To: Distribution

From: Benson I. Margulies

Date: 12/02/82

Subject: Bootload Multics Status and New Initialization PLM material.

1 ABSTRACT

The following is a status report for Bootload Multics, describing work done and design decisions made.

This is the first revision of MTB 598. It contains technical corrections to the description of collection zero, more details on collection 1, and much more discussion of command environment issues.

This MTB will be published in several revisions. Each revision will completely document the design work done to date in a format suitable for inclusion in a rewritten System Initialization SDN, and discuss the current open design questions.

---

2 WHERE WE ARE

As of MR10.1, the majority of Sibert's code has been integrated into the system, producing a hardcore system that is compatabable with the existing hardcore, runs on top of BOS, and includes the first set of console changes neccessary to run the IPC-CONS-2. The one major site-visible change has been to formally require 512K of contiguous low order memory. The reasons for this will be explained below.

Since the MR10.1 version of the code was stabilized, work has been done to complete and debug bootload without firmware. In collection zero, tape firmware loading has been debugged, the bootload_io package has been upgraded to work with multiple IOM's, and some questionable ORION support removed. In collection 1, facilities for the additional early initialization pass, including many command environment utilities, have been implemented.

This system will be sufficient to serve as a sort of initial go/no-go test of Dipper bootload. If this system can be booted with the Dipper OS bootload facility, and its primitive I/O package manages to read tape and print a message, then that much functionality has been verified. To make a releasable system with an IIOC as the bootload IOM requires the rest of the BOS replacement functionality to be designed and implemented.

3 GENERAL SUMMARY

Bootload Multics can be thought of as divided into the following parts:

* Collection 0
* Collection 1 Multiple Initialization
* Command Environment
* Crash Handler.

3.1 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. As such, they replace BOS FWLOAD and bootstrap1. Since they must be capable of failing gracefully on an otherwise bare machine, they have a substantially more complex error reporting environment than bootstrap1, which depends on BOS for such amenities.

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.

3.1.1 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 compatability mode, the BOS BOOT command simulates the IOM, leaving the same information. However, it also leaves a config deck and 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 a sentinel in the flagbox.

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 eibecause ther of because 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. If this is successful, it transfers to the base of bound_bootload_0. This is the beginning of bootload_abs_mode. This program copies the template descriptor segment assembled into template_slt_ to the appropriate location, loads the DSBR and the pointer registers, enters appending mode, and transfers to bootload_1.

3.1.2 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.

3.2 Module Descriptions

3.2.1 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 its position as the first module in the bound segment. This program runs in absolute mode, and makes heavy use of IC modifiers to achieve position independance. It first clears the memory used by the Collection 0 data segments, then copies the template descriptor segment 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_1$begin, the basic driver of collection zero initialization.

3.2.2 BOOTLOAD_1.ALM

Bootload_1's contract is to set up the I/O, fault, and console environments, 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_1 makes a series of tsx2 calls to set up each of these facilities in turn. It calls bootload_io$preinit to interpret the bootload program left in low memory by the IOM/IIOC/IOX; bootload_flagbox$preinit to check for the presence of BOS, and set flags appropriately; bootload_faults$init to fill in the fault vector; bootload_slt_manager$init_slt to copy the data from template_slt_ (a cds program) 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_2.

3.2.3 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 current correspondance between collection numbers and sys_info$initialization_state values, provisions have been made for sub-collections. 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 amongst 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 not 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.

3.2.4 BOOTLOAD_CONSOLE.ALM

Bootload_console uses bootload_io to do console I/O. Its initialization entry, init, finds the console on the bootload IOM 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 convienience. 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.

3.2.5 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.

3.2.6 BOOTLOAD_ERROR.ALM

Bootload_error is responsable for all the error messages in collection 0. It is similiar 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 return to BOS if there is a BOS to return to. 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.

3.2.7 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.

3.2.8 BOOTLOAD_FLAGBOX.ALM

Bootload_flagbox tests the flagbox sentinel to see if BOS is present, and sets the flag bootload_info$assume_config_deck appropriately. It also sets a flag in the flagbox that informs BOS that Multics has been booted, and that there is a core image worth saving and dumping.

3.2.9 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

3.2.10 BOOTLOAD_INFO.CDS

Bootload_info.cds, which produces both bootload_info and sys_boot_info, contains state and configuration information for the collection zero and later environment. It has provisions to contain site-supplied configuration information specifying the type and location of the bootload tape and disk MPC. If these values are provided, as it is expected they will be, no operator queries will be needed to bring the system up. Only cold site boots or disk problems will require operator intervention during boot. As of this writing, no interface has been defined to fill these values in. The most promising possibility is to specify this data in the MST header file, and have generate_mst set the values into the segment. The checker could then report their settings in the checker listing.

3.2.11 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 in abs mode, and its callers must fill in their DCW lists with absolute addressses. 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. If we could detect paged mode, we could set up paged I/O, and use rel mode otherwise. The net result would be that all DCW adresses could be relative to the base of bound_bootload_0. Lacking this feature, absolute addressing is used, ugly though it may be.

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.

3.2.12 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.

3.2.13 BOOTLOAD_LOADER.ALM

Bootload_loader is the collection zero loader. 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. 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.

3.2.14. 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.

3.2.15 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 (see below) 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 that is is to be used for. 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.

3.2.16 TEMPLATE_SLT_.CDS

This CDS program contains 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. Also present are the absolute addresses, lengths, and pointers to each of the collection 0 segments for use elsewhere in bound_bootload_0.

4 COLLECTION 1

The changes to collection 1 fall into three categories. The first group are changes to use sdw_util_ and ptw_util_ to construct SDW's and PTW's, for ORION compatability. The second are changes to reflect the different environment constructed by collection zero. For example, oc_data_init uses the bootload console information that is present in sys_boot_info. These include changes needed to support multiple initialization for disk firmware load, a topic addressed in greater detail below. Most of these either allow a collection 1 program to be run twice, or to provide a special "early" entrypoint for use in the early initialization environment. The third category consists of changes to support the Command Environment. These include moving facilities from collection two to collection one and adding the minimal command environment facilities.

4.1 Interesting individual modules

Bootload Command Environment modules are not included in this section.

4.1.1 BOOTLOAD_2.ALM

Bootload_2.alm replaces the former bootstrap2.alm. Its function is to fill 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. The prelinking function of bootstrap2 has been assumed by bootload_linker in collection 0.

4.1.2 INITIALIZER AND REAL_INITIALIZER.PL1

In pre-Bootload-Multics systems, the program initializer resided in bound_active_1, a permanent hardcore segment, even though none of its code was executed after initialization. The problem was that some program had to call init_proc after the init_segs were deleted. This has been fixed by splitting the program into two pieces. Initializer.pl1 now consists of

only calls to real_initializer, delete_segs$delete_segs_init, and init_proc. Real_initializer.pl1.pmac, in bound_init_1, contains the vast bulk of the code.

Real_initializer includes an upgraded 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 || "NNN"b4, where NNN is the error number in BCD, BOS is called.(1) When BOS is eliminated the call will result in a return to the Bootload Multics crash handler.

There are two paths through real_initializer: early initialization and service initialization. The early initialization path is used for native (no BOS) boots. It must initialize without a config deck in order to read the config deck from disk using paging I/O. Service initialization assumes that an accurate config deck is already in memory. Both paths set an any_other handler of init_error_handler.pl1 to catch unclaimed signals. More detail on this follows.

4.1.3 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.

4.1.4 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.

4.1.5 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:

---

(1) This is BCD as the rest of the world understands it, not GEBCD. On Multics it is equivalent to pl1 unsigned fixed decimal. Each four bits take on a value from 0 to 9.

```
    rpv lcc MPC_model RPV_model RPV_device
or cold lcc MPC_model RPV_model RPV_device
```

for example:

```
    rpv A20 601 451 7
```

would boot the current CISL configuration.

If the operator makes a mistake, or types help, she 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 base channel is sent a reset status command. 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.

4.1.6 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 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.

4.1.7 HC_LOAD_MPC.PL1

Hc_load_mpc embodies the protocol for loading all but unit record MPC's. It is an io_manager client. It would not be hard to change it to load UR and EUR MPC's, and to run on either io_manager or ioi. If that were done, it could replace the code in the user ring in the load_mpc command. For now, it is buried in the hardcore. 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.

4.1.8 INIT_CLOCKS.PL1

This program replaces the BOS time facility. If the bootload memory is an SCU, 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", and the operator is invited to type "s" to set this time, or "new" 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 replaces the TIME command by translating times to SC switch settings.

The time zone is a source of trouble here. If the program is run in early initialization, before the config is available, it would either have to demand a time zone from the operator on each boot, or use one stored in bootload_info. The later is clearly superior. However, the question then arises as to whether the time zone in the config deck should override that in the cds. If we choose the cds approach, it would be better to eliminate the zone on the config card, but allow the zone to be changed during operation (either bootload command level or via an hphcs_ gate). Otherwise the interaction of the two will prove too confusing.

If we choose to put the clock information in bootload_info rather than the config deck, we may want to identify all the other parameters that are currently in the config deck which it would be appropriate to set in the header file instead.

4.1.9 ESABLISH_CONFIG_DECK.PL1

The config deck is stored in the config partition on the RPV in between bootloads. In service, the segment config_deck is effectively an abs_seg onto the partition. This program establishes the segment, by filling in the aste and page table appropriately. When called in early initialization, (at the $early entrypoint), it does not bother to set up the aste, but just uses read_disk to read the contents into the wired segment. In the case of a cold boot, it writes the fabricated config deck into the partition. In BOS compatability, it writes the BOS config deck to the partition and then establishes the segment.

4.1.10 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. The command environment will presumably have its own any_other handler that uses less drastic means.

4.1.11 INIT_EMPTY_ROOT.PL1

This program has been changed to provide an improved cold boot mechanism that will be available, with any luck, in 10.2. The special format part cards that specified the RPV layout are no longer needed. Instead, fill_vol_extents_, the subroutine used by the user ring init_vol command, has been adapted. 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 she changes it, she is required to define at least the hardcore and (for the moment) bos partitions.

4.2 Early Initialization

The sequence of events in an early_initialization is given here. Init_early_config.pl1 constructs 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, and the bootload tape controller.

Scs_and_clock_init$early fills in the initial scs data from the config deck. It also fills the bootload CPU port number into the config deck, which is how it differs from scs_and_clock_init$normal.

Initialize_faults$fault_init_one resets the fault vector from the collection zero handlers to the collection one handlers, and sets up the fim and condition signalling.

Get_io_segs, iom_data_init, and scas_init are run as in a service initialization. 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.

Init_sst$early differs from init_sst$normal in that the sst and core map segments are allocated with the slt allocation pointers. This is done because the top of the 512K image is already occupied with init segs, so that the usual strategy of locating these segments at the top of the bootload controller would fail.

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

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 query her in detail for each item.

The BCE command processor, bce_command_processor_, is used to parse the "cold" and "rpv" request lines described above.

Firmware is booted in the RPV controller with boot_rpv_subsystem.pl1, 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.

Cold boot is also requested in the find_rpv_subsystem dialog. If a cold boot is requested and confirmed, init_cold_boot.pl1 (not yet written) will query the operator for the information needed to initialize the RPV. This data will be recorded in the config deck using the same cards that are currently prepared under BOS for a cold boot. This reduces the amount of work to be done to init_empty_root.

The RPV data is filled into the config deck, and initialization continues. disk_init, init_pvt, init_root_vols, read_disk$init, and init_partitions together have the net effect of setting up disk and page control. No segments are paged, though, except for rdisk_seg. In particular, make_segs_paged and collect_free_core are not run so that the memory image as of this point could be used as the crash handler. There is more discussion of this issue later on.

Once paging is available, and the RPV has been found valid (or cold-initialized), the real service config deck is read from the RPV config partition. In the case of a cold boot, the fabricated config deck is stored in the partition as an initial config.

A compatability issue arises here -- how does a site booting Bootload Multics without BOS for the first time get a deck into the partition? The solution is that service Multics will be storing the config deck in the partition as much as an entire release before the field is running without BOS. MR10.2 will, if all goes well, start storing config decks into the partition. The first release in which the Bootload Multics command enviromnent and crash handler are active should follow in MR11, with BOS eliminated sometime afterwards.

At this point, early initialization's work is done. The real config deck is available, 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 service initialization.

4.3 Service Initialization

Service initialization is much like early initialization. So much so, that save for addition of move_non_perm_wired_segs, there is no need to describe it in detail in this revision. When this MTB is fleshed out into the full Initialization PLM it can be added.

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. It may yet prove possible to cut this down, depending on how much memory it takes to set up the crash handler. However, 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 neccessary 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.(2)
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. The net result is the same memory layout that the existing collection 1 initialization produces.

The program depends on the knowledge that the loader assigns segment numbers in monotonically increasing order to sup (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.

4.4 Facilities moved to collection 1

The are two reasons for moving facilities into collection one. First, to provide a reasonable user interface for any queries in collection one, formatting and similiar facilities are needed. Second, there are good reasons (discussed below) for trying to fit the entire command environment into the wired environment. In fact, the facilities provided by the command environment are exactly those needed to communicate with the operator, so the installation of the command environment in collection one fulfills both needs.

The command environment needs two classes of facilities previously left for collection two. First, real fault handling, condition signalling, and non-local gotos are required. The alternative would be far worse in terms of net complexity, as various facilities tried to simulate these features. Thus signal_, condition_, and a few other programs are moved to bound_sss_wired_ (in the part of the segment that eventually gets made paged).

Second, a variety of utilities usually though of as a part of the user ring environment are needed. These range from cu_$generate_call to cv_dec_check_ to define_area_. Since many of there programs reference the error table, error_table_ was moved as well.

5 THE BOOTLOAD COMMAND ENVIRONMENT

To replace BOS, Bootload Multics must provide a certain number of facilities when the storage system is not available. Examples are fdumps, disk saves and restores, interactive hardcore debug (patch and dump), and automatic crash recovery. The mechanism described in the original Bootload Multics MTB for this was a command environment running in a paged environment.

---

(2) Why? 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 we de-commit non-paged-mode IOM support, we can dispense with this whole business.

5.1 Initialization

There are two facets to a paged command environment. One is the use of paging I/O to reference data on disk, such as firmware images, config deck files, crashed memory images and the like. It is desirable to use paging I/O to avoid the need to support another disk I/O subsystem. The second facet is the use of demand paging for the command environment programs themselves. This is desirable for the usual reason; more data can be used at one time. On the other hand, it complicates initialization of the environment considerably.

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 maintainance and the like. When the service system crashes, the command environment becomes the crash recovery environment that, like BOS today, oversees dumping and automatic restart. A full cold boot is a special case of a cool boot.

There are two ways that a collection of paged programs can be loaded. They can be loaded wired into collection 1, and then made paged, or they can be loaded into a paged environment using the collection 1 loader, segment_loader. If the command environment needed only to be entered as part of bootload, then the choice would not make much difference. There would be plenty of memory (as much as contiguously addressable) for wired segments, and they could be read from the MST.

Crash recovery is another problem. Any program to run in crash recovery conditions must either be part of the perm-wired supervisor, or be read in off of disk. Since the amount of memory saved on disk and therefore available for the crash handler is limited (and currently assumed to be 512K), the command environment would have to fit in that core image. As of this writing, most of the basic facilities of the command environment have been written, using 261 out of 512K. The majority if the remainder will be available for the commands themselves. Thus it is now certain that the basic environment can be loaded wired, and likely that the entire environment can be so loaded, as well.

Once the core image is loaded, it will make itself paged. When booting, it can use the hardcore partition for paging. When entered at crash time, though, it must leave the hardcore partition undisturbed, since much of the state of the crashed system is there. It will use a seperate paging partition instead.

If some of the command environment cannot be loaded wired, then it will have to be loaded paged. This is how collection 2 is loaded today. At boot time this is easy, since it will be on the MST. At crash time the part of the environment that did not fit into the wired image would have to be loaded from disk one segment at a time, using an as yet undefined format for storing an MST in a disk partition.

5.2 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, so that allocate statements with no "in" qualifiers work.

* standard input and output 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_data$put_chars and bce_data$get_line. 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_.
* ioa_ support. ioa_ is moved to collection 1. In addition to the rs entrypoints, which would work anyplace, ioa_ and ioa_$nnl provide formatted output using the bce_data$put_chars output entry variable.
* bce_query and bce_query$yes_no. Each takes a response argument, ioa_ control string, and arguments, and askes the question on the console.
* bce_error is the local surrogate for com_err_. It does not signal any conditions in its current implementation.
* a command processor. The command processor provides an ssu_-like primitive subsystem facility. Individual facilities can define tables of the requests they accept, and the entrypoints that implement them. The command processor calls them with a standard calling sequence.

bce_command_processor_ had the following interface:

```
declare bce_command_processor_ entry (char (*),
                                      entry (char (*), fixed bin (35))
                                      returns (entry),
                                      pointer, fixed bin (35);

call bce_command_processor_ (Line, Command_finder, SS_info_ptr, Code);
```

Where:

Line is the command line to process.

Command_finder is the entry that locates the command. It is called with the name of the command, the SS_info_ptr, and a return code, and is expected to return the entry to call to execute the command.

SS_info_ptr is a pointer to a control info structure. This structure is passed to the procedures that implement the requests, and records the current state of the subsystem environment. This structure includes a pointer to an ssu_ standard request table and other data. The command itself is called with this structure as an argument, and is expected to use the arg_list_ptr in the structure to find its arguments. Trivial subsystems omit the request table, and supply an ad-hoc Command_finder.

Code is an error table code. If the Command_finder returns a code it is returned here.

```
/* Begin include file bce_subsystem_info_.incl.pl1 BIM 11/82 */
/* format: style3 */
declare   ss_info_ptr          pointer;
declare   1 ss_info            aligned based (ss_info_ptr),
            2 request_table_ptr
                               pointer,
            2 abort_label      label,
            2 ioa              entry options (variable),
            2 ioa_nnl          entry options (variable),
            2 query            entry options (variable),
            2 error            entry options (variable),
            2 name             char (32) unaligned,
            2 arg_list_ptr     pointer,
            2 info_ptr         pointer, /* per subsystem */
            2 flags            aligned,
              3 forbid_semicolons bit (1) unaligned;
/* End include file bce_subsystem_info_ */
```

Any program that wants to parse lines using standard syntax (without quotes, parens, or active functions, for now) calls this 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.

* Listener and Find Command. bce_listener_ and bce_find_command_ together with bce_command_processor_ make up the command loop at top level and inside any subsystems. The ss_info_ptr for the top level is kept in bce_data$root_ss_info_ptr, thus allowing the ".." request to be supported. The listener reads lines from the console, and calls the command processor using the find command procedure. The request table bce_top_level_requests_ is used. The command processor then calls the commands with the structure above as a parameter. The arg_list_ptr in the structure is set to an arg_list consisting of all the command line arguments.

No discussion of individual commands is undertaken. See the original Bootload Multics MTB.

6 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. It would also be desirable for it to be able to handle one extra level of crash; if the command environment crashes it would be very useful to be able to get an fdump of that.

6.1 How BOS does it

When a crashing Multics returns BOS, the toehold does the following:

* Save the machine conditions.
* Save the low 64K of memory in the BOS partition
* Loads BOS into memory
* Initializes BOS.

Since BOS is not a paged environment, initializating BOS is the same whether you are crashing or using the BOS WARM loader command. No previous state other than the contents of the config deck need be retained.

6.2 How Bootload Multics should do it

The Bootload Multics crash handler must, one way or another, set up the command environment, which is paged. There are two ways to get a paged environment. One is to load up a wired memory environment and then to let it make itself paged.

The other is to load a paged environment. This is inordinately difficult: A paged environment, to be any use, must have some pages that are not in memory. If all the pages could be fit into the memory image, then it could be loaded unpaged. Those pages out of memory are recorded as devadds. They cannot be devadds in the service hardcore partitions and paging pool, because these must remain inviolate (except for patching). Arranging for them to be anyplace else would require the system to run make_segs_paged multiple times, once against the service hardcore partitions, and once against the partition(s) to be used in the crash recovery environment. Running make_segs_paged twice, though, presupposes a program called make_segs_unpaged. This process would bear a distinct resemblance to putting the toothpaste back into the tube. Once the system is running paged, control would have to be transferred to a perm-wired program that would turn paging off. But the wired environment packs segments to mod 16 boundaries to save space, so that it would not be possible to just set the wired bits in all the PTW's and touch all the pages. Instead, collection 1 would have to be reloaded, and loading such a thing would require rerunning the impure collection zero loader.

Worse even than an extra load of collection 1 at initialization, saving the crash handler as a paged environment requires one paging partition per level of crash. If a crash of the crash handler is to be handled at all, a pristine paging partition containing the original result of make_segs_paged must be available. In short, a mess.

The alternative is to let the crash toehold load a 512K (or some other size) unpaged memory image that was snapshot to disk just before make_segs_paged was run. It can then run make_segs_paged against the crash handler paging partition. If it is a crash of the crash handler, it can save the old contents of the crash paging partition before it does so, in yet another disk partition. This still requires a partition per level of crash, but the partition need not contain anything special.

This has some other advantages. Given the additional effort of writing a program that runs is a user process and prepares a collection 1 memory image, boot-from-disk becomes a possibility. At very least, the current MST could be replaced by a wired image and one or

more collections of programs loaded paged.(3) If, in fact, Dipper relieves us of all firmware loading responsabilities, this would be a practical course of action.

6.2.1 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 will include a crash handler that can write a core image to tape, prompting the operator for the drive number. In the extreme case that T&D and the colsole output are not enough to resolve the hardware failure, the tape could be sent to Phoenix (or Paris) for analysis. Once a site has successfully booted a system without BOS, we assume that they will always have a functional hardcore tape around thatthey can boot and examine the dump. The existing tool for converting BOS CODE SAVES to fdumps will be adapted to work against these.

6.2.2 THE TOEHOLD

The toehold, toehold.alm, is an inpure, unpaged, wired, privileged program that resides in a known location in absolute memory (10000o). 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. To store a complete machine state, it is neccessary to use scpr (store central processor register) instructions, which are difficult to use in absolute mode. They interpret their tags not as modifiers, but as indications of which register to store. In absolute mode, then, they must be assembled with the actual absolute address they are to store into. This make debugging the toehold difficult, since in early debugging it is desirable to load the toehold someplace other than 10000o, so that if it fails the BOS toehold at 10000o can still be used to get back to BOS. To avoid this and other problems, toehold enters appending mode on a trivial set of descriptors the describe it and the low 256K of memory.

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 price paid for this strategy is that the toehold cannot be activated until after enough of early initialization has run to read the config deck, since the config is needed. Crashes before that time can print copiuous messages (assuming the

---

(3) To "load paged." in this context is to read successive segments out of the partition into paged segments. This is quite different from trying to prepare a partition containing the on-disk images of a set of paged segments. This process is analogous to that used by segment_loader to load collection 2. The partition would be set up under service, not during initialization.

console is functional) and make core tapes, as descrived above, but the crash handler cannot be entered.

6.2.3 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.

7 TIME ESTIMATES FOR THE REMAINING WORK

The material that follows assumes a target of FCS of BOS replacement functionality (with or without malingering BOS coexistance) in MR11. Having learned our lesson from the integration of Sibert's code, any work done between the 10.1 release and the real release should be integrated as soon as possible. This requires synchronization of the Bootload Multics schedule with the 10.2 and perhaps 11.0 schedules.

7.1 Collection Zero

Collection zero is essentially complete.

7.2 Command Environment

* Tuning of utility interfaces. Two weeks, scattered amongst other work.
* The command loop itself, including command table. One month.
* "exec_coms". Two weeks.
* Tuning and integrating the existing bootload file system code. One week.
* The COMMANDS THEMSELVES (all the BOS functions worth replacing). This estimate is pure SWAG as yet, but conservative. 6 months. Important highlights: fdump replacement: one month. tape save/restore compatable with volume dumper, potentially including work in the online reloader: one month. Everything else pales in comparison to these two.

7.3 Crash Handling

Design and construction of the toehold swapping mechanism. one month.

7.4 Interface from the crash to the command environment

An as yet vaguely designed modification to page control to allow references to segments in the dump some of whose pages were part of the 512K swap. two weeks.

7.5 Online Facilities

There is work to be done to the dump analyzers and the MST generation tools. There is a month here.

7.6 Documentation

There are various kinds of documentation -- MOH successor documentation of the operator interface: three weeks for the developers plus however much time for writers and terminal operators; PLM/SDN documentation of all the facilities so that the code stays maintainable: three weeks.

7.7 Grand Total

These estimates total 11.25 months of effort. They are relatively conservative. That is, they are not off by more that a factor of two. Several of these tasks can be done in parallel. Thus a proper PERT chart would reduce the real time to produce this.

7.8 A final note

Up until now, this project has been run on a 'design and document as we go' basis. At such point as we commit to some delivery schedule, it would behove us to spend some time firming up the list of tasks and the work involved in each. Continuing design work will make that task easier.