Classification: 000/0
Date: 5/1/78
Doct=1 Vers=2

Lecture 1

The first section covers pieces and the relationships between system components, the basic interface, and some common tables. This section will not cover the internal details of any one interface or any one piece; they will be covered later. Thus, for example, some of the SVCs in the task-to-supervisor interface will be mentioned, but not necessarily all of them.

Looking at the system as a whole, the first question is "who's in control?" Usually the answer is, "The supervisor." But the supervisor, doesn't seem to have any entry point--it never begins and ends. "How does the supervisor get entered?" An I/O interrupt or a program interrupt occurs. Who causes the interrupts? A task that's running. Who starts a task? It's the supervisor. It's the chicken-egg problem. Essentially, the whole system is interrupt-driven. Someone initiates things, usually the operator pressing the request key on the console typewriter.

This section is an overview of what could be called the Steady State System. In other words, it is assumed that the system is there, loaded and running. How the system is built, how it is loaded from disk once it has been built, and things like that will be covered later.

Figure 1 is the picture around which discussion in this section will revolve. Most of the items here are covered in detail in later sections. For now they will be treated as black boxes and only the connections discussed. The connecting lines are the interfaces, and the numbers on each line are solely to identify the interface for discussion.

At the bottom of this picture is the hardware machine. Above this is a box called supervisor. Note that this picture is carefully stratified in a number of manners. At this point, note the boundary indicated at the left between supervisor state at the bottom and, above it, problem state. Problem state and supervisor state refer to the hardware definition of the supervisor state and problem state for the 360 and 370.

This means that the supervisor, or anything else below that line, runs in supervisor state and nobody else does. That, essentially, is the definition of the supervisor, although one usually considers the supervisor to be a particular assembly listing. There are, however, other things below the line. There's a series of what will be called supervisor subroutines which the supervisor causes a task to call. For example, there's one called JBRP, which stands for Job Request Processor, called in the process of task initiation. The interface (2) between the supervisor and the supervisor subroutines is that the supervisor causes the task to start there before going about its business by virtue of setting the task's PSW to the entry point of the subroutine before dispatching the task. There's another entity which lives below the line which is hard to classify exactly where it belongs in the system. This is the machine check recovery code which as its name suggests, gets control when a machine check occurs.

The interface labelled "1" between the supervisor and hardware is well defined by the Principles of Operations manual and it means that the supervisor owns the PSWs for the old and new various interrupt states. It gets entered whenever an I/O interrupt or external interrupt occurs, etc.

Above the supervisor in this diagram are the tasks in the system. In order to describe the interface and say a few things about the historical wording or terminology that occurs, a simple task will be discussed first.

The task used will be the REWIND task, and the interface between it and the supervisor (labelled "3") is the one being discussed. [At this point a slight digression is necessary. Back at the time this all started (1966), we obtained from Lincoln Labs a small supervisor called LLMPS which managed jobs of this variety. The terminology they picked used "job program" to represent the code in the machine, and "job" to represent an activation of that, and there may be several activations of that if it's a re-entrant job program. Since then, computer terminology has evolved so that normally "task" is used for "job". But for the purposes of this manual, jobs and tasks are used interchangeably, for terminology.]

On this interface (3) there are three areas to cover. One is getting started, in other words, how does it start a task? Another is how it obtains services while it's running, and third is how it terminates.

This discussion is applicable to all tasks in the system, although the example is relatively simple. MTS has a job program. It's started many times and it has the same interface, although the job program is larger than most. The main interrupt that starts everything is the request button on the operator's console typewriter, pressed because the operator wants to start something. Every line entered by the operator is a request to start a task. There is no command language at the supervisor level--it only starts jobs.[1] The first thing that the operator types in is the name of the job he's starting. As a slight digression, one might ask how do you stop a task, once it's started? There's a job called STOP. If the operator wants to stop a job he presses the REQUEST button and enters "stop" and a

[1] That's actually not quite the truth. Lines beginning with $ are passed to HASP as commands, and lines beginning DIS,MOD,SE, or TRCTP are swallowed by the supervisor as actual supervisor commands. But everything else starts a job.

parameter designating what is to be stopped. This starts the STOP job whose purpose is to stop another job. Hopefully, it gracefully stops by itself to eliminate cascading problem?

To get back to starting a job, the first thing the supervisor does is allocate a job table entry for this invocation. This is a fixed-length area where all variable information pertaining to a job (or else a pointer to same) is kept. For example, the job's registers are stored here when it is not executing. At the front of the job table are stored task number and the 8-character task name (MTS, HASPLING,...). [A task number of zero means this job table entry is not in use.]

Two items relating to the initiation of jobs must now be discussed. The Job Header is information attached to the front of the job program. The Job List Entry is essentially a "symbol-table" entry to the list of job programs in the system specifying the name of the job program and where its code is to be found.

Each job program is prefixed by a job header. The job header specifies the location in the job program of the first instruction to be executed, the number of preallocated devices and buffers which the job requires, the device type required for each device, and the size of each required buffer. The format of the job header is:

```
┌──────────────────────────────────┐
│                                  │
│Location of First Job Instruction│
│                                  │
├─────────────────┬────────────────┤
│                 │                │
│     NJBDVU      │     NJBBFU     │
│                 │                │
├─────────────────┴────────────────┤
│                                  │
│Names of required Devices         │
│            (4 Bytes)             │
│               ...                │
│               ...                │
│               ...                │
│                                  │
├──────────────────────────────────┤
│                                  │
│Sizes of Required Buffers         │
│            (4 bytes)             │
│               ...                │
│               ...                │
│               ...                │
└──────────────────────────────────┘
```

NJBDVU	=	Number of Devices Used
NJBBFU	=	Number of Buffers Used

An illustrative 360 coding sequence of a job using two devices and three buffers is shown below.

```
JOB    START   0
       DC      A(BEGADR)
       DC      H'2'
       DC      H'3'
       DC      CL4'PTR'
       DC      CL4'7TP'
       DC      F'128'
       DC      F'2048'
       DC      F'2048'
       .
       .
       .
BEGADR DS      0H
```

The number of required devices are specified in the field NJBDVU and the number of required buffers are specified in the field NJBBFU. The device types for each required device must be given in the full words following the word specifying the number of devices. The size of each required buffer must be given in the full words following the device types. Device types, specifying the device requirement, are four characters, left justified, with trailing blanks.

The order in which the device names are specified determines a logical device number (LDN) for each device, where the first device specified is logical device one. When a job program issues a supervisor call, the device to which the call is associated is indicated by the logical device number. In this way a job program can be written independently of the physical address of a device.

This preallocation of devices and storage is used only by small jobs (such as REW). The MTS job-program obtains its devices and storage dynamically. The MTS job header specifies no preallocation. Thus, all items entered by the operator after "MTS" are considered parameters and are stashed away for MTS to look at when it’s given control.

Entry to a Job Program

When a job program is successfully initiated from the console typewriter, control is passed to the first instruction as specified in the Job Header. Three locations in the Job Table associated with the job are placed in General Registers 0, 1, and 2: General Register 0 contains the address of the pseudo Sense Switches, General Register 1 contains a pointer to the list of buffer addresses, and General Register 2 contains a pointer to the list of input parameters. If an input parameter is alphanumeric, it is right justified with leading blanks. If an input parameter is a decimal integer, it is converted to a four byte signed binary number. The list of input parameters is terminated with a fence of FFFFFFFF. If there are no input parameters, the first word of the parameter list will be the fence.

The Job List

For each Job which is to be run under the supervisor, there must be an entry in the Job List. The job List consists of a collection of fixed-length (16 byte) Job List Entries. The Job List Entry indicates whether the job is re-entrant and whether it runs relocatable, and gives the location of the job. Each Job List Entry is assembled as a separate subprogram, and contains the entry name of the job program as an Extern in the Job List Entry subprogram. The format of a job list entry is:

```
┌───────────────────────────────────────┐
│          8 CHARACTER                  │
│            JOB NAME                   │
├────────┬─────────┬────────────────────┤
│VERS NO │ BITS    │ JOB NUMBER         │
├────────┴─────────┴────────────────────┤
│             JOB  PROGRAM ADDRESS      │
└───────────────────────────────────────┘

BIT 0 INDICATES THE JOB IS RE-ENTRANT
BIT 1 INDICATES THE JOB IS RELOCATABLE
```

The job list entry for the REW job is:

```
ARM5  START 0
      EXTRN JBREW                 ENTRY
      DC    CL8'     REW'         JOB NAME
      DC    C'1'                  VERSION
      DC    X'80'                 REENTRANT
      DC    H'0'                  NUMBER
      DC    A(JBREW)              ENTRY ADDRESS
      END
```

The byte labeled Version can be used to indicate that a modification has been made to the job program. In the REW coding above, Version 1 is indicated. The "Bits" byte specifies whether the job is re-entrant and/or relocatable. Job programs may be written as re-entrant, whereby a single copy of a job program can be active for more than one task. If a job is re-entrant, the left-most bit of the "Bits" byte is set to 1. If the job-program's activation is to run in relocate mode, the second left-most bit is set to 1. MTS is an example of a job-program which is both reentrant and runs relocatable.

Associated with every active job is a task number which is used to identify the particular activation of the job throughout the system. If a job is not re-entrant, it can be active for only a single task. To indicate that a non re-entrant job is active, the task number is inserted in a field of the job list entry.

The Job List Entries are all collected together and sandwiched between a first-job (JOBLST), which defines the beginning of the "table", and a dummy last job (LSTJOB). The dummy last job has a blank name and version, and an all-ones job program address:

```
LSTJOB  START 0      LAST ENTRY IN JOB LIST
        DC    CL9' '
        DC    7X'FF'
        END
```

One thing should be mentioned about the parameter scan at this point. There are 14 words in the job table for parameters. The supervisor scans the input line from the operator; it doesn't just put four characters into a word in the job table; it actually scans for blanks as delimiters. If it finds (between the blanks) any characters that are non-numeric it assumes it’s a character string and it takes the last four characters and puts it in the word (right-justified with leading blanks). If it finds something that is all numeric, assumes it’s a decimal number and it converts it, and puts it into the next word of parameters. If there are long names (such as *INIT) to feed to the program that’s receiving these, such as MTS, they can’t be entered directly. If five characters are typed in a row, the last four characters, ("INIT") are stuffed into the parameter. The characters must be separated:

```
*INI   T,,,
```

The supervisor will put the characters into two contiguous words in the parameters. Trailing commas are used since MTS treats these as FDname delimiters. Trailing commas on the "T" are needed because otherwise the supervisor, (bless his heart), would right-justify it with leading blanks. [Another anomaly, device names in the system are left-justified with trailing blanks. Device types are right-justified with leading blanks.] This splitting is rarely used because it’s such an annoyance that most of the pertinent file names are four characters, like *RST. [The string that HASP issues to start up an MTS batch job is rather astonishing.]

When the job is started the base register is established, and it suddenly finds itself at the front of its code. Three registers are set up: GR0 points to switches in the job table, GR1 points to the series of words which keep track of the storage buffers requested, and GR2 points to the first word of the parameters. Requested devices are referred to as logical devices 1, 2, and 3, for example. It’s strictly in order of which they were specified and hence, the order of the parameters.

A job gets services -- that’s the line marked "3" -- from the supervisor, by issuing SVC instructions, which cause an interrupt in the supervisor. That’s the only way to get to the supervisor. The supervisor then processes the request and restarts the task. Anything that the task wants the supervisor to do is done by means of an SVC. There are about 100 SVCs now. The original Lincoln Lab supervisor has 20. A couple have since disappeared, and things have grown.

A job terminates by using an SVC. There’s an SVC EXIT which says "I’m done." The supervisor calls a subroutine to clean up things, release things, and so forth. There’s another SVC to intercept job stoppages (which includes SVC EXIT). MTS uses this SVC for maintaining control of things. Therefore, issuing an SVC EXIT does not always mean the job is stopping. For example, if the subroutine SYSTEM is called, the the first thing it does is issue an SVC EXIT, because the code to save all the registers, change state, and everything else is rather complex. Thus, there is only one copy of the code, and the first thing that happens on entry to SYSTEM is an SVC EXIT. The next thing MTS knows is that it is entered through the intercepted-exit section of code, and it finally discovers that someone did an SVC EXIT with a particular address. Therefore, it calls SYSTEM.

HASP communicates in approximately the same way. HASP is initiated by the operator typing HASP and giving as parameters, possible drive names for disk packs, which theoretically should have disk packs mounted on those drives. When HASP starts off, it actually does a little more than the standard "3" interface and issues an SVC to tell the supervisor that HASP is running. This SVC gives the location of some special words in HASP since the supervisor sometimes has to make a special entry to HASP. If the operator types in a line beginning with a dollar sign, it's considered a command to HASP and the supervisor just passes it on by chaining it to a chain of messages for HASP to process, setting the appropriate flag bytes and posting HASP. If HASP is well behaved, it will look at the messages. So there's a slight additional interface here. That will be discussed more in the sections about HASP and HASPLING.

It was decided at the time HASP was being installed, that we would create a little entity called a HASPLING. HASP is not re-entrant. It multiprograms within itself, but there's only one HASP job running. It has lots of code that represent the "job programs" and it has something akin to a job table, which are called processor control elements. The HASPLING is a job program which is re-entrant, and one is activated for every device that HASP has doing things for it; i.e., one for each reader, printer, punch, remote SDA line, etc. There's also one for each HASP disk and one to handle messages to the operator's console (from HASP to the operator's console). The interface between the HASPLING and the supervisor is the standard one (3), and consists mainly of an SVC to start an I/O operation and a SVC to wait until the I/O operation is complete. HASP is the one that starts the HASPLING, by issuing an SVC which starts a task. The communication between the two (interface "4") is that HASPLING gets passed as parameters in the job request, the name of the device to manage and the location in HASP of some control information, a lock byte, and some pointers. This lock byte and buffer pointers is how the HASPLING gets its information of when it's supposed to write things out, or read things in. And that's also how HASP tells the HASPLING to go away, if it's through with it.

When the HASPLING has nothing to so, it does a variant of the SVC WAYT type of wait, SVC SLEEP. Both types wait for some bits to change to 0, and a return from that SVC does not occur until all the bits specified are 0. But for the SVC WAYT, the job has to stay on the CPU queue, and every time the supervisor goes to dispatch anybody, it checks those bits to see if they have changed, which is expensive. So, the SLEEP and AWAKE mechanism was generated. The SVC SLEEP says "we're doing a WAYT type of suspension, but take me off the CPU queue because someone will do an explicit type of interrupt to get me started again". When HASP wants to initiate something on a HASPLING it takes the task number and it does an SVC AWAKE which tells the supervisor to put that task back into the CPU queue. It also zeros the WAYT byte, of course, before it can go on.

The PDP (Paging Drum/disk Processor) interfaces with the supervisor (interface 10). It runs in absolute mode; it can't page itself. It runs in problem state as an absolute task, so it behaves like any other task, except that for efficiency there are some special things done in the supervisor. It uses a lot of standard SVCs, but there are some added exclusively for interfacing with the paging drum processor. It has a number of SVC's only it uses because it has to get information about where the queues are, which the supervisor is keeping. It's not re-reentrant.

There's also the JOBS program (now called SSRTN) which is really an external scheduler to the system. HASP and MTS get information from it, but don't pass information to it (interface 6). There is a region in storage with bits and numbers which HASP looks at to decide to start a new batch job, and MTS looks at for limited service state determination. The Jobs program performs the external scheduling (i.e., when should a task be started), and the supervisor does the internal scheduling.

The right hand side of Figure 1 shows the separation between absolute and relocatable. An interface between absolute and relocatable is necessary, and complex. This interface is supervisor assisted, in the sense that there's a series of SVC's to perform the moving across that boundary. This interface is less used now that HASP and the HASPLINGs are relocatable.

Proceeding further in Figure 1, on top of MTS, we have the collection of the device support routines. These do all the I/O to and from the MTS tasks, particularly terminal support, tape support, etc. That's the interface labeled "7". There's also another set of interfaces, the Command language Subsystem interface (12). One of these interfaces is the user program. A CLS is just a program but each CLS can be run independently of the others. For example, there's no distinction between the editor and a user program. The editor is written as a program, and it runs as a program but independently from the user execution program. A special case is one CLS, namely SDS, which has hooks into the user program CLS, since it has to monitor what is going on. There are a number of real CLS's plus two more. Level 0 CLS is MTS itself, the command mode, 1 is the user program, and 2 on up are the actual CLSs. (Editor, SDS,...) For symmetry's sake, it was made all the same.

Another interface is with the file routines. MTS calls the file DSR which communicates with the file routines (interface 8). They are designed so they could be called by an absolute task, although that's not done yet. MTS also calls some of the file routines directly (interface 9).

The loader is also called from MTS (interface 11), although it's also called when running the system from scratch and there isn't anything around but the boot-strap loader to load the loader. Then the loader loads everyone else. The loader interface is such that anybody can call it since it is entirely self-contained. The loader looks at what it's given, decides if it's a good record and shoves it into storage.

For user programs, there are some SVC's that the user program will issue by means of a macro. These are the time of day, etc. But generally this type of interface is not used very much. The majority of supervisor services are obtained through MTS. (The DSR's however call the supervisor.)

This represents the minimum overall view of the system. Eleven subcomponents and several other things are included.

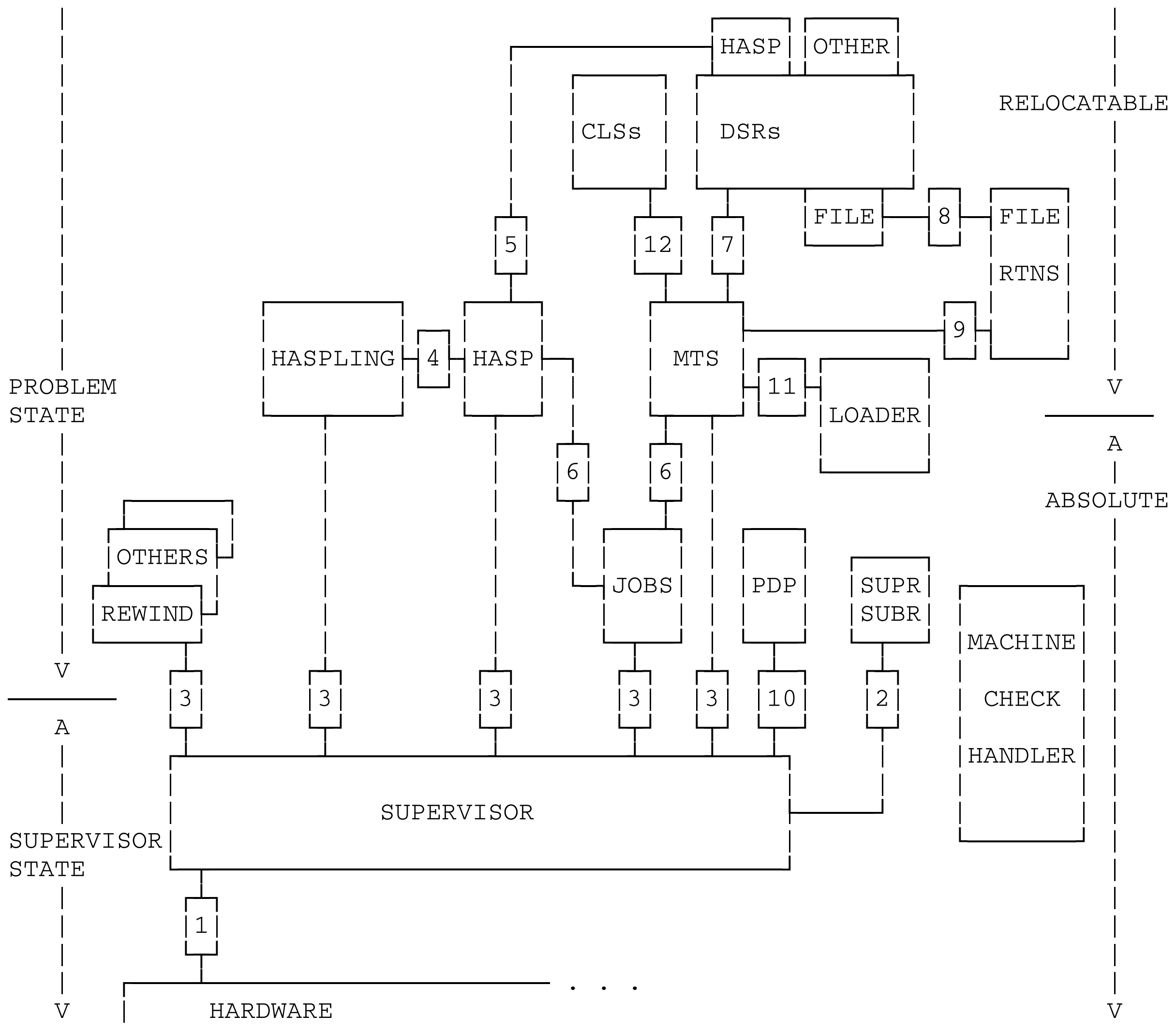
HASP
OTHER
CLSs
DSRs
RELOCATABLE
FILE
8
FILE
RTNS
5
12
7
HASPLING
4
HASP
MTS
9
11
LOADER
PROBLEM STATE
V
A
ABSOLUTE
6
6
OTHERS
REWIND
JOBS
PDP
SUPR SUBR
MACHINE CHECK HANDLER
V
A
3
3
3
3
3
10
2
SUPERVISOR
SUPERVISOR STATE
1
HARDWARE
. . .
V
V

FIGURE 1