SUBJECT:

Internal Organization of Multics System Initialization

SPECIAL INSTRUCTIONS:

This document supersedes the previous edition of the manual, order number AN70-00, dated February 1975.

This System Designers' Notebook 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) and Subroutines (Order No. AG93).

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

DATE:

05/29/84

ORDER NUMBER:

AN70-01

PREFACE

Multics System Designers' Notebooks (SDNs) 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 SDNs 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.

This SDN contains a description of the software that initializes the Multics system. This description is by no means complete in all its details; for a thorough understanding of Multics initialization, or of any particular area within this system, this SDN should be used for reference in conjunction with the source of the relevant programs.

In addition to this manual, the volumes of the _Multics Programmers' Manual_ (MPM) should be referred to for details of software concepts and organization, external interfaces, and for specific usage of Multics Commands and subroutines. These volumes are:

MPM Reference Guide, Order No. AG91

MPM Commands and Active Functions, Order No. AG92

MPM Subroutines, Order No. AG93

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

SECTION 1

SUMMARY OF INITIALIZATION

Multics initialization, as described in this SDN, can be thought of as divided into the following parts:

* Hardware and PL/1 Environment initialization (Collection 0)
* Page Control initialization (Collection 1 service pass)
* Bootload Command Environment (bce) (Collection 1 multiple passes)
* Crash Handler (toehold)
* File System initialization (Collection 2)
* Outer ring Environment initialization (Collection 3)

The parts listed before collection 2 are collectively called "Bootload Multics."

A collection is simply a set of initialization routines that are read in and placed into operation as a unit to perform a certain set, or a certain subset, of the tasks required to initialize a portion of the Multics supervisor. Each collection consists of a distinct set of programs for reasons discussed throughout this SDN. Even though each collection mostly exists to perform a particular set of functions, they are normally referred to by their number (which have only historical significance) rather than the name of their function.

Initialization may also be thought of as having three separate functions:

Bringing up the system
: This role is obvious. The description of this role follows along the functions needed to perform it. Each portion of initialization runs, utilizing the efforts of the previous portions to build up more and more mechanism until service Multics itself can run.

Providing a command environment before the file system is activated from which to perform configuration and disk maintenance functions

Providing an environment to which service Multics may crash which is capable of taking a dump of Multics and initiating recovery and reboot operations

These last two functions are the role of bootload Multics (bce). They take advantage of the fact that during initialization an environment is built that has certain facilities that allow operations such as disk manipulation to occur but it is an environment in which the disks themselves are not yet active for storage system operations. This environment, at an intermediate point in initialization, forms the bootload command environment (bce).

The bootload command environment is saved before further initialization operations occur. When service Multics crashes, service Multics is saved and this bce "crash" environment is restored. This safe environment can then examine or dump the service Multics image and perform certain recovery and restart operations without relying on the state of service Multics.

HARDWARE AND PL/1 ENVIRONMENT INITIALIZATION

The purpose of collection 0 is to set up the pl/1 environment and to start collection 1. It has a variety of interesting things to perform in the process. First of all, collection 0 must get itself running. When Multics is booted from BOS, this is an easy matter, since BOS will read in the beginning of collection 0, leaving the hardware in a known and good state and providing a description of the configuration (config_deck) around. When not booted from BOS, that is, when booted via the IOM boot function, collection 0 has the task of getting the hardware into a good and known state and finding out on what hardware it is working. Once collection 0 has set up the hardware, it can load collection 1 into memory. Collection 1 contains the modules needed to support programs written in pl/1; thus, this loading activates the pl/1 environment. After this time, more sensible programs can run and begin the true process of initialization. The result of this collection is to provide an environment in which pl/1 programs can run, within the confines of memory.

PAGE CONTROL INITIALIZATION

The main task of collection 1 is to make page control operative. This is necessary so that we may page the rest of the initialization programs (initialization programs all have to fit into memory until this is done). The initialization of page control involves setting up all of the disk and page control data bases. Also, the interrupt mechanism must be initialized. The result of this collection is to provide an environment in which i/o devices may be operated upon through normal mechanisms (i.e., via page faults or direct calls to the standard device control modules) but in which the storage system is not active. At the final end of collection 1, this environment becomes paged, using a special region of the disks (the hardcore partition) so that the storage system is not affected.

Collection 1 can be run multiple times. The effect of making a pass through collection 1 is to set up the device tables (and general configuration describing tables) to reflect a new configuration. The various passes of collection 1 are the key to the operation of bce. There are several times when the running of collection 1 is necessary. It is necessary when we first start up, to allow accessing the hardware units "discovered" by collection 0. Once the correct configuration is determined via bce activities, collection 1 must be re-run to allow all of the devices to be accessible during the rest of initialization and Multics service proper. Finally, when the crash environment is restored (see below), another pass must be made to provide accessibility to the devices given the state at the time of the crash.

FILE SYSTEM INITIALIZATION

With paging active, collection 2 can be read into a paged environment. Given this environment, the major portion of the rest of initialization occurs. Segment, directory and traffic control are initialized here, making the storage system accessible in the process. The result of this collection is an environment that has active virtually all hardcore mechanisms needed by the rest of Multics.

OUTER RING ENVIRONMENT INITIALIZATION

Collection 3 is basically a collection of those facilities that are required to run in outer rings. In particular, it contains the programs needed to provide the initializer's ring one environment, especially the code to perform a reload of the system (especially the executable libraries). After the execution of this collection, the Initializer enters into a ring one command environment, ready to load the system (if necessary) and start up the answering service. (Activities performed from ring one onward are not covered in this SDN.)

BOOTLOAD COMMAND ENVIRONMENT (BCE)

The bootload command environment is an environment that can perform configuration and disk management functions. It needs to be able to support i/o to devices in a pl/1 environment. Also, since bce must be able to operate on arbitrary disks, it must be capable of running before the storage system is active. Thus, it is equivalent to the collection 1 environment before the environment becomes paged. In this environment, built by a special run of collection 1, a series of facilities provides a command environment that allows pl/1 programs to run in a manner similar to their operation in the normal Multics programming environment.

CRASH HANDLER (TOEHOLD)

When Multics has crashed, Multics is incapable of performing the types of analysis and recovery operations desired in its distressed state. Thus, a safe environment is invoked to provide these facilities. Since bce is capable of accessing memory and disks independently of the storage system (and the hardcore partitions), it becomes the obvious choice for a crash environment. When Multics crashes, bce is restored to operation. Facilities within bce can perform a dump of Multics as well as start recovery and reboot operations. The crash environment consists of the mechanisms needed to save the state of Multics upon a crash and to re-setup the bootload command environment. These mechanisms must work in the face of varying types of system failures; they must also work given the possibility of hardware reconfiguration since the time the safe environment was saved.

SECTION 2

COLLECTION 0

Collection 0 in Bootload Multics is an ensemble of ALM programs capable of being booted from BOS or the IOM, reading themselves off of the boot tape, loading tape firmware if needed, setting up an I/O and error handling environment, and loading collection 1.

Collection 0 is organized into two modules: bootload_tape_label, and bound_bootload_0. The first is an MST label program designed to read the second into its correct memory location, after being read in by the IOM bootload program. The second is a bound collection of ALM programs. bound_bootload_0 takes extensive advantage of the binder's ability to simulate the linker within a bound unit. The programs in bound_bootload_0 use standard external references to make intermodule references, and the binder, rather than any run-time linker or pre-linker, resolves them to TSR-relative addresses. Any external references (such as to the config deck) are made with explicit use of the fact that segment numbers for collection 0 programs are fixed at assembly time.

GETTING STARTED

bootload_tape_label is read in by one of two means. In native mode, the IOM or IIOC reads it into absolute location 30, leaving the PCW, DCW's, and other essentials in locations 0 through 5. The IIOC leaves an indication of its identity just after this block of information.

In BOS compatibility mode, the BOS BOOT command simulates the IOM, leaving the same information. However, it also leaves a config deck and flagbox (although bce has its own flagbox) in the usual locations. This allows Bootload Multics to return to BOS if there is a BOS to return to. The presence of BOS is indicated by the tape drive number being non-zero in the idcw in the "IOM" provided information.

The label overlays the interrupt vectors for the first two IOM's. Because the label is formatted as a Multics standard tape record, it has a trailer that cannot be changed. This trailer overlays the interrupt vectors for channels B9 and B10. Without a change in the label format, the bootload tape controller cannot use either of these channels as a base channel, because the label record wipes out the vectors that the IOM bootload programs sets up. This prevents control from transferring to the label program.

The label program first initializes the processor by loading the Mode Register and the Cache Mode Register, and clearing and enabling the PTWAM and the SDWAM. It then reads all of bound_bootload_0 off the tape. This action places the toehold and bootload_early_dump into their correct places in memory, in as much as these two modules are bound to be the first two objects in bound_bootload_0. If this is successful, it transfers to the beginning of bootload_abs_mode through an entry in the toehold. (This entry contains the address of bootload_abs_mode, via the linking performed by the binder.) This program copies the template descriptor segment assembled into template_slt_ to the appropriate location, copies int_unpaged_page_tables and unpaged_page_tables to their correct locations, loads the DSBR and the pointer registers, enters appending mode, and transfers to bootload_0.

PROGRAMMING IN COLLECTION 0

Collection 0 programs are impure assembly language programs. The standard calling sequence is with the tsx2 instruction. A save stack of index register 2 values is maintained using id and di modifiers, as in traffic control. Programs that take arguments often have an argument list following the tsx2 instruction. Skip returns are used to indicate errors.

The segment bootload_info, a cds program, is the repository of information that is needed in later stages of initialization. This includes tape channel and device numbers and the like. The information is copied into the collection 1 segment sys_boot_info when collection 1 is read in.

MODULE DESCRIPTIONS

bootload_abs_mode.alm

As mentioned above, bootload_abs_mode is the first program to run in bound_bootload_0. The label program locates it by virtue of a tra instruction at a known place in the toehold (whose address is fixed); the tra instruction having been fixed by the binder. It first clears the memory used by the Collection

0 data segments, then copies the template descriptor segment, int_unpaged_page_tables and unpaged_page_tables from template_slt_. The DSBR is loaded with the descriptor segment SDW, the pointer registers are filled in from the ITS pointers in template_slt_, and appending mode is entered. bootload_abs_mode then transfers control to bootload_0$begin, the basic driver of collection zero initialization.

bootload_0.alm

bootload_0's contract is to set up the I/O, fault, and console services, and then load and transfer control to collection 1. As part of setting up the I/O environment, it must load tape firmware in the bootload tape MPC if BOS is not present. bootload_0 makes a series of tsx2 calls to set up each of these facilities in turn. It calls bootload_io$preinit to interpret the bootload program left in low memory by the IOM/IIOC/IOX, including checking for the presence of BOS; bootload_flagbox$preinit to set flagbox flags according to the presence of BOS; bootload_faults$init to fill in the fault vector; bootload_slt_manager$init_slt to copy the data from template_slt_ to the SLT and name_table; bootload_io$init to set up the I/O environment; bootload_console$init to find a working console and initialize the console package; bootload_loader$init to initialize the MST loading package; bootload_tape_fw$boot to read the tape firmware and load it into the bootload tape controller; bootload_loader$load_collection to load Collection 1.0; bootload_loader$finish to copy the MST loader housekeeping pointers to their permanent homes; and bootload_linker$prelink to snap all links in Collection 1.0.

Finally, the contents of bootload_info are copied into sys_boot_info. Control is then transferred to bootload_1.

The firmware collection.

As described below under the heading of "bootload_tape_fw.alm", tape firmware must be present on the MST as ordinary segments. It must reside in the low 256K, because the MPC's do not implement extended addressing for firmware loading. The tape firmware segments are not needed after the MPC is loaded, so it is desired to recycle their storage. It is desired to load the MPC before collection 1 is loaded, so that backspace error recovery can be used when reading the tape. The net result is that they need to be a separate collection. To avoid destroying the old correspondence between collection numbers and sys_info$initialization_state values, this set exists as a sub-collection. The tape firmware is collection 0.5, since it is loaded before collection 1. The segments in collection 0.5 have a fixed naming convention. Each must include among its set of names a name of the form "fwid.Tnnn", where "Tnnn" is a four character controller type currently used by the BOS FWLOAD facility. These short names are retained for two reasons. First, they are the controller types used by Field Engineering. Second, there is no erase and kill processing on input read in this environment, so that short strings are advantageous. Note that if the operator does make a typo and enters the wrong string, the question is asked again.

bootload_console.alm

bootload_console uses bootload_io to do console I/O. Its initialization entry, init, finds the console on the bootload IOM. This is done by first looking in the config deck, if BOS left us one, or, if not, by trying to perform a 51 (Write Alert) comment to each channel in turn). Only console channels respond to this command. When a console is found, a 57 (Read ID) command is used to determine the model.

The working entrypoints are write, write_nl, write_alert, and read_line. write_nl is provided as a convenience. All of these take appropriate buffer pointers and lengths. Read_line handles timeout and operator error statuses.

There are three types of console that bootload_console must support. The first is the original EMC, CSU6001. It requires all its device commands to be specified in the PCW, and ignores IDCW's. The second is the LCC, CSU6601. It will accept commands in either the PCW or IDCW's. The third type is the IPC-CONS-2. In theory, it should be just like the LCC except that it does NOT accept PCW device commands. Whether or not it actually meets this specification has yet to be determined.

To handle the two different forms of I/O (PCW commands versus IDCW's), bootload_console uses a table of indirect words pointing to the appropriate PCW and DCW lists for each operation. The indirect words are setup at initialization time. The LCC is run with IDCW's to exercise the code that is expected to run on the IPC-CONS-2.

bootload_dseg.alm

bootload_dseg's task is to prepare SDW's for segments loaded by bootload_loader, the collection zero loader. bootload_dseg$make_sdw takes as an argument an sdw_info structure as used by sdw_util_, and constructs and installs the SDW. The added entrypoint bootload_dseg$make_core_ptw is used by bootload_loader to generate the page table words for the unpaged segments that it creates.

bootload_early_dump.alm

When an error occurs during early initialization, bootload_early_dump is called. It is called in three ways. First, if bootload_error is called for an error (as opposed to a warning), this routine is called. Secondly, if a crash should occur later in initialization (after collection 0) but before the toehold is set up (and bce running), the toehold will transfer here. Third, the operator can force a transfer to this routine through processor switches any time up until collect_free_core runs. (This includes while bce is running.) This is done by force executing a "tra 30000o" instruction.

bootload_early_dump starts by reestablishing the collection 0 environment (masked, pointer registers appropriately set, etc.). It then uses bootload_console to ask for the number of a tape drive on the bootload tape controller to use for the dump. When it gets a satisfactory answer, it dumps the first 512k of memory (that used by early initialization and bce), one record at a time, with a couple of miscellaneous values used by read_early_dump_tape (which constructs a normal format dump). If an error occurs while writing a record, the write is simply retried (no backspace or other error recovery). After 16 consecutive errors, the dump is aborted, a status message printed, and a new drive number requested.

bootload_error.alm

bootload_error is responsible for all the error messages in collection 0. It is similar in design to page_error.alm; there is one entrypoint per message, and macros are used to construct the calls to bootload_formline and bootload_console. bootload_error also contains the code to transfer to bootload_early_dump. There are two basic macros used: "error", which causes a crash with message, and "warning", which prints the message and returns. All the warnings and errors find their parameters via external references rather than with call parameters. This allows tra's to bootload_error to be put in error return slots, like:

```
        tsx2      read_word
        tra       bootload_error$console_error
                                           " error, status in
                                           " bootload_console$last_error_status
        ...                                " normal return
```

Warnings are called with tsx2 calls.

bootload_faults.alm

bootload_faults sets up the segment fault_vector. All faults except timer runout are set to transfer to bootload_error$unexpected_fault. All interrupts are set to transfer control to bootload_error$unexpected_interrupt, since no interrupts are used in the collection zero environment. The same structure of transfers through indirect words that is used in the service fault environment is used to allow individual faults to be handled specially by changing a pointer rather than constructing a different tra instruction (also, instructions do not allow "its" pointers within them). The structure of the scu/tra pairs (but not the values of the pointers) formed by bootload_faults is that used by the rest of initialization and service.

bootload_flagbox.alm

bootload_flagbox zeroes the bce flagbox. It also zeroes the cold_disk_mpc flag when BOS is present for historical reasons. Various values are placed in the flagbox that no one looks at. This program is responsible for the state of the BOS toehold as well. It copies the BOS entry sequences into the bce toehold and sets the bce entry sequence into the BOS toehold for the sake of operators who enter the wrong switches.

bootload_formline.alm

This program is a replacement for the BOS erpt facility. It provides string substitutions with ioa_-like format controls. It handles octal and decimal numbers, BCD characters, ascii in units of words, and ACC strings. Its only client is bootload_error, who uses it to format error message. The BCD characters are used to print firmware ID's found in firmware images. Its calling sequence is elaborate, and a macro, "formline", is provided in bootload_formline.incl.alm

bootload_info.cds

The contents of this segment are described under data bases.

bootload_io.alm

bootload_io is an io package designed to run on IOM's and IIOC's. It has entrypoints to connect to channels with and without timeouts. It always waits for status after a connection. It runs completely using abs mode i/o, and its callers must fill in their DCW lists with absolute addresses. This is done because

NSA IOM's do not support rel mode when set in PAGED mode, and there is no known way to find out whether an IOM is in paged mode. Under normal operation, the config card for the IOM is available to indicate whether the IOM is in paged mode or not, relieving this difficulty.

The preinit entrypoint is called as one of the first operations in collection 0. Besides setting up for i/o, it copies and determines from the IOM/IIOC/BOS provided boot info the assume_config_deck (BOS present) flag and the system_type value.

bootload_linker.alm

bootload_linker is responsible for snapping all links between collection one segments. It walks down the LOT looking for linkage sections to process. For each one, it considers each link and snaps it. It uses bootload_slt_manager$get_seg_ptr to find external segments and implements its own simple definitions search.

bootload_loader.alm

bootload_loader is the collection zero loader (of collections 0.5 and 1). It has entrypoints to initialize the tape loader (init), load a collection (load_collection), skip a collection (skip_collection), and clean up (finish). The loader is an alm implementation of segment_loader.pl1, the collection 1 loader. It reads records from the mst, analyzes them, splitting them into slt entries, definitions and linkage sections, and segment contents. Memory is obtained for the segment contents using allocation pointers in the slt. Page tables are allocated for the segment within the appropriate unpaged_page_tables segment. When proper, the breakpoint_page is added as another page to the end of the segment. Definitions and linkage sections are added to the end of the proper segments (ai_linkage, wi_linkage, ws_linkage, as_linkage). The loader has a table of special segments whose segment numbers (actually ITS pointers) are recorded as they are read in off of the tape. These include the hardcore linkage segments, needed to load linkage sections, definitions_, and others. The loader maintains its current allocation pointers for the linkage and definitions segments in its text. bootload_loader$finish copies them into the headers of the segments where segment_loader expects to find them.

bootload_slt_manager.alm

bootload_slt_manager is responsible for managing the Segment Loading Table (SLT) for collection zero. It has three entries. bootload_slt_manager$init_slt copies the SLT and name table templates from template_slt_ to the slt and name_table segments. bootload_slt_manager$build_entry is called by bootload_loader to allocate a segment number and fill in the SLT and name table from the information on the MST. bootload_slt_manager$get_seg_ptr is called by bootload_linker to search the SLT for a given name. It has imbedded in it a copy of hash_index_ used to maintain a hashed list of segment names compatible with the list for slt_manager in further collections.

bootload_tape_fw.alm

bootload_tape_fw is responsible for loading the bootload tape MPC. It begins by loading collection 0.5 into memory with a call to bootload_loader$load_collection. By remembering the value of slt.last_init_seg before this call, bootload_tape_fw can tell the range in segment numbers of the firmware segments. Firmware segments are assigned init_seg segment numbers by bootload_loader, but are loaded low in memory, for reasons described above. bootload_tape_fw then determines the correct firmware type. If bootload_info specifies the controller type, then it proceeds to search the SLTE names of the firmware segments for the appropriate firmware. If bootload_info does not specify the firmware type, then bootload_tape_fw must ask the operator to supply a controller type. This is because there is no way to get a controller to identify itself by model.

Each of the firmware segments has as one of its SLTE names (specified in the MST header) the six character MPC type for which it is to be used. bootload_tape_fw walks the slt looking for a firmware segment with the correct name. If it cannot find it, it re-queries (or queries for the first time) the operator and tries again.

Having found the right firmware, the standard MPC bootload sequence is initiated to boot the tape MPC. The firmware segments' SDW's are zeroed, and the slt allocation pointers restored to their pre-collection-0.5 values. bootload_tape_fw then returns.

template_slt_.alm

This alm program consists of a group of involved macros that generate the SLTE's for the segments of collection zero. It is NOT an image of the segment slt, because that would include many zero SLTE's between the last sup seg in collection zero and the first init seg. Instead, the init seg SLTE's are packed in just above the sup segs, and bootload_slt_manager$init_slt unpacks them. It also contains the template descriptor segment, packed in the same manner, and the template name table. The initial contents of int_unpaged_page_tables and unpaged_page_tables are also generated. Also present are the absolute addresses, lengths, and pointers to each of the collection 0 segments for use elsewhere in bound_bootload_0.

SECTION 3

COLLECTION 1

The basic charter of collection 1 is to set up paging, fault handling, as well as various data bases needed for paging and other like activities. Collection 1 can run multiple times, for various reasons.

SUMMARY OF COLLECTION 1 PASSES

The first run through collection 1 is known as the "early" pass which is described below. It is a run in which we are restricted to work within 512K and in which only the rpv is known; in fact, it is this pass which finds the rpv and the config deck. If BOS is present, this pass is not needed. The end of this pass is the arrival at "early" command level, used to fix up the config deck, in preparation for the "boot" pass.

The second pass, which is known as "bootload Multics initialization", also runs only within 512K. It, however, has knowledge of all disks and other peripherals through the config deck supplied either by BOS or the early initialization pass. This pass is made to generate a crash-to-able system that can be saved onto disk for crash and shutdown purposes. After the crash handler (this image) is saved, the bootload Multics "boot" command level can be entered. This level allows the booting of Multics service. After Multics has shut down, a slight variant of this pass, the "shut" pass, is run in a manner similar to that for the "crash" pass, described below.

The third pass (which actually comes after the fourth) is another run of bootload Multics initialization performed after Multics has crashed. This pass is made to re-generate various tables to describe the possibly different configuration that now exists after having run Multics. Bootload Multics "crash" command level is then entered.

The fourth pass through collection 1 is called "service initialization", which runs using all memory and devices. The result of this pass is suitable for running the later collections, and bringing up service.

The "early" pass creates a safe environment consisting of a set of programs in memory and a synthesized config deck that describes known hardware. This is saved away to handle crashes during the "boot" pass. If the "boot" pass fails, the toehold restores this earlier saved environment which then runs a "re_early" pass. This is really a normal pass, but using the saved away config deck of known good hardware. The "re_early" pass comes back to an "early" command level to allow the operator to fix the deck or hardware.

When the "boot" pass succeeds, it also saves a good memory image and the now confirmed site config deck. After the "boot" pass saves this image, the "boot" command level is entered and eventually it boots Multics, running the "service" pass. If this fails, the toehold restores the saved image. A "bce_crash" pass then runs. This is a normal pass but one in which the saved config deck is used. This pass is run on the assumption that, either a bce command died and the operator may now examine it, or that the "service" pass found a problem. The "bce_crash" level allows the operator to fix things.

Once the boot of service Multics completes collection 1, a crash or shutdown will invoke the toehold to restore bce. This time, however, the current config deck is used to utilize any reconfigurations that have occured. bce will come to the "crash" or "boot" command levels.

We'll start by looking at the basic initialization pass, that used to come to the normal ("boot") bce command level.

NORMAL (BOOT) PASS

The sequence of events in a normal initialization pass is given here. As of the time of the start of a normal initialization pass, the config deck has been found, either by BOS or the early initialization pass. All other data bases besides sys_boot_info and sys_info set or created during previous initialization passes have been deleted. The pass starts with saving certain attributes, such as free core extents, for later restoration at the end of the pass (before running another).

scs_and_clock_init fills in the initial scs (system configuration segment) data from the config deck. This is information on the processors and the memory controllers.

get_io_segs, iom_data_init, ocdcm_$init_all_consoles, and scas_init are run to set up the disk_seg, pvt, iom_data, ioi_data, oc_data and the system controller addressing segment.

tc_init initializes tc_data's apte and itt lists.

init_sst generates the sst and core map appropriate for the pass. This is the last real memory allocation. After this time, allocation of memory based upon the data in the slt is deactivated. The remaining tables either have memory already allocated for them or are generated paged, once paging is started. announce_chwm announces memory usage.

initialize_faults$interrupt_init initializes the interrupt vector. With iom_data and oc_data set up, this permits ocdcm_ to be used for console I/O. The interrupt mask is opened with a call to pmut$set_mask.

The basic command environment facilities (I/O interfaces and a free area) are set up in a call to init_bce. (BCE is an acronym for Bootload Command Environment). This allows programs that query the operator to do so in a more friendly fashion than raw calls to ocdcm_. Further descriptions of bce facilities follow later.

load_disk_mpcs runs (only during a "boot" pass and only when we do not have BOS present) to make sure that all disk mpcs have firmware active within them.

init_pvt, read_disk$init and init_root_vols together have the net effect of setting up disk and page control. No segments are paged at this time, though, except for rdisk_seg. Once we reach here, we know that the config deck describes a set of hardware sufficient (and valid) enough to reach command level and so we save the config deck as safe_config_deck.

establish_temp_segs maps the bootload paged temp segments onto the reserved area for them in the "bce" partition. find_file_partition maps the bce file system area (bootload_file_partition) unto the "file" partition.

load_mst$init_commands maps the pagable bce programs onto the areas of disk in which they were read by load_mst earlier.

If this is a "early" or "boot" pass, this environment is saved and the toehold setup to invoke it. This is done by init_toehold. The "early" pass saves the entire environment; the "boot" pass simply saves the safe_config_deck so determined by this pass.

bce_get_to_command_level can now be called to provide the appropriate bce command level. At the "early" command level, the config deck must be made to be correct. At the "boot" command level, the mpcs (other than the bootload tape mpc and all of the disk mpcs) need to be loaded.

Within the command level, the config deck (on disk, disk_config_deck) may have been modified. This is read in, via establish_config_deck, for the next initialization pass. For cold boots, the generated config deck is written out instead.

When the pass is over, the states saved at the beginning of the pass are restored, the system is masked, and we proceed to perform another pass.

SERVICE PASS

The sequence of events in a service pass differs from the normal pass in many ways.

After initialize_faults$fault_init_one runs, move_non_perm_wired_segs is called to move the segments loaded by collection 0 to their proper places, thereby utilizing all of the bootload memory.

[Collection 0 assumes 512K of bootload memory, for two reasons. First, if BOS and the config deck are not present, there is no easy way of finding out how much memory there is, so some assumption is needed. Second, the crash handler will have to run in some amount of memory whose contents are saved on disk. 512K is a reasonable amount of space to reserve for a disk partition. At current memory and disk prices it is hard to imagine anyone with a bootload controller with less that 512K, or a problem with the disk partition.

When setting up the service environment, though, it is necessary to move the segments that have been allocated in the 512K limit. It is desirable to have sst_seg and core_map at the high end of the bootload memory controller. (On the one hand, the controller they reside in cannot be deconfigured. On the other hand, only the low 256K of memory can be used for I/O buffers on systems with IOM's not in paged mode. While we could just start them at the 256K point, that might produce fragmentation problems. So the top of the controller is best.) If the controller really has 512K of memory, collection 1 paged segments will be there. move_non_perm_wired_segs takes the segments that the collection zero loader allocated high (paged segments and init segments that are not firmware segments) and moves them to the highest contiguously addressable memory, hopefully leaving the top of the low controller for the sst_seg and core_map.]

tc_init sets the number of aptes and itt entries on the basis of the tcd card. A normal bce pass really needs no such entries.

init_sst generates the sst and core map appropriate for all of memory at the top of the bootload memory. A normal pass allocates these tables through normal off-the-slt allocation (because the top of the 512k area is filled with temp segs).

Since the service pass does not come to bce command level, establish_temp_segs, find_file_partition and load_mst$init_commands are not run.

init_toehold is not run since upon a crash we want to return to the bootload environment and not to a state in which we are booting.

init_partitions checks the "part" config cards.

Now, the routine we've all been waiting for runs. make_segs_paged causes all pagable segments to be paged into the various hardcore partitions thereby no longer needing memory. We can then run collect_free_core to regain the freed space.

delete_segs$temp deletes the segments temporary to collection 1. We can then load, link, and run collection 2 (performed by segment_loader, pre_link_hc and beyond).

EARLY PASS

The early initialization pass is a pass through collection 1 whose job is to set up paging and obtain the config deck from its disk partition so that a normal initialization pass may be run which knows about the complete set of hardware.

It starts with init_early_config constructing a config deck based on assumptions and information available in sys_boot_info. This config deck describes the bootload CPU, the low 512K of memory, the bootload IOM, the bootload tape controller and the bootload console. Given this synthetic deck, we can proceed through scs_and_clock_init, etc. to setup the environment for paging. scs_and_clock_init$early fills the bootload CPU port number into the config deck, which is how it differs from scs_and_clock_init$normal.

scas_init and init_scu (called from scas_init) have special cases for early initialization that ignore any discrepancy between the 512K used for the bootload controller and any larger size indicated by the CPU port logic.

During the early pass (or, actually during the first "boot" pass, if an early pass is never run), init_bce$wired sets up references in bce_data to wired objects. This allows bce_console_io and other friendlier routines to run.

To locate the RPV subsystem, find_rpv_subsystem looks in sys_boot_info. If the data is there, it will try to boot the RPV subsystem firmware (if needed). If not, it queries the operator for the data. If, later in initialization, the data should prove suspect (e.g. RPV label does not describe the RPV), control returns here to re-query the operator. The operator is first asked for a command line specifying the RPV subsystem model and base channel, and the RPV drive model and device number. The operator may request that the system generate a query in detail for each item. Cold boot is also requested in the find_rpv_subsystem dialog. The simple command processor, bce_command_processor_, is used to parse the "cold" and "rpv" request lines described above.

The RPV data is filled into the config deck, and initialization continues with init_pvt and friends. init_root_vols is called through its early entrypoint so as to allow for an error return. Errors occuring during the initing of the rpv will cause a re-query of the rpv data by returning to the call to get_io_segs.

Firmware is booted in the RPV controller by boot_rpv_subsystem, called from find_rpv_subsystem, which finds the appropriate firmware image and calls hc_load_mpc. A database of device models and firmware types and other configuration rules, config_data_.cds, is used to validate operator input and, for example, translate the subsystem model into a firmware segment name.

init_roots_vols checks for the presence of and creates certain key partitions on the rpv. The "conf" partition, if not present, is created by trimming 4 pages off of the hardcore partition. The "bce" (bce crash handler, temporary area and MST storage) and "file" (bootload file system) partitions are created, if any is not found, by a call to create_rpv_partition. This program shuffles the disk pages to find enough contiguous space at the end of the disk for the partitions.

After running establish_temp_segs and find_file_partition, the rest of the MST is read. This step is performed during the "early" pass or whatever is the first boot pass. tape_reader$init sets up tape reading. load_mst reads in collection 1.2 (config deck sources and exec_coms) into bce file system objects, collection 1.5 (bce paged programs and firmware images) into mst area pages leaving around traces for load_mst$init_commands (which maps them into the bce address space) and saves collections 2 and 3 on disk for warm booting. tape_reader$final shuts down the tape. load_mst$init_commands then runs.

The early or the first boot pass then initializes bce_data references to paged objects with init_bce$paged.

An early command level is now entered, using a subset of the real bce command level commands. This level is entered to allow editing of the config deck.

After leaving command level, init_clocks is called. This is the time when the operator sets the clock. Up until this time, the times shown were random. If the operator realizes at this time that he must fix the config deck, or whatever, he has a chance to return to the early command level. When the clock is set, control proceeds.

At this point, early initialization's work is done. The real config deck is read in (by establish_config_deck), and the system can rebuild the wired databases to their real sizes. Interrupts are masked, completion of pending console I/O is awaited, and the slt allocation pointers are restored to their pre-collection-1 values. Control then moves to the "boot" pass.

CRASH PASS

The crash pass recreates a "boot" environment from which dumps can be taken and emergency_shutdown can be invoked. It differs from the "boot" pass only in the verbosity (to avoid printing many messages at breakpoints) and in the command level that is reached.

RE_EARLY PASS

A re_early pass is run to restore a safe environment following a failure to boot to the "boot" command level. It is identical to a "boot" pass except that it uses a saved config deck known to be good and reaches a "early" command level.

BCE_CRASH PASS

The bce_crash pass is run to restore a safe environment following a failure to boot the "service" pass. This may also be the result of a failure of a bce utility invoked at the "boot" command level. This pass is identical to the boot pass except that it uses a saved config deck known to be good and reaches the "bce_crash" command level.

SHUT PASS

The shut pass is run when Multics shuts down, as opposed to crashing. It differs from the boot pass only in that load_disk_mpcs is not run, because it shouldn't be ncessary (Multics was using the mpcs okay) and because it would interfere with possible auto exec_com operation.

MODULE DESCRIPTIONS

Bootload Command Environment modules are not included in this section.

announce_chwm.pl1

The name of this program means announce_Core_High_Water_Mark. It will announce the extent to which memory is filled during the various passes of collection 1 when the "chwm" parameter appears on the "parm" card in the config deck. Near the beginning of each pass, this program announces the amount of memory used, based upon information in the slt. At the end of service initialization, it walks down the core map entries, looking for pages that are available to page control and those that are wired. The difference between the memory size and the total figure given here is the amount taken up by non-page control pages, the sst for example. As a side bonus, the before entrypoint announces the usage of int_unpaged_page_tables; the after entrypoint announces the usage for unpaged_page_tables.

boot_rpv_subsystem.pl1

boot_rpv_subsystem is the interface between find_rpv_subsystem and hc_load_mpc, the hardcore firmware loading utility. All that it really has to do is find the appropriate firmware segment in collection 1. config_data_ is used to map the controller model to a firmware segment name, of the usual (T&D) form (fw.XXXnnn.Ymmm). The segment and base channel are passed to hc_load_mpc, and the results (success or failure) are returned to find_rpv_subsystem.

boot_tape_io.pl1

This is the program that performs reading of the MST by collections 1 and beyond. It uses the physical record buffer as an i/o area. io_manager is used to perform the i/o, with dcw lists generated within this program.

bootload_1.alm

bootload_1 is the first collection 1 program, called directly by collection 0. It fills in the stack headers of the prds and inzr_stk0 to initialize the PL/1 environment. It then calls initializer.pl1 which pushes the first stack frame.

collect_free_core.pl1

At the end of collection 1 service initialization, this program is called to free the storage taken up by the previously wired initialization segments. It does this by marking all core map entries for pages still unpaged (judged from the address field of the sdws of all segments) as wired and marking all of the rest as free (available for paging). It special cases breakpointable segments to avoid freeing references to breakpoint_page.

create_rpv_partition.pl1

To save the effort of creating the new Bootload Multics partitions by requiring all sites to perform a rebuild_disk of their rpv, this program was created. It creates partitions on rpv (high end) by shuffling pages about so as to vacate the desired space. The pages to move are found from the vtoces. The vtoces are updated to show the new page location and the volmap is updated to show the new used pages. This program uses read_disk to read and write the pages. No part of the file system is active when this program runs.

delete_segs.pl1

delete_segs is called after the various collections to delete the segments specific only to that collection (temp segs). It is also called at the end of collection 3 to delete segments belonging to all of initialization (init segs). It scans the ast list for the appropriate segments, uses pc$truncate to free their pages (in the hardcore partition) or pc$cleanup to free the core frames for abs-segs and then threads the astes into the free list. This program is careful not to truncate a breakpoint_page threaded onto a segment.

disk_reader.pl1

disk_reader is used by the collection 1 loader (of collection 2), segment_loader, and by the collection 2 loader, load_system, to read the mst area of disk. It operates by paging disk through disk_mst_seg. The init entrypoint sets up disk_mst_seg unto the first 256 pages of the mst area to be read. As requests come in to read various words, they are paged from this segment. When a request comes in that is longer than what is left in this segment, the remainder is placed into the caller's buffer, and disk_mst_seg re-mapped onto the next 256 pages. This continues as needed.

establish_config_deck.pl1

The config deck is stored in the "conf" partition on the RPV in between bootloads. It runs in one of two ways, depending on whether it is setting up for service or bce use. For bce use, a abs-seg is created which describes the disk version. config_deck still describes the memory version. If it is necessary to read in the disk version, abs_seg is copied to config_deck. Likewise, if some program (config_deck_edit_ in particular) wants to update the disk version, abs_seg is again used, receiving the contents of config_deck. During service, config_deck is itself both wired an an abs-seg on the disk partition. This is done by creating an aste whose ptws describe memory. We make the core map entries for the pages occupied by config_deck describe this aste and the disk records of the conf partition. These cme's are threaded into page controls list (equivalent of freecore) providing a valid wired segment, at the address of config_deck.

fill_vol_extents_.pl1

This is the ring 1 program that obtains, through the infamous "init_vol loop", the desired parameters of a disk to initialize. It is called in initialization by init_empty_root when performing a cold boot to determine the desired partitions and general layout desired for the rpv.

find_rpv_subsystem.pl1

find_rpv_subsystem initializes configuration and firmware for the RPV disk subsystem. When available, it uses information in sys_boot_info. When that information is not present, the operator is queried. The basic query is for a request line of the form:

rpv Icc MPC_model RPV_model RPV_device

or

cold Icc MPC_model RPV_model RPV_device

as described in the MOH.

If the operator makes a mistake, or types help, the operator is offered the opportunity to enter into an extended, item by item dialog to supply the data.

The information is checked for consistency against config_data_, a cds program that describes all supported devices, models, etc. The mpc is tested through hc_load_mpc$test_controller, to see if firmware is running in it. If the response is power off, then boot_rpv_subsystem is called to load firmware. Then init_early_config$disk is called to fill this data into the config deck. If a later stage of initialization discovers an error that might be the result of an incorrect specification at this stage, control is returned here to give the operator another chance.

The operator is also allowed to enter "skip_load" or "skip", as a request before entering the rpv data. This forces a skip of the firmware loading, regardless of the apparent state of the mpc.

get_io_segs.pl1

A scan through the config deck determines the sizes of the various hardcore i/o databases which this program allocates. This program also fills in some of the headers of these databases as a courtesy for later initialization programs. The key determiners of the sizes of the tables allocated are the number of subsystems, the number of logical channels to devices, the number of drives, the number of ioms, etc. get_main is used to allocate the areas, using entries in the slt to find the memory. Areas allocated are: the pvt, the stock_segs, the disk_seg, ioi_data, iom_data and io_config_data.

get_main.pl1

get_main is used to create a segment that is to reside in main memory. It runs in one of two ways, depending on whether allocation off the slt (slt.free_core_start) is allowed. When this is not allowed (later in initialization), make_sdw$unthreaded is used to generate the segment/aste. pc_abs$wire_abs_contig forces this segment to be in memory. Earlier in initialization (before page control is active), the segment is allocated from the free core values in the slt. These values determine the placement in memory of the to be created segment. get_main allocates a page table for this segment in either int_unpaged_page_tables or unpaged_page_tables (depending on whether the segment will eventually be made paged). The ptws are filled in and an sdw made. The given_address entrypoint of get_main can be used to utilize its unpaged segment page table generation capabilities (as in init_sst).

hc_load_mpc.pl1

hc_load_mpc embodies the protocol for loading all MPC's. It is an io_manager client. Since the firmware must be in the low 256K, a workspace is allocated in free_area_1 and the firmware image is copied out of the firmware segment and into this buffer for the actual I/O. The urc entrypoint is used to load urc mpcs. This entry accepts an array of firmware images to load. It scans the list to determine to which channels each overlay applies. The extra entrypoint test_controller, used by find_rpv_subsystem and load_disk_mpcs, tests a controller by executing a request status operation. The results of this are used to see if the mpc seems to be running (has firmware in it).

init_aste_pools.pl1

This program is called exclusively from init_sst and really does most of its work. It builds the four aste pools with empty astes appropriately threaded. Each aste is filled in with ptws indicating null pages.

init_clocks.pl1

This program performs the setting of the system clock. It starts by providing the time and asking if it is correct. If it is, fine. If the operator says it's not, the operator is prompted for a time in the form:

yyyy mm dd hh mm {ss}

The time is repeated back in English, in the form "Monday, November 15 1982". If the bootload memory is a SCU, the operator is invited to type "yes" to set this time (when the time is met), or "no" to enter another time. The time is set in all the configured memories, to support future jumping clock error recovery. On 6000 SC's, the program translates times to SC switch settings. The program gives the operator time to set the clock by waiting for an input line. At any time, the operator may enter "abort", realizing that something is wrong. init_clocks then returns. real_initializer will re-enter the early command level in this case.

init_early_config.pl1

init_early_config fabricates a config deck based on the information available after collection zero has completed. The bootload CPU, IOM, console, and tape controller are described. The port number of the bootload CPU is not filled in here, since it is not easily determined. Instead, scs_and_clock_init$early fills it in. Appropriate parm, sst, and tcd cards are constructed, and placeholders are filled in for the RPV subsystem, so that iom_data_init will reserve enough channel slots. init_early_config$disk is used to fill in the real values for the RPV subsystem once they are known.

init_empty_root.pl1

fill_vol_extents_, the subroutine used by the user ring init_vol command, has been adapted to provide the main function of this program. It provides a request loop in which the operator can specify the number of vtoces, partition layout, etc. The operator is provided with a default layout, including the usual set of partitions and the default (2.0) average segment length. If it is changed, the operator is required to define at least the hardcore and bce required partitions and (for the moment) the bos partition.

init_hc_part.pl1

init_hc_part builds the appropriate entries so that paging and allocation may be done against the hardcore partition. It builds a pseudo volmap (volmap_abs_seg) describing the hardcore partition (which is withdrawn from the beginning thereof) allowing withdrawing of pages from the partition. A record stock is also created of appropriate size for the partitions.

init_partitions.pl1

This program makes sure that the partitions the operator specified in the config deck are really there. It checks the labels of the config deck specified disks for the specified partitions. Disks that do have partitions so listed are listed as un-demountable in their pvt entries.

init_pvt.pl1

The pvt contains relatively static data about each disk drive (as opposed to dynamic information such as whether i/o is in progress). init_pvt sets each entry to describe a disk. No i/o is done at this time so logical volume information, etc. can not be filled in. Each disk is presumed to be a storage system disk, until otherwise determined later.

init_root_vols.pl1

init_root_vols finds the disks that will be used for hardcore partitions. It mostly finds the disks from root cards and finds the hardcore partitions from the labels. For the rpv, it will also call init_empty_root, if a cold boot is desired, call create_rpv_partition, if various required partitions are missing (MR11 automatic upgrade), and set various pvt entries to describe the rpv. During the service pass, init_hc_part is called to establish paging (and allow withdrawing) against the hardcore partition.

init_scu.pl1

This routine is used within scas_init to init a given scu. It compares the scu configuration information (from its switches) with the supplied size and requirements. When called for bootload Multics purposes, the size of the scu may be larger than that specified (generated) in the config deck without a warning message. It generates ptws so it can address the scu registers (see the description in the glossary for the scas). The execute interrupt mask assignment and mask/port assignment on the memories is checked here.

init_sst.pl1

init_sst starts by determining the size of the pools. Normally, this is found in the sst config card (although init_sst will generate one of 400 150 50 20 if one isn't found). For early and bootload Multics initialization, though, the pools sizes are determined from the current requirements given in figures in bootload_info. The size of the core_map is determined from the amount of configured memory for normal operation and is set to describe 512K for early and bootload Multics operation. The area for the sst is obtained, either from the top of the bootload scu for normal operation, or from the slt allocation method for early and bootload Multics operation. The headers of the sst and core map are filled in. init_aste_pools actually threads the astes generated. The pages of memory not used in low order (or bootload (512k)) memory are added to the core_map as free. For normal operation, the other scu's pages are also added to the free list. collect_free_core will eventually add the various pages of initialization segments that are later deleted.

init_vol_header_.pl1

init_empty_root uses this program to initialize the rpv. This routine writes out the desired label (which describes the partitions filled in by fill_vol_extents_), generates an empty volmap and writes it out, and generates empty vtoces and writes them out.

initial_error_handler.pl1

This any_other handler replaces the fault_vector "unexpected fault" assignments. It implements default_restart and quiet_restart semantics for conditions signalled with info, and crashes the system for all other circumstances.

initialize_faults.pl1

initialize_faults has two separate entries, one for setting things up for collection 1, and one for collections 2 and beyond. This description is for collection 1 (initialize_faults$fault_init_one). initialize_faults_data describes which faults have their fault vectors set to fim$primary_fault_entry (scu data to pds$fim_data), fim$signal_entry (scu data to pds$signal_data), fim$onc_start_shut_entry (scu data to pds$fim_data) or wired_fim$unexp_fault (scu data to prpds$sys_trouble_data) (all others). Special cases are: lockup and timer runout faults are set to an entry that will effectively ignore them. Derails go to fim$drl_entry to handle breakpoints and special drl traps. Execute faults are set to wired_fim$xec_fault (scu data to prds$sys_trouble_data). Page faults are set to pagefault$fault (scu data to pds$page_fault_data). And connect faults are set to prds$fast_connect_code (scu data to prds$fim_data). Write access is forced to certain key programs to set values within them. Access is reset afterwards. These are pointers which must be known by certain programs when there will be no mechanism for the programs themselves to find them. An example is the pointers within wired_fim specifying where scu data is to be stored. The last thing done is to set the signal_ and sct_ptr in the inzr_stk0 stack header so that signalling can occur in collection 1.

initialize_faults_data.cds

This cds segment describes which faults go to where so that initialize_faults can so set them. For collection 1, the major faults set are: command and trouble to fim$primary_fault_entry (scu data in pds$fim_data), access violation, store, mme, fault tag 1, 2 and 3, derail, illegal procedure, overflow, divide, directed faults 0, 2 and 3, mme2, mme3, mme4 to fim$signal_entry (scu data to pds$signal_data), shutdown, op not complete and startup to fim$onc_start_shut_entry (scu data to pds$fim_data) and the rest to wired_fim$unexp_fault (scu data to prds$sys_trouble_data).

initializer.pl1

initializer consists of only calls to real_initializer, delete_segs$delete_segs_init, and init_proc. real_initializer is the main driver for initialization. It is an init seg. initializer exists as a separate program from real_initializer because, after the call to delete init segs, there must still be a program around that can call init_proc. This is the one.

iom_data_init.pl1

The function of this program is to set up the data bases used by io_manager. These include iom_data and the actual mailboxes used in communicating with the iom. The iom cards are validated here. The overhead channel mailboxes are set for the described channels.

load_disk_mpcs.pl1

During the "boot" pass, all disk mpcs must have firmware loaded into them. This is done by load_disk_mpcs. This program scans the config deck, searching for disk mpcs. It tests each one (with hc_load_mpc$test_controller) to determine a list of apparently non-loaded disk mpcs. If this list is not empty, it prints the list and asks the operator for a sub-set of these to load. bce_fwload is used to perform the actual loading.

load_mst.pl1

load_mst reads in the MST. It contains a routine which understands the format of a MST. This routine is supplied with various entry variables to do the right thing with the objects read from the various collections. For collection 1.2, the objects are placed into the bce file system through bootload_fs_. For collection 1.5, the segments have linkages combined, etc. just as in segment loader. The objects are placed on disk, in locations recorded in a table. These are paged bce programs. Collections 2 and 3 are simply read in as is, scrolling down the mst area of the "bce" partition using the abs-seg disk_mst_seg. The init_commands entrypoint uses the table built while reading collection 1.5. The appropriate bce segments are mapped onto disk using the locations therein.

make_sdw.pl1

make_sdw is the master sdw/aste creation program for collection 1 and beyond. It contains many special cases to handle the myriad types of segments used and generated in initialization. It's first job is to determine the size of the desired segment. The size used is the maximum of the slte's current length, maximum length and the size given on a tbls card (if the segment's name is in variable_tables). Also, an extra page is added for breakpoints when needed. Given this size, an appropriate size aste is found and threaded into the appropriate list, either init segs, temp segs, or normal segs. Wired segs aren't threaded; they are just listed as hardcore segments. The page table words are initialized to null addresses. If the segment is wired and is breakpointable, the last ptw is instead set to point to breakpoint_page. For abs-segs, this is the end;

abs segs and other "funny" segs must build their own page tables and a real sdw to describe them. For a normal segment, however, the page table entries are filled as follows: an appropriate hardcore partition to hold the pages is chosen. abs_seg's sdw is set to indicate this null address page table. The various pages are touched, causing page control to be invoked to withdraw an appropriate page against the hardcore partition whose drive index is in the aste. (abs_seg's sdw is then freed.) make_segs_paged and segment_loader, the main clients of make_sdw, will then copy the desired data (either from wired memory or from the tape) into these new (pagable) pages.

make_segs_paged.pl1

make_segs_paged, that most famous of initialization programs, actually, in a way, has most of its work performed by make_sdw. make_segs_paged examines all of the initialization segments, looking for those it can page (i.e., not wired, not already made paged, non-abs-segs, etc.). It walks down this list of segments from the top of memory down, using make_sdw to generate an aste, an sdw, and a page table full of disk pages for it. The sdw is put into dseg, and the contents of the wired segment is copied into the paged version. The pages of memory are then added to page control's free pool The dseg is also copied with a new dbr generated to describe it.

Breakpointable segments are special cased in two ways. First of all, when the pages of the old segment are freed, occurences of breakpoint_page are not. Also, when copying the segment, breakpoints set within it must be copied. All of breakpoint_page cannot be copied since it includes breakpoints in other segments. Thus, we must copy each breakpoint, one at a time by hand.

move_non_perm_wired_segs.pl1

This program takes the segments allocated high addresses by collection 0 (paged segments and init segments that are not firmware segments) which were put at the top of the 512K early initialization memory, and moves them to the top of the contiguously addressable memory, leaving the top of the low controller for the sst_seg and core_map.

This program depends on the knowledge that the loader assigns segment numbers in monotonically increasing order to permanent supervisor and init segs, and that the high segments are allocated from the top of memory down. Thus it can move the highest segment (in memory address) first, and so on, by stepping along the SLTE's.

The copying of the segment can be tricky, though, since not only must the contents be moved but the page table must be changed to reflect the new location. For this, we build abs_seg0 to point to the new location. The segment is copied into abs_seg0. We now make the sdw for the segment equal to that for abs_seg0. The segment is now moved, but we are using the page table for abs_seg0 for it, not the one belonging to it. So, we fix up the old page table to point to the new location, and swap back the old sdw. This starts using the new ptws in the old place.

Segments that were breakpointable (had breakpoint_page in them) must be special cased not to move the breakpoint page.

ocdcm_.pl1

Within initialization, the init_all_consoles entrypoint of ocdcm_ is called. This entrypoint sets up oc_data to a nice safe (empty) state. The various console specific parms are found and saved. The main loop examines all prph opc cards. They are validated (and later listed if clst is specified). For each console, a console entry is filled describing it. The bootload console, when found, is specifically assigned as bootload console. As a last feature, the number of cpus is found. This is because the longest lock time (meaningful for determining time-outs) is a function of the number of processors that can be waiting for an i/o.

ocdcm_ also provides for bce a special function. It maintains wired_hardcore_data$abort_request, set to true whenever the operator hits the request key when this was not solicited (no read pending). This flag is used by bce_check_abort to conditionally abort undesired bce operations.

prds_init.pl1

This program simply initializes certain header variables in the prds. This includes inserting the fast_connect_code, the processor tag, etc.

pre_link_hc.pl1

The linker for collection 2, this program performs a function analogous to that performed by bootload_linker. It walks down the linkage sections of the segments in question, looking for links to snap. slt_manager is used to resolve references to segments. A definition search is imbeded within this program.

read_disk.pl1

read_disk is the routine used to read a page from or to write a page to disk. The init entry point sets up rdisk_seg as a one page paged abs segment for such purposes. Actual page reading and writing consists of using disk_control to test the drive (unless the no_test entrypoints were used), and then page control to page the page. For reads, we construct a page table word describing the page of disk. Touching rdisk_seg then reads it in. For writing, we generate a null address page table entry. When we write to it, a page of memory is obtained. By forcing the core map entry to describe the desired page of disk, unwiring the page and performing a pc$cleanup (force write), the page makes it to disk.

read_disk_label.pl1

To read a disk label, we call read_disk_label. It uses read_disk to preform the i/o. Several such reads will be performed. if necessary. The label is validated through a simple check of label.Multics, label.version and label.time_registered.

real_initializer.pl1.pmac

real_initializer is the main driver for initialization. It largely just calls other routines to set things up, in the proper order.

There are many paths through real_initializer as described above. All paths set an any_other handler of initial_error_handler to catch unclaimed signals, which eventually causes a crash.

The main path through real_initializer calls collection_1 (an internal subroutine) multiple times and then passes through to collections 2 and 3. Each call to collection_1, in the normal case, "increments" sys_info$collection_1_phase, thus producing the main set of collection 1 passes. Various deviations from this exist. Aborting disk mpc loading resets the phase to re_early and branches back to the "early" command level. A failure when finding the rpv during the "early" pass retries the "early" pass. The reinitialize command resets the phase to "early" and then simulates the bce "boot" function, thus making the next pass become a new "boot" pass.

When Multics crashes or shuts down, the toehold restores the machine conditions of bce saved in the toehold. These return the system to save_handler_mc, which quickly returns through init_toehold to real_initializer. The routine collection_1 senses this and returns to the main collection_1 calling loop.

real_initializer keys off the memory_state (determines between crashing and shutting down) and old_memory_state (state of crashed memory - determines crashed collection 1 phase) in the toehold to determine the pass to run next.

real_initializer includes a stop-on-switches facility. pl1_macro is used to assign a unique number to each step in initialization. This number can also be used in the future to meter initialization. Before each step in initialization, a call is made to the internal procedure check_stop. If the switches contain "123"b3 || "PNNN"b6, where PNNN is the error number in binary coded decimal (P is the collection 1 phase, NNN is the stop number obtained from a listing), bce is called (if the toehold is active).

scas_init.pl1

scas_init inits the scas (system controller addressing segment). It is the keeper of things cpu and scu. The config deck is searched for cpu and mem cards which are validated and the boxes' switches validated against the cards. The scs$cow (connect operand words) are filled in here with values so that we may send connects to the various processors. init_scu is called to set masks and such for the various scus. The port enables are set for the ioms. The cpu system controller masks are checked. Finally, if the cpus and ioms do not overlap in port numbers, the cyclic priority switches are set on the scus.

scs_and_clock_init.pl1

This program initializes most of the data in the scs. In previous systems, the scs was mostly filled in its cds source. To support multiple initializations, though, the segment must be reset for each pass. This program also has the task of setting sys_info$clock_ to point to the bootload SCU. Finally, at its $early entrypoint, it fills in the bootload SCU memory port number in the config deck, since it used that data in scs initialization. Initializing the scs consists of initiating data about cpus and scus.

segment_loader.pl1

segment_loader is used to load collections 2.0 and beyond. It uses disk_reader to read records from the MST of disk. The various records from the MST are either collection marks, header records (denoting a segment) or the data forming the segments. Given information in the segment header, an appropriately sized area in wi_linkage$, ws_linkage$, ai_linkage$ or as_linkage$ is generated. slt_manager$build_entry chooses the next segment number (either supervisor of initialization) for the segment and creates the slt entry. make_sdw creates an sdw an the page table and allocates disk space in the hardcore partition for the segment. With read/write access forced for this new (pagable) segment, the segment is read from disk. Access is then set as desired in the header record. We loop in this manner until we encounter a collection mark when we stop.

slt_manager.pl1

This is a relatively simple program. slt_manager$build_entry looks at the header read from an MST and builds a slt entry. The header defines whether this is a supervisor or an initialization segment (which defines from which set of segment numbers (supervisory start at 0, initialization start at 400 octal) it is given), what names to add to the name table, and whether this segment has a pathname which needs to be added to the name table (so that init_branches can thread them into the hierarchy). While it is building the entry, it hashes the names in the same manner as bootload_slt_manager. slt_manager$get_seg_ptr uses this hash list to search for the segment name requested.

sys_info.cds

sys_info is described under data bases.

tape_reader.pl1

tape_reader uses boot_tape_io to read MST tape records. It is capable of reading several tape records and packing them into a user supplied buffer. It validates the tape records it reads for Multics-ness, performing the (old) reading re-written record error recovery mechanism.

tc_init.pl1

tc_init is run in two parts, the second called part_2 run in collection 2. Part one, just called tc_init, allocates an appropriately sized tc_data (see the description of tc_data_header, above) given the supplied number of aptes and itt entries. The workclass entries are initialized to their defaults. Workclass 0 is set up for the initializer as realtime, etc. Everyone else is put initially into workclass 1. The aptes and itts are threaded into empty lists. Initial scheduling parameters are obtained from the schd card. The length of the prds is set (either default or from tbls card). The stack_0_data segment (which keeps track of the ring 0 stacks given to processes when they gain eligibility) is initialized. Apte entries for the initializer and idle (bootload cpu) are created.

Finally, memory is allocated for the pds and dseg of the various idle processes (which won't actually be started until tc_init$part_2).

SECTION 4

THE BOOTLOAD COMMAND ENVIRONMENT

Bootload Multics must provide a certain number of facilities when the storage system is not available. Examples are system dumps to disk, disk saves and restores , interactive hardcore debug (patch and dump), and automatic crash recovery.

INITIALIZATION

There are two ways that the command environment is entered. When an existing system is booted from power-up (cool boot), the command environment is entered to allow config deck maintenance and the like. When the service system crashes, the command environment becomes the crash recovery environment that oversees dumping and automatic restart. A full cold boot is a special case of a cool boot.

The heart of the bootload Multics command environment (bce) runs mostly wired. The paged segments are paged temp segments, managed by get_temp_segment_ and friends, for such purposes as qedx buffers and active function expansion. The bce file system is paged. Also, some bce command programs are paged, through the grace of load_mst. These are mapped onto an area of the bce partition. bce does not use the storage system, nor the hardcore partition.

Certain special programs are run so as to initialize bce. These are: init_bce to enable the basic facilities of switches and areas and such; find_file_partition to enable the bootload Multics file system; establish_temp_segs to provide paged temp segments; and, load_mst$init_commands to allow references to paged bce programs. load_mst was described under the bootload Multics initialization pass in collection 1.

ENVIRONMENT AND FACILITIES

The basic facilities of the command environment are:

*

a free area. free_area_1 is initialized with define_area_, and a pointer left in stack_header.user_free_area and stack_header.system_free_area, so that allocate statements with no "in" qualifiers work. get_system_free_area_ () will return a pointer to this area. This area is used for global data needed between commands. Each command normally finds its own local area, normally on a paged temp segment.

* standard input, output and error entries that hide the distinction between console and "exec_com" input. These are entry variables in the cds program bce_data.cds. They are hardly ever called directly, as more sophisticated interfaces are defined atop them. The entry variables are bce_dataget_line, bce_dataput_chars and bce_data$error_put_chars. get_chars is not sensible in the console environment, for the console will not transmit a partial line. The module bce_console_io is the usual target of the entry variables. It uses ocdcm_, oc_trans_input_ and oc_trans_output_. bce_data also contains the pointers get_line_data_ptr, put_chars_data_ptr and error_put_chars_data_ptr which point to control information needed by the target of the entry variable. The pair of values of an entry variable followed by the data pointer is what constitutes a bce switch. A pointer to this switch is passed around much as an iocb pointer is passed around in Multics. Both ioa_ and formline_ understand these bce switches so that normal calls may be made.

* bce_query and bce_query$yes_no. Each takes a response argument, ioa_ control string, and arguments, and asks the question on the console. An active function interface is provided.

* bce_error is the local surrogate for com_err_, used by various non command level programs. It does not signal any conditions in its current implementation. com_err_ and active_fnc_err_ simply call bce_error appropriately when in bce.

* a command processor. The standard command_processor_ is used to provide a ssu_-like subsystem facility. The various command programs are called with a pointer to bce_subsystem_info_, of which the arg_list_ptr is the important information.

* a request line processor. Any program that wants to parse lines using standard syntax (without quotes, parentheses, or active functions, for now) calls bce_command_processor_ with the command line, a procedure that will find the command, and a return code. find_rpv_subsystem, for example, calls it with an internal procedure that checks that the command is either "rpv", "cold", "help", or "?", and returns the appropriate internal procedure to process the command.

These procedures use the usual cu_ entrypoints to access their arguments.

* The paged temp segments bootload_temp_1 .. bootload_temp_N. These are each of 128/N pages long, and mapped as abs-seg's onto a part of the bce partition. N is established by the number of such segments listed in the MST header (and computed by establish_temp_segs). These segments are managed by get_temp_segments_ and friends.

* A primitive file system. bootload_fs_ manages a simple file system mapped onto the "file" partition on the rpv. This file system can hold config files or exec_coms. It is writable from within Multics service. The objects in the file system have a max length of 128/N pages, matching that of the temp segments, and have a single name.

* The standard active function set.

* Disk i/o facilities. Several exist. Some utilities call (read write)_disk. If they do not need the disk test that this routine performs (as when accessing the (already) trusted rpv), they call the no_test versions of these entrypoints. Another mechanism is to build a paged segment onto the desired disk area, normally via map_onto_disk. This mechanism trusts the built in mechanisms of page control (and traffic control disk polling) to ensure that the i/o is noticed. A final mechanism is to call dctl$bootload_(read write), which allows the queueing of multiple i/os to different disks. This is used for high volume operations, such as pack copying.

RESTRICTIONS

Various Multics facilties are not present within bce. Some are listed below.

* No operations upon the file system hierarchy are allowed (except for indirect references by bce_probe to segments in the Multics image).

* Normal segment truncation/deletion/creation is not allowed. The ptws must be manually freed.

* Segments may not be grown (no withdrawing of pages is allowed). They must be explicitly mapped onto the desired free area of disk or memory.

* No iox_ operations are allowed. Pseudo-iocb's do exist, though.

* Only a finite (and small) number of paged/wired work areas can exist. They also have comparatively small lengths.

* Dynamic linking is not done. References to object names are done with slt_manager$get_seg_ptr.

* Wakeups and waiting for wakeups can not be done. A program must loop waiting for status or use pxss facilities.

* Timers (cput and alrm) may not be set. Programs must loop waiting for the time.

* There are no ips signals so no masking is involved. The real question is the masking of interrupts (pmut$set_mask).

* Any routine that itself, or through a subsidiary routine, calls bce_check_abort (which includes any output operation), must be prepared to be aborted at these times. Thus, they must have a pending cleanup handler at these times, or simply have nothing that needs to be cleaned up.

MODULE DESCRIPTIONS

bce_abs_seg.pl1

This relatively uninteresting program maintains a list of abs-segs built during an initialization pass. This is done so that real_initializer can free them, en masse, when it needs to reinitialize before another pass.

bce_alert.pl1

Console alert messages (mostly for bce exec_com's) are produced by bce_alert. It simply appends its arguments, separated by a space) into one string which it prints through bce_data$console_alert_put_chars. This prints the message with audible alarm.

bce_alm_die.alm

bce_alm_die wipes out the bce toehold and enters a "dis" state.

bce_appending_simulation.pl1

All references to absolute and virtual addresses within the saved Multics image are performed by bce_appending_simulation. It has multiple entrypoints for its functions.

The "init" entrypoint must be called before all others. It initializes certain purely internal variables, for later efficiency. As an added bonus, it sets the initial dbr for the appending simulation based on whether it is desired to examine the crash image or bce itself.

The entrypoint "new_dbr" sets a new dbr for the simulation. This entrypoint takes apart the dbr supplied. The main purpose of this entrypoint is to find this new address space's dseg, so it can evaluate virtual addresses. This fetching of the description (aste/page table/sdw) of dseg can be done using the absolute fetching routines of bce_appending_simulation and by manually disecting sdws and ptws. This entrypoint must also find the core_map, if present, which is needed by the virtual entrypoints to find out-of-service pages.

The "(get put)_(absolute virtual)" address entrypoints actually perform the fetching or patching of data. They take the input address and fetch or replace data in pieces, keeping each piece within a page. This is done because different pages desired may reside in totally different locations.

"get_absolute" and "put_absolute" work in relatively simple ways. They examine the address to determine its location. Some low memory pages will be in the image on disk and fetched through the paged abs-segs multics_(low high)_mem. Other pages are in memory (above 512k). These are fetched through the abs-seg abs_seg0 which this program slides onto a 256k block as needed. References to absolute locations in examine-bce mode always use the abs_seg0 approach to fetch everything from memory. These entries keep a page_fault_error handler to catch disk errors, a store handler to handle memory addreses not enabled at the processor ports and an op_not_complete handler to catch refernces to scu's who have our processor disabled.

Before virtual addresses may be fetched/patched, the "new_segment" entrypoint must be called. The purpose of this entrypoint is to fetch the sdw/aste/page table for the segment for later ease of reference. This is done by using the "get_virtual" entrypoint, referencing dseg data given the previously discovered description of dseg (in the "new_dbr" entrypoint). For efficiency in fetching the sdw (meaningful for the dump command which calls this entrypoint for every segment number valid in a process and ends up fetching null sdws), a dseg page is kept internal to this routine.

Virtual addresses are manipulated by the "(get put)_virtual" entrypoints. These entrypoints break apart the request into blocks that fit into pages. For each page of the segment that it needs, it examines its ptw (found in the segment description found and provided by the "new_segment" entrypoint) to determine its location. Pages flagged as in memory are obtained by the absolute entrypoint. Pages on disk can be easily manipulated by mapping rdisk_seg onto the page and paging it. If it is in neither catagories, something is either wrong or the page is out of service. For out of service pages (pages with i/o in progress upon them), the "correct" page is found (the page at the source of the i/o) and this manipulated. If this is a put operation, it is necessary to replace this page in both locations (both memory and the disk page in use) to make sure that the effect is felt. Also, for any put operation, the proper page table word must have its modified bit set so page control notices the modification.

bce_check_abort.pl1

bce_check_abort contains the logic for possibly aborting bce functions upon operator request. When called, it checks wired_hardcore_data$abort_request, which is set by ocdcm_ whenever an unsolicited request is hit. If this bit is set, bce_check_abort prompts the operator with "Abort?" to which the response determines the degree of abort. Both this query and the response i/o are performed through bce_data$console_[whatever] to force them to appear on the console. A response of "no" simply returns. "yes" and "request" signals sub_request_abort_, which is intercepted by the bce_exec_com_ and bce_listen_, or by a bce subsystem. Entering "command" signals request_abort_, handled by bce_exec_com_ and bce_listen_ to abort a subsystem. Entering "all" performs a non-local goto to <sub-sys info>.abort_label, which returns to bce_listen_ at top level.

bce_check_abort is called on the output side of bce_console_io and other output oriented bce i/o modules. Thus, most operations will notice quickly the operator's intent to abort. However, any program that can enter an infinite computational loop (such as the exex_com processor trying to follow an infinite &goto ... &label loop) must call bce_check_abort within the loop to provide a way out.

bce_command_processor_.pl1

This routine is a scaled down version of command_processor_. It does not support active functions or iteration sets. Written as such, it does not need the various work areas that command_processor_ needs and can run completely wired. It separates the command line into the usual tokens, forming an argument list of the various argument strings. It uses a routine supplied in its call to find an entry variable to perform the command found. It is used in various very early initialization programs like init_clocks and find_rpv_subsystem (which obviously cannot page) as well as some bootload Multics programs that can deal with the simplicity and wish not to power up command_processor_.

bce_console_io.pl1

bce_console_io is the interface to the console dim ocdcm_. Its function is to perform translation appropriate to the console (oc_trans_input_ and oc_trans_output_) and to call ocdcm_$priority_io to perform the i/o. bce_console_io$get_line is the routine normally found in the entry variable bce_data$get_line and bce_console_io$put_chars is the routine normally found in bce_data$put_chars and bce_data$error_put_chars.

bce_continue.pl1

bce_continue restarts the interrupted image. It flushes memory and uses pmut$special_bce_return to invoke the toehold. As it passes, it resets all rtb flags in the flagbox except ssenb. This is so that the next return to bce does not show the current rtb flags.

Also present in this module is the bos command, which flushes memory and uses pmut$special_bce_return to invoke the BOS toehold.

bce_data.cds

This cds segment contains data pertinent to the command environment activities of bce. It holds the entry and data pointers used to perform i/o on the pseudo switches bce_dataget_line, bce_dataput_chars, bce_data$error_put_chars and bce_data$exec_com_get_line. It keeps track of the current exec_com level, through bce_data$command_abs_data_ptr (part of the exec_com_get_line switch). It also holds the top level subsystem info for the command level in bce_data$subsys_info_ptr.

bce_die.pl1

This module just checks to see if it is okay to die, which is actually performed by bce_alm_die.

bce_display_instruction_.pl1

One of the bce_probe support utilities, bce_display_instruction_ displays one (possibly multi-word) instruction. It uses op_mnemonic_ for its information. The result is to print an instruction and to return the number of words dumped.

bce_display_scu_.pl1

bce_display_scu_ is another bce_probe utility. It displays the scu data found in machine conditions supplied to it. bce_display_instruction_ is used to interpret the instruction words from the data.

bce_dump.pl1

The disk dumping facility of bce is found in bce_dump. It is actually a rather simple program but with a few tricky special decisions made within it. After parsing the command line arguments, it figures out the process and segment options to use. These options are merged together in a hierarchical fashion; that is, options applying to all processes apply to eligible; all that apply to elgible apply to running, etc. The dump header is filled in with machine state information from the toehold. The dump header on disk is flagged as invalid. An abs-seg (dump_seg, created by establish_temp_segs) is built to run down the dump partition during segment placing. Given this out of the way, dumping can start. Each apte is read from the saved image (through bce_appending_simulation). For each, the segment options applying to each are determined. Given the segment limits in the dbr for this process, each segment is examined to see if it meets the segment options. Most of the options are self-explanatory. When it comes to dumping non-hardcore segments, though, it is desired to dump any hierarchy segment only once. This is done by keeping a pseudo bit-map of the sst, where each bit says that a segment has been dumped. (Since the smallest possible aste in the sst is 16 words, there can be at most 256K/16 astes. Given an address within the sst from a segments' sdw, we assume that any aste that crosses the mod 16 boundary near this address describes the same segment as this and need not be dumped again.) If a segment is to be dumped, we read pages from its end, looking for the first non-null page. All pages from the beginning of the segment up to and including this page are appended to the dump. (The dump_seg abs-seg is adjusted to indicate these pages.) When all is dumped, we update the header and write it out.

bce_error.pl1

A simplified form of com_err_, bce_error simply fetches the text of an error message from error_table_ and constructs an error message which is printed through bce_data$error_put_chars. The com_err entrypoint is used to format a com_err_ style message, used by com_err_ when called during initialization.

bce_esd.pl1

An emergency shutdown of Multics is initiated by bce_esd. It uses bce_continue to invoke the toehold to restart the image. However, before doing this, it patches the machine conditions in the toehold to force the image to transfer to emergency_shutdown|0, to perform an esd.

bce_exec_com_.pl1

bce_exec_com_, along with bce_exec_com_input, form the bce equivalent of version 1 exec_com's. bce_exec_com_ is a merging of functions found in exec_com with those found in abs_io_$attach. It finds the ec and builds an appropriate ec_info and abs_data structure to describe it. The ec attachment is made (bce_data$exec_com_get_line) is made to refer to this ec invocation, after saving the previous level. Commands are read from the ec through bce_exec_com_input and executed through command_processor_$subsys_execute_line. Once bce_exec_com_info returns a code for end of file, the ec attachment is reverted.

bce_exec_com_input.pl1

bce_exec_com_input performs the parsing of exec_coms. It is a pseudo i/o module, in the style of bce_console_io$get_line. It is called in two possible cases. The first is to fetch a command line for execution by bce_exec_com_. In this case, the switch is bce_data$exec_com_get_line. When an &attach appears in an ec, bce_exec_com_input will have attached itself (by making bce_data$get_line point to itself) and then calls to bce_data$get_line will call bce_exec_com_input for a line where the switch (bce_data$get_line) will point to the abs_data for the ec that performed the &attach. The basic code is stolen from abs_io_v1_get_line_. The major changes are to delete non-meaningful operations like &ec_dir.

bce_execute_command_.pl1

This routine is the caller for the various bce command programs. It is passed as an argument to, and is called, from command_processor_$subsys_execute_line. It is given a pointer to an argument list generated by command_processor_, as well as the request name. bce_execute_command_ uses bce_map_over_requests_ to scan through bce_request_table_ to find the entry to call. It understands the difference in calling between Multics routines (like active functions stolen from Multics) and bce routines. It also understands the flags indicating within which command levels a command is valid.

bce_fwload.pl1

Firmware is loaded into various mpcs by bce_fwload. Its objective is to find, for each mpc desired, the set of firmware images needed for it. hc_load_mpc does the actual loading. For a normal (disk, tape) mpc, this involves just finding the mpc card which shows the model. The model implies the firmware module needed (config_data_$mpc_x_names.fw_tag). The desired module is found through slt_manager. (Firmware images for disk were part of collection 1 and are wired (they needed to be in memory to be able to load the rpv controller); other images were part of paged collection 1.5.) For urc controllers, the main firmware can also be derived from the mpc's mpc card. However, it is necessary to check all prph cards to find peripherals accessible through that urc. For each, and depending on the urc channel it is attached to, the appropriate firmware overlay is found and put in the correct slot in the list of firmware to load.

bce_get_flagbox.pl1

This module performs the bce (get set)_flagbox commands/active functions. It is basically a version of the corresponding Multics routine, modified to make direct references to the flagbox instead of a gated access.

bce_get_to_command_level.pl1

The routine to get from real_initializer into command level is bce_get_to_command_level. It builds a bce_subsystem_info_ structure which it passes to bce_listen_. It examines the current state to determine if the initial command should be null (manual entry), the flagbox bce command (normal) or probe (breakpoint entry). Since it is the routine below real_initializer on the stack, it is the routine to which control must return so that real_initializer can be returned to to perform boot and re_initialize functions. Thus, boot and re_initialize are entrypoints within this program. re_initialize just returns, setting the collection_1_phase to "early" so that real_initializer will end up running another boot pass. This will cause bootload Multics to pick up any changes that have been made to the config_deck. boot scans the arguments which are inserted into the intk card. It then returns.

bce_inst_length_.pl1

Another bce_probe utility. This routine is used to determine the length of an instruction, so that it may be correctly relocated. It differs from the real probe's version in that it does not attempt to deal with xec instructions.

bce_list_requests_.pl1

This program implements the list_requests (lr) bootload Multics command. It does a simple minded walk down the bootload Multics request table, using bce_map_over_requests_, with a printing routine to print the request names and the description within the table. It understands the dont_list flag, as well as understanding flags indicating at which levels a given command is valid.

bce_listen_.pl1

bce_listen is a simple loop that reads a command line from bce_data$get_line and executes it through command_processor_ (using bce_execute_command_ to actually execute the request). It contains the sub_request_abort_ and request_abort_ handlers to work with the operation of bce_check_abort.

bce_map_over_requests_.pl1

Programs that wish to walk down the bootload Multics request table (bce_list_requests_ and bce_execute_command_) call bce_map_over_requests_ with a routine that is called on each entry in the table. As such, the format of the table itself is known only to this routine.

bce_name_to_segnum_.pl1

This bce_probe utility maps segment numbers to names. It searches the slt and name_tables from the saved image. Entrypoints exists to convert a segment number to a hardcore segment name (bce_segnum_to_name_), a segment pointer to a virtual name (bce_segptr_to_name_), and a segment name to a segment number (bce_name_to_segnum_).

bce_probe.pl1.pmac

The main portion of bce's probe support, bce_probe contains the main drivers for most of probe's facilities. It contains the request line parser, address and value parsers and most of the functional routines.

bce_probe starts by examining its arguments and its environment to determine its operating mode. It defaults to examining the breakpoint image if the flagbox indicates a break, to examining the crash image, when at bce_crash or crash command levels or to examining bce otherwise. Given its operating mode, it initializes the appending simulation package accordingly and establishes a few initial constants. If in break mode, it determines the point of break for operator information.

bce proceeds to read request lines from the console. The first "string" in the line (or partial line left, if this is a multiple request line) found by internal routine get_string becomes the request name. This is looked up in a table and dispatched through a "case" statement.

REQUEST ROUTINES

The before request finds the desired address. It is validated to ensure that it is virtual and that the segment named is breakpointable. Finding the breakpoint page for this segment, this request looks for an empty break slot. The original instruction is relocated there (bce_relocate_instruction_) and replaced by a transfer to the break block. The break block consists of a "drl -1" instruction, which causes the break, followed by the relocated instruction, followed by a transfer back to just after the original instruction in the code. This break block and the transfer to the block are patched into the segment such that failure at any time will not damage the segment.

The continue request validates itself and calls bce_continue.

The dbr request fetches its arguments. Constructing a new dbr, it calls internal routine new_dbr.

The display request gets and validates its arguments. It loops, fetching (through bce_probe_fetch_) at most a page at a time to display (since we only allocate a one page buffer for the fetch). The internal routine "display" displays the data in the specified mode. Since data to be displayed may cross page boundaries, any data "display" cannot display (because it would need data from the next page to fill out a line) is "scrolled" in front of the page buffer and a new page worth's of data fetched. This continues until the last page is fetched.

The let request finds the address and sets up for patching of same. It then loops, finding values from the request line, converting them to binary. These are appended unto a word based buffer. When all are fetched, they are patched into place.

The list_requests request simple prints a canned list of requests.

The mc request gets its address and uses bce_display_scu_.

The name request uses bce_segnum_to_name_.

The proc request fetches the desired apte from tc_data in the image. A new dbr value found therein is passed to internal routine "new_dbr".

The quit request quits.

The reset request performs the inverse of the before request. After validating its address (for virtualness, breakpointability, etc.), it undoes the effect of before, in reverse order to prevent damage to the segment.

The segno request uses bce_name_to_segnum_.

The stack request validates its argument. Given the word offset therein, it decides whether to start from the specified stack header or frame. The needed data is fetched and displayed in interpreted form. Each stack pointer fetched is validated, not only to insure that it is a valid pointer, but to insure that stack frame loops do not cause bce probe loops.

The status request uses the internal routine "status" to display breakpoints set. It simply validates its argument and decides between listing breakpoints for a segment versus listing breakpointed segments.

INTERNAL ROUTINES

check_no_more_args insures that no more arguments appear on the request line; that is, that we are looking at a semi-colon or new-line.

display displays data in a specified mode. It determines the bit sizes to display, alignments, etc. Its only trick is when processing the end of a buffer full that doesn't fill a display line. This causes it to not finish its display. Its caller (the display request) then appends what was not displayed to the front of the next buffer full so that it may appear in the next group.

function is used to parse functional references, such as "reg(ralr)". function extracts the arguments to the function (whose identity was determined by its caller), builds an argument list from these strings, and calls the function.

get_address contains the logic to parse a bce probe address. It fills in the structure, bce_probe_data$address to define the current address. It special cases the dot (".") forms, checks for virtual forms (those with a "|" in them), notices absolute addresses (single octal number) and uses function for the pseudo-variable type of addresses (reg and disk).

Internal routines to get_address, called by function, build the address structure for these types.

get_string finds the next "string" in the request line. Its basic job is to pass whitespace and find string delimiters.

get_value finds a let request value. It looks for ascii strings (values starting with a quote character), which it must parse separately (since quoted strings confuse the notion of string contained in get_string), finds virtual pointers (strings containing "|"), and finds the various numeric types.

line_error is used to print error messages. Besides printing the given message, optionally with or without the current request line arg or error code, it also aborts the current request line.

new_dbr is the counterpart to the new_dbr entrypoint to the appending package. It exists to set up references to a few popular segments (slt and name_table) whenever the dbr changes.

pass_white passes whitespace.

status displays breakpoint status. Since break blocks are zeroed when not in use it is possible to find them easily. For any segment listed in the image's slt as being breakpointable, status fetches the last page (that which holds the breakpoints) and examines each break block. Any with a valid original_instr_ptr are displayed.

<u>bce_probe_data.cds</u>

Information communicated between probe and its support routines is done so through bce_probe_data. This cds contains the current value of "." (current address), as well as pointers to bce_appending_seg_info structures describing key segments in the image used by the support routines.

<u>bce_probe_fetch_.pl1</u>

This support utility to bce_probe fetches data, given a length and the current address (in bce_probe_data$address). It simply uses bce_appending_simulation for absolute and virtual address and read_disk for disk addresses. Register addresses must be specially handled by the caller.

<u>bce_query.pl1</u>

bce_query is a simple-minded counterpart to command_query_. It uses bce_data$put_chars to print a question and bce_data$get_line to read an answer. The main entrypoint accepts any answer and bce_query$yes_no accepts only yes or no which it returns as a bit. This routine is called with no prompt by some routines who find its return result (char (*)) to be better that the buffer and length and return length returned by bce_data$get_line.

bce_ready.pl1

bce_ready prints the bce ready message:

bce (BCE_COMMAND_LEVEL) TIME:

It has a nnl entrypoint to print the message without new-line (as a prompt). The normal entry prints the line (for ready message within exec_com).

bce_relocate_instruction_.pl1

This is another support routine for bce_probe. It differs from the standard Multics version in that it does not allow relocation of "xec" instructions. (Service probe allows this by attempting to examine the target of the xec, something bce_probe does not attempt.)

bce_request_table_.alm

The bootload Multics request table is a normal ssu_ request table built with ssu_request_macros. Each entry contains a pointer to the routine that performs a request, the name and short name of the request, and a short description of the request. The actual threading of the entries is known only to bce_map_over_requests_, which performs the walking down of this table. The last three flags in each rq_data entry is used to specify whether the command is valid at the three main bce command level types: early, boot and crash.

bce_severity.pl1

This is the bce counterpart to the Multics severity command/active function. It does not work as the Multics routine does, however. Instead, it knows the set of programs that recognize a severity indicator. For the desired one, it calls the severity entrypoint thereof to find the severity.

bce_shutdown_state.pl1

The current shutdown state of the storage system (rpv label.shutdown_state) is found by this routine. It uses read_disk to find this information.

bce_state.pl1

This command/active function simply returns the name of the current bce state.

bootload_disk_post.pl1

This routine is used in conjunction with the high volume disk facility of bce (dctl$bootload_(read write)). Whenever a disk i/o queued through this means is posted for completion, it is done so through bootload_disk_post, called by either dctl or disk_control. The result is posted in a structure described by bootload_post_area.incl.pl1. This area must be maintained by the caller.

bootload_fs_.pl1

bootload_fs_ contains various routines to act upon the bootload Multics file system. The format of the bootload Multics file system is known only to this program. The file system is kept in a single abs-seg (bootload_file_partition), mapped (and paged) off the bce partition on the rpv. A two page header at the start of the partition contains a directory of 174 entries (max that fits) listing the name, size and placement of the file within the segment. Also present is a free block map. Files are allocated as a contiguous series of blocks (64 word blocks) within the segment. The segment is automatically compacted by this routine when necessary. Entrypoints to this routine are: lookup (find the length of a file given its name), list (allocates a list of file names and sizes within a user supplied area), get (copies a file into a user supplied buffer), get_ptr (returns a pointer and length to a given file (hcs_$initiate?)), put (allocates area within the file system for a file and copies a user supplied buffer into it), put_ptr (allocates an area within the file system large enough for a given file and returns a pointer to it) (both put and put_ptr take an argument allowing for the deletion of a file with the same name as the one desired), delete (deletes a directory entry and frees the space used), rename (renames a file (does not allow name duplication)), and init (clear out the bootload file system entirely).

bootload_fs_cmds_.pl1

This program simply calls bootload_fs_ to perform the functions of the bootload Multics commands print, list, delete, rename, and initialize. This routine supports the star and equal conventions for most of its operations through match_star_name_ and get_equal_name_.

bootload_qedx.pl1

bootload_qedx is a modified version of qedx. It differs in its use of file system operations (bootload_fs_) and its use of temp segs.

config_deck_data_.cds

The config deck editor's source of config card descriptions is found in config_deck_data_. This cds provides labels for the fields, numbers and types of fields, etc.

config_deck_edit_.pl1

This is the program that edits config decks. It calls qedx_ to perform text editing, specifying the caller_does_io option. With this option, qedx_ calls config_deck_edit_ to perform read and write operations on buffers. Any read/write not to the config deck uses bootload_fs_. Reads/writes to <config deck> (buffer 0) use the config deck conversion routines. This program makes use of config_deck_parse_, the routine that can convert from ascii (possibly labeled) form to and from binary form. The conversions are performed using a set of tables (config_deck_data_) that describe the names of the fields, the required and optional number thereof, the data types of the fields, etc. Also allowed by the conversion routines are cards of types not recognizable starting with a dot (.) which are not validated. This is to allow for future expansion and site formatted cards.

When a command line argument is supplied, the file specified is accessed (bootload_fs_$get_ptr) and the object obtained is supplied to the internal routine write_config_deck which sets this new deck.

establish_temp_segs.pl1

Whenever bce needs (paged) temp segments, it calls get_temp_segments_. get_temp_segments_ gets these segments from the pool of segments bootload_temp_1..N. establish_temp_segs divides the temp seg pages allocated in the bce partition (128 pages) up into the N segments (N is determined from the number of such segments listed in the mst header). The paged segments are built as abs-seg's onto this area of the determined length. This size is saved in sys_info$bce_max_seg_size. establish_temp_segs also creates the bce segments multics_(low high)_mem, used to access the saved image, dump_seg, used to access the dump partition and disk_config_deck, used to access the rpv (real?) copy of the config_deck (as opposed to our running copy in config_deck).

find_file_partition.pl1

find_file_partition maps the bootload Multics file system abs-seg (bootload_file_partition) onto the bce partition on the rpv in much the same manner as establish_config_deck maps the config deck. It also calls bootload_fs_$init to begin accessing the segment. If bootload_fs_ states that the file system is bad, find_file_partition will call bootload_fs_$init again, this time to clear out the file system.

init_bce.pl1

init_bce initializes the bootload Multics command environment features required for future programs. It is called early in initialization. At its wired entrypoint, it sets up free_area_1 as an area, setting the inzr_stk0 stack header to point to it so that allocates without an area work correctly and so that get_system_free_area_ also works. This routine also initially sets bce_dataget_line, bce_dataput_chars and bce_data$error_put_chars to their appropriate entry values (bce_console_io$get_line, bce_console_io$put_chars and bce_console_io$put_chars, respectively) so that calls to bce_query, bce_error and especially ioa_, will work. At its paged entrypoint, it finishes up references to paged objects, in particular, to the exec_com routines.

SECTION 5

CRASH HANDLING

Bootload Multics must be able to save the salient state of a crashing system and set up the command environment for dumping and other intervention.

EARLY CRASHES

Crashes in collection 0 or the early initialization pass of collection one should be very rare. Since the system uses a generated config deck, the set of possible operator inputs is small, and it is possible to do a much more thorough job of testing than can be done with BOS or service initialization. However, hardware problems will happen, and software bugs will sneak through. To cover these cases, collection 0 includes a crash handler that can write a core image to tape, prompting the operator for the drive number.

THE TOEHOLD

The toehold, toehold.alm, is an impure, wired, privileged program that resides in a known location in absolute memory (24000o). It has entrypoints at the beginning that can be entered in one of two ways: with the execute switches processor function, or by being copied into the fault vector. The toehold, therefore, is entered in absolute mode. It must save the 512K memory image off to disk, and then load in the crash handler.

The memory image includes the complete machine state. All absolute addresses, channel programs, port and channel numbers, and other configuration dependent information is stored into the toehold by a PL/I program, init_toehold.pl1. Thus the alm code does not have to know how to do any of these things, which simplifies it considerably.

The toehold starts with the various entry sequences; one for manual entry, one for Multics entry (which differs from manual entry in that the means of entry is to execute the entry through a fault vector entry; it is necessary to update the machine conditions in this case to pass the instruction that caused the fault vector execution) and one for restarting the machine image. The crash entries save the entire machine state. This is done under the protection of the memory_state so that the machine state is not overwritten if the toehold is invoked again after being invoked after a crash. An internal routine performs i/o given a set of dcw lists (built by init_toehold). After the memory is saved and the crash handler read in, the machine state of bce is restored. (It was saved by save_handler_mc.) This causes a return into save_handler_mc, which quickly returns to init_toehold, which quickly returns to real_initializer who quickly starts the appropriate crash initialization pass.

On the restore side, the system is masked and the internal routine called to read back the saved image. The machine conditions are restored from the toehold (which is not saved/restored during the memory shuffle).

MODULE DESCRIPTIONS

fim.alm

fim is listed in the crashing set of modules in as much as that it contains the bce breakpoint handler. A bce breakpoint consists of a "drl -1" instruction. fim's drl handler special cases these (in ring 0), saves the machine state in breakpoint_page (after advancing the ic to pass the drl instruction) and calls pmut$bce_and_return. It also performs the restart from a breakpoint.

init_toehold.pl1

This pl1 program constructs the channel programs to save and restore the 512K memory image, and fills it and other data into the text of toehold. After saving the bce image (crash handler) on disk, it calls save_handler_mc to save the current machine state of bce in the toehold. When bce is invoked upon a crash, the bce restore operation will return to the return in save_handler_mc which will return to this point in init_toehold. init_toehold notices this and quickly returns to real_initializer who will perform the desired crash initialization pass.

save_handler_mc.alm

The save_handler_mc program, called from init_toehold right after it saves the crash handler to disk, saves in the toehold the machine conditions appropriate for bce. Besides register contents and such, it saves the return address to the return in save_handler_mc.

SECTION 6

COLLECTION 2

The main task of collection 2 is to make the storage system accessible. Along its way, it loads collection 3 into the storage system and places the appropriate entities from collections 1 and 2 into the hierarchy. The sub-tasks are to enable segment control and directory control. The real traffic control is also started. Since collection 2 runs in a paged environment, it does not have the memory restrictions that collection 1 had. This is the reason why it is in a different collection from collection 1.

ORDER OF EXECUTION

The operations performed in collection 2 are described below.

initialize_faults$fault_init_two is called to change the fault vectors into the desired values for normal service operation, now that the code for such has been loaded.

Initialization now runs performing several intermingled functions. All hardcore segments must be created now, before traffic control is fully initialized. This is so that the address space inherited by the new processes (idle in particular) encompasses all of hardcore.

tty_buf, tty_area and tty_tables are generated through a call to fnp_init. They won't be needed at this time but must be allocated before tc_init$part_2.

Unique id (uid) generation is initialized by a call to getuid$init. This is required before segments in the hierarchy (in particular, >sl1 and >pdd) can be created.

init_vtoc_man allocates and initializes the vtoc_buffer_seg. We are therefore eligible to read and write (and create) vtoces.

dbm_seg is allocated and initialized to an area by dbm_man$init. init_scavenger_data allocates the scavenger_data segment, used by the volume scavenger. The page control data base, dm_journal_seg_, used to control synchronous page operations (data management), is initialized by init_dm_journal_seg. dir_lock_seg, used to keep track of directory lockings and waitings thereupon, is initialized by dir_lock_init. Again, these are created before tc_init$part_2 is run.

After this point, changes to the hardcore descriptor segment may not be reflected in idle process and hproc descriptor segments. This is because init_sys_var, which sets various system variables, uses the number of supervisor segments present (which is the expected total set thereof) to set the stack base segment number in various variables and in the dbr.

We can now run tc_init$part_2, which creates the idle processes and starts multiprogramming. At this time, only the bootload cpu will be running but the idle process will be enabled to run on it.

With multiprogramming active, syserr_log_init can create the syserr hproc (after it makes the syserr partition accessible). We then log a message to the effect that this was done.

The activation of segment control, which began with the creation of the sst, continues now with the creation of the system trailer seg (str_seg) by init_str_seg. If the astk (ast track) parm was specified, init_sst_name_seg initializes the sst_names_ segment with the names of paged hardcore segments.

The entrybounds of hardcore gates are set via a call to init_hardcore_gates, which also stores linkage pointers into the gates for a reason described under the description of the program.

We can finally make the volumes of the rlv accessible for storage system activity by a call to accept_rpv. This sets up the volume and vtoc maps and stocks for the drives, allowing vtoc_man and the page creation/destruction functions to work against the paging region of the disks.

The logical volume table (lvt) is initialized to describe the rlv by init_lvt.

bad_dir_ and seg_fault_handlers are now set up as we are about to access our first directory. init_root_dir makes the root directory known in the Initializer's process, creating it if this is a cold boot. The functions performed here are those that will allow future hierarchy segment references through segment control (kst creation, in particular). kst_util$garbage_collect is called just to make the kst neat. At this time, we can consider segment control to be active. We can call upon it to create, delete or whatever. The presence of the root will allow these activities by virtue of the special casing performed by segment control when it discovers a segment with no parent (the root).

The hardcore entities which need to be placed into the hierarchy (deciduous segments) are done so by init_branches, which also creates >sl1 and >pdd appropriately. These entities will be needed when we try to leave ring zero. Of course, other required segments are needed; these are the contents of collection 3.

init_stack_0 then runs to create the various stack_0's to be shared between eligible processes, now that it has a place to put them.

delete_segs$temp can now run, deleting collection 2 temporary segments. This ends collection 2.

MODULE DESCRIPTIONS

accept_fs_disk.pl1

A disk is accepted into the file system by accept_fs_disk. It validates the pvte for the disk. The label is read. (If this is a pre-MR10 pack, salvage_pv is called to convert the vtoc region for stock operations.) The pvid and lvid of this disk are copied into the pvt, finally making this data valid. The volmap and vtoc map are initialized and the stocks made active by init_volmap_seg. If this fails, the volume salvager is called and we try again. The partition map from the label is checked against the volmap to make sure that no partition claims pages in the paging region. The updated disk label is written out as we exit.

accept_rpv.pl1

The volumes of the rlv are accepted for storage system use by accept_rpv. First, the various disks that have hardcore partitions are validated, from their labels, to be part of the rlv. We then scan the intk card to see if the rpv or rlv desire salvaging; these facts are stored in the pvt. If the rpv needs salvaging, this is done now (salvager$volume_salvage). For information purposes, we log (or print, if the hcpt parm was specified), the amount of the hardcore partition used on the various disks. accept_fs_disk is called to accept the rpv in the normal way. wired_shutdown is enabled as the storage system is considered to be enabled. Appropriately, make_sdw$reset_hcp is called to prevent further attempts to allocate from the hardcore partition. Contrary to the name (accept_rpv), the entire rlv is accepted next by calling the salvager, if necessary, and accept_fs_disk for the other rlv volumes. We can then clear salv_data$rpv to keep the salvager from salvaging the rpv later.

create_root_dir.pl1

During a cold boot, the root is initialized by create_root_dir. It locks the root, setting its uid to all ones. The various dir header variables are set, pvid, master_dir flag, etc. A directory style area is set up along with a directory hash table. The dir is then unlocked and we exit.

create_root_vtoce.pl1

create_root_vtoce creates a vtoce for the root directory during a cold boot. The vtoce created describes the root as a master directory of appropriate length, maximum quota limit, created as of the current time, primary name of ">", etc. vtoc_man is used to allocate space in the vtoc map for this and to write it out.

dbm_man.pl1

dbm_man manages the dbm_seg (dumper bit map) for the volume dumper. The init entrypoint used during initialization allocates and initializes the dbm_seg. Its size is determined from the number of disk drives configured and allocated out of the hardcore partition by make_sdw. This routine changes dbm_seg from its MST status (an abs_seg) to being a real segment.

dir_lock_init.pl1

The segment used to keep track of directory lockings and waitings thereupon, dir_lock_seg, is allocated and initialized by dir_lock_inid. The size of this segment is based upon max_max_eligible (the maximum number of readers of a lock) and sys_info$max_tree_depth (maximum lock depth one can hold). The dir_lock_seg is converted from an abs_seg to a real seg, paged out of the hardcore partition. Initially, ten dir_lock's are allocated, threaded appropriately.

fnp_init.pl1

fnp_init initializes the data bases used in Multics-fnp communication. tty_buf is allocated in wired memory either with a default size or a size specified by the ttyb parm. Various header variables are set up. If a tty trace table is called for by a config parm, it is allocated in the tty_buf free_space area.

tty_area is initialized as an empty area. tty_tables also has its header filled in and its table_area set to an empty area. The config file is scanned for fnp cards; each one sets the fnp_config_flags appropriate to it. The hardware fixed dn355_mailbox for each fnp is zeroed. fnp_info is set. Finally, io_manager$assign is called to assign each fnp with an interrupt handler of dn355$interrupt.

getuid.alm

getuid is the generator of uid's (unique identifiers) for storage system objects. It operates by effectively incrementing tc_data$id under its own form of lock. The init entrypoint used during initialization stores an initial uid "seed" in tc_data$id generated from the clock value.

init_branches.pl1

The program that places the appropriate hardcore segments into the hierarchy, creating >sl1 and >pdd as it goes, is init_branches. To start with a clean slate, it renames the old >process_dir_dir and >pdd to a screech name. append then creates a new >process_dir_dir (added name of >pdd) which is then initiated. The per_process sw is set on for this dir. It is given the maximum quota possible. The old >system_library_1 (>sl1) is also renamed and a new one created and initiated. Access is set to s for *.*.* on it. We then walk down the various sst pools looking for segments to have branches created. The sst entry leads us to the slt entry for the segment to be placed in the hierarchy. create_branch is called (running recursively) to create a branch for the segment (it creates all necessary containing directories and a vtoce for the segment). A pointer to the parent directory and its aste is found. The aste for the hardcore segment is threaded into the parent entry. The per_process sw, max_length and uid fields are set in the aste. It is then threaded out of the hardcore lists and into the appropriate segment list. The vtoc index provided for the segment (found in its entry in the parent directory) is copied into the aste so vtoc_man will work. The entrybound of the segment is placed into the directory entry. If aste tracking is going on, a sstnt entry is added. Its vtoce is updated, putting the correct information from the initialization created aste into the vtoce. The parent directory is then unlocked and terminated.

The per_process sw is turned on in the aste for >pdd so that it can propogate down to sons activated off it. We walk down >pdd to propogate this switch. The maximum length of the slt and name_table are explicitly set, not trusting the slte fields for them. A maximum quota is reset on >pdd. The default acl term of sma *.SysDaemon is removed from >pdd and the acl term of sma Initializer.SysDaemon.z is added. >dumps is created and salvaged if needed. The hierarchy is now properly created and active.

init_dm_journal_seg.pl1

init_dm_journal_seg initializes the page control data base dm_journal_seg_ used to control synchronous page operations. This routine parses the dbmj card. This card describes the sizes of the various journals needed. Once the size of dm_journal_seg_ is found, its memory (wired) is obtained from make_sdw. Various header parameters (pool thresholds, pages held, events) are filled in. The various journal entries have their time stamp initialized to tc_data$end_of_time. The various page_entry's are threaded into a list. After this, sst$dm_enabled is set for the world to know.

init_hardcore_gates.pl1

init_hardcore_gates performs a variety of functions to make those things which are hardcore gates into future usable entities. It recognizes anything in the slt with ring brackets of 0, 0, n as a hardcore gate. It finds within the text (given the definitions) the segdef .my_lp and stores there (having forced write access) the linkage pointer for the gate. This is done because, the gate, known in outer rings by a segment number different from the hardcore number, would not be able to find its linkage by indexing into the lot by its segment number as normal outer ring programs do. Given the segdef .tv_end found for the gate, the entrybound is set in the gate's sdw. Finally, the ring brackets for restart_fault and return_to_ring_0_ are set from their slt values so that these segments may be used in outer rings with their hardcore segment numbers. (return_to_ring_0_ has a pointer to it stored as the return pointer in the stack frame by signaller. return_to_ring_0_ finds restart_fault through a text imbeded pointer.)

init_lvt.pl1

The logical volume table is initialized by init_lvt. It sets up the header and then uses logical_volume_manager$add to add the entry for the rlv.

init_processor.alm

A processor is inited by init_processor. The init entrypoint stores the absolute address of various variables into init_processor itself for execution within absolute mode when started on other cpus. When run to start a cpu, it performs some collection of tests, enters appending mode, fiddles with associative memories and cache, informs pxss that it is running (through its apte), initializes pds and prds time values, sends out a connect to preempt the processor and then opens the mask to allow interrupts. (We will be interrupted at this time (by the connect we sent). This will cause us to find our way back to pxss to schedule something to run on this processor.) The idle loop for a processor is contained within init_processor following this. The idle loop flashes a moving pattern in the aq lights when it is on the processor. At this time, x4 contains the number of eligible processes, x5 the term processid and x6 the number of ready processes for the sake of checking system operation.

init_root_dir.pl1

The root directory is made known by init_root_dir. We start by checking to see if this is a cold boot. If so, create_root_vtoce is called. The root vtoce is read. An aste is obtained for the root dir (64 pages), which is initialized from the data in this vtoce. pc is used to fill the page table. search_ast hashes in this aste. We can now begin the process that will allow future segment accessing activity through segment control. The Initializer's kst is built, by initialize_kst. The pathname "associative memory" used to map segment numbers to pathnames is initialized by pathname_am$initialize. makeknown_ is called to make the root (uid of all ones) known (found in the kst). If this is a cold boot, this segment just made known must be initialized to a directory by create_root_dir. Finally, this directory is salvaged, if necessary.

init_scavenger_data.pl1

The segment scavenger_data is initialized by init_scavenger_data.

init_sst_name_seg.pl1

The sst_names_ segment is initialized by init_sst_name_seg whenever the astk parm appears. It walks down the slt, looking for segments that are paged with page tables in the sst. For each, it copies the primary name into the sst_names_ segment.

init_stack_0.pl1

The various ring zero stacks (stack_0) are created by init_stack_0. Since a process cannot lose eligibility while in ring 0, the number of processes that can have frames down on ring zero stacks is equal to the maximum possible number of eligible processes (max_max_eligible). We thus create this many ring 0 stacks which are used by eligible processes. The various stack_0.nnn segments are created in >sl1. They are, in turn, initiated, truncated, and prewithdrawn to be 16k long. The vtoce is updated accordingly. The stack header from the initializer's ring zero stack is copied into the header of these stacks. The stack is then terminated. The acl for Initializer is removed. The first stack slot is claimed for the Initializer; the current stack being put into the slot in stack_0_data.

init_str_seg.pl1

init_str_seg initializes the system trailer segment (str_seg) into a list of free trailer entries.

init_sys_var.pl1

Now that all of the hardcore segments have either been read in or created, we can now stand back and observe hardcore. The next supervisor segment number (mod 8) becomes the ring 0 stack segment number (stack base) which is stored in active_all_rings_data$stack_base_segno and hcscnt. We make sure that the dsegs for the idle processes will be big enough to describe these segments. The stack base is stored in the dbr value in the apte. Various other system variables are set: sys_info$time_of_bootload, sst$pvhtp (physical volume hold table pointer), sst$rqover (record quota overflow error code, which is moved to this wired place from the paged error_table_), and sst$checksum_filemap (depending on the nock parm).

init_volmap_seg.pl1

init_volmap_seg initializes a volmap and vtoc map segment allowing us to reference such things on a given physical volume. It starts by acquiring an aste for the volmap_seg (for the segment abs_seg) and one for the vtoc header (for the segment volmap_abs_seg) (vtoc map) which are then mapped onto the desired areas of the disk. (This is done under the ast lock, of course.) The free count of records is redetermined from the volmap. The same is done for the vtoc map. If this is a member of the rlv and volume inconsistencies were previously found and the number of free vtoces or records is below a certain threshold, a volume salvage is called for. If we will not salvage, we can accept the disk. Use of the hardcore partition on the disk is terminated through a call to init_hc_part$terminate_hc_part. Vtoc and record stocks are allocated. The pointers in the pvte to these stocks are set as are various other status and count fields. The number of free records and the base address of the first record in each stock page is computed. The dumper bit map from the disk is allocated into the dbm_seg (previously created by dbm_man$init_map). Finally, under the ast lock, we clean up the abs_seg and volmap_abs_seg segments (free their sdws).

init_vtoc_man.pl1

The vtoc_buffer_seg is initialized by init_vtoc_man. This routine acquires enough contiguous memory for the vtoc_buffer_seg, determining the number of vtoc buffers either from the config vtb parm or from a default. Various vtoc buffer headers are initialized here.

initialize_faults.pl1

initialize_faults was described earlier, under collection 1. The entry point fault_init_two, used by collection 2, sets up fault vectors for normal (file system) operations. It prevents timer run-out faults during operation through a call to pmut$ldt. initialize_faults_data is used to set the main faults. Faults set are: command, trouble, segment and linkage to fim$primary_fault_entry (scu data to pds$fim_data), store, mme, ft1, lockup, ipr, overflow, divide, df3, mme2, mme3, mme4 and ft3 to fim$signal_entry (scu data to pds$signal_data), and fault numbers 26 to 30 to wired_fim$unexp_fault (scu data to prds$sys_trouble_data). Access violations are routed specially to fim$access_violation_entry which maps the acv fault into our sub-faults. Timer runouts are sent to wired_fim$timer_runout (who normally calls pxss) with the scu data stored in prds$fim_data. Parity goes to fim$parity_entry. Finally, we set up the static handlers for the no_write_permission, isot_fault and lot_fault conditions.

kst_util.pl1

kst_util performs utility functions with regard to maintaining the kst. The garbage collect entrypoint cleans up the kst by terminating any segment not known in any ring or a directory with no active inferiors.

start_cpu.pl1

start_cpu might best be described as a reconfiguration program. It is used during initialization to start a idle process on each configured cpu (at the appropriate time). When starting the bootload cpu in collection 2, it fills in the apte entry for the idle process for the cpu in question. Some more variables in init_processor are set (controller_data). A simple call out to init_processor$start_bootload_cpu can be made.

syserr_log_init.pl1

The syserr logging mechanism is made operative by syserr_log_init. It creates the segment syserr_log which it maps onto the log partition, wherever it is. A consistency check is made of the partition; if the check fails, the partition is re-inited. The syserr hproc (SyserrLogger.Daemon.z)'s ring 0 stack (syserr_daemon_stack) is initialized. The hproc is created by create_hproc$early_hproc with a stack of syserr_daemon_stack, dseg of syserr_daemon_dseg, pds of syserr_daemon_pds, and procedure of syserr_logger. A fast channel is defined for communication through syserr_data to the hproc. Logging is now enabled.

tc_init.pl1

tc_init was described earlier to set up and initialize tc_data. tc_init$part_2, in collection 2, starts up multiprogramming by creating the idle processes. This entry can only be called once the initialzer's dseg is completely filled in by all those who read or create hardcore segments. Various variables in template_pds are filled in which are applicable to the idle processes. For each configured processor, a copy of template_pds and the initializer's dseg is made into appropriate entries in idle_dsegs and idle_pdses. The stack_0 for these processes is made to be the prds for the given processor. The initial process for the bootload processor (the initializer himself) is created by threading in an apte specifying init_processor as an initial procedure. It is placed in work class zero. tcm is initialized to indicate only this one process running. Various polling times are set for when polling becomes enabled as we start multiprogramming. init_processor$init sets up the rest of the state. We can now call start_cpu to start the bootload cpu idle process.

SECTION 7

COLLECTION 3

The main task of collection three is to read itself into the hierarchy. Collection three consists of those programs that are necessary to reach ring one in the initializer's process and to be able to perform a reload function (and other maintenance functions). A few extraneous functions are also performed in collection three.

ORDER OF EXECUTION

Collection three starts with its main function: load_system is called to read the remaining mst entities into the hierarchy. At this time, the mst reading function is shut down.

io_config_init initializes the data in io_config_data for use in later econfiguration activities. ioi_init is called to prepare for outer ring usage of physical devices.

tc_init$start_other_cpus starts up the other processors. We now consider collection three done and set sys_info$initialization_state to 4.

real_initializer finally finishes, returning to initializer. initializer can then delete init segs through delete_segs$init, real_initializer being part of one. Initialization then finishes by a call to init_proc, to call out to ring one command level.

MODULE DESCRIPTIONS

init_proc.pl1

init_proc is the first program run in ring zero in a normal process. It calls out to the initial procedure for a process in the outer ring. For the Initializer, the initial_proc is made to be system_startup_. The setting of the working dir is skipped, since we can't be sure it's there yet. The ring one stack is created explicitly, by makestack. system_startup_ is initiated. call_outer_ring_ is called to "return" out to ring one (outward calls are not allowed) to transfer to system_startup_.

io_config_init.pl1

io_config_data is initialized by io_config_init. (It was allocated memory and its base pointers set up by get_io_segs.) The tables are initialized in the order: iom and mpc, channel and then devices (as it indeed must be).

Filling in the iom and controller entries is easy; they are one for one with iom and mpc cards.

A walk is made of prph cards twice. The first pass is made to fill in the channel entries. Each prph card is found. If the peripheral is a disk or tape (has an mpc), we also find a chnl card (if present). Each channel is added to the channel list. The internal routine controller_idx_from_chanid looks up the index into the controller array for the controller owning this channel (via ioi_config$find_controller_card). The internal routine iom_idx_from_chanid finds the corresponding iom array entry. After all of this, each channel is linked to its base physical channel via calls to ioi_config$find_base_channel.

A second pass over prph cards is made to fill in the device entries. For each device, we start by finding its physical channels. (This is done by walking down all the channels (from the prph and chnl cards), looking up the base channel (from the channel entries) and making an array of the physical channels found (template_pchan_array). If any of these channels is configured (it was marked configured above because its iom was on), the device becomes configured on. The device entry is filled in from the card. For disks and tapes, though, we add a device entry for the controller and one each for each drive.

ioi_init.pl1

ioi_init sets up the various ioi_ data bases. It walks the config deck, allocating group table entries for each channel group. Each device whose channel is accessed through a controller has its group entry flagged as a psia. The device table entries and channel table entries are allocated from information on the prph card. Then, for each chnl card, the group table entry corresponding is found and the channel table entries allocated from the information on the chnl card. The base logical channel for each group is found. The group entries are then traversed to find storage system disk channels. All non-storage system disk channels are assigned to ioi_ through io_manager. As a final gesture, the ioi_ page tables are setup (ioi_page_table$init).

ioi_page_table.pl1

The init entrypoint of ioi_page_table is called during initialization to set up the io_page_tables segment. It starts by abs wiring the segment as one page (initially) and zeroing it. The header is initialized. Sixty-four word page tables are allocated and initialized within this page, as many as will fit.

load_system.pl1

Collection three is loaded into the hierarchy by load_system. It reads the mst source (disk_reader) looking for segments. For each, init_branches$branch is called to create the branch (init_branches is described under collection two). The appropriate acl is set up, given the mst information. The segment contents are copied into the created branch. If the Initializer does not have write access to the final segment, the acl is cleared of this acl entry.

tc_init.pl1

tc_init was described earlier. The entrypoint start_other_cpus, starts cpus other than the bootload cpu at the end of collection three (after their interference won't matter). A prds for the various non-bootload processors is created and entry-held. The pds and dseg for the other cpu's idle processes was already created so we can now call start_cpu on this new cpu as we would normally during reconfiguration.

SECTION 8

MECHANISMS

This chapter describes certain tricky and not so tricky mechanisms used within initialization to get things done. Also included is a look at the mechanism by which the various parts of the supervisor come into operation.

HARDCORE SEGMENT CREATION

There are various ways that segments come into being within the hardcore. These mechanisms are usually quite distinct from the normal method of creating a segment within the hierarchy (append$foo).

The first group of segments that are created are those needed by collection zero. Collection zero itself is read in in absolute mode; no segments exist other than those hardware supplied. To save collection zero the problem of generating segments for its use in absolute mode, its segments are generated by macros within template_slt_.alm. These macros generate not only the slt entries for collection zero segments (and various segments at fixed absolute memory addresses); they also generate the page tables and the segment descriptor words for the segments. A much simpler program in absolute mode moves these page tables and sdws (the dseg) to appropriate places and loads the dbr (also generated by template_slt_). Thus, these early segments come quickly and magically into being. All of the segments described by the template_slt_ are data segments with no initial content except for bound_bootload_0 itself, which was loaded into the correct memory address by the bootload tape label, and toehold, by virtue of being the first part of bound_bootload_0.

The second group of segments to come into being are the collection one segments loaded by collection zero. These segments are created through a mechanism imbeded in bootload_loader and bootload_dseg. When the segment header (actually a slt entry) is read from the MST, the need for a segment of a certain size is called for. Values in the slt header keep track of the extent of memory allocated. The type of segment (permanent "unpaged" or not) determines from what end of memory the space will be obtained. A page table of appropriate size is constructed in the proper area (either the segment unpaged_page_tables for permanent "unpaged" segments or int_unpaged_page_tables for temporary or to be made paged segments). A new sdw pointing to this page table is tacked onto the appropriate end of dseg (low segment numbers for permanent segments, high for temporary or init segs). With write access set on in this sdw, the segment contents can be loaded from tape into the memory area. Proper access is then set in the sdw. The segment is now existent.

Collection one creates certain data segments that are wired and contiguous. The most obvious is the sst. These are created by the routine get_main. get_main might be considered the counterpart of the collection zero segment creation mechanism when called in collection one. It also allocates memory space from values in the slt header. A page table of appropriate length in one of the two unpaged page table segments is constructed and a sdw fabricated to this page table. The caller of get_main forces this sdw into dseg and performs the appropriate associative memory clearing function.

The other type of segment created by collection one is a paged segment. There are two cases of this. The first is a paged segment that is to be mapped against a previously defined area of disk. This is done when we want to access a partition or part thereof, as when we want to read the config deck from disk. To do this, make_sdw is called, specifying that we want an sdw for an abs-seg. make_sdw finds us an aste of appropriate size and threads it into the hardcore lists, but senses the abs-seg switch and does not allocate pages or whatever. The caller of make_sdw builds its own page table within the aste obtained by calling ptw_util_$make_disk to make each page table word point to the correct disk record. The pvtx of the desired disk is inserted into the aste. Thus, references to this segment (whose sdw points to the page table in this aste) will wake up page control who will page in the proper pages. This mechanism appears in several places; the desired way of generating such a segment is to call map_onto_disk.

The second type of paged segment created by collection one (or two for that matter) is a segment paged off the hardcore partition. In this case, allocation of pages is done by page control. make_sdw is called as before, but, this time, it not only creates an aste for the segment, but it finds space for it. A disk with a hardcore partition with enough free space to hold the segment is selected. This pvtx is put into the aste. As an added bonus, since such segments will not have trailer entries, the trailer pointer in the aste is set to the hardcore segment number (for those programs that need to map the hardcore aste list entries to slt entries). The page table words are set to a nulled state. make_sdw then touches each page, causing page control, when the page fault occurs, to withdraw a page from the partition. (init_hc_part created a vol map and record stock that page control can use which describes only the hardcore partition.) With the segment now in existence, the caller of make_sdw can now load the segment. For collection one or two, this involves either initializing the data segment or copying in the segment contents read from the mst.

When collection two needs a wired contiguous data space, it calls get_main also. In this case, though, get_main calls make_sdw$unthreaded which will obtain an aste and sdw and page space. pc_abs$wire_abs_contig is then called to wire this segment into contiguous memory pages. A paged segment to be mapped onto a particular area of disk is created as described for collection one.

Hardcore segments that need to be placed into the hierarchy (deciduous segments) are so placed as follows. append is called to create a branch. This creates a vtoce for the segment and makes active, creating if necessary, all parent directories. Normally, segment control activities would then create an aste for this being created segment which would be threaded as a son of the parent directory's aste. In this initialization case, though, the aste for the new segment already exists. We hand thread this aste into the normal segment lists and thread it as a son of the parent directory's aste. The directory entry for this segment created by append gives the vtoc index of the vtoce for it. By placing this vtocx into the old aste for the new segment, vtoc_man can make the vtoce for this now deciduous segment reflect the placement of this segment in the hardcore partition (where it was allocated during hardcore initialization). The segment is now properly active and accessible from the hierarchy.

HARDWARE AND CONFIGURATION INITIALIZATION

The initialization of the hardware and configuration information pertaining to it (basically scs (and also iom_data)) is a little understood process. To better understand the method of initialization, it is necessary to start with an understanding of the operation of the hardware on which Multics runs. This description pertains to the DPS-8 hardware series. The description for the Level-68 series is similar but is not included.

Interconnection of Multics hardware

A Multics system consists of a set of system control units (SCU's), central processing units (CPU's) and input/output multiplexors (IOM's).

A SCU controls access to memory. Each SCU owns a certain range of (absolute) memory. Any active unit (a CPU or an IOM) that requires access to memory does so by requesting the access from the SCU that owns the given range of memory.

A CPU performs the actual computations within the system. It operates by requesting instructions and data from the appropriate SCUs, operating upon them, and placing the results into appropriate locations in SCUs.

An IOM performs input and output to physical devices. It requests data from SCUs to send to devices and takes data from devices, storing it into SCUs.

IOMs and CPUs are not directly connected to one another. The only method of communication between active modules is through a SCU. The connection of modules in a Multics system is therefore something like the following.

```
                     -------    -------
                    | IOM A |  | IOM B |
                     -------    -------
                       |    \/    |
                       |    /\    |
 -------             -------    -------        -------
| MEM A |---| SCU A |  | SCU B |---| MEM B |
 -------             -------    -------        -------
                       |    \/    |
                       |    /\    |
                     -------    -------
                    | CPU A |  | CPU B |
                     -------    -------
```

The crosses indicate that both IOMs and both CPUs connect to both SCUs; the CPUs and IOMs are not themselves connected.

The active modules (CPUs and IOMs) have up to four ports that go to SCUs. These are referred to as the memory ports of the active module in question. The SCUs have up to eight ports that can go to active modules. These are referred to as the active module ports of the SCU or just simply as SCU ports.

All CPUs and IOMs must share the same layout of port assignments to SCUs. Thus, if memory port B of CPU C goes to SCU D, the memory port B of all other CPUs and IOMs must go to SCU D. All CPUs and IOMs must describe this SCU the same; all must agree in memory sizes. Also, all SCUs must agree on port assignments of CPUs and IOMs. Thus, if port 3 of SCU C goes to CPU A, then port 3 of all other SCUs must also go to CPU A.

Configuration of Multics hardware

The various hardware modules need varying amounts of configuration description information with which to run.

CPU AND IOM HARDWARE CONFIGURATION

The CPUs and IOMs require access to main memory. They resolve their own internal concept of memory address (virtual or io page table) into an absolute main memory address. This address must describe a location in one and only one memory store unit, which itself must be connected to only one SCU. The IOM or CPU must determine which SCU owns the memory location desired, and supply that SCU with the address relative to its base of the location desired. The CPU and IOM do this with the memory configuration information known to them by configuration switches and changed under software control.

The configuration data known to the processor (at the hardware level) is found via the rsw instruction with operands of 1 and 2, which can be obtained by calling pmut$rsw with these operands. The format of the data returned is described in rsw.incl.pl1 and also shown below.

The data returned by the rsw 2 instruction is shown below.

bits	meaning
0-3	4-word/2-word interlace (if enabled)
4-5	processor type (01 for DPS-8)
6-12	seven msb's of the fault base
13-13	id prom installed
19-19	dps (marketing) option
20-20	8k cache option
23-23	Multics model CPU
24-24	Multics mode enabled
29-32	cpu speed (0 = 8/70, 4 = 8/52)
33-35	cpu number

The data returned by rsw 1 consists of four nine bit bytes describing each of the four possible memory (SCU) ports of the processor. The bytes appear in order in the result, SCU 0 in the high order bits. The format of the byte is:

bits	meaning
0-2	port assignment
3-3	port is enabled
4-4	system initialize is enabled
5-5	port is interlaced with neighbor
6-8	memory size

The actual memory size of the memory attached to the SCU attached to the processor port in question is 32K * 2 ** (encoded memory size). The port assignment couples with the memory size to determine the base address of the SCU connected to the specified CPU port (absolute address of the first location in the memory attached to that SCU). The base address of the SCU is the (actual memory size) * (port assignment).

The IOM has similar port description information interpreted similarly. This information is not readable from the CPU.

SCU HARDWARE CONFIGURATION

The SCU also has description of its ports (to CPUs and IOMs) as well as description of the store units attached to it. This information is determined by the rscr instruction (pmut$rscr), given the SC_CFG argument. (The explanation of the rscr instruction appears later.) The portions of the result that pertain to SCU port and store unit configuration are shown below.

bits	meaning
09-11	lower store size
12-15	store unit (A A1 B B1) on-line
21-21	SCU in program mode (vs manual)
22-22	non-existant address checking enabled
23-29	non-existant address limit
30-30	store unit interlace enabled
31-31	B is lower addressed store (vs A)
32-35	port enable mask for ports 0-3
57-63	cyclic priority (0/1-6/7)
68-71	port enable mask for ports 4-7

A DPS-8 SCU may have up to four store units attached to it. If this is the case, two store units form a pair of units. The size of a pair of units (or a single unit) is 32K * 2 ** (lower store size) above.

If the non-existant address flag is on, any address to a store unit whose high order bits (above the lower 15) is greater than or equal to the non-existant address limit generates a non-existant address SCU illegal action.

A SCU will respond to and provide information to only those ports that are enabled (port enable mask above).

SCU ADDRESSING

There are three ways in which an SCU is addressed. In the normal mode of operation (memory reading and writing), an active unit (IOM or CPU) translates an absolute address into a memory port (on it) and a relative memory address within the memory described by the memory port. The active module sends the address to the SCU on the proper memory port. If the active module is enabled by the port enable mask in the referenced SCU, the SCU will take the address given to it and provide the necessary memory access.

The other two ways pertain to reading/setting control registers in the SCU itself. For each of these, it is still necessary to specify somehow the memory port on the CPU whose SCU registers are desired. For the rmcm, smcm and smic instructions, this consists of providing a virtual address to the processor for which bits 1 and 2 are the memory port desired.

The rscr and sscr instructions, though, key off the final absolute address to determine the SCU (or SCU store unit) desired. Thus, software needs a way to translate a memory port number into an absolute address to reach the SCU. This is done with the paged segment scas, generated by init_scas (and init_scu). scas has a page corresponding to each SCU and to each store unit in each SCU. pmut$rscr and pmut$sscr use the memory port number desired to generate a virtual address into scas whose absolute address (courtesy of the ptws for scas) just happens to describe memory within that SCU.

The cioc instruction (discussed below) also depends on the final absolute address of the target operand to identify the SCU to perform the operation. In the case of the cioc instruction, though, this has no particular impact in Multics software. All target operands for the cioc instruction when referencing IOMs are in the low order SCU. When referencing CPUs, the SCU performing the connecting has no real bearing.

Inter-module communication

As mentioned earlier, communication between active modules (CPUs and IOMs) can only be performed through SCUs.

CPUs communicate to IOMs and other CPUs via the cioc connect i/o channel) instruction. The operand of the instruction is a word in memory. The SCU containing this operand is the SCU that performs the connect function. The word fetched from memory contains in its low order bits the identity of a port on the SCU to which this connect is to be sent. This only succeeds if the target port is enabled (port enable mask) on the SCU. When the target of the connect is an IOM, this generates a connect strobe to the IOM. The IOM examines its mailbox in memory to determine its course of action. When the target of the connect is another CPU, this generates a connect fault in the target processor. The target processor determines what course to follow on the basis of information in memory analyzed by software. When a connect is sent to a processor (including the processor issuing the connect), the connect is deferred until the processor stops executing inhibited code (instructions with the inhibit bit set).

Signals sent from an IOM to a CPU are much more involved. The basic flow is as follows. The IOM determines an interrupt number. (The interrupt number is a five bit value, from 0 to 31. The high order two bits are the interrupt level.

0 - system fault
1 - terminate
2 - marker
3 - special

The low order three bits determines the IOM and IOM channel group.)

0 - IOM 0 channels 32-63
1 - IOM 1 channels 32-63
2 - IOM 2 channels 32-63
3 - IOM 3 channels 32-63
4 - IOM 0 channels 0-31
5 - IOM 1 channels 0-31
6 - IOM 2 channels 0-31
7 - IOM 3 channels 0-31

It also takes the channel number in the group (0-31 meaning either channels 0-31 or 32-63) and sets the <channel number>th bit in the <interrupt number>th memory location in the interrupt mask word (IMW) array in memory. It then generates a word with the <interrupt number>th bit set and sends this to the bootload SCU with the SXC (set execute cells) SCU command. This sets the execute interrupt cell register in the SCU and sends an XIP (execute interrupt present) signal to various processors connected to the SCU. (The details of this are covered in the next section.) One of the processors (the first to get to it) sends an XEC (execute interrupt cells) SCU command to the SCU who generated the XIP signal. The SCU provides the interrupt number to the processor, who uses it to determine the address of a fault pair in memory for the "fault" caused by this interrupt. The processing of the XEC command acts upon the highest priority (lowest numbered) bit in the execute interrupt cell register, and also resets this bit in the register.

Interrupt Masks and Assignment

The mechanism for determining which processors are candidates for receiving an interrupt from an IOM is an involved topic. First of all, a processor will not be interrupted as long as it is executing inhibited instructions (instructions with the inhibit bit set). Beyond this, though, lies the question of interrupt masks and mask assignment.

Internal to the SCU are two sets of registers (A and B), each set consisting of the execute interrupt mask register and the interrupt mask assignment register. Each execute interrupt mask register is 32 bits long, with each bit enabling the corresponding bit in the execute interrupt cell register. Each interrupt mask assignment register has two parts, an assigned bit and a set of ports to which it is assigned (8 bits). When a bit is set in the execute interrupt cells register, the SCU ands this bit with the corresponding bit in each of the execute interrupt mask registers. If the corresponding bit of execute interrupt mask register A, for example, is on, the SCU then looks at the A interrupt mask assignment register. If this register is not assigned (enabled), no further action takes place in regards to the A registers. (The B registers are still considered (in parallel , by the way).) If the register is assigned (enabled), then interrupts will be sent to all ports (processors) whose corresponding bit is set in the interrupt mask assignment register. Thus, only certain interrupts are allowed to be signalled at any given time (based on the contents of the execute interrupt mask registers) and only certain processors will receive these interrupts (as controlled by the interrupt mask assignment registers).

In Multics, only one processor is listed in each of the two interrupt mask assignment registers, and no processor appears in both. Thus, there is a one for one correspondence between interrupt masks that are assigned (interrupt mask registers whose assigned (enabled) bit is on) and processors who have an interrupt mask (SCU port number appears in an interrupt mask assignment register). So, at any one time only two processors are eligible to receive interrupts. Other processors need not worry about masking interrupts.

The contents of the interrupt mask registers may be obtained with the SCU configuration information with the rscr instruction and set with the sscr instruction.

bits	meaning
00-07	ports assigned to mask A (interrupt mask assignment A)
08-08	mask A is unassigned (disabled)
36-43	ports assigned to mask B (interrupt mask assignment B)
44-44	mask B is unassigned (disabled)

The contents of a execute interrupt mask register are obtained with the rmcm or the rscr instruction and set with the smcm or the sscr instruction. The rmcm and smcm instruction only work if the processor making the request has a mask register assigned to it. If not, rmcm returns zero (no interrupts are enabled to it) and a smcm is ignored (actually, the port mask setting is till done). The rscr and sscr instructions allow the examining/setting of the execute interrupt mask register for any port on a SCU; these have the same effect as smcm and rmcm if the

SCU port being referenced does not have a mask assigned to it. The format of the data returned by these instructions is as follows.

bits	meaning
00-15	execute interrupt mask register 00-15
32-35	SCU port mask 0-3
36-51	execute interrupt mask register 16-31
68-71	SCU port mask 4-7

Operations upon masks

Since at most two processors have interrupt masks assigned to them, not all processors can manipulate their own masks. But, to remove the need for processors to ask whether they have a mask before operating upon them (in partiuclar, to mask interrupts), a mechanism has been devised. It's execution is carried out by by pmut$set_mask and pmut$read_mask. The code fragment of pmut that reads/sets the mask follows.

```
read_mask:
          lxl1      prds$processor_tag
          lprpab    scs$mask_ptr,x1
          xec       scs$read_mask,x1

set_mask:
          lxl1      prds$processor_tag
          lprpap    scs$mask_ptr,x1
          xec       scs$set_mask,x1
```

For each processor tag, then, there is a set of data pointers and instructions in scs$mask_ptr, scs$read_mask and scs$set_mask that either operate upon the processor's mask or pretend they did. When the processor in question does not have an interrupt mask, the data is as follows:

```
mask_ptr - packed pointer to
          prds$simulated_mask

read_mask:
          ldaq      ab|0

set_mask:
          staq      ab|0
```

which will succeed in doing nothing. When the processor does have an interrupt mask, the data is as follows:

```
mask_ptr - packed pointer to
          scs$port_addressing_word(bootload scu)
```

```
read_mask:
          rmcm      ab|0,*

set_mask:
          smcm      ab|0,*
```

which will read and set the mask. The array scs$port_addressing_word contains the data words required as operands for the rmcm, smcm and smic instructions. They contain the memory port number in their low order bits (i.e., their array index is their contents). The smic instruction uses scs$interrupt_controller (the low order memory port (address 0)) as ian array index to perform the smic against the low order SCU.

The operands of the pmut$read_mask and pmut$set_mask operations (rmcm and smcm instructions, respectively) were described above. The value scs$sys_level masks all interrupts. It has zeroes for all bits loaded into the execute interrupt mask register but has all ones for all ports of the SCU to which enabled active modules are connected. scs$open_level has the same SCU port enable bits but has ones for all interrupts of all levels from both channel sets of all IOMs currently active.

Sequence of Initialization

Configuration initialization occurs primarily within scs_and_clock_init, iom_data_init, scas_init and init_scu called from within scas_init.

The name of this routine should probably be just scs_init. The clock portion is really just a check of clock functioning (and setting up clock data in general). It fills in the scs$port_addressing_word's as described above. scs$processor_switch_data is read to get the configuration and data switch values. scs$bos_processor_tag is set to indicate this cpu (currently the only one running) as the bootload cpu. scs$read_mask, scs$set_mask and scs$mask_ptr are set to the dummy values mentioned above. When scs_and_clock_init is run, all interrupts are masked, and no one really needs to think about its masks. The various processor ports are examined looking for memories. The port number of the low order memory so far is set into scs$interrupt_controller and sys_info$clock_. When scs_and_clock_init is finsihed, then, the configuration data for the bootload cpu is known, as well as for the various memories attached to it. Examination of this data and setting of masks waits for later programs.

iom_data_init initializes the data needed by io_manager. This includes descriptions of the various IOMs and their channels. The basic setup of this information (numbers of IOMs, numbers of channels) was set up by get_io_segs who obtained this data from the config_deck. Most description of IOMs appears in iom_data so no major changes take place to scs within iom_data_init.

Aside from filling in scw's and lpw's for each channel_table and mailbox entry, the more interesting part of iom_data_init is the main IOM card processing loop. It examines each IOM card, making sure that no IOM is duplicated, that the field values are reasonable, that no card claims an SCU port claimed by another IOM (and sets scs$port_data to claim the IOM) etc. The iom_data.per_iom data is initialized as to configured, on_line, paged, etc. This routine adds to scs$open_level the necessary bits to enable interrupts from the IOMs. (Interrupts are not enabled until initialize_faulst$interrupt_init.)

The conclusion of configuration initialization occurs in scas_init and its servant, init_scu. At its entry, scs$port_data has been set up to only describe the IOMs. This routine will set these for processors. It also initializes scas, as its name implies. This requires determining all memories and store units. Aside from this, the routine checks the port enable switches for the processor ports for correctness.

The first loop of interest scans all CPU cards. It checks them for reasonableness, that no CPU is mentioned twice, that no other active module claims this SCU port, etc. The cow's (connect operand words) used when perfoming cioc's to this processor are set.

What follows this is the SCU scanning loop. It takes each MEM card and checks it for reasonableness, whether tags are duplicated, whether the memory extent (from rsw_util) matches and does not overlap any other memory, etc. init_scu is then called.

init_scu initializes an SCU. This is the routine that sets up scas for a particular SCU. This is done by installing ptw's into the page table for scas to describe the SCU. Reading the configuration from the SCU, the data is compared against the computed data given the processor configuration information (which scas_init compared against the config_deck description of the memory). If the configuration from the SCU indicates aditional store units, the scas pages for them are set (to allow getting the store unit mode registers with an rscr).

The mask checking part of init_scu makes sure that each interrupt mask that is assigned on the SCU is assigned to a processor (as opposed to an IOM) and that no more than one mask indicates a given processor. This is done by walking down the CPU data in scs and comparing the mask data recorded for the other processor ports for duplication. This also records which masks assigned for this SCU are claimed by processors. Any mask that is assigned that does not appear in the description of a processor is mis-assigned.

After the SCUs have been initialized in this way, a little more work is left. The bootload CPU's ports are checked, so that no extra port is enabled. For each IOM (and the bootload CPU), the port enable bit is set in each SCU.

For each processor, we find the processors with masks assigned. For these, we set scsset_mask, scsread_mask and scs$mask_ptr to actually perform the rmcm and smcm instructions as described above to manipulate their masks. We check to be sure that the bootload CPU owns one of the masks.

The final loop examines the ordering of active modules on the SCUs to see if the cyclic priority switches can be set. This is only done if the IOM group does not overlap the CPU group.

PAGE CONTROL INITIALIZATION

Page control initialization consists of a variety of activities run during collection one. init_sst build the sst and core_map. The sst is needed since we need to have an aste for page control so that it can find what disk needs i/o (from the pvtx within the aste). The core_map is needed since it shows the status of memory pages (initially free between the groups of initialization segments, currently wired). Page control needs this information so it can find a free memory frame into which it can read a desired page. init_pvt performs the function of creating the pvt. It is the index into the pvt for the device from which a page (or other i/o) is desired that is needed by disk_control (dctl). read_disk$init is needed to initialize page reading/writing through rdisk_seg. This routine builds the paged segment rdisk_seg, which can be mapped onto the desired page of disk to read. The aste for rdisk_seg contains the pvtx of the disk to read. The page table word for rdisk_seg provides the disk address. At this point, we can actually read or write a page by touching rdisk_seg within read_disk. read_disk sets up the aste and page table word, as described. When the page is touched, a page fault will wake up page control. It will find a free memory frame, read the page in, and resolve the page fault.

read_disk_label uses read_disk, then, to read a disk label. init_root_vols uses read_disk_label to read the label of hardcore partition volumes. Given the label, it finds the partition map and finds the hardcore partition. A small volmap is built that describes this partition and is mapped onto the beginning of the partition. A small record stock is built to describe the volmap. Given this initial stock, attempts to create or free pages on a disk (within the hardcore partition) can succeed. Now, we can create hardcore segments by building null page tables and taking page faults. Page control will find a free page from the volmap for the partition (whose pvtx is in the aste) and resolve our page fault. At this point, all of the services we need of page control are available. For the ease of later activities who need various partitions to map paged areas onto, init_partitions is called to validate the part information. We now page happily.

Later, in collection two, the real volmaps and record stocks are set up by accept_rpv. After this point, page control will simply shift its page creation/freeing activity to that described by the paging region. All hardcore segments had their pages pre-withdrawn from the hardcore partition, so no possibility exists that we will accidentally put a paging region page into a hardcore segment.

SEGMENT AND DIRECTORY CONTROL INITIALIZATION

Segment and directory control are initialized in stages throughout collections one and two. It started in collection one when the sst was built. It continues into collection two with getuid$init. This allows us to generate unique ids for newly created segments and directories. init_vtoc_man paves the way for vtoc_man to perform i/o on vtoces. Segment control's trailer segment is created by init_str_seg. accept_rpv sets up the real vtoc maps and vtoc stocks. Now vtoc_man can really read and write vtoces, as well as create and free them. Now, if we were to try a normal activation of a segment, given its pvtx/vtocx, we could find the segment and thread the segment into the right astes and trailers. init_lvt builds an initial rlv (in the lvt) out of the disks listed as having hardcore partitions. This allows segment control's disk selection algorithm to be able to find a disk to use when segments try to be created. We now have enough mechanism in place to utilize most of the facilities of segment control, but we cannot yet access and activate hierarchy segments.

The initialization of directory control is imbedded within the initialization of segment control. It started with dir_lock_init providing us with an initially empty list of locked directories. The real start up of directory control, though, occurs in init_root_dir. This builds the kst (used at segment fault time to resolve segment numbers into an understanding of what needs activation) and creates (if need be) and activates and initiates by hand the root directory. Directory control can now reference hierarchy objects with segment control's help. Any attempt to create a hierarchy segment (append) can succeed by selecting a disk (lvt lookup), vtoce creation (vtoc_man using vtoc stock, vtoc map and vtoc buffers) and aste creation (using sst and the trailer seg). Also, deactivation is possible since the trailer is built to describe what to setfault and the kst is present to be able to re-activate. At this point, we are able to handle segment faults, given the information in the kst and by recursively traveling down the hierarchy by virtue of the fact that the root is now and always active.

SEGMENT NUMBER ASSIGNMENT

There are basically three classes of segments as far as segment number assignment is concerned. The first is segments that will be a permanent part of the supervisor. These are assigned consecutive segment numbers, starting at 0. dseg is always 0, of course.

The second class is initialization and collection temporary segments. These are assigned consecutive numbers starting at 400 octal. Although temporary segments are deleted at the end of each collection, their numbers are not re-used. We continue to assign the next non-used number to the next temporary or initialization segment.

The order of assignment of these numbers is purely according to the order that the segments are encountered. The first few segments are assigned numbers by template_slt_; but, again, this is in order of encounterance. The only requirements are that dseg must be segment 0 and that the slt must be segment 7 (assumed by all dump analyzers).

Normal hierarchy segments fall into the third class of segments, as far as segment number assignment is concerned. As for these, the sequence is as follows. The next higher mod 8 segment number after the last permanent supervisor segment is chosen as the stack base (ring zero stack number). The next seven numbers are assigned to the outer ring stacks, in order. Since the root is made active after this, and the root becomes the first real hierarchy segment initiated, it gets the segment number after stack_7. Other segments are assigned progressively higher segment numbers according to segment control's normal rules. We do not need to worry about running into segment number 400 octal since these segments will be deleted before we ever get that far. Only permanent supervisor segments will show up in one's dseg.

Some supervisor segments (deciduous segments) get initiated into the normal user's address space. Regular stacks are initiated by special handling (makestack called from the segfault handler) and are directly referred to by the reserved stack segment numbers. A normal segment like bound_library_1_ is activated through normal segment control means. Thus, it will appear in two places in the user's address space; one in the supervisor segment number range (with ring brackets of 0, 0, 0, by the way) and once in the user ring segment number range (greater than the root's segment number) (with ring brackets of 0, n, n).

This is a problem for hardcore gates, though, relative to their linkages. A user ring call to bound_library_1_ will cause modules within it to find their linkage section from the lot entry for this segment. Any module called from bound_library_1_ will also be in the user ring, so the user ring linkage section for the segment number corresponding to the user ring version of bound_library_1_ will find the called module. Hardcore gates, however, don't call hierarchy entities but instead call entities that can only be found through the linkage section generated via pre-linking during initialization which resides in the ring zero linkage section corresponding to the hardcore segment number. To make it possible to find this easily, init_hardcore_gates stored into the hardcore gate segdef .my_lp the pointer to this linkage section. Thus, when called from the outer ring with the outer ring segment number, hardcore gates will quickly switch over to the hardcore linkage section and function properly.

TRAFFIC CONTROL INITIALIZATION

All three collections contribute efforts toward enabling traffic control. Collection one starts by building the tc_data segment in tc_init, full of empty aptes to describe processes. At this time, though, a flag in tc_data indicates that mult-programming is not active. Any call to traffic control to pxss$wait will simply loop for notification (which will come from a call to pxss$notify in some interrupt routine). No polling routines are run at this time. Other initialization activities proceed to build the supervisor address space.

Collection two starts up multi-programming. It does this through tc_init$part_2. Multi-programming requires multi-processes; initially this is the Initializer and an idle process, but it soon encompasses answering service created processes and hardcore processes (hprocs). Creating an idle process requires creating a pds, stack_0 (prds) and dseg for it. The dseg and pds are simply copies of those for the Initializer, now that they are filled in. apte entries for the Initializer and for idle are created. We can now consider multi-programming to be on. start_cpu is called to start the processor. For the bootload processor, this means calling init_processor in a special case environment (non-absolute mode, if nothing else). init_processor (the idle loop) marks itself as a running processor, sends itself a connect, and unmasks the processor. The connect will go to traffic control, who will pre-empt idle and return control to Initializer.

In collection three, start_cpu is called (from tc_init$start_other_cpus) in the same manner as would be done for adding a cpu during reconfiguration. This is somewhat described in the reconfiguration manual.

SECTION 9

SHUTDOWN AND EMERGENCY SHUTDOWN

The goal of shutdown, obviously enough, is to provide an orderly cessation to service. A normal shutdown is one in which the system shuts itself down, following the direction of the operator's "shut" command. An emergency shutdown is that operation invoked by bce which forces Multics to run emergency_shutdown, which performs the clean up operations below.

One could consider the system to be shutdown if one simply forced a return to bce, but this is not enough. Proper shutdown involves, at first, the answering service function of logging out all users. The answering service then shuts itself down, updating final accounting figures. Now with just the Initializer running, the task of shutdown described here follows.

The major goal of shutdown and emergency_shutdown is to maintain consistency of the storage system. It is necessary to move all updated pages of segments to disk, to update all directories in question with new status information, to update vtoces of segments referenced, and to clear up any effects caused by the creation of supervisor segments.

These functions must be performed in several stages. Also, the ordering of operations is such as to minimize the degree of inconsistency within the storage system that would occur if a failure were to occur at any point.

Since these same functions are performed for an emergency shutdown, the operations are performed so as to assume as little as possible from the information in memory.

ORDER OF EXECUTION OF SHUTDOWN

The module shutdown is called via hphcs_$shutdown. It starts by removing the fact that we were called from an outer ring, so we won't accidentally return. An any_other handler is set up to flag any possible error, later. The first action of shutdown is to force itself to run on the bootload cpu and to stop the others (stop_cpu).

disk_emergency$test_all_drives checks out all of the storage system drives at once to avoid errors later.

tc_shutdown destroys the remnants of any processes and turns off multi-processing.

scavenger$shutdown cleans up any scavenges that were in progress.

We then switch over to the stack inzr_stk0 for the rest of shutdown. This is performed through the alm routine, switch_shutdown_file_system, which starts the file system shut down.

shutdown_file_system is the first program called on inzr_stk0. It is a driver for the shutdown of the file system. It starts by updating the rpv volmap, vtoc header (and vtoc map) and label of the rpv to show the current state (in case problems occur later).

The most important step, from the user's point of view, is to flush all pages in memory (considered to be part one of shutdown) with pc$flush. This is relatively easy and safe to perform since it only requires walking down core map entries; sst threads, etc. do not have to be trusted. This marks the completion of (emergency) shutdown, part 1.

The stack zero segments are released so that demount_pv can deactivate them.

deactivate_for_demount$shutdown deactivates all non-hardcore segments and reverts deciduous segments (removes from the hierarchy those supervisor segments put into the hierarchy during initialization). This updates the directories containing those segments that were active at shutdown time (and their vtoces).

Our next task is to remove the pages of these updated directories from memory. We start by demounting all operative disks (other than the rpv) with demount_pv. After this, if any locks remain set, we set the shutdown state to three; it is normally four.

If any disks are inoperative, we just perform another memory flush (to remove rpv directory pages), wait for console i/o to finish (ocdcm_$drain_io) and return to bce.

If all was okay, we demount the rpv with demount_pv. The storage system is now considered to be shut down. The ssenb flag in the flagbox is reset to show this. We flush memory once more, to get the last log messages out. The message "shutdown complete" is printed; we wait for console completion. Shutdown can now return to bce.

ORDER OF EXECUTION OF EMERGENCY SHUTDOWN

emergency_shutdown is called from bce. bce modified the machine conditions of the time of return to bce to cause a return to emergency_shutdown|0. This module initializes itself through text imbeded pointers to its linkage section, etc. and enters appending mode.

Multi-programming is forced off (tc_data$wait_enable).

The apt, metering and various apte locks are forced unlocked.

The return to bce earlier stopped all of the other cpus. scs$processor is set to show this fact.

The connect lock is forced unlocked.

Various trouble pending, etc. flags are reset in case of another failure.

scs masks, etc. are set up for single (bootload) cpu operation. We mask down to sys_level.

A switch is made to the idle process. This is done by using scs$idle_aptep to find the idle's apte. Its dbr is loaded.

All other cpus are set to delete themselves, in case they try to start.

The idle process has prds as its stack. A stack frame is pushed onto this stack by hand.

The ast and reconfiguration locks are forcibly unlocked.

The first external module is called. ocdcm_$esd_reset resets oc_data, and the console software. syserr_real$syserr_reset resets the syserr logger and the syserr_data segment and flags.

io_manager$reset resets iom_data status.

page$esd_reset resets its view of the disk dim.

pc_recover_sst recomputes the page control state. page$time_out is called.

disk_emergency$test_all_drives_masked runs as for normal shutdown, but in a masked state.

The prds is abandoned as a stack (it is reset) and the stack pointer set to null (idle process). The first page of template_pds is wired and the sdw for pds set to point to template_pds (hopefully a good pds). The first page is touched, hopefully successfully paging in the page. The stack pointers are then set to inzr_stk0. We then call wired_shutdown$wired_emergency.

wired_shutdown sets an any_other handler and unmasks the processor. It makes a few checks to see if the storage system was enabled. If a vtoc_buffer is in the unsafe state, its physical volume has its trouble count incremented.

For each pvte, the scavenger data is reset as in a normal shutdown. page$reset_pvte is called. Emergency shutdown part 1 is started.

fsout_vol updates the rpv information on disk as for shutdown.

Pages of segments are flushed from information in the core map entries (pc$flush). The rpv information is again written. This ends part one of emergency shutdown.

vtoc_man$stablilize gets vtoc buffers into shape.

We can now call shutdown_file_system and let normal operations carefully try to update directories and vtoces, as for a normal shutdown.

MODULE DESCRIPTIONS

deactivate_for_demount.pl1

Other than the flushing of pages themselves, the deactivation of segments (updating their directory entries and vtoces) performed by deactivate_for_demount is one of the most important functions of shutdown. The deactivations are performed by hand so as not to disturb aste threads. The operation consists of walking down the ast hierarchy (tree)-wise, recognizing that each active segment has all of its parent directories also active. We start at the root. For each segment to consider, we look down its inferior list. Each look at an aste and an inferior element is performed with a variety of validity checks on the aste (within pool boundaries, parent/son pointers correct, etc). If inferiors exists, they are pushed onto a stack (max hierarchy depth deep) of astes to consider. When we push an aste with no inferiors, we consider it directly.

If it was a hardcore segment (deciduous), it is removed from the aste list it is in and its vtoce freed. Non-hardcore segments have their pages flushed (pc$cleanup) if they are not entry-held (entry-held segments, such as pdses had their pages flushed earlier and will be caught in the final flush) and their vtoces updated (update_vtoce$deact). After a segment is considered, its brothers are considered. When they are done, we return back to their parent for consideration. We proceed in this manner until we consider and pop the root aste off the stack. Segment control is now no longer active.

demount_pv.pl1

demount_pv demounts a physical volume. It starts by waiting for everyone to relinquish the drive; that is, no one can be in the middle of a physical volume operation. All segments on the volume are deactivated. For the shutdown case described here, a special deactivation is performed to avoid possible problems in the case of emergency shutdown. Each aste pool is traversed (by numerical order, not link order because of possible mis-linkings). All non-hardcore segments (except the root) are deactivated in-line by calling pc$cleanup and update_vtoce$deact on the segment. We then wait for all vtoc i/o to complete to the disk. fsout_vol is called to update the volmap, vtoc header and map and the label. Finishing, we clean up the pvt entry.

disk_emergency.pl1

To ease the burden on shutdown of drives being inoperative, disk_emergency$test_all_drives is called. It tests all storage system drives by first assuming that each one is good, then running disk_control$test_drive. If the drive is declared inoperative this time, it is marked as such with an error report printed. Shutdown of objects on this drive will be suspended.

emergency_shutdown.alm

bce, when crashed to, received the machine conditions at the time of the call to bce. For an emergency shutdown (esd), bce patches these to force a transfer to emergency_shutdown|0. Multi-programming is forced off (tc_data$wait_enable). The apt, metering and various apte locks are forced unlocked. The return to bce earlier stopped all of the other cpus. scs$processor is set to show this fact. The connect lock is forced unlocked. Various trouble pending, etc. flags are reset in case of another failure. scs masks, etc. are set up for single (bootload) cpu operation. We mask down to sys_level. A switch is made to the idle process. All other cpus are set to delete themselves, in case they try to start. The idle process has prds as its stack. A stack frame is pushed onto this stack. The ast and reconfiguration locks are forcibly unlocked. ocdcm_$esd_reset resets oc_data, and the console software. syserr_real$syserr_reset resets the syserr logger and the syserr_data segment and flags. io_manager$reset resets iom_data status. page$esd_reset resets its view of the disk dim. pc_recover_sst recomputes the page control state. page$time_out is called. disk_emergency$test_all_drives_masked runs as for normal shutdown, but in a masked state. The prds is abandoned as a stack (it is reset) and the stack pointer set to null (idle process). The first page of template_pds is wired and the sdw for pds set to point to template_pds (hopefully a good pds). The first page is touched, hopefully successfully paging in the page. The stack pointers are then set to inzr_stk0. We then call wired_shutdown$wired_emergency.

fsout_vol.pl1

fsout_vol is called whenever a volume is demounted. This includes the shutdown equivalent function. It endeavors to update the volume map, vtoc header and map and label for a physical volume. It drains the vtoce stock for the disk (vtoc_stock_man$drain_stock) to return those vtoces withdrawn previously. The vtoc map is then forced out to disk. We can then free the vtoc stock. We similarly drain, write out and free the record stock/map. The dumper bit map is freed and updated to disk. The time map updated and mounted is updated in the label. If this is the root, this is the program that records in the label such useful information as the disk_table_vtocx and uid and the shutdown and esd state.

scavenger.pl1

The shutdown entrypoint to scavenger is called during shutdown to clean up any scavenge operations in progress. It walks down scavenger_data looking for live entries. For each, it clears the corresponding pvte fields deposit_to_volmap, scav_check_address and scavenger_block_rel which affects the operation of page control.

shutdown.pl1

This is the starting driver for shutdown operations. It is called from hphcs_$shutdown from the Initializer command shutdown. It forces itself to run on the bootload cpu and it stmps the others. disk_emergency$test_all_drives test the drives before use. tc_shutdown stops and destroys the other processes. scavenges are stopped (scavenger$shutdown). We then switch stacks back to inzr_stk0 and proceed through shutdown within switch_shutdown_file_system.

shutdown_file_system.pl1

shutdown_file_system is the driver for the shutdown of the file system. It runs on inzr_stk0. Its operations include: fsout_vol updating of the rpv, flushing pages of segments, releasing stack_0 segments for deactivation purposes, running deactivate_for_demount$shutdown to deactivate non-hardcore segments and revert supervisor segments threaded into the hierarchy at initialization (updating directories as a result) and then flushing memory again (by calls to demount_pv for the various disks). This module keeps track of the state of operativeness of drives; if any are inoperative, we just perform a final flush and quit; otherwise we can demount the rpv also. A final flush is performed to get syserr log pages out. After console i/o has drained, we can return to bce.

switch_shutdown_file_system.alm

switch_shutdown_file_system is the first program in a set to shut down the file system. It moves us back to inzr_stk0, the initialization stack for our processing. While it is fiddling with stack pointers, it also sets pds$stack_0_ptr and pds$stack_0_sdwp. On this new stack, it calls shutdown_file_system.

tc_shutdown.pl1

Traffic control is shutdown by tc_shutdown. It flags the system as being in a shutting down state (tc_data$system_shutdown). It also sets wait_enable to 0, disabling multi-programming. For each process in the apt, deactivate_segs is called, destroying the process and finishing our task.

wired_shutdown.pl1

The module wired_shutdown is the counterpart to shutdown in the esd case. It starts by setting an any_other handler and unmasking the processor. It makes a few checks to see if the storage system was enabled. If a vtoc_buffer is in the unsafe state, its physical volume has its trouble count incremented. For each pvte, the scavenger data is reset as in a normal shutdown. page$reset_pvte is called. Emergency shutdown part 1 is started. fsout_vol updates the rpv information on disk as for shutdown. Pages of segments are flushed from information in the core map entries (pc$flush). The rpv information is again written. This ends part one of emergency shutdown. vtoc_man$stablilize gets vtoc buffers into shape. We can now call shutdown_file_system and let normal operations carefully try to update directories and vtoces, as for a normal shutdown.

APPENDIX A

GLOSSARY

abs-seg
: An abs-seg is a reserved segment number in the hardcore address space used to access disk or memory outside of the normal mechanisms. That is, they are not built by the normal functions that append to the storage system nor are they built by the functions that create segments out of the hardcore partition or initialization memory. Examples of abs-segs are segments mapped onto an area of disk to allow paging to be used to read/write them (such a mechanism is used to read the config deck from disk) or segments mapped onto an area of memory for examination (page control does this to examine pages being evicted). abs-segs are managed (i.e., created and deleted), each in its own way, by a set of software created for the purpose; One may not use the standard system functions to operate upon them (such as segment deletion). However, the contents of the segments are addressed through normal mechanisms; that is, memory mapped abs-segs are referencable via the hardware and abs-segs built with an aste/page table pair in the sst are allowed to have page faults taken against them.

bce
: The Bootload Command Environment within bootload Multics, that is, the collection of programs and facilities that make up a command level that allows certain critical functions to be performed before storage system activation occurs during system initialization.

bootload Multics
: Those early parts of initialization that are capable of booting bce from a cold, bare machine, including bce itself.

cold boot
: A bootload in which the state of all hardware and peripherals is unknown. In particular, the Multics file system is either non-existant or has been destroyed. This is also known as an initial boot.

collection

A "collection" is a set of programs read in as a unit that together perform a function during initialization. Collections are referred to by number, starting with zero. Each collection depends on the mechanisms initialized by the collections that preceded it. As each collection finishes its task, some of that collection is deleted and some is kept, depending on the requirements of future collections.

There are also fractionally numbered collections, which consist of support entities for the preceding collection.

The division of initialization into collections is done based upon various restrictions imposed by the course of initialization. For example, since the first few collections must run entirely within memory, restrictions on available memory (and the amount that can be required of a system) force unessential programs into later collections.

contiguous

A contiguous segment is one whose memory locations describe contiguous absolute memory locations. Most segments do not have this requirement; their pages may appear arbitrarily in memory. Certain segments, though, such as the sst_seg must have their locations in order, due to hardware requirements for placement of their contents.

cool boot

A bootload in which the Multics file system is on disk and believed to be good but in which the state of memory and other peripherals is unknown. In particular, bootload Multics is not running. The mpc's may or may not have firmware running in them. The system is loaded from the MST (tape) and initiated via iom switches.

crash

A failure of Multics. This may be the result of a hardware or software failure that causes Multics to abort itself or the result of an operator aborting it. A crash of Multics during early initialization can produce a tape dump of memory. Crashes after this time can be examined with bce utilities or saved to disk by bce and analyzed later.

deciduous segments

These are segments generated or read in as part of initialization which are given branches in the hierarchy (by init_branches). Although they become part of the hierarchy, their pages remain in the hardcore partition and are therefore destroyed between bootloads. Examples are the segments in >sl1 and the Initializer's pds. (The name suggests the leaves of trees.)

deposit
A page control concept. It means to add an object to a list of free objects.

dseg
descriptor segment (see data bases)

dump
A subset of Multics segments saved after a crash that can be examined through various dump analysis tools to determine the cause of the preceding crash. A dump is either a disk dump, a dump performed to the dump partition of disk by the dump facility of bce, or an "early dump", one performed to tape during early initialization.

early initialization
Those parts of initialization needed to reach bootload Multics command level. All activities after leaving bootload Multics command level are referred to as service initialization.

emergency shutdown
A Multics operation, invoked by bce, that runs a subset of the hardcore facilities to shut down the file system (put the storage system into a consistent state) after a crash.

esd
emergency shutdown

hardcore
The supervisor of Multics, loosely defined. This is a collection of programs and segments generated or read in during initialization.

hproc
A hardcore process. Such a process is created by a call to create_hproc, as opposed to being created through the answering service. Such hprocs (currently SyserrLogger.Daemon and MCS_Timer_Daemon.SysDaemon) perform activities integral to the system operation and must be created prior to, and independent of, the answering service.

init segments
Segments needed only during the course of initialization. These are deleted after the end of the last hardcore collection.

initialization
The action of starting Multics. This consists of placing the appropriate software modules in the appropriate places and constructing the appropriate software tables such that an event, such as someone trying to dial a login line, or a page fault occuring, etc. will invoke the proper software

which will be in a position to perform the necessary operation.

kst
: known segment table (see data bases)

lvt
: logical volume table (see data bases)

MST
: Multics system tape

Multics system tape
: The "tape" is the set of Multics programs that will make up the supervisor in un-pre-linked form. This set of programs originates on a tape; some of them spend part of their lives in a disk partition.

nondeciduous
: A hardcore segment not mapped into the hierarchy. These segments live in the hardcore partition and are known only by having sdw's in the hardcore address space.

partition
: An area of a storage system disk, other than the label, vtoc, volume map and paging area. These areas can be accessed by paging mechanisms but are not used to hold pages of storage system segments. Hardcore segments are mapped onto the hardcore partition so that they may be used, and early initialization can run, without touching the file system proper.

pre-linking
: As the Multics supervisor is read from the MST, the various modules are linked together. This operation, called pre-linking, is similar to linking (binding) that occurs during normal service operation for user programs, except that it consists of running through all segments and finding all external references and resolving them. This is done during initialization for efficiency, as well as for the fact that the dynamic linker cannot be used to link itself.

ptw
: page table word

ptwam
: page table word associative memory

pvt
: physical volume table (see data bases)

root physical volume
: The main disk drive. It can never be deleted. This drive

is used to hold the original hardcore partition as well as the partitions required by bce and is therefore required at an early point in Multics initialization.

rpv
: root physical volume

scas
: system controller addressing segment (see data bases)

scs
: system communications segment (see data bases)

sdw
: segment descriptor word

sdwam
: segment descriptor word associative memory

shutdown
: The orderly cessation of Multics service, performed such as to maintain consistency of the storage system.

slt
: segment loading table (see data bases)

supervisor
: A collection of software needed for operation of user's software and support software provided for the user. This would include software to make disk accessing possible, to provide scheduling activity, etc. The supervisor in Multics is referred to as "hardcore".

temp segments
: Segments needed only during one collection. They are deleted at the end of the major collection, before loading the next collection.

uid
: unique identifier (of a segment)

unpaged
: A segment that is not paged under the auspices of page control. Such a segment has its page table either in unpaged_page_tables or int_unpaged_page_tables. Except for the possible presence of the breakpoint_page, these segments are contiguous. During early initialization, all segments are generated to be of this type. The program make_segs_paged forms paged segments that are copies of the pagable initialization segments. Certain wired segments, though, are left unpaged.

In previous releases, unpaged segments were literally unpaged, that is, they had no page table and had the unpaged flag set in their sdw. Currently only fault_vector, iom_mailbox, dn355_mailbox, isolts_abs_seg, abs_seg and abs_seg1 are of this type but will receive page tables in a future release.

vtoc
: The volume table of contents of a storage system volume. The vtoc is divided into entries (vtoce), each of which describes a hierarchy segment contained on that volume.

warm boot
: A bootload in which the Multics file system is present on disk and believed good, and in which bootload Multics is running on the processor. This type of bootload of Multics is performed from disk.

wired
: A page, or set of pages, is wired if it cannot be moved from memory by page control.

withdraw
: A page control concept, said of records and vtoces. It means to remove an object from a list of free objects.

APPENDIX B

INITIALIZATION AND INITIALIZED DATA BASES

This appendix describes various data bases kept by initialization or that are generated by initialization. As such, this list incorporates the most significant data bases within the system.

AI_LINKAGE (ACTIVE INIT LINKAGE)

This initialization segment corresponds to area.linker for initialization programs that will be paged. This area is built by bootload_loader and segment_loader from linkage sections found on the MST.

AS_LINKAGE (ACTIVE SUPERVISOR LINKAGE)

This hardcore segment corresponds to area.linker for paged supervisor programs. It is shared across processes, and can therefore contain only per-system static such as initialization static variables (when only one process is running) or system wide counters, etc. The linkage areas are formed in here by the various MST loading programs.

BCE_DATA (BOOTLOAD COMMAND ENVIRONMENT DATA)

bce_data keeps information that pertains to the command environment features of bootload Multics. It contains entries that describe the main pseudo i/o switches (input, output and error) as well as the state of exec_com and subsystem execution.

BOOTLOAD_INFO

bootload_info, generated initially from bootload_info.cds, contains various information about the state and configuration of early initialization. It contains: the location of the bootload tape (iom, controller channel, drive number and drive and controller type provided by the IOM boot function), status about firmware loading into the bootload controller, the location of the rpv (iom, controller, drive number and drive and controller type provided in the find_rpv_subsystem dialog), the location of the bootload console (and type), a variety of pointers to other data bases, as well as the master flags indicating the presence of BOS and the need for a cold boot. All of this data is copied into sys_boot_info during generation and during system initialization. Most references to this data are therefore to sys_boot_info.

bootload_info.cds has provisions to contain site-supplied configuration information. If these values are provided, no operator queries will be needed to bring the system up. Only cold site boots or disk problems would require operator intervention during boot. It is intended that an interface will be provided to fill in these values, such that generate_mst could set the values into the segment and the checker could report their settings in the checker listing.

CONFIG DECK

Historically named, the config_deck contains the description of the configuration. It contains one entry (card) for each iom, cpu, memory, peripheral subsystem, etc. in the configuration. It also describes various software parameters. These entries are referenced by programs too numerous to count. It is built initially by init_early_config to describe enough of the system to find the rpv and read in the real config_deck saved in a partition thereon. (If this is a cold boot, in which there would be no config_deck, the config_deck is entered manually or from the MST through the config deck editor.) After this time, it becomes a wired (at its initialization address) abs-seg mapped onto the "conf" partition. Various reconfiguration programs modify the entries.

CORE MAP

One of the page control data bases, the core_map describes frames of memory available for paging. Each entry describes a page frame. When a frame is used (it has a ptw describing it), the disk address of the page occupying the frame is kept in the core_map entry. init_sst initially builds the core_map. It is updated to accurately describe the state of pagable memory by make_segs_paged, which frees certain unpaged segments and collect_free_core which works to find various holes between segments. Page control maintains these entries.

DBM SEG (DUMPER BIT MAP SEG)

dbm_seg holds the dumper bit maps used by the volume dumper. It is paged off the hardcore partition. Its initialization as an area was performed by dbm_man$init. Each configured disk drive has two maps here, one for the incremental dumper and one for the consolidated dumper. The segment starts with the usual lock, control information, and meters. After this comes an area in which the bit maps are allocated. Each bit map consists of a bit corresponding to each vtoce on the volume in question. The bits indicate the need to dump the various segments.

DIR LOCK SEG

dir_lock_seg keeps track of lockings of directories and on processes waiting thereupon. It has a header with a lock and various status. Each dir_lock entry contains the uid of that which is locked, various flags, threads to a more recently locked entry, and the array of process ids for the various lockers (more than one only for all readers).

DISK POST QUEUE SEG

A part of page_control, disk_post_queue_seg is an obscure data base used to keep track of disk i/o postings that could not be made because the page table was locked at the time of i/o completion.

DISK SEG

The disk seg contains the various tables (except the pvt) used by disk_control and dctl to perform i/o to disks. It is split into the tables disk_data, disktab, chantab, devtab as well as the queue of disk i/o requests. disk_data contains entries giving the names and locations within disk_seg of the disktab entry for each configured disk subsystem. The disktab entry contains various subsystem meters, as well as holding the queue entries for the subsystem. Also contained herein are the chantab and devtab entries for the subsystem. Each chantab entry lists a i/o channel to use to perform i/o to the subsystem, given as an io_manager index. It also holds various per channel meters, and, most importantly, the dcw list that performs i/o on the channel. The devtab entries, one per subsystem drive, describe the drives. This consists of status information (inoperative, etc.) as well as per drive statistics.

DM_JOURNAL_SEG_

A page control data base, dm_journal_seg_ is used to keep track of page synchronization operations for data management. It is allocated and initialized by init_dm_journal_seg. It starts with a lock for manipulating the journal entries as well as the usual wait event information. Also present are information about the number of pages held in memory, the maximum pages held, the number of journals, etc. Corresponding to each aste pool is a structure containing a threshold and number of active, synchronized segments. Following this are various meters. Then comes the journal entries and then the page entries. Each journal entry contains the time stamp that determines when pages of the segment being held can be written (when the journal was written), the number of pages held, and a relative thread to the list of page entries for the pages being held. A page entry contains the threads that make up this list, a relative pointer to the core map entry for the page, and a relative pointer to the journal entry for the page.

DN355_DATA

This data seg, initialized by fnp_init, contains global information on each configured fnp. Data for each fnp includes: a pointer to the hardware mailbox, pointers to the dcw lists and the physical channel blocks (pcb), the number of subchannels, the iom/channel info, indexes into the pcb for lslas and hslas (hmlcs), status of the delay queues, various flags about the state of fnp operations, the lct (logical channel table) entry pointer, status of bootloading, and various counts of free blocks, input and output data and control transaction counts, etc.

DN355_MAILBOX

The dn355_mailbox is a set of mailboxes at fixed hardware addresses. They start with the fnp pcw. Also present are various counts of requests and the fnp crash data. Following this are 8 Multics initiated sub-mailboxes and 4 fnp initiated sub-mailboxes. The sub-mailboxes describe the line for which the operation is being performed along with the data for that operation.

DSEG (DESCRIPTOR SEGMENT)

The descriptor segment is a hardware known data base. It contains a sdw (segment descriptor word) for each segment within a process' address space. The ultra important processor register dsbr (descriptor segment base register), also called the dbr, indicates the absolute address of the page table describing it.

The sdw of a segment indicates the address of the page table of the segment (which contain the locations of the pages of the segment) and other information, such as the length of the segment, accesses allowed, etc. dseg must be segment 0. The initial dseg is generated by template_slt_ and copied into dseg by bootload_abs_mode. Entries are added by bootload_dseg, get_main and make_sdw as segments are loaded from the MST. The generation of sdws is integrated with the creation of slt entries, and the allocation of memory/disk that the sdw/page tables effectively describe.

FAULT VECTOR (FAULT AND INTERRUPT VECTORS)

This is another hardware known data base, at a fixed absolute memory address (0). It contains two words for each possible fault and interrupt. Normally, each entry contains a scu instruction, to store all machine conditions, and a tra instruction, to transfer to the code that handles the fault/interrupt. These two instructions are force executed in absolute mode on the processor. The entries are filled in by bootload_faults and initialize_faults. During some phases of initialization, when a particular fault/interrupt is to be ignored (such as a timer running out), the fault vector entry is set to a scu/rcu pair, which stores machine conditions and then reloads them, returning to the point of interruption. The scu and tra instructions actually perform indirect references through "its" pointers that are present following the interrupt vectors within this segment. During normal operations, only these pointers are changed.

FLAGBOX

The flagbox is an area of memory, at a known address, that allows communication between Multics operation and bootload Multics. This area contains information from Multics to bootload Multics such as the fact that we are crashing, and here's what exec_com to run. Bootload Multics can pass information up when booting, such as being in unattended mode so that Multics will know how to boot. The area is examined by various programs and set through commands/active functions in both Multics and bootload Multics operation. This area is within the bce toehold.

INZR_STK0 (INITIALIZER STACK)

This is the stack used by initialization and shutdown. The name stands for initializer stack. Originally wired, it becomes paged during initialization. Once the actual ring 0 stacks are created and after collection 3, initialization will leave this stack (in init_proc). Shutdown will return to this stack for protection as the stack_0's are deleted.

INT UNPAGED PAGE TABLES

The page tables for init and temp segments are kept here. It gets an initial value through template_slt_ and is managed by the various segment creation routines. Once make_segs_paged is run, no unpaged segments exists whose page tables are here. So, we delete this segment. The page table for this segment is contained within it.

IO CONFIG DATA

The inter-relationship between peripherals, mpc's and iom's is described in io_config_data. It contains a set of arrays, one each for devices, channels, controllers and ioms. Each entry, besides giving the name of each instance of said objects, gives indexes into the other tables showing the relationship between it and the rest. (That is, for example, each device shows the physical channels going to it; each channel shows the mpc for it, etc.)

IO PAGE TABLES

The page tables referenced by a paged mode iom for ioi_ operations are found in io_page_tables. It is a abs-wired segment, maintained by ioi_page_table. It starts with a lock and indexes of the start of free page table lists. The header ends with the size and in_use flags for each page table. The page tables themselves are either 64 or 256 words long; each page table of length N starts at a 0 mod N boundary and does not cross a page boundary within the segment.

IOI DATA

ioi_data contains information pertinent to ioi_ (the i/o interfacer). It holds ioi's data itself (ioi_data), as well as group channel and device entries for ioi handled devices. ioi_data contains counts of groups, channels and devices, reconfiguration lock, some flags, and then the channel, group and device entries. A channel/device group entry describes a group of devices available through a channel. It contains a lock, subsystem identifier, various flags describing the device group, the number of devices and some counters. A channel table entry describes the state of a channel. It holds status flags, the io_manager index for the channel, and a place for detailed status. A device table entry holds the wired information for an ioi device. Besides pointers linking it to the group and channel entries, it contains various status bits, workspace pointer, ring, process_id and event channels for communication with the outer ring caller, timeout and other time limits, offsets into the user's workspace for status storage, and the idcw, pcw, tdcw and status areas.

IOM DATA

iom_data describes data in use by io_manager. It starts with lpw, dcw, scw and status area for stopping arbitrary channels. This is followed by various meters, such as invalid_interrupts. Following this, for each iom are various pieces of state information, on-line, paged mode, etc. It concludes with more meters and ending with devtab entry indices. For each device, a status are is followed by various flags (in_use), channel identification, pcw, lpw and scw, status queue ptr, and various times and meters.

IOM MAILBOX

This segment is another hardware known and fixed segment. It is used for communication with the various ioms. The segment is split into the imw area, which contains a bit per channel per iom per interrupt level indicating the presence of an interrupt, followed by the mailboxes for sending information to the ioms and receiving status back.

KST (KNOWN SEGMENT TABLE)

The known segment table is a per-process segment that keeps track of hierarchy segments known in a process. Hardcore segments do not appear in the kst. The kst effectively provides the mapping of segment number to pathname for a process. It is the keeper of the description of segments that are initiated but not active within a process (as well as those that are active). The Initializer's kst is initialized by init_root_dir. It starts with a header providing the limits of the kst, as well as information such as the number of garbage collections, pointers to the free list, what rings are pre-linked, the 256k segment enable flag, a uid hash table, the kst entries and finally a table of private logical volumes connected to this process. Each kst entry contains a used list thread, the segment number of the segment, usage count per ring, uid, access information, various flags (directory, transparent usage, etc), an inferior count for directories or the lv index for segments and the pointer to the containing directory entry. It is this pointer that allows the name of the segment to be found. Also, the segment number of the directory entry pointer allows us to find the kst entry for the containing directory, etc., allowing us to walk up the hierarchy to find the pathname of a segment.

LVT (LOGICAL VOLUME TABLE)

The logical volume table consists of an array of entries that describe the various logical volumes. It starts with a count of entries as well as a maximum count limit. Following this is a relative pointer to the first entry and a hash table for hashing lvid (logical volume ids) into lvt entries. The entries that follow, one per logical volume, contain a relative pointer to the threaded list of pvt entries for the logical volume, the lvid, access class info for the volumes and then various flags like public and read_only. It is initialized by init_lvt to describe the rlv and maintained by logical_volume_manager.

NAME TABLE

The name_table contains a list of all of the various names by which the segments in the slt (see below) are known. This table is used by the slt management routines but especially by the various pre-linkers, who use it to resolve references to initialization modules. It is generated from template_slt_ and by the slt management routines, who read in the names from entries on the system tape.

OC_DATA

oc_data describes data used by ocdcm_ to handle consoles. It starts with the required lock, version, device counts, etc. Various flags are kept, such as crash on recovery failure. The prompt, discard notice are kept here. Status pointers, times, etc. are followed by information on the process handling message re-routing. Following this are indices into queues of entries followed by the queues. An entry exists for priority i/o (syserr output, which always forces a wait until complete), one for a pending read, and 8 for queued writes. After this are meters of obscure use. The segment ends with an entry for each configured console followed by an entry for each element of a event tracing queue. Each console entry provides its name, state, type, channel, status, etc. Each i/o queue entry provides room for the input or output text, time queued, flags (alert, input/output, etc), and status.

PHYSICAL RECORD BUFFER

The physical_record_buffer is a wired area of memory used by collection 0's and collection 1's MST tape reading routine for i/o buffers.

PVT (PHYSICAL VOLUME TABLE)

One of the disk describing tables, the physical volume table contains an entry for each configured disk drive. It can in some ways be considered the master disk describing table in as much as performing i/o to a disk drive requires the pvtx (pvt index) of the drive (the index number of the entry in the pvt for that drive). The pvt entry contains the physical and logical volume id for the drive, various comparatively static flags about the drive (such as storage_system, being_demounted, device_inoperative, etc.), information for the volume dumper and information about the size, fullness, volmaps and stocks (both record and vtoc) of the drive. This table is allocated by get_io_segs, and built by init_pvt. The various brothers in a logical volume are chained together in a list by the logical_volume_manager so that the vtoc_man can have a set of disks from which to select a target for a new segment. During initialization, make_sdw$thread_hcp (for init_root_vols) uses these threads (before the disk_table is accessed) to form the list of drives which contain hardcore partitions (those eligible to contain hardcore segments).

SCAS (SYSTEM CONTROLLER ADDRESSING SEGMENT)

This is a very curious pseudo-segment, built by scas_init out of page table words generated into scs. It contains one pseudo-page for each memory controller (and another page for each individual store other than the lowest). The address of the page is the base address of the store/controller. This segment makes references to it of the form n*1024 to form an absolute address to controller n. Thus, instructions like rscr (read system controller register) can use this segment (as indeed they do inside privileged_mode_ut) to reference the desired system controller registers.

SCAVENGER DATA

scavenger_data contains information of interest to the volume scavenger. Its header is initialized by init_scavenger_data. The segment starts with the usual lock and wait event. Following this is the pointer to the scavenger process table. Then come the meters. The scavenger process table, which follows, describes the processes performing scavenging operations. Each entry contains a process id of a scavenging process, the pvtx of the drive being scavenged, and indices of scavenger blocks in use. Scavenger blocks contain record and overflow blocks used to keep track of pages of a disk (its claiming vtoce and its state).

SCS (SYSTEM COMMUNICATIONS SEGMENT)

The scs is a hodge-podge of information about configuration and communication between active elements. It contains information about the scus and the cpus. It contains the cow's (connect operand words) needed to connect to any given cpu/iom, the interrupt masks used to mask/unmask the system, the various smic patterns (set memory interrupt cells), instructions to clear associative memories and the cache, connect and reconfiguration locks, various trouble flags/messages used for keeping track of pending communication of faults to bce, cyclic priority switch settings, port numbers for controllers, configuration data from the controllers, processor data switch values/masks, controller sizes, and the scas page table (see scas).

SLT (SEGMENT LOADING TABLE)

One of the most significant initialization data bases, the slt describes each initialization segment. It is built initially from template_slt_, an alm program that not only builds the appropriate slt entries for collection 0 segments, but also generates the dseg for collection 0. Each entry in the slt contains: pointers into name_table of the names and the final storage system pathname (and acl), if any, for the segment; access modes, rings, etc. for the segment; various flags used for generation/loading of the segment, such as abs/init/temp/supervisor segment, wired/paged, etc.; the length and bit_count, etc. It is maintained by bootload_slt_manager and slt_manager, who build entries based on information on the MST. These entries are maintained so that the various pre-linkers (bootload_linker and pre_link_hc) can find the target segments of the various references.

SST (SYSTEM SEGMENT TABLE)

The sst (which contains the active segment table) is one of the most important tables in Multics. It is the keeper of active segments. Each active segment has an entry describing it (its aste). The aste contains information used by segment control and communicated with page control on the state of a segment. The most important part of the entry is the page table words (ptws) describing the disk/memory location of each page. There are four pools of astes of different lengths to hold page tables of four possible maximum lengths: 4, 16, 64 and 256 ptws. The entries are threaded into various lists. The free entries of the various pools are threaded into lists. Active segments have their own lists. Separate lists are generated for temp and init (supervisor) segs. Aside from these threads, each aste also contains threads used to link segments to their parents and their trailer seg entry. Status information includes: the segment's uid, the current length, maximum length and records used, the pvtx and vtocx of the segment (which couple with the ptws to find the pages of the segment), various status bits of more obscure use, and finally the quota computation information. init_sst originally builds this table. The page table words are maintained by page control. The entries themselves are maintained by segment control.

SST NAMES_

The sst_names_ segment contains the names of paged segments described by the sst. It is initialized by init_sst_name_seg during collection 2 and maintained by segment control only if the astk parm appears. It starts with information describing the four aste pools followed by the paged segment primary names.

STACK 0 DATA

stack_0_data contains information keeping track of the ring 0 stacks (stack_0.nnn) that are shared between processes (one per eligible process). It is initialized by init_stack_0. It has a lock used to control threading of a pool of such stacks. Each entry contains a list thread, a relative pointer to the aste for the segment, a relative pointer to the apte for the holding process, and the sdw for the stack. When this stack is given to a process, this sdw is forced into its dseg; the acl of the stack is kept as a null acl.

STOCK SEG

stock_seg contains the record and vtoce stocks, a part of the reliable storage system. Whenever a new page or vtoce is needed for a drive, it is obtained from these stocks. The stocks are filled by pre-withdrawing a number of records or vtoces from the drive. This mechanism is used so that, upon a crash, it is guaranteed that any records or vtoces being created were marked in the record or vtoc maps as in use. This prevents re-used addresses.

STR SEG (SYSTEM TRAILER SEGMENT)

The str_seg is a paged segment used by segment control to perform setfault functions. It is initialized into a list of free entries by init_str_seg. Each entry contains the usual backward and forward threads forming a list of trailers for a given segment (the list itself is found by a relative pointer in the aste for the segment). When needing to fault a segment, this list shows all processes containing the segment. The entry shows the segment number, for a process with this segment active, of the segment and a relative pointer to the aste for the dseg of that process (which is where we need to fault the sdw).

SYS_INFO

sys_info is a keeper of all sorts of information about the state of the system. The most important entries to initialization are sys_info$initialization_state, which takes on values of 1, 2, 3 and 4 corresponding to whether we are running initialization collection 1, 2, 3 or whether we are running service (beyond collection 3), and sys_info$collection_1_phase, which takes on values defined in collection_1_phases.incl.pl1 corresponding to running early, re_early, boot, bce_crash, service and crash passes through collection 1. Also included are key things like: the scu keeping the current time, the current time zone, various limits of the storage system, and some ips signal names and masks. The variable "max_seg_size" records the maximum length of a segment. This value is changed during bce operation to describe the maximum length of a bce paged temp segment. This allows various Multics routines to work without overflowing segments. Also in sys_info is "bce_max_seg_size", this bce maximum segment length. This is available for any user ring programs who desire to limit the size of objects they prepare for the bce file system.

SYS_BOOT_INFO

See bootload_info, above.

SYSERR_DATA

The syserr_data segment is part of the syserr logging mechanism. syserr actually just writes messages into this segment and not to the paged log to avoid problems of paging during possible system trouble. It is up to the syserr hproc to move these messages from syserr_data to the log.

SYSERR_LOG

The paged abs-seg syserr_log, which describes the log partition of disk, is used to hold the syserr log. It is mapped onto the log partition by syserr_log_init. The syserr mechanism involves putting syserr messages into syserr_data (which are possibly written to the console) and then waking up the syserr hproc which copies them into the paged partition. This is done so that page faults are taken by the hproc, not by the syserr caller who may be in trouble at the time. It starts with a header providing the length of the segment, a lock, relative pointers to the first and last messages placed there and also copied out (by the answering service), the threshold that shows how full the partition can get before the answering service is notified to copy out the messages, the event channel for notification (of the answering service) and the event for locking. Following this are entries for the various syserr messages. Each message is threaded with the others; it has a time stamp, id number, and the text and optional data portions of the message.

TC_DATA

tc_data contains information for the traffic controller. The most obvious entry list herein is the list of aptes (active process table entries). There is one apte for every process. The apte lists activation information for the process, such as its dbr, its state (blocked/running/stopped/etc.), various per-process meters (such as cpu usage), its work class, and other per-process scheduling parameters. Following the apt is the itt (inter-process transmission table), maintained by pxss (the traffic controller) to hold wakeups not yet received by a target process. The call to hcs_$wakeup (or its pxss equivalent) places an entry in the itt containing the target process id, the event channel, the message data, etc. The next call to hcs_$read_events obtains the events waiting for the target process. Also present in tc_data is various meters (tcm.incl) and other flags. Imbeded within this is the wct (work class table) which keeps track of the status of scheduling into work classes. tc_init builds these tables (see tc_data_header).

TC_DATA_HEADER

This is a trick initialization segment. tc_data_header is allocated wired storage by tc_init to hold the real tc_data. tc_data, originally build just from a cds segment and therefore just describing the header of tc_data, is copied in. The sdws for tc_data and tc_data_header are then swapped. As such, the initialization segment tc_data_header (which describes the read in tc_data) is deleted, but tc_data (now mapped onto the allocated tc_data_header area) remains.

TOEHOLD

The toehold is another area for Multics/bootload Multics communication. (In particular, the flagbox is contained within it.) The toehold is a small program capable of getting to bootload Multics from a crashing/shuting down Multics service. (Its name is meant to suggest holding on by one's toes, in this case to bootload Multics.) init_toehold builds a dcw (device control word) list that, when used by the toehold program, can write the first 512k of memory (those used by bootload Multics) out to the bce partition and read in bootload Multics (saved in the bce partition by init_toehold). The program runs in absolute mode. It is listed here because it contains the flagbox and the all important dcw lists.

TTY_AREA

Terminal control blocks (tcb's) are allocated in tty_area. It is initialized to an area by fnp_init and managed by the various communication software.

TTY_BUF

The tty_buf segment contains, obviously enough, the tty buffers used for manipulating data communicated with the fnp. It contains various meters of characters processed, number of calls to various operations, echo-negotiation, etc., trace control information and timer information. Following this is the tty_trace data, if present, and the tty_buffer_block's, split into free blocks and blocks with various flags and characters in chains. The layout of this segment into empty areas is done by fnp_init.

TTY_TABLES

tty_tables is an area in which tables (conversion and the like) are allocated. It has the usual lock and lock event. It is initialized by fnp_init.

UNPAGED_PAGE_TABLES

All permanent non-per-process unpaged segments have their page tables in unpaged_page_tables. The page table for this segment is also within it. It is generated initially by template_slt_ and added to by the various segment creation routines. The header of unpaged_page_tables contains the absolute address extents of all hardcore segments that contain page tables; these are unpaged_page_tables, int_unpaged_page_tables and sst_seg. Dump analyzers look here to resolve absolute addresses from sdws into virtual addresses of page tables.

VTOC_BUFFER_SEG

vtoc buffers live in the vtoc_buffer_seg. The segment is allocated and initialized by init_vtoc_man. It starts with the usual global lock and wait event. Following this are various parameters of the amount and usage of the vtoc buffers, including information about the vtoc buffer hash table. Then comes the vtoc_man meters. Finally comes the hash table, the vtoc buffer descriptors (pvtx - vtocx info, etc.) and the vtoc buffers themselves.

WI_LINKAGE (WIRED INIT LINKAGE)

This initialization segment corresponds to area.linker for wired initialization segments. It is built by the MST loading routines.

WIRED_HARDCORE_DATA

Another collection of data for hardcore use, this segment is permanent. It contains the size of a page, the amount to wire for temp-wiring applications, the history register control flag, the trap_invalid_masked bit, a flag specifying the need for contiguous i/o buffers (if a non-paged iom exists), the debg card options, the fim fault_counters and the bce abort_request flag.

WS_LINKAGE (WIRED SUPERVISOR LINKAGE)

This wired hardcore segment, shared between processes, corresponds to area.linker for wired hardcore programs. It is built by the MST loading routines.

APPENDIX C

MEMORY LAYOUT

In the memory layout charts below, the starting absolute address and length for each data area is given (in octal). When a number appears in brackets ([]), this means that it is really a part of the segment listed above it.

The memory layout after the running of collection 0 (the loading of collection 1) follows.

start	length	contents
0	600	fault_vector
1200	2200	iom_mailbox
3400	3000	dn355_mailbox
10000	2000	bos_toehold
12000	10000	config_deck
24000	22000	bound_bootload_0
[24000]	[4000]	[(bootload Multics) toehold]
[24000]	[2000]	[flagbox (overlays the toehold)]
[30000]	[n]	[bootload_early_dump]
46000	4000	toehold_data
52000	2000	unpaged_page_tables
54000	2000	int_unpaged_page_tables
56000	2000	breakpoint_page
60000	6000	physical_record_buffer
66000	2000	dseg
70000	10000	name_table
100000	4000	slt
104000	2000	lot
106000	and up	wired segments
		fabricated segments
1777777	and down	all other segments

The absolute addresses of most of these segments is arbitrary. Hardware known data bases must be at their proper places, though; also, the toeholds are placed at addresses known to operators. Except for these exceptions, the segments may be moved. Their addresses are contained in bootload_equs.incl.alm.

All programs refering to this include file must be reassembled if these addresses are changed. Certain interdependencies exist that one must be aware of. First of all, the toehold is placed at a 0 mod 4 page address. physical_record_buffer must be the last of the fixed memory address segments. The length of all segments is an integral number of pages. The two unpaged page tables segments must be large enough to meet the demands on them; refer to announce_chwm. Also, the length given for bound_bootload_0 must hold the text thereof.

After collection 1 has finished, segments have been made paged and collection 1 temp segments have been deleted, the memory layout is as follows.

start	length	contents
0	600	fault_vector
1200	2200	iom_mailbox
3400	3000	dn355_mailbox
10000	2000	bos_toehold
12000	10000	config_deck
24000	4000	toehold (bootload Multics)
[24000]	[2000]	[flagbox (overlays the toehold)]
46000	4000	toehold_data
52000	2000	unpaged_page_tables
56000	2000	breakpoint_page
60000	and up	paging area
high mem		sst_seg

INDEX

C

E

K

L