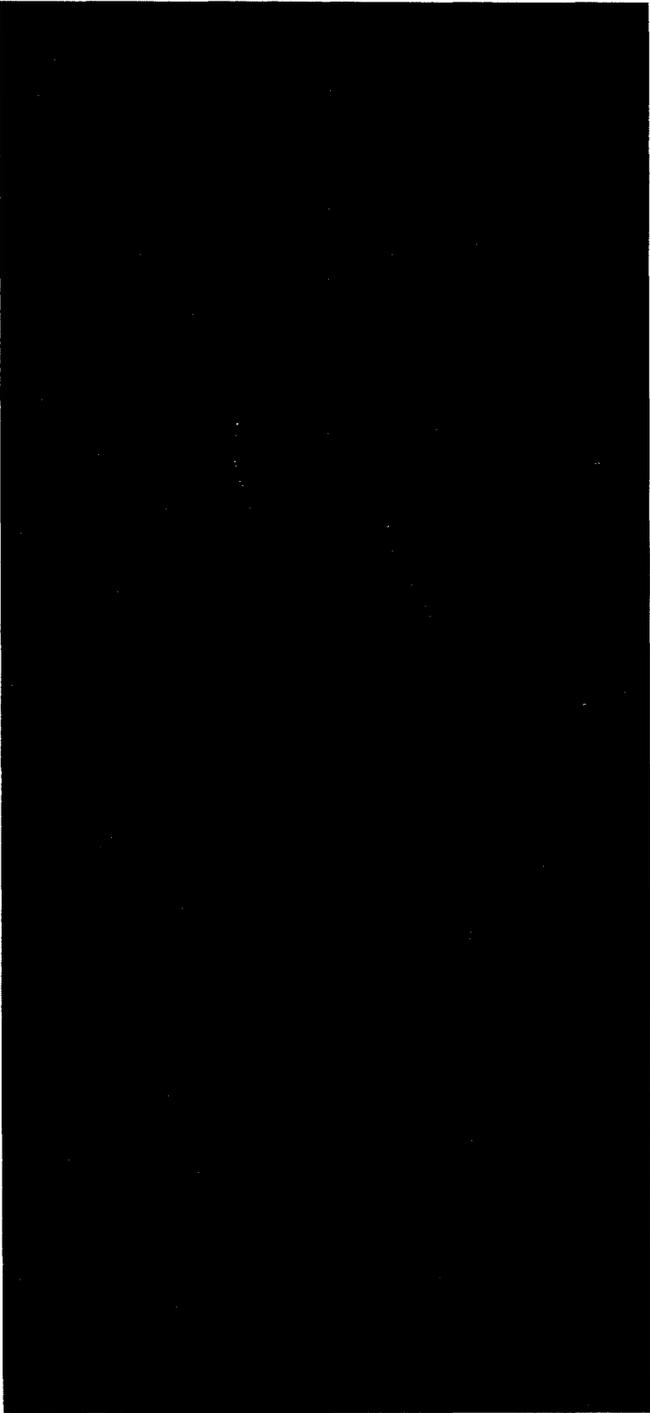RECONFIGURATION
PROGRAM LOGIC MANUAL

MULTICS

SOFTWARE

Honeywell

MULTICS

RESTRICTED DISTRIBUTION

SUBJECT:

Dynamic Reconfiguration Software for the Major Hardware Modules (Processor, System Controller, and Bulk Store).

SPECIAL INSTRUCTIONS:

This Program Logic Manual (PLM) describes certain internal modules constituting the Multics System. It is intended as a reference for only those who are thoroughly familiar with the implementation details of the Multics operating system; interfaces described herein should not be used by application programmers or subsystem writers; such programmers and writers are concerned with the external interfaces only. The external interfaces are described in the Multics Programmers' Manual, Commands and Active Functions (Order No. AG92), Subroutines (Order No. AG93), and Subsystem Writers' Guide (Order No. AK92).

As Multics evolves, Honeywell will add, delete, and modify module descriptions in subsequent PLM updates. Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This PLM is one of a set, which when complete, will supersede the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96).

DATE:

June 1974

ORDER NUMBER:

AN71, Rev. 0

PREFACE

Multics Program Logic Manuals (PLMs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation. They are not intended for application programmers or subsystem writers.

The PLMs contain descriptions of modules that serve as internal interfaces and perform special system functions. These documents do not describe external interfaces, which are used by application and system programmers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions. To help maintain accurate PLM documentation, Honeywell publishes a special status bulletin containing a list of the PLMs currently available and identifying updates to existing PLMs. This status bulletin is distributed automatically to all holders of the System Programmers' Supplement to the Multics Programmers' Manual (Order No. AK96) and to others on request. To get on the mailing list for this status bulletin, write to:

Large Systems Sales Support
Multics Project Office
Honeywell Information Systems Inc.
Post Office Box 6000 (MS A-85)
Phoenix, Arizona 85005

File No.: 2L13

AN71

CONTENTS

SECTION I

INTRODUCTION

This document describes the implementation and design of the Multics dynamic reconfiguration software for the major hardware modules of the system. Although there are many more hardware and software switchable modules in the system, this document is limited to processor, system controller and bulk store memory reconfiguration.

Dynamic reconfiguring in Multics, on a per-module basis, is done only under explicit operator request. The facility of the system that automatically deconfigures selected subregions of core or bulk store when hardware problems arise uses the same basic mechanism as module deconfiguration where appropriate. There is currently no way the system will automatically deconfigure a faulty processor. The software to automatically deconfigure core is incomplete. The software to automatically deconfigure a faulty record of the bulk store is operational.

SECTION II

TERMINOLOGY

Terms and phrases frequently used in discussions of dynamic reconfiguration are defined below.

system controller	is a hardware module that interfaces an active module to the main memory of the configuration. The system controller manages system interrupts, passes on connect signals, contains the system calendar clock and provides memory functions to its active users.
memory controller	same as system controller
memory	same as system controller
controller	same as system controller
processor	is a major processing unit. A processor (CPU) is one of the three standard active modules. (The others are the IOM and bulk store controller.)
port	is a point of connection from a system controller to an active module. A processor port is one of eight connection spots on a processor to which a system controller can be attached. A controller port or memory port is one of eight points on a system controller for a connection to an active module. Each active module contains hardware known as port logic which determines (usually from absolute address) which processor port contains the connection to the appropriate system controller.

There exists hardware in each system controller to enable and disable requests, etc. over one of its (memory) ports. This port control logic contains among other logic a port enable register describing exactly which memory ports are enabled and hence which active modules can interact with the system controller.

control processor

There is a feature of the system controller hardware which allows the active modules on only certain selected ports to change port control and interrupt masking values of the system controller. Any processor so selected at the system controller maintainance panel is said to be a control processor. The switch which defines a processor to be control is the execute interrupt mask assignment (EIMA) switch. Each system controller has four such switches, and hence, each system controller can have up to four control processors. By convention, Multics allows only one control processor per controller. Three of the EIMA switches are disabled. The fourth selects the port of the control processor.

control memory

Each processor in the system has, by software convention, a control memory which that processor uses when it needs a system controller function (other than reading the clock) performed. The typical system controller functions which may be needed by a processor are 1) setting interrupt masks, 2) sending interrupts to other processors, and 3) receiving interrupts from system controllers. The control memory for a processor is established at bootload time or when the processor is added to the system. An accessing mechanism (described) later is established in the per-processor data base, the PRDS. Note that each processor has at least one control memory and that, by convention, processors will receive interrupts only from control memories. All control memories for a processor have the EIMA

switch selecting that processor and hence send interrupts only to that processor.

channel mask

Associated with each system controller is an eight-bit mask register that has a bit for each port of the controller. If this bit is a 1 the active module on the respective port can use the system controller. If the bit is 0 the respective active module will fault if it attempts to use the controller. The channel mask bit can be forced _on_ or _off_ by the port control switches on the maintainance panel of the system controller. These port control switches are three position switches that can be set (on a per port basis) to either ENABLE, DISABLE, or PROG CONTROL. The first two positions have the corresponding affect. If the switch is in the PROG CONTROL position, however, the channel masks can be set or reset by any _unmasked_ processor. It is a convention of the Multics reconfiguration software to attempt to mask all active modules currently not being used. For normal Multics running, therefore, all port control switches should be in the PROG CONTROL position.

The channel masks are changed as part of the SMCM processor instruction that also changes the interrupt masks (see later).

interrupt mask

Associated with each EIMA switch on a system controller is an interrupt mask register. There are therefore four interrupt mask registers for each system controller. (Note there is only one channel mask register.) The interrupt mask register contains a bit for each of 32 possible interrupt cells within the system controller. The interrupt cells are set on (and off when the interrupt is picked up) by the active modules of the system. When an interrupt cell is _on_ the system controller broadcasts a signal to all active modules selected by EIMA switches on the system controller that have the corresponding interrupt

unmasked. An interrupt cell will remain on, in general, until an interrupt mask register is unmasked enabling it to be sent to a processor. (Note that EIMA switches should only select ports that contain processors.)

By convention, since Multics currently uses only one EIMA switch per system controller, there will only be one interrupt mask register in a system controller that is being used. This interrupt mask register will be used by the corresponding processor to enable and disable the signalling of interrupts to that processor.

A feature to be considered when dynamically reconfiguring is that before an EIMA switch can safely be changed from one processor to another the interrupt mask must be masked down so that no interrupts are lost during the physical movement of the switch.

system interrupt

A system interrupt is an interrupt needed by the system in order to carry out its orderly functions of driving I/O devices and communicating between I/O devices. In addition there are special interrupts the system software uses to bring an orderly stop to the entire system or to delay processing of some data until a later time when better features of the system are available. All I/O interrupts are sent to the system controller containing the first word of the mailboxes and by software convention this will always be the first or low order system controller in the system. This means that all I/O interrupts will be sent to the processor selected by the EIMA switch on the low order system controller. This low order controller is often called the system interrupt controller or bootload memory and its selected processor is called the bootload processor or bootload CPU. It is possible to change the bootload CPU but it is not possible to change the bootload memory.

process interrupt

A process interrupt is an interrupt directed to a particular process within the Multics system. They are used to force the processor to execute some specific code on behalf of the process; for example, when the process has used up its scheduling quantum or the process is to be destroyed. Unlike system I/O interrupts, process interrupts can be directed toward any CPU by directing them toward one of the control memories for the CPU. By software convention whenever it is desired to send a process interrupt to a particular processor the same system controller is always used (assuming no reconfiguration has taken place).

internal interlace

The system controllers have a feature that allows interleaving of double-words of data between the low order and high order internal stores within the controller. This is termed internal interlace and, except for timing changes, is invisible to all active modules.

external interlace

In addition to internal interlace within the system controller each active module of the system has, as part of its port logic the capability to distribute apparently contiguous data between different system controllers. This external interlace can be either two words at a time or four words at a time and only can be used between system controllers that are on an even-odd port pair. The affect of this external interlace in conjunction with the internal interlace within the system controller leads to a four-way interlace mechanism. This four-way interlace is not to be confused with the four-word interlace implemented within the port logic of the active modules. All active modules must have their external internal switches set in the same way.

processor tag

Each processor has a two-bit switchable register that can be read by a special processor instruction. This register,

called the <u>cpu tag register</u> should be set differently for each processor in the configuration. The value of the register the <u>cpu tag</u> is used as the name of the processor by the reconfiguration software. The current implementation expects the first cpu tag to be 1 (0 is not recognized) and hence only three processors can currently be configured.

processor index

It is convenient to remap the cpu tags into a contiguous series of processor indices. This is done by software. The first processor configured is assigned the index one regardless of the value of the cpu tag. The correspondence between processor index and processor tag is kept in the system configuration segment (SCS).

core block

A <u>core block</u> is a contiguous region of core starting on a <u>page boundary</u> that is one <u>page</u> long. All of core is thus divided into fixed length regions the size of a page. Some core is <u>permanently wired</u> and can never contain paged data. Other core contains data or code that is <u>temporarily wired</u>, i.e., it is temporarily forced to remain in core. The core may later be freed up and reused for some other page. The term <u>wired</u> applies to anything that must remain in core for some time for some reason. The terms <u>latched</u>, <u>locked</u> and <u>core resident</u> are also used in the literature for what is here called wired.

core used list

The <u>core used list</u> or simply the <u>used list</u> is a threaded list of all <u>core map entries</u> for the core blocks in the paging pool. A core map entry describes which page, if any, is currently occupying the associated core block. The <u>core map</u> consists of all core map entries that are threaded together. The core map, however, can also be indexed by absolute core block number as an alternative method for scanning core map entries.

record

A record is a contiguous region of a secondary storage device that begins on a page boundary and that is one page long. Satisfying a page fault, for example, consists in moving the data of a page from a given record of secondary storage to a given block of core and performing the necessary connections.

abs_usable

The term abs_usable applies to that attribute of a core block that permits the core to be used for I/O. This concept is needed by I/O modules as they must set up DCW lists that have absolute addresses in them. The core blocks of the bootload memory can not be dynamically deconfigured (for several unrelated reasons) and therefore all core blocks of the bootload memory that are part of the paging pool are marked as abs_usable. In addition, core blocks of other system controllers will also be so marked if there are not enough abs_usable blocks in the bootload memory.

abs_wired

The term abs_wired applies to a block of core that contains a page that is wired down because it may contain absolute addresses. Such a page can not be moved either to make room for another abs_wired page or to remove the memory. Any memory that contains one abs_wired page can not be dynamically deconfigured until that page is no longer required to be abs_wired.

paging device map

The paging device map (or pdmap) is used as part of the bulk store management algorithms and is analogous to the core map. It is kept ordered by time of recent reference and hence is the key to the bulk store replacement algorithm.

read/write sequence   A _read/write sequence_ (or _rws_) is that
                             mechanism used to move a modified page
                             from the bulk store to secondary
                             storage. The mechanism consists in
                             finding a block of core, reading in the
                             page from the bulk store and then
                             writing the page out to disk.

SECTION III

DATA STRUCTURES

The several key data structures used by the reconfiguration software are kept in the segments SCS and SST. These are initialized as described in Section IV and modified as described in Sections VI, VII and VIII.

The following declarations of data structures describe the structures that are primarily used during processor reconfiguration.

```
declare 1 scs$processor_data ext aligned,
        2 int_port(8) bit(3) unal;

declare 1 scs$change_contr(8) ext aligned,
        2 flag bit(1) unaligned;

declare 1 scs$delete_cpu(8) ext aligned,
        2 flag bit(1) unaligned;

declare 1 scs$processor_tag(8) ext aligned,
        2 index fixed bin(3);

declare (scs$new_tag,
         scs$new_index,
         scs$new_port,
         scs$new_contr) fixed bin(3) ext;

declare scs$new_contr_ptr ptr ext;

declare scs$lock bit(36) aligned ext;

declare scs$mask bit(1) aligned ext;

declare scs$last_call fixed bin ext;
```

```
declare scs$bootload_cpu_tag fixed bin(3) ext;

declare scs$nprocessors fixed bin ext;

declare scs$processor_port(8) fixed bin(3) ext;
```

The variables declared above have the following meaning:

1. processor_data.int_port   is an array, indexed by processor index, of port numbers for the system controllers sending process interrupts to the given processor.

2. change_contr.flag   is an array, indexed by processor index, indicating whether ("1"b) or not ("0"b) a given processor should change its control memory, i.e., update any internal data bases (such as its PRDS) to indicate that a new system controller is controlling process interrupts for the processor. (A processor must always know which system controller is its control memory in order to know which one to mask when the processor must run protected from interrupts.)

3. delete_cpu.flag   is an array, indexed by processor index, indicating whether ("1"b) or not ("0"b) a given processor should remove itself from the active configuration. This bit is turned ON by the reconfiguring process to indicate to the idle process for the given processor that the processor is being deconfigured. After the idle process successfully stops its processor it resets the flag to indicate to the reconfiguration process that the processor has been deleted.

4. processor_tag.index   is the array giving the mapping between processor tag and processor index. It is indexed by processor tag and yields processor index. When the processor tag for a given processor index is desired, the array is linearly searched. Similarly when a new processor

index is desired, the array is searched for an available value. All unassigned indices have the value -1.

5. new_tag

is a temporary used to record the processor tag of the processor to be reconfigured. It is used both in adding and deleting a processor as well as in deleting a system controller.

6. new_index

is analogous to new_tag. It records the processor index of the processor of interest in adding or deleting a process or in deleting a system controller

7. new_port

is analogous to new_tag and holds the system controller port (of the processor) of interest, i.e. it specifies which system controller is being deleted or which system controller is (to be) control for a processor being reconfigured.

8. new_contr

is used in conjunction with change_contr to indicate which system controller is to be used by a given processor.

9. new_contr_ptr

is used in conjunction with change_contr and contains a pointer to be used when the processor is to reference the new system controller. It usually is a pointer pointing indirectly to a port addressing word (a word with the high order three bits indicating the system controller being referenced).

10. lock

is the global reconfiguration lock. This lock must be set whenever processor or system controller reconfiguration is being done.

11. mask

is a bit indicating whether a processor must mask or unmask itself during critical stages of reconfiguration. In particular, a

processor must mask before the EIMA switch of its control memory is changed.

12. last_call   is a variable used to synchronize calls to the supervisor side of the reconfiguration software. Usually each entry in reconfig (the supervisor reconfiguration program) can only be called after another specific entry has been called. This variable holds the number associated with the previous call and is thereby used to check for an invalid sequence of calls.

13. bootload_cpu_tag   is the processor tag of the bootload processor. Since special actions must be taken to guard against the loss of interrupts when the bootload processor is being deleted, a convenient method for determining the bootload processor was established.

14. nprocessors   is simply a count of the number of processors configured in the system.

15. processor_port   is an array, indexed by processor index, indicating the system controller (via processor port number) which is control for the processor.

The following data structures are used both during processor reconfiguration and system controller reconfiguration.

```
declare scs$port_addressing_word(0:7) bit(3) aligned ext;

declare scs$proc_contr_ptr(8) ptr ext;

declare 1 scs$controller_data(0:7) aligned ext,
           (2 size bit(18),
            2 base bit(18),
            2 contr_proc bit(18),
            2 pad1 bit(2),
            2 sys_int_sw bit(1),
            2 pad2 bit(2),
            2 clock_in_use bit(1),
```

```
                2 pad3 bit(5),
                2 abs_wired bit(1),
                2 ext_interlace bit(1),
                2 four_word_interlace bit(1),
                2 int_interlace bit(1),
                2 pad4 bit(3) ) unaligned;

        declare 1 rcd aligned,
                2 locker_id char(32),
                2 controller_data(0:7) like scs$controller_data,
                2 processor_data,
                    3 int_port(8) bit(3) unaligned,
                    3 processor_port(8) fixed bin(3),
                    3 processor_index(8) fixed bin(3),
                (2 tag,
                2 index,
                2 port,
                2 contr) fixed bin(3),
                2 mask bit(1) initial("0"b),
                2 channel_mask (0:7) bit (1);
```

16. port_addressing_word

is an array, indexed by system controller name (i.e. processor port number) of words containing a bit pattern that when indirected through will yield a reference to the given system controller. This addressing mechanism is used by the RCCL, RMCM, SMCM and SMIC instructions. The entries of this array are pointed to by the pointers in scs$proc_contr_ptr. A copy of the appropriate proc_contr_ptr entry is made in each processor's PRDS to facilitate the use of the above instructions. (The PRDS also contains the processor index so the scs$proc_contr_ptr could be used.)

17. controller_data

is a structure, indexed by system controller name (i.e. processor port number), containing data about the given system controller. This data is initialized during bootload and updated during reconfiguration.

18. controller_data.size

is the number of (1024-word) core blocks attached to the given system controller.

19. controller_data.base   is the absolute core address (mod 1024) of the first word of addressable core in the system controller. This value and the controller_data.size value above will both be all 1's if the system controller is not currently configured.

20. controller_data.contr_proc   is the processor index of the processor controlling this system controller. (The EIMA switch on the controller points to the port on which the processor with this processor index is connected.)

21. sys_int_sw   is ON only for the bootload memory. This flag is used to determine if special action must be taken when the processor controlling the bootload memory is deconfigured.

22. controller_data.clock_in_use   is ON for the single system controller whose clock is being used. There is only one clock of the system controllers which is used and this is the one in the bootload memory.

23. controller_data.abs_wired   is ON for all controllers which can contain "abs_wired" pages, i.e. pages which can not be moved due to some process containing absolute addresses pointing to within the page. This bit is ON by default for the bootload memory and will be turned ON for other system controllers if there are not enough abs_wirable pages in the bootload memory.

24. controller_data.ext_interlace   is ON if this system controller is externally interlaced.

25. controller_data.four_word_interlace   is meaningful only if the controller is externally interlaced in which case it indicates, if ON, that the interlace is four-word interlace as opposed to two-word interlace.

26. controller_data.int_interlace   is ON if the system controller is internally interlaced.

27. rcd   is a structure used by reconfig to communicate with reconfigure, the user-ring part of the reconfiguration software.

28. rcd.locker_id   is the process_group_id of the process that last locked the reconfiguration lock and hence the process currently performing reconfiguration if reconfiguration is in progress.

29. rcd.controller_data   is a direct copy of the corresponding data from the SCS.

30. rcd.processor_data   is a direct copy of the corresponding data from the SCS.

31. rcd.tag   is the processor tag of the processor of interest for the particular reconfiguration request being performed. This variable may be input or output.

32. rcd.index   is the processor index of the processor of interest. This variable may be input or output.

33. rcd.port   is the system controller number (processor port number) of the system controller being reconfigured; or it is the system controller port number (thereby selecting a processor) of interest.

34. rcd.contr   is analogous to rcd.port and refers to the system controller that is having its control processor changed.

35. rcd.mask   is used to communicate the need to mask or unmask a system controller.

36. rcd.channel_mask   reflects the current state of the (eight-bit) channel mask assumed by the supervisor and hence the latest value set in all controllers.

THE SYSTEM CONTROLLER ADDRESSING SEGMENT

The system controller addressing segment (SCAS) is a
specialized data base used to read and set certain registers in
system controllers. In particular the SCAS segment is used to
generate the correct final (absolute) address needed by the RSCR
and SSCR instructions. These instructions operate on the system
controller that contains the final absolute address generated by
the address preparation logic ff the processor. The SCAS
"segment" is really nothing more than an eight-word page-table
with each page table word (PTW) pointing to a block of core in
the currently configured system. The first "page" of SCAS is
located in the system controller on port 0 of the processor, the
second "page" is located in the system controller on port 1, etc.
Note that there may be "holes" in SCAS due to certain system
controllers not being configured.

THE SYSTEM SEGMENT TABLE (SST) AND RELATED DATA

The core memory data base, the core map, consists of all of
the core map entries (CME's) threaded into a circular list. The
"head" of the core map is pointed to by the variable sst.usedp
and is the core map entry for what the system considers the least
recently referenced page. The paging device map entry (PDME) is
completely analogous to the core map entry. These are both
described below:

```
declare   1 cme based (cmep) aligned,
          (2 (fp, bp) bit(18),
           2 devadd bit(22),
           2 pad1 bit(2),
           2 io bit(1),
           2 rws bit(1),
           2 pad2 bit(1),
           2 removing bit(1),
           2 abs_w bit(1),
           2 abs_usable bit(1),
           2 pad3 bit(3),
           2 contr bit(3),
           2 ptwp bit(18),
           2 pad4 bit(18),
           2 dblw_devadd bit(22),
           2 pad5 bit(14) ) unaligned;
```

```
        2 pad2 bit(1),
        2 truncated bit(1),
        2 notify_requested bit(1),
        2 pad3 bit(1),
        2 removing bit(1),
        2 not_on_disk_yet bit(1),
        2 pad bit(1),
        2 ptwp bit(18),
        2 ht bit(18),
        2 pad4 bit(36) ) unaligned;
```

37. cme.fp, cme.bp

are the forward and backward thread pointers used to chain the CME's together.

38. cme.devadd

is the device address associated with the page residing in the given core block. This may be "null", a bulk store device address or a disk device address. If it is a bulk store device address, the disk address can be found in pdme.devadd for the corresponding paging device map entry.

39. cme.io

is "0"b if the last (or pending) I/O request for the core block was a read and is "1"b if the last request was a write.

40. cme.rws

is ON only if this block of core is being used for a read/write sequence.

41. cme.removing

is set ON only if this block of core is currently being deleted from the system. If it is ON, the core map threading algorithm does not thread the entry into the used list, but rather leaves the entry where it is (in the remove list).

42. cme.abs_w

is ON only if the page residing in this block of core is abs_wired

43. cme.abs_usable

is ON only if this block of core is abs_usable.

44. cme.contr

is the controller name (processor port number) of the controller containing this block of core.

45. cme.ptwp is a pointer to the PTW for the page residing in this core block or is (18)"0"b if the core block is free.

46. cme.dblw_devadd contains the secondary storage device address to be used in the store-through cycle of a double write request. If this field is (22)"0"b then no secondary write should be queued.

47. pdme.fp, pdme.bp are the forward and backward thread pointers used to chain the PDME in the paging device used list. However, if a read/write sequence is in progress for this paging device record the PDME is <u>not</u> threaded into the used list and the pdme.bp field is used instead to hold a pointer to the core map entry for the block of core being used for the RWS.

48. pdme.devadd contains the secondary storage device address associated with the page residing in the corresponding record of the paging device. This address is never null as page control always quarantees a disk address has been assigned for any pages residing on the paging device.

49. pdme.modified is ON if the page on the paging device has been modified and hence must be written back to disk (via an RWS).

50. pdme.incore is ON if the page associated with this PDME is in core.

51. pdme.rws is ON if a read/write sequence is in process for this PDME.

52. pdme.used is ON only if the corresponding paging device record is being currently used. The record is free and available for use if this is OFF.

53. pdme.abort is ON if a process took a page fault on a page which was in the

process of being written back to disk (i.e. an RWS was in process). This flag is tested when the RWS completes; if it is ON, the page is left on the paging device, the core used for the RWS is given to the page, the PTW is updated to indicate that the page is in core and the faulting process is notified that this page is in core.

54. pdme.truncated

is ON if a page was truncated during a RWS for this page.

55. pdme.notify_requested

is ON if some process wants to be notified when the RWS in progress completes.

56. pdme.removing

is ON if the paging device record is currently being deconfigured.

57. pdme.not_on_disk_yet

is ON for any (new) pages on the paging device which have never been flushed to the disk. This bit is used during automatic deconfiguration of paging device records to determine if the disk copy of the page is valid. If the copy is not valid (has never been written) the disk address is freed up and the page is given a null address. This prevents unauthorized access of data.

58. pdme.ptwp

points to the PTW for the page corresponding to this PDME if the segment which contains the page is active. If the segment is not active this field is (18)"0"b.

59. pdme.ht

is a hash thread used to thread all PDME's with the same hash index (generated from secondary storage device address) together.

59. pdme.ht   is a hash thread used to thread all
                                         PDME's with the same hash index
                                         (generated from secondary storage
                                         device address) together.

SECTION IV

DATA BASE INITIALIZATION

This section describes the initialization of the data bases used by the reconfiguration software. Some of these data bases will not be changed after the bootload, others will be changed all the time and still others will only be changed when reconfiguration is explicitly requested.

SCS INITIALIZATION

The system communication segment (SCS) described in Section III is initialized primarily by the programs scs_init, scas_init, initialize_faults, init_sst, start_cpu and tc_init. (The program scas_init fills in scs$bootload_cpu_tag, which is not again changed unless the bootload processor is deconfigured.)

The controller_data structure is filled in in stages as various programs learn more about the configuration. The program scas_init reads the system controller registers of each configured controller to determine internal interlace, etc. The processor switches are also read to determine external interlace and to verify that the actual configuration corresponds to the configuration deck.

The clock reading mechanism consists of a pointer in sys_info pointing to a port addressing word (a word with the high order three bits being a port number) for the port connected to the system controller whose clock we want to use. During early stages of initialization this pointer and the target port

addressing word are set up to point to the bootload system controller. This function is performed first by initialize_faults so that the clock reading mechanism will be enabled early in the bootload. The clock reading mechanisms are initialized "officially" in scs_init.

The program scs_init initializes many structures in SCS but its primary concern is the initialization of the interrupt handling mechanism. This includes setting up the various interrupt mask patterns, the interrupt handler array, and (of importance to processor deconfiguring) the variable scs$simulate_pattern is set up to have a bit ON for each system interrupt the system can not afford to lose for an extended length of time.

SCAS INITIALIZATION

SCAS is not really a data base but rather a page table that points to pages in each of the configured system controllers. The actual content of the pages is not of importance and in general changes as pages are moved in and out of the particular region pointed to by a given SCAS PTW (the actual page that SCAS is set up to point to in each system controller is the first page in the controller.) As mentioned earlier SCAS is used by the RSCR and SSCR instructions and indeed scas_init issues these instructions as soon as SCAS is initialized to verify the actual configuration corresponds to the configuration deck. In addition other information about the system controllers (internal interlace, etc.) is saved at this time.

SST INITIALIZATION

The initialization of the SST is described fully in System Initialization, Order No. AN70. The two important features relevant to reconfiguration are:

1. the abs_usable bits in core map entries are set ON for all core blocks in the bootload system controller (it can't be deleted anyway because it contains mailboxes and fault and interrupt vectors) and

2. the bulk store (paging device) map is initialized as described on the "page" configuration card.

OTHER DATA BASE INITIALIZATION

The initialization of the PRDS, done mainly by prds_init, tc_data, tc_init and start_cpu, is straightforward and simple. The primary interaction between the traffic controller and reconfiguration consists in the creation, running and deletion of the idle processes.

SECTION V

RECONFIG AND RECONFIGURE

The processor and system controller reconfiguration software is split into two main programs: reconfig in the hardcore ring and reconfigure in the user ring. These programs are so intimately tied together that even the most trivial change to one often must be reflected somehow in the other. They are really one logical program which has been split across rings for reasons independent of the reconfiguration problem. In particular, reconfig must be in ring zero because it performs (via calls) privileged functions such as masking, sending interrupts and changing hardcore data structures. Similarly, reconfigure must be in the user ring because it must perform normal terminal I/O and, of prime and nonobvious importance, it must be possible to be interrupted with a preempt interrupt from the reconfiguration processor. This preempt interrupt (for other unrelated reasons) cannot be allowed to go off (and be handled) in the hardcore ring.

The communication between the two programs is done through an entry in the hphcs_ gate. This entry is declared as follows:

    declare hphcs_$reconfig entry (ptr, fixed bin, fixed bin(35));

    call hphcs_$reconfig (rcd_ptr, entry_no, code);

1. rcd_ptr   is a pointer to the rcd structure described in Section III. (Input)

2. entry_no   is a code telling reconfig which function is being requested. (Input)

3. code   is an error code indicating what kind of problem has been encountered, if any. (Output)

The various "entries" into reconfig are therefore implemented by one large "do case -n-" type construct where n is the entry number. There is extensive cross checking done by reconfig to insure that the proper sequence of calls is maintained.

All processor and system controller reconfiguration is controlled by a lock which prevents more than one process from executing a reconfiguration sequence at the same time. This lock can be forced to the unlocked state (bypassing any sequencing tests) via a call to the user ring program reconfigure$force_unlock. This entry expects no arguments and prints out the process group ID of the last process to lock the lock.

The error code returned by reconfig is decoded by reconfigure to determine the details of a particular error. In particular, the code is a two-digit decimal number. The first digit gives a general indication of what type of problem was encountered and the second digit gives more detail. The following tables interpret all returned error codes.

First Digit

0	general
1	error from start_cpu
2	error from stop_cpu
7,8,9	general

Error Code   Meaning

1	reconfiguration mechanism is locked
2	reconfiguration mechanism is not locked
3	improper sequence of calls
4	argument is inconsistent with last call
5	invalid entry number
11	processor would not start
12	processor has wrong CPU tag in switches
13	wrong EIMA switch (reconfiguring process got initialize interrupt itself)
14	no APTE available for idle process
15	no idle process for this CPU
16	no processor for the given controller
21	no idle process for this CPU
22	no processor for the given controller
70	cpu to be deleted not in system
71	cannot find enough CPUs

73	memory already configured
75	controller to be added is not defined
74	controller cannot be removed
76	not enough controllers left
77	not enough core blocks left
78	abs wired page in controller to be removed
79	core block not in used list in controller being removed
80	processor already configured
81	controller not found for CPU
82	port already assigned
90	illegal value for processor tag
91	illegal value for controller
92	illegal value for port
93	program bug - no controller available

SECTION VI

PROCESSOR RECONFIGURATION

This section describes the workings of the processor-adding and processor-deleting functions. Before this can be fully described, however, the mechanism of _idle processes_ must be briefly explained.

IDLE PROCESSES

There is one idle process for each processor on the system. In general, the idle process for a processor is run whenever that processor cannot find another process to run, either because no other process wants service or because all processes that want service are either running on other processors or are _waiting_ for some system event such as a page fault to be satisfied. A processor will never be run in another processor's idle process.

An idle process is a full-fledged normal Multics process in that it has its own stack (pds), its own descriptor segment, and its own APT entry. In particular, an idle process can be threaded in at the head of the eligible queue if necessary. This technique, in fact, is used to force a given processor to execute in its idle process by issuing a preempt interrupt to the processor after threading the idle process APT entry to the head of the list. However, most of the time the idle processes are threaded at the tail of the ready list where they will be selected only if all other processors do not want to or cannot run.

The idle process for a processor must be created before a processor is added to the system. (This is not quite true for the bootload CPU that, of course, must somehow be bootstrapped into the normal state. See _System Initialization_, Order No. AN70, for a complete description of this bootstrap mechanism.)

AN70, for a complete description of this bootstrap mechanism.) Similarly, each processor on the system must have a processor data segment (the PRDS) before it can be run. Both the idle process and the PRDS for a processor therefore must be created as part of the operation of dynamically adding a processor to the system. After a processor is deleted from the system its idle process and PRDS are deleted as a cleanup measure.

ADDING A PROCESSOR TO THE SYSTEM

To add a processor to the system up to four calls to reconfig may have to be performed in addition to the "status" call which returns general information about the current configuration.

The first "entry" called (number 10 or 11) communicates to ring zero the intent to add a processor to the system. The caller (reconfigure) may or may not specify which controller the processor is to be assigned. If the caller does not specify a controller (entry number 11) the system will attempt to find one. (Recall that a controller is needed for each processor in the system and that there must therefore be at least as many controllers in a given configuration as processors.) At any rate a controller is selected and if the controller receives system interrupts, the fact is noted. A processor index is then generated and a return to the user ring is made. Note that the user ring was returned a switch indicating whether or not the new controller receives system interrupts.

The second "entry" called when adding a processor (number 15) is called to mask the controller selected above, as well as to set the channel mask in all controllers. The channel masks are set by updating all the masks in SCS and then forcing the masks to be loaded. For controllers which get interrupts for some CPU it is enough to cause that CPU to take an interrupt as the interrupt interceptor will mask and hence set the channel mask. In this case the preempt interrupt is sent. For controllers which do not get interrupts an explicit call to set their masks (to "open level") is made. Note the controller is masked to "sys_level" and hence all normal interrupts from hardware devices are delayed if the controller is the system interrupt controller. In this case any abnormal terminations of the reconfigure request (wrong switch settings, etc.) must be acted upon quickly (by the operator) as any significant delay in unmasking will impair efficient operation of the system.

After returning from the entry which masks the new controller, the operator is instructed to change the EIMA switch on the selected controller to the CPU being added. These instructions (which require operator reaffirmation) are followed by the third call into ring zero (entry number 16).

This third call is the call that actually adds the processor to the system and starts it running. The data base SCS is changed to reflect the added CPU and the program start_cpu is called. This program creates and initializes the PRDS for the new processor as well as the new idle process. start_cpu in turn calls init_processor to get the processor going. This is done by patching the interrupt vector for a special "processor initialize" interrupt and then sending this interrupt to the new processor. The patched interrupt vector forces the new processor to enter code in init_processor which loads the DBR register of the new idle process. After certain other initial steps the newly added processor sets a flag saying it is successfully running and enters the idle loop in search of a normal user process to run. The processor which initiated the newly added processor loops until confirmation that the new processor is running normally. If this confirmation never comes (it waits a few milliseconds) an error code is returned which is reflected to the reconfiguration programs.

If the processor did not add successfully (wrong switch setting, etc.) the operator is instructed to change the EIMA switch back and the controller is unmasked. (If the add was successful the idle process would have unmasked). When the add is successful the initiating process resets the interrupt vector to its original state and returns control from init_processor to start_cpu to reconfig.

The program init_processor (see System Initialization, Order No. AN70) is called to start all processors on the system - even the bootload process at initialization time. For this reason the program makes special checks to see if it interrupted itself (with the processor initialize interrupt). In addition to this check, the processor reads his configuration switches to make sure the tag is set correctly and makes a check to see if the processor was already initialized. (This would be the case if the bootload processor, for example, sent the initialize interrupt to another processor but received it itself do to an incorrectly set EIMA switch.)

The program init_processor consists of two logically different sections. The call side is called by start_cpu when everything has been set up for the new processor. This entry sets up the interrupt vector to transfer to the second section of

init_processor at the label "first_steps". It is the code at this label that first gets executed by the new processor. The code runs in absolute mode until the DBR is loaded. After the DBR is loaded the program checks its configuration as mentioned above and if all is well the program enters the "idle" code of init_processor after sending itself a preempt interrupt. Before beginning to idle, however, the reconfiguration process is told (via a variable in init_processor) that the new process is running. Also before idling the new processor unmasks (to open_level) its control memory so that it can receive process interrupts.

DELETING A PROCESSOR FROM THE SYSTEM

The first "entry" invoked when deleting a processor is number 20. This entry checks that the CPU to be deleted is actually in the configuration and then searches for (1) another CPU to take the interrupts directed toward the one being deleted and (2) the first of possibly several controllers pointing to the CPU. If there is only one controller controlled by the CPU being deleted, the CPU is stopped at this time. The CPU will mask the controller when it stops (so the EIMA switch can be moved to another processor).

If there is more than one controller pointing to the CPU, one of the controllers that is _not_ used for process interrupts is selected and its port number is returned to the user ring. For each such controller found in the system a pair of calls into ring 0 is made. The first call (number 21) is done prior to directing the operator to change the EIMA switch. It is a call to mask the controller. The second call (number 22) is performed after the switch has been changed. It first unmasks the controller and then searches for another controller pointing to the CPU.

The controller that gets interrupts for the CPU is done last in the sequence. In this case, instead of finding the next such controller, the CPU is actually stopped. The CPU will mask its controller prior to stopping. Entry number 23 is then invoked. This entry waits until the CPU says it has masked and then returns. Entry number 24 is then called to finish up. It checks if the CPU deleted was the bootload CPU and if so redefines scs$bootload_cpu_tag. It then checks to see if the processor has really been deleted by checking scs$delete_cpu. If the processor has not stopped, the ring 0 code returns control to the user ring so that the effect of a preempt interrupt that was apparently

final step, the SCS data base is updated and the channel masks are set to reflect the absence of the processor from the system.

There are some interesting control sequences which must be followed when deleting a CPU. In particular the code which the CPU being deleted must execute is all isolated in the idle process and in particular in the program init_processor. In order to force the processor to execute this code, the idle process is given ultimate high priority (by moving to the head of the scheduler's ready list). The actual code of the idle process is a small loop consisting of: (1) code to flash the lights on the maintenance panel in a recognizable pattern, (2) some DIS instructions to enable the panel, and (3) some tests to see if the processor is being deleted or if it should change its control memory. The check to see if the processor should be deleted results in the idle process invoking some code which eventually results in the processor executing a DIS. In particular, the processor:

1. Gets the port number for its controller (used as an index into scs$controller_data).

2. Checks to see if the processor is the bootload CPU. and, if so, sends a batch of interrupts to the controller for the new bootload CPU using scs$simulate_pattern.

3. Masks the controller.

4. Checks to see if the controller for the CPU received system interrupts. If so, the CPU to get the controller is told to change its controller and given high priority so it will run and see the request. The CPU being deleted then loops until the processor to get the controller has changed controllers.

Note that the deletion of the bootload CPU requires special action to be taken. This is because system interrupts (such as those that drive the disks and typewriters) are only handled by the bootload CPU because this is the CPU which receives interrupts from the bootload (low-order) memory and all I/O interrupts are sent to this system controller. Since it is necessary to mask all interrupts in an interrupt mask register whenever its EIMA switch is changed, real system interrupts must be prevented from reaching any processor while this switch is being changed (the new processor must be directed to receive system interrupts). Since the system can not in general survive an extended period with system interrupts being processed, these interrupts must be simulated by sending every interrupt which may be of interest to the processor which is to receive system

interrupts. These interrupts are remembered in the scs$simulate_pattern cell at initialization time, and broadcast to the _new_ bootload CPU every second or so by the CPU being deleted.

Another item to be noted is that the several places in the reconfiguration software where masks are set in controllers are bracketed by calls to force the running (reconfiguring) process to run on a particular processor -- the processor which is control for the system controller whose mask is being changed. The "processor required" mechanism built into the dispatcher of the traffic controller is used for this purpose. Therefore, if reconfiguration is attempted from a process already restricted to a given processor, that restriction may not be in effect after the reconfiguration is complete. (The "processor required" mechanism does not "stack" requests.)

Since adding a processor to the system requires the operator (see Section IX) to interact at his terminal, it is not possible to add a processor to the system _automatically_. This means that it is not possible to bootload the system and have it come up with two processors running. The second and subsequent processors must be explicitly added _after_ the system is booted.

SECTION VII

MEMORY RECONFIGURATION

This section describes the mechanisms used to dynamically reconfigure primary memory (core or MOS). The first two subsections describe system controller reconfiguration and the third subsection describes the core block reconfiguration within a controller.

ADDING A SYSTEM CONTROLLER TO THE SYSTEM

At system initialization time the data bases scs$controller_data and the main memory map in the SST are initialized. These are initialized from the configuration deck (and active register values); since the core map can not easily be grown it is required that any system controllers that will ever be configured to the system for a given bootload must be specified in the configuration deck for the bootload. This is done by using an ON or OFF field of the MEM configuration cards. All system controllers actually configured and to be used at bootload time are indicated as being ON. Other system controllers are OFF.

When the core map is initially set up, only core blocks which are in configured system controllers are threaded into the used list. Core blocks for system controllers that are not yet configured are left alone and threaded into no list. To add a system controller (and its memory) to the system, all that need be done is to thread the (hopefully unused) core blocks for the controller into the core-used list. This is exactly what add_memory does as invoked by reconfig after an addmem request is given. Before the memory is enabled the operator is told to set the switches on all active modules so the system controller can be referenced. The switches on the controller itself must also

particular the EIMA switch is set to the processor that controls the system controller. Note that the entire system can be prepared (i.e., all switches set correctly) before the addmem request is given by the operator.

The actual code in reconfig to add a system controller to the system, entry 30, merely checks consistency of all arguments and calls add_memory to actually add the system controller to the system. The program add_memory first sets the interrupt mask in the memory -- to load the appropriate channel mask -- and then threads the core map entries for core blocks in the controller into the used list. Before calling add_memory, reconfig forces the executing process to run on the processor that controls the memory so the mask can be set.

REMOVING A SYSTEM CONTROLLER FROM THE SYSTEM

The mechanism to remove a system controller from the system is complicated by two features. First, a mechanism must be provided to remove all references to any pages in the system controller by processors. Second, a mechanism must be provided to remove all references to the memory of the system controller by other active modules, particularly IOM's.

The first major problem in removing main memory, i.e., preventing processors from referencing the memory, is not hard to solve in that all processor references to the memory are indirect through PTW's over which the system software has control. (It is not possible to remove a system controller that contains permanently wired code or data.) It is thus necessary only to remove access in PTW's or copy pages into core that is not being removed. This in fact is just what is done. There are three cases to consider:

1. Core blocks that contain wired pages.

2. Core blocks that contain pages that are not wired but are modified.

3. Other core blocks.

Pages that are temp-wired must remain in main memory but need not remain in the same location in main memory. Such pages are copied from the region of memory being removed to a region of memory remaining. After the copy is complete, the PTW is changed to point to the new copy and all processors are forced to clear

their associative memories so that they will refetch the PTW with the new address and make all subsequent references to the copy. If the page is modified while the copy is being performed, all processors are stopped (forced to loop, via a connect fault) and the copy is made while no one can modify the data. The entire mechanism to move a wired page is implemented in the program evict_page.

Pages that are modified are simply written out and evicted when the I/O completes. This process continues until a page does not get modified while the I/O is going on in which case the block of memory can be claimed.

Pages that are neither wired nor modified are evicted immediately (the PTW is set to fault) unless I/O is in progress, in which case the I/O is waited for and the block is claimed on the next pass.

The program (pc_abs$remove_core) that does all this work loops through all blocks in the given controller until a pass is completed that leaves no blocks unclaimed.

The second major problem in removing main memory, that of preventing other active modules from referencing the memory, is solved before it even becomes a problem. This is via the abs_wiring technique, which requires that any pages that are referenceable via nonprocessor active modules (e.g., the IOM) cannot reside in a deconfigurable system controller. In order to do this, certain controllers are set up as "abs_usable" and hence nondeconfigurable. For most configurations, the bootload memory alone is abs_usable, but the system dynamically chooses other controllers as necessary if there is not enough abs_wireable memory in the bootload controller.

Therefore any program that uses a page for I/O (that is not permanently wired) must call a special program to have the page wired down. This program is pc_contig. See The Storage System, Order No. AN61, for a complete description of this mechanism. The dynamic deconfiguration software need not be concerned about pages wired for I/O activity.

When a system controller is deleted from the system, a check must be made to see if the system controller is used by a processor as its process interrupt controller. If this is the case, another controller must be found for the processor that in turn may require the changing of the EIMA switch on the newly selected controller if it is controlled by a CPU other than the one losing its interrupt controller. All of these cases are handled in the controller deletion software; the general case

will take several calls into the hardcore ring to: (1) communicate with the operator via the typewriter, and (2) mask and unmask as the EIMA switch is changed. The variable scs$change_contr is analogous to the variable scs$delete_cpu and is sampled by the idle process (that is given priority so it will run) in its normal idle loop. If the change_contr flag is set, the program init_processor will perform the necessary changes (in the correct PRDS) to reflect the change. It will also signal when it is done by resetting the flag. The program stop_cpu$switch_contr is used to initiate this mechanism.

AUTOMATIC MEMORY DELETION

To be supplied.

SECTION VIII

BULK STORE RECONFIGURATION

The bulk store reconfiguration mechanisms differ from the system controller mechanisms in that records of the bulk store are reconfigured rather than the active module, the bulk store controller, being reconfigured. However, the software is set up so that if _all_ records on a given bulk store are removed from active use by the system, the controller can safely be deconfigured for offline test or for use by another local system. No switches need be changed during this reconfiguration mechanism. (It is, of course, necessary that any parts of the bulk store that are to be used by a system be configured to that system -- the bulk store reconfiguration software assumes that necessary configuration switches have been set and does not remind the operator about these switches.)

During system initialization (see _System Initialization_, Order No. AN70) the "page" configuration card is read to determine which bulk store records are to be initially used by the system. The paging device map is read in; if it has not been successfully cleaned up, the system is crashed. Otherwise, there is no valid data on the bulk store and all records that are to be used have the paging device map entries (PDME's) threaded into the paging device used list and marked as free. Until reconfiguration time all of these records will remain in this used list unless:

1. The record is removed for a moment as part of a rethreading operation,

2. The record is removed because a read/write sequence is in progress for the given page, or

3. The record is dynamically removed automatically because of a fatal read request.

The paging device map (like the core map) can be searched either by following the used list thread or by indexing into the map with a given record number. The latter method is used for bulk store reconfiguration under operator control.

The main supervisor program that controls bulk store record reconfiguration is delete_pd_records. This program, callable through the hphcs_ gate, has two entries as described below:

```
declare delete_pd_records entry (fixed bin, fixed bin,
      fixed bin(35));

call delete_pd_records (first, count, code);
```

1. first   is the record number of the first of count
                    contiguous records to be removed from active
                    use by the system. (Input)

2. count   is the number of records being deconfigured.
                    (Input)

3. code   indicates, if nonzero, that the input
                    parameters were inconsistent with the current
                    configuration. (Output)

```
declare delete_pd_records$add_pd_records entry (fixed bin,
      fixed bin, fixed bin(35));

call delete_pd_records$add_pd_records (first, count, code);
```

1. - 3.   are analogous to above.

A request to delete a record that is already deconfigured is not considered fatal. In fact, it is convenient to be able to delete an entire core storage module (CSM) after several records within it have been automatically deleted by page control. The paging device map always resides on the first few records of the paging device region that is potentially usable for a given bootload. It is again nonfatal to request that these records be deleted. However, they will not be deleted, because the current implementation does not provide for moving the paging device map copied onto the paging device. In particular, if the first CSM is to be deleted (for offline work) the entire bulk store must be disabled.

ADDING BULK STORE RECORDS

To add a region of the bulk store to the current configuration the operator must specify which regions of the bulk store should be added. As mentioned earlier, all configuration switches must have previously been set correctly before the bulk store add request is given. This includes the various switches on the bulk store controller as well as all the port-enable switches on all system controllers. (The normal operation is to have the port-enable switches under program control.) Since the bulk store controller is not the target of the operator requested reconfiguration, the channel masks in the system controllers are not changed even if the entire set of bulk store records are deconfigured.

The actual mechanism to add bulk store records to the system is quite similar to the main memory add mechanism. It is necessary that all of the paging device map that will ever be needed for a bootload be allocated at system initialization time. Those records of the bulk store that are not initially part of the system do not have their PDME's threaded into the paging device used list. The "addpage" request issued by the operator merely threads the PDME's for records being added into the paging device used list and updates the two system-wide variables in the SST, pd_free and pd_using. The variable pd_free reflects the number of records actively configured and which are free for use. The variable pd_using indicates the number of records actively configured. Both of the variables are updated by the internal procedure "set_pd_free_and_using" (under control of the global paging lock) in the main bulk store reconfiguration program delete_pd_records. Note that when pd_using reaches zero, i.e., there are no records actively being used, the automatic update of the paging device map is disabled making it possible to physically deconfigure the bulk store controller.

The internal procedure "build_page_card" of delete_pd_records updates the page configuration card (if possible) to reflect the current bulk store configuration for both adding and deleting bulk store records.

DELETING BULK STORE RECORDS

Deleting bulk store records is quite analogous to deleting main memory blocks. The entire mechanism is controlled by the program delete_pd_records in the hardcore ring. This program

first checks its parameters for consistency and then loops through the specified region of the paging device map (indexing by paging device record number), cleaning out pages as it goes. The entire process is under control of the global paging lock; since the various control bits (such as the modified bit) of the PDME are simulated and under control of the same lock these bits will not change as long as the lock remains set.

There are five cases of interest. These are:

1. The record is not used.

2. The page for a given record is in main memory.

3. The page is not in main memory but has been modified since last written to secondary storage.

4. The page is not in main memory and has not been modified.

5. A read/write sequence is in progress for the given page.

If the record is not used it is merely removed by threading its PDME out of the paging device used list.

If the page is in main memory the core map entry is updated to include the secondary storage device address rather than the paging device address; if the modified bit is ON in the PDME, the modified bit is set ON in the corresponding PTW. This can cause a slight anomaly in the value of date-time-modified for the segment owning the page.

If the page is not in main memory but has been modified since last being written to secondary storage, a read/write sequence is initiated for the page. In addition, a flag is set in the PDME so that when the read/write sequence completes the paging device record will be marked as being deconfigured (i.e., the PDME will not be threaded into the paging device used list). The flag used to indicate this is pdme.removing.

If the page is not in main memory and has not been modified, the record is taken as if it weren't used unless the segment containing the page is active. In this case the device address saved in the PTW is changed to the secondary storage device address from the paging device address.

If a read/write sequence is in progress for a page the pdme.removing flag is set ON so that the record will not be threaded into the paging device used list when the RWS completes. Any pages that have RWS's in progress are remembered;

at the end of scanning all PDME's, the last RWS noticed is waited for. The scan is then started again from the beginning until no RWS's are seen in one pass.

When a record that was used is taken, the secondary storage address for the record must be hashed out of the paging device hash table.

AUTOMATIC PAGING DEVICE RECORD REMOVAL

When page control encounters a fatal read error from the paging device, the removal of that paging device record is triggered automatically.   This is done at page_fault$done time and consists of hashing the PDME out, threading the PDME out of the used list and reporting the event on the operator's console. This type of dynamic deconfiguration of paging device records is not reflected in the page configuration card, because it is done at a time when no page faults can be taken.   A special flag, pdme.not_on_disk_yet, is checked to determine if a copy of the page has ever been written to secondary storage.   If so, that (probably slightly out of date) copy is preserved as <u>the</u> copy of the page. If the page has never been written to secondary storage since it was created, the block of main memory to contain the page is zeroed.   If the fatal read error occurs during a read/write sequence, a similar action is taken with respect to using a valid secondary storage copy only if it exists.

SECTION IX

THE COMMAND INTERFACE

The reconfiguration of main memory, processors and bulk store is under operator control either from an "initializer" terminal or from a normal logged-in (privileged) user. The initializer commands are

        addcpu
        delcpu
        addmem
        delmem
        addpage
        delpage

and are described fully in the <u>Multics Operator's Handbook</u>, Order No. AM81. The normal user commands are

        reconfigure$addcpu
        reconfigure$delcpu
        reconfigure$addmem
        reconfigure$delmem
        addpag
        delpag

and, of course,

        reconfigure$force_unlock

Note that there is no initializer command to unlock the reconfiguration lock.

TITLE: | MULTICS RECONFIGURATION PROGRAM LOGIC MANUAL

ORDER No.: AN71, REV. 0

DATED: JUNE 1974

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME _____ DATE: _____

COMPANY _____

TITLE _____

_____

_____

Honeywell

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

The Other Computer Company:

Honeywell

HONEYWELL INFORMATION SYSTEMS