ZZ77-9058

TECHNICAL INFORMATION EXCHANGE

IBM®

January 15, 1969

AN INTRODUCTION TO OS/360 MVT CONTROL LOGIC AND DEBUGGING WITH MVT CORE DUMPS

Mr. David G. Norris
IBM Corporation
1720 Zollinger Road
Columbus, Ohio 43221

This paper presents, in a logical, step-by-step manner, an approach to debugging an OS/360 MVT program using a core dump. Techniques for both programming and data management errors are presented. A discussion of MVT control program logic and the use of control blocks is also included.

IBM Corporation, Technical Publications Dept., 112 E. Post Road, White Plains, N.Y. 10601

ZZ77-9058

CONTENTS

PREFACE

OS/360 is an awesome creation. In the process of designing, generating and installing an MVT system, one vital area of the effort is often overlooked - in depth debugging by the application programmer. Then one day this programmer timidly enters the systems engineer's office carrying a card deck and a three inch listing of an ABEND dump. His eyes declare an obvious state of panic and desperation. "What is this? - What does it mean? - What do I do?" he implores.

Suddenly the realization sinks in that the poor devil has no idea what actually goes on in the guts of OS/360. He knows the "concepts" learned from OS/360 class, but is totally bewildered when faced with a dump and required to find the error.

The present literature is, in the author's opinion, inadequate - it merely lists the contents of a dump and deals superficially with its interpretation. This paper is an attempt to fill this (in the author's opinion) deficiency.

The author bases his paper upon three axioms:

1) A good applications programmer must have a good, if superficial, knowledge of the internal workings of OS/360;

2) An applications programmer feels extremely uneasy with an ABEND dump, and is interested in but a very few of the fields and control blocks displayed;

3) What is needed by applications programmers is not only an understanding of the contents of a dump, but far more importantly a method of attack, an approach, to debugging with a core dump.

This manual, then, is based upon these three axioms, and is divided into three general corresponding areas:

1) An overview of the physical and logical system;

2) System control and the use of control blocks;

3) Debugging with core dumps.

This paper arises from the author's experience and notes used in teaching a debugging class to applications programmers and consultants at the Ohio State University. Thanks is given to that legion of OSU students whose ABEND dumps became the object of this writer's efforts, and appear in some form in this paper. No attempt is made herein at exhaustiveness or rigor. The object is to get the programmer's feet on the ground. Skill comes only with necessity's practice.

AN OUTLINE OF THE MVT SYSTEM

GENERAL:

An outline of the Operating System falls easily into two major categories: The physical system, and the logical use of this system.

In the broadest sense, the physical Operating System consists of a number of system data sets and the devices upon which they reside. In main storage the system is physically divided into discrete regions of core which are used for various specific reasons. When we study the use of these regions and data sets, we are concerned with the logical system.

THE PHYSICAL MVT SYSTEM:

When considering the physical Operating System two areas should be discussed: First the system data sets and devices; Secondly core storage. We must note, however, that as in figure 1 , communication exists between all these areas.

System Data Sets and Devices:

Generally we refer to the devices upon which the system data sets reside (i. e. direct access devices) as "SYSRES" or systems residence.

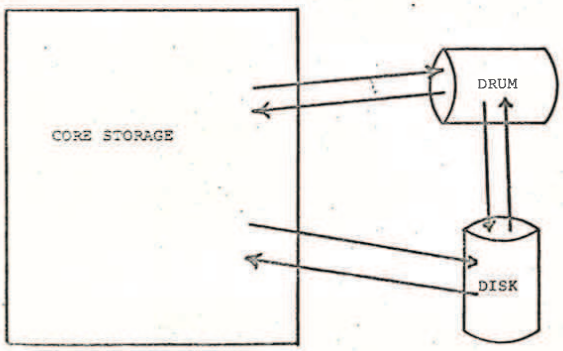

Figure 1: The overview of the logical OS/360.

Every MVT system consists of a number of required system data sets. The more important of these are discussed below.

SYS1.NUCLEUS: The NUCLEI of the system are kept in this library. This data set is referenced only at IPL (initial program load) time when the desired nucleus is loaded into core storage by NIP (NUCLEUS initialization program). The NUCLEUS is discussed in the next section.

SYSCATLG: Data set and index pointers are kept in the system catalogue, called SYSCATLG. Using the catalogue, references to existing data sets may be made by data set name alone. For a discussion of cataloguing, see OS/360 Concepts and Facilities, and OS/360 Supervisor and Data Management Services SRL's.

SYS1. LINKLIB: This is the main system program library; it contains the language processors, such as FORTRAN and the assembler, the job scheduling modules, task management routines and other system programs and utilities.

SYS1.SVCLIB: Transient supervisor call routines (SVC's) are kept in this library until they are called by the system, at which time they are loaded into core storage. Transient SVC's are discussed in the section describing the NUCLEUS. The I/O access routines and error recovery routines are also kept in this data set.

SYS1. PROCLIB: This is the system procedure library; it contains the catalogued procedures called by the 'PROC=' parameter on the EXEC job control statement. The reader, writer, initiator and remote job entry procedures are also kept in this data set.

SYS1.SYSJOBQE: The system input and output queues are kept in this data set. Here the system keeps track of which jobs are to be executed, and which jobs have yet to have their output processed. SYSJOBQE is discussed in the section describing job management.

SYS1.LOGREC: When a machine error is encountered, a record of the failure and the machine environment at the time of failure is made in this data set.

In addition to the required system data sets, a number of optional data sets may be included on SYSRES at the user's discretion. Some of these are discussed below.

SYS1.MACLIB: Assembler language macro definitions are kept in this library.

SYS1.SORTLIB: Many of the SORT/MERGE routines are kept in this library; they are loaded into core by the SORT/MERGE control routines which are kept in LINKLIB.

SYS1.ALGLIB: ALGOL subroutines are kept in this library.

SYS1.FORTLIB: In this library are kept FORTRAN subroutines.

SYS1.COBLIB: This data set contains the COBOL subroutines.

SYS1.PL1LIB: Subroutines required by PL/I are kept in this library.

SYS1.TELCMLIB: Modules which are used to handle telecommunication lines are stored in this data set.

SYS1.SYSVLOGX and SYS1.SYSVLOGY: When the write to log (WTL) macro is used, the message referenced is written into one of these data sets. The operator may display these messages upon the system console as it pleases him.

SYS1.ROLLOUT: This data set is used to roll out low priority jobs when a high priority job needs the storage occuppied by the low priority job. When the storage required by the high priority job is no longer needed, the low priority job is rolled back into core from this data set.

SYS1.ASRLIB: When the asynchronous machine check handler is is included in the system, this data set is used. For further discussion, see the OS/360 Concepts and Facilities SRL.

In MVT, two additional groups of data sets are used; these are generally called IEFDATA data sets. The MVT job scheduler SPOOL's input stream data (i. e. data following a DD * card) to these data sets. SYSOUT data is also placed in these data sets, to be SPOOL'ed at a later time to the SYSOUT device.

MVT in Core Storage:

Our second area of concern in understanding MVT is the physical layout of core storage. Four general areas comprise the MVT system in storage: The NUCLEUS; the SYSTEM QUEUE SPACE; the DYNAMIC AREA; and the LINK PACK AREA.

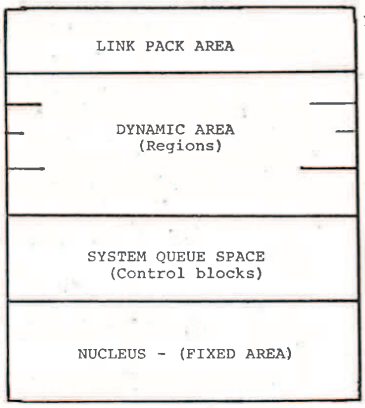

Figure 2:
Core storage layout of an MVT system.

THE LOGICAL MVT SYSTEM:

THE NUCLEUS

Part of MVT remains in storage at all times; that is, it is permanently resident. This is called the NUCLEUS, and is loaded by NIP at IPL from SYS1.NUCLEUS into the low address portion of storage. In this area reside the most frequently used MVT routines, such as interrupt handlers, the input/output supervisor, I/O channel scheduling routines, task schedulers, storage supervisors etc. The NUCLEUS has a storage protection key of zero, and its routines execute in the supervisor state.

I/O error analysis and recovery routines are brought into the I/O supervisor transient area in the NUCLEUS; these are brought in from SYS1.SVCLIB. This transient area is 1024 bytes long, and is used for all transient error recovery routines.

Other transient areas exist in the NUCLEUS; these consist of pairs of 1024 byte areas. A minimum of one pair exists; more pairs may be generated into the system at the user's option. These are called the supervisor call transient areas.

There are four types of SVC's in OS/360.

TYPE I SVC's are resident in the NUCLEUS and are executed with all interrupts masked off (i. e. disabled). The WAIT, EXCP, POST and GETMAIN SVC's are examples.

TYPE II SVC's are resident in the NUCLEUS and are executed in an enabled state (i. e. interrupts are allowed). The LINK, XCTL and LOAD SVC's are type II.

TYPE IIISVC's are transient, and consist of a single load into the transient area (i. e. they are less than or equal to 1024 bytes long). The WTO and WTL SVC's are type III.

TYPE IV SVC's are transient, and consist of multiple loads into the transient areas; each load LINKS to the following load. ABEND, OPEN and CLOSE are examples of type IV SVC's.

Hence the NUCLEUS consists of a fixed area of core, containing a fixed portion of MVT, with the exception of the I/O supervisor transient area and pairs of supervisor call transient areas which contain, of course, variable routines loaded dynamically.

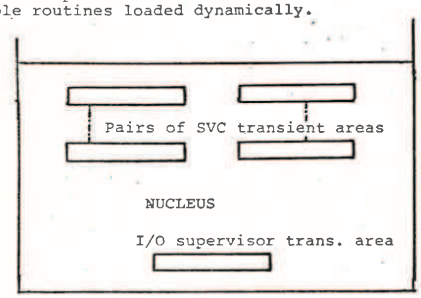

Figure 3 : The NUCLEUS.

THE SYSTEM QUEUE SPACE

Immediately above the NUCLEUS in core is the SYSTEM QUEUE SPACE; it is set up at IPL by NIP. This area is fixed in size (the size may be changed at IPL) and has protect key zero.

Here MVT creates and maintains control blocks in order to control multi-programming and supervise the system. The SYSTEM QUEUE SPACE may be likened to a work area for the MVT system in core; that is, SYS1.SYSJOBQE is the work area for controlling the jobs before and after execution and the SYSTEM QUEUE SPACE for maintaining the jobs once selected and brought into core for execution.

These control blocks are formed into queues which are maintained by MVT to manage storage, protect keys, to keep track of what programs are in storage, where they are, their status, protect keys, where free core exists and how much, what devices are allocated etc. When multi-programming, task switching and task control is maintained through these queues.

Each region in the DYNAMIC AREA has a queue of control blocks in the SYSTEM QUEUE SPACE; by using these queues the system manages each region and also multi-programs between regions. These queues and their uses are discussed in the section describing MVT control logic and control blocks.

THE LINK PACK AREA

Beginning at the top core address and extending downward is the LINK PACK AREA. This area is built at IPL by NIP; it is assigned a protect key of zero. The contents of the LINK PACK AREA may be altered at IPL; the contents remain fixed from one IPL to the next.

At the top of the LINK PACK AREA is the system BLDL list. This is a list of TTR's (relative track and record address) of commonly used routines in SYS1.LINKLIB and SYS1.SVCLIB. When these routines are needed, the system can directly access the module through the TTR; this eliminates the time consuming search through the VTOC for the data set, the search through the data set directory for the member, then the actual reading of the member. The BLDL area is built by NIP at IPL.

Directly below the BLDL list in core is an area in the LINK PACK AREA which contains re-entrant routines from SYS1.LINKLIB and SYS1.SVCLIB; these are not SVC routines. Usually these routines are common access routines, such as GET locate mode fixed, parts of the MVT initiator; step start and termination routines, parts of the READER/INTERPRETER and OUTPUT WRITERS. User written re-entrant routines may be included. All regions and the NUCLEUS will share these routines as they are re-entrant.

The reader should be familiar with the three kinds of code in OS; these are: Non-re-usable; serially re-usable; and re-entrant. Non-re-usable code is that which alters itself, in the form of switches, data etc., in such a way that for each execution of the module a fresh copy must be loaded; it is neither re-usable or may it be shared.

Serially re-usable code is that which is self initializing - i. e. which starts by resetting counters to zero, resetting switches and clearing data areas. This code may be re-used without loading a fresh copy, but only in a serial manner; that is, task 2 must wait for task 1 to complete its use of the code before it may use it. Tasks may not be restarted in the code other than at the beginning.

Re-entrant code is that which in no way alters itself. Any switches or data areas are kept external from the code by using GETMAIN's and registers to isolate any changes. A task using this code may, at any time, be interrupted by a higher priority task which uses this code. As the code is in no way altered, and the data is isolated and pointed at by a register, the higher priority task does not alter the code, as it obtains its own data area through a GETMAIN during execution. When it relinquishes the code, the registers of the lower priority task are restored, which point to the lower priority task's data area. In this way the lower priority task may resume execution at the point of interruption rather than at the beginning of the code. This code may be shared and re-used by tasks in a priority fashion, each interrupted task resuming execution at the point of interruption. It is this type code which is in the LINK PACK AREA.

The third and final section of the LINK PACK AREA, immediately under the second section, contains re-entrant modules of type III and type IV SVC's. Usually these will be the commonly issued SVC's such as OPEN, CLOSE etc. By residing 'permanently' in core much time is saved by eliminating searches through SYS1.SVCLIB for them. Not all type III or type IV SVC's are re-entrant; great care must be taken in selecting routines to reside here. This area, like the rest of the LINK PACK AREA, is set up and filled by NIP at IPL.

These three areas, then, considered together, make up the LINK PACK AREA.

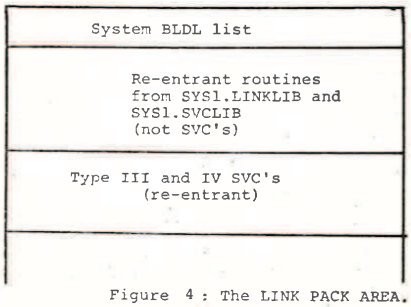

Figure 4 : The LINK PACK AREA.

THE DYNAMIC AREA

The remaining area of core, that which is between the SYSTEM QUEUE SPACE and the LINK PACK AREA, is called the DYNAMIC AREA. It is into this area that processing programs and data are loaded and executed. The DYNAMIC AREA is divided into dynamically assigned regions; roughly speaking there is one region per job in the system. These regions are allocated dynamically as needed, and will vary in size and location; there will be one region per initiator active in the system. Initiators are discussed in the section on job management in an MVT system.

Let us now examine the make up of an individual region. A region consists of three general areas: Subpool 251; subpool 252; and the data area. These areas are assigned dynamically within the region as the need arises.

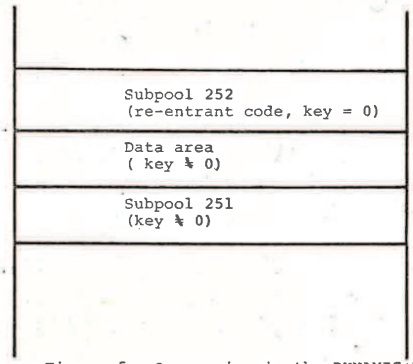

Figure 5 : One region in the DYNAMIC AREA.

Subpool 252 is assigned from the top end of the region. Here re-entrant code for the job will be loaded; subpool 252 is assigned a protection key of zero. This is analagous to the LINK PACK AREA; in fact, the LINK PACK AREA is actually the subpool 252 for the NUCLEUS.

From the bottom of the region is assigned subpool 251; this will be assigned a non-zero protect key. Here will be placed non-re-usable and serially re-usable routines for the job. These two subpools, 251 and 252, are sometimes lumped together in the term JOB PACK AREA.

Between these two subpools is the data area. It is from this area that GETMAIN's will be satisfied. Usually I/O buffers will reside in this area. The same non-zero protect key assigned to subpool 251 will be used for the data area.

There is one region which is permanently assigned; that of the MASTER SCHEDULER. This is assigned and loaded by NIP at IPL; it is always immediately below the LINK PACK AREA. The reader will recall that the MASTER SCHEDULER is the communication link between OS/360 and the operator: it accepts and schedules commands, and displays the system status upon request.

MVT CONTROL LOGIC AND CONTROL BLOCKS

GENERAL:

A job in an OS/360 system is controlled through queues of control blocks. In order to analyse or debug a job it is necessary to re-construct the flow of control and I/O; to do this one must have a basic understanding of the contents and usage of some of these control blocks. We will discuss these in two stages: Task control and I/O control.

TASK MANAGEMENT AND CONTROL BLOCKS:

The reader should be familiar with the concept of a task. A TASK is work to be done; a program is merely a series of steps to perform a function. There is not a one to one relationship of TASK's in the system and programs in the system. For example, consider the square root program, SQRT. The program SQRT is simply a series of machine instructions; the actual performance or taking of the square root is a TASK. Hence the difference: The taking of the square root is a TASK; the TASK uses the program SQRT. Now, if many jobs (TASKS) in the system require the taking of the square root, and SQRT is serially re-usable or re-entrant, is it necessary for each TASK to have its own copy of SQRT? Of course not; we will load one copy of SQRT into the LINK PACK AREA, and allow all TASK's to share this code; we will QUEUE TASK's on this code. This will be particularly true in the case of common I/O routines. Hence we will have many TASK's and one program. This QUEUING and sharing and controlling of TASK's and programs is what is meant by TASK CONTROL.

For each TASK in the system, a TCB (Task Control Block) is created. There is at least one TCB per region. The TCB is created by an ATTACH macro, and consists of roughly 190 bytes; it resides in the SYSTEM QUEUE SPACE. The TCB contains a multitude of flags indicating the status of the TASK, and pointers to every conceivable queue of related control blocks and tables; it is the communication and control center for the TASK. The contents of the TCB will be discussed in the section on core dumps in MVT; for more detail refer to the OS/360 System Control Blocks SRL.

For each PROGRAM required for the performance of the task, an RB (Request Block) is created in the SYSTEM QUEUE SPACE. These RB's contain program control information, PSW's, size etc. RB's are created by the LINK and XCTL macros. Four types of RB's are created: PRB's for programs; SVRB's for resident and non-resident SVC's; IRB's for system interrupt blocks (asynchronous requests such as STIMER); and SIRB's for asynchronous I/O error recovery routines. We will concern ourselves with PRB's and SVRB's.

These RB's are formed into the ACTIVE RB QUEUE for the task; the TCB points to the ACTIVE RB QUEUE. This queue represents programs which are active or have been active.

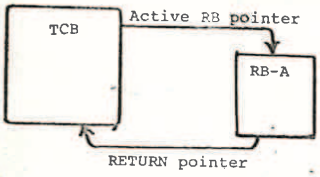

Figure 6:
Active RB queue with one avtive RB in the queue.

For example, if program A is the only program needed by the task, the active RB queue would look like that in Figure 6.

Consider, however if program A contains a LINK to program B. At the conclusion of this "LINK B" macro execution, the active RB queue would look as the example in Figure 7.

The TCB always points to the RB which is ACTIVE (i. e. the RB for the program currently executing); in Figure 7 this is RB-B. This RB points back (via the RETURN pointer) to the next most recent RB (in this case RB-A).

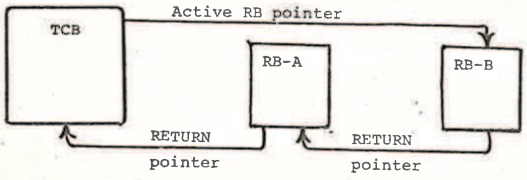

Figure 7:
Active RB queue with two RB's in the queue. RB-B is the active RB at this time; it was the object of a LINK macro in program A.

Upon completion of program B, the RETURN is made, via the RETURN pointer, to program A: After execution of this RETURN, the active RB queue would appear as in Figure 8.

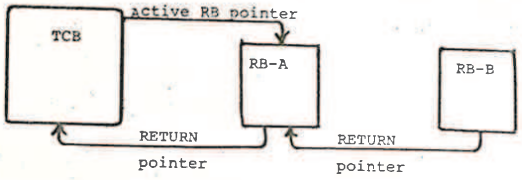

Figure 8:
Active RB queue after execution of the RETURN in program B via the RETURN pointer in RB-B. The active RB queue now has one RB in it.

In this manner, the system controls the flow of control of the task through the required programs; return of control will follow the RETURN pointers. Hence we can reconstruct the control flow through the programs necessary for the completion of the task.

One should be aware that an XCTL macro is handled differently. If in our example program B executed an XCTL to program C, the active RB queue would appear as in Figure 9.

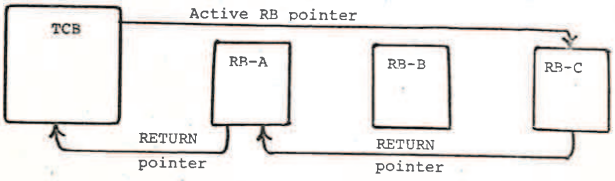

Figure 9:
Active RB queue after program A LINK'ed to program B which XCTL'ed to program C.

Note that in Figure 9 the RETURN pointer in RB-C indicates return to program A; this is consistent with the fact that RETURN is always made to the next higher level of control, and XCTL brings in a program at the SAME level of control. LINK, however, brings in the program at the next lower level of control. Note, however, that the system is now unaware of the existence of RB-B; there is no way of returning to it, as it has in effect disappeared.

For programs which are brought into core as the result of a LOAD macro, a control block similar to an RB is constructed, called an LLE (Load List Element). Since there is no such thing as an "active" LOAD'ed program, LLE's are simply chained together into an LLE queue. The TCB points to the LLE queue, which resides in the SYSTEM QUEUE SPACE.

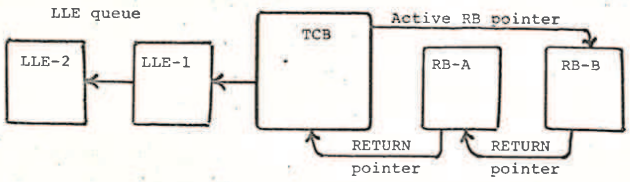

Figure 10:
Active RB queue and LLE queue, pointed at by the TCB.

One other important control block for task management should be discussed. The CDE (Contents Directory Entry) queue contains one CDE for every load module in the region, or being used by the region. There is one CDE queue for each region; this queue is used by MVT to manage the contents of storage. Three important pieces of information are kept in the CDE: 1) The pointer to the corresponding RB; 2) the entry point address of the related program; 3) the module name. Hence we can quickly "inventory" our region by scanning the CDE queue. This queue resides in the SYSTEM QUEUE SPACE.

In the section on core dumps in MVT we will use these control blocks to pinpoint the program in error.

Figure 11 illustrates the relationships of all the control blocks so far discussed.

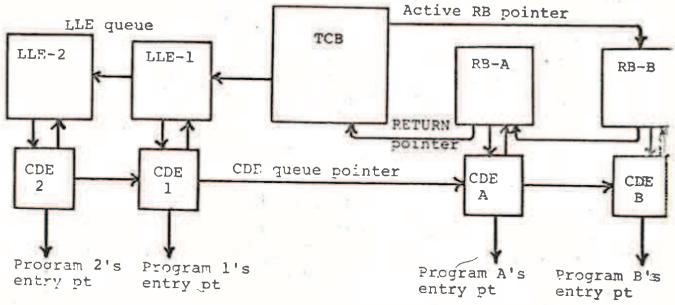

Figure 11:
MVT task control queues and control blocks for a region. All contro blocks and queues shown reside in the SYSTEM QUEUE SPACB.

One additional, extremely important area must be discussed and understood to completely master program control - that of register SAVE areas. The reader should know the OS/360 register conventions:

Register 0: Used for passing parameters;
1: Used for a pointer to parameter lists;
2-12: For general program use;
13: Points to the current SAVE area;
14: Used for the RETURN address;
15: For entry point address or return code.

Also he must know and follow the OS/360 conventions concerning forward and backward chaining of SAVE areas; only if he knows and follows these conventions will register contents be of any value to him in debugging. For a discussion of chaining conventions, see the OS/360 Supervisor and Data Management Services SRL.

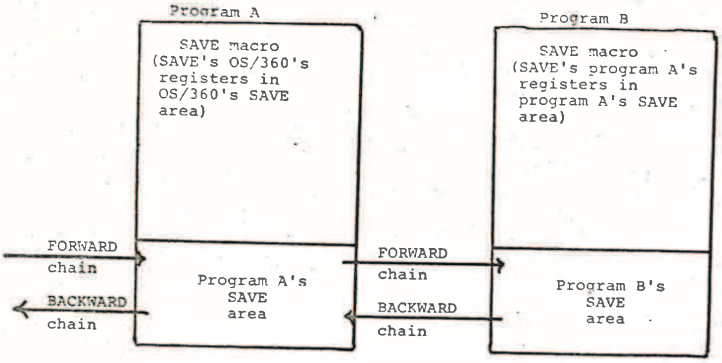

Figure 12:
SAVE area conventions, showing forward and backward chains; those to the left of program A point to and from OS/360's SAVE area.

DATA MANAGEMENT AND I/O CONTROL BLOCKS:

Many times the source of an error appearing in a program lies in the data or data management, such as improper data definition, illegal data or improper data manipulation. OS/360 performs data management in a manner similar to task management; that is, with a series of control blocks and queues.

Five major control blocks need be considered; these are located through the task's TCB.

The DCB (Data Control Block) is the final residing place of the control information for logical data manipulation. This control block is assembled into the program, and may be filled in and supplemented from the DD card and data set label at OPEN time. The reader should be familiar with the DCB. For further information see the OS/360 Concepts and Facilities manual.

While the DCB does contain most of the information concerning the data set, it does not contain device dependent information or the channel program necessary to actually perform the I/O operation. Other I/O blocks and I/O routines must be present; to co-ordinate and manage these, the OPEN routine constructs in the SYSTEM QUEUE SPACE a DEB (Data Extent Block) for each DCB. This control block is the "communications center" for IOS (Input Output Supervisor). It contains the addresses of the I/O modules such as start I/O (SIO), end of extent and channel end. The DEB also contains pointers to the TCB, DCB, IRB and UCB (Unit Control Block) and the next DEB in the DEB queue. The TCB points to the first DEB in the DEB queue. For direct access data sets, the extents (beginning and ending TTR's) are kept in the DEB, giving it its name. No I/O operations may take place outside this extent.

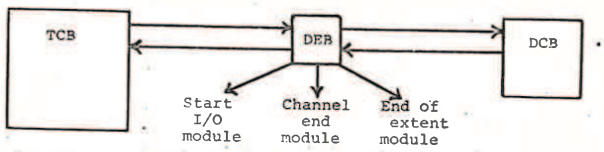

Figure13: The DCB and DEB.

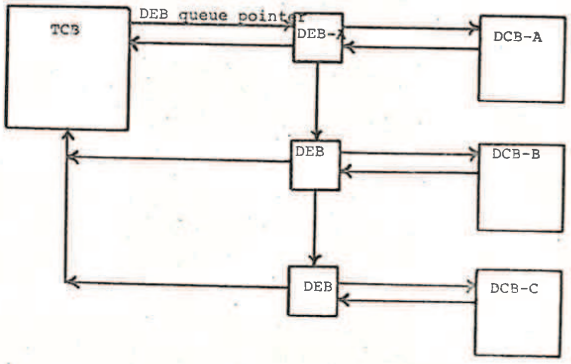

Figure 14: The DEB queue.

IOS uses the DEB to locate the proper start I/O module for the allocated device, and the proper channel end routine for interrupt handling and interpretation.

Communication between IOS and the problem program takes place through the IOB (Input Output Block); here the CCW list (the channel program) and the completion code for an I/O request are placed for program inspection. Contents of the IOB include a pointer to the DCB, a pointer to the ECB (Event Control Block) and a pointer to the CCW list. Also such things as the block count, sense bytes, the last word of the CSW and error counts are kept in the IOB. In most cases the actual channel program is built in the IOB. For direct access devices, the initial seek TTR is kept here. There is one IOB for each I/O request; it is constructed in the user's region by the user himself. Normally the programmer will re-use the same IOB for all similar requests. The READ, WRITE, GET and PUT routines usually construct this IOB.

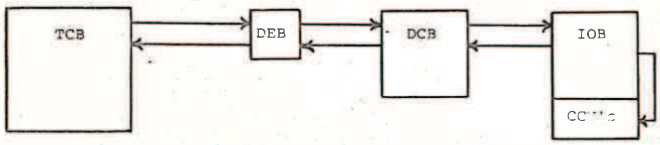

Figure 15: The DEB, DCB and IOB.

At this point the only thing missing is the link between these control blocks and the actual physical device. This is accomplished through the UCB (Unit Control Block). The UCB contains all information necessary pertaining to the device and its status, such as unit type, unit address, volume serial of the volume presently mounted, TTR of the VTOC for direct access devices and the sense bytes last read. Flags in the UCB are set and used by IOS to indicate whether the device is ready, on-line, busy, seeking or has encountered a permanent I/O error. Contained in the NUCLEUS, there is one UCB per device, which is placed there during system generation; the DEB contains a pointer to the allocated UCB for the DCB.

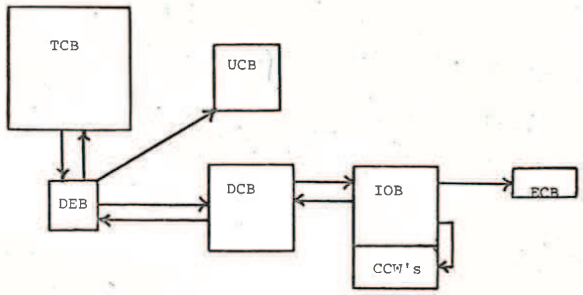

Figure 16: The DEB, DCB, IOB, ECB and UCB.

One additional table should be mentioned which is of minimal value for debugging but is of considerable importance in programming. This is the TIOT (Task Input Output Table), constructed by the INITIATOR in the SYSTEM QUEUE SPACE during I/O allocation. The TIOT contains the jobname, stepname, and one entry for each DD card containing the DD card name and the address of the allocated UCB for that DD card. The TCB points to the TIOT.

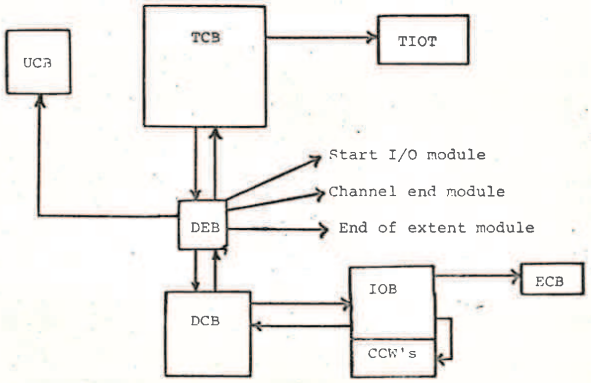

Figure 17: Complete I/O control blocks for one data set.

JOB MANAGEMENT IN AN MVT SYSTEM

GENERAL:

A JOB passing through an OS/360 MVT system encounters five distinct stages. These stages involve systems routines grouped under the function of JOB MANAGEMENT; to understand the system we should be familiar with JOB MANAGEMENT.

The five stages in which a JOB is involved are:

* Entry into the MVT system;
* Initiation of the JOB;
* Execution of the JOB;
* Termination;
* The writing of standard output classes to punches, printers and tapes.

Handling these stages are the following routines:

* A JOB is read into the system by a READER/INTERPRETER;
* Selecting a JOB for execution and initiating it is an INITIATOR (sometimes called an INITIATOR/TERMINATOR);
* While executing, use is made by the program of supervisor services such as OPEN, CLOSE etc.;
* The INITIATOR terminates the JOB by disposing of the data sets etc.;
* Standard output is SPOOLED to the printers and punches by an OUTPUT WRITER or SYSOUT WRITER.

THE READER/INTERPRETER:

The main function of the READER/INTERPRETER is to read the JOB STREAM. The JOB STREAM consists of JCL (Job Control Language) statements and DD * data (that is, data following a //ddname DD * card and preceding a /* card), sometimes called SYSIN data. In other words, the READER/INTERPRETER reads the INPUT STREAM, which is usually cards or tape.

As the JCL statements are read, the READER/INTERPRETER converts them to internal table form and places these tables on SYS1.SYSJOBQE; these will form the input queue.

The JOB statement is converted to a JCT (Job Control Table), containing the information necessary to control the JOB. JOB name, priority, message class, region size, condition codes and a pointer to the accounting information are contained in the JCT.

As the EXEC card is encountered it is converted into an SCT (Step Control Table), containing the step name, PARM field values, program name, condition codes and region size. With the SCT we begin the real building of the input queue entry for this JOB; the JCT points to the first SCT, each SCT points to the next SCT. For the following discussion we will consider a three step JOB; our queue so far would appear as in figure 18.

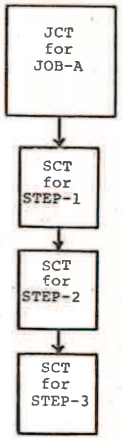

Figure 18:
Entries built in SYS1.SYSJOBQE from a JOB card and three EXEC cards by the READER/INTERPRETER.

Each DD card encountered by the READER/INTERPRETER is converted into a JFCB (Job File Control Block). The JFCB contains all information for allocating and disposing of the referenced data set during the job's execution. OPEN and CLOSE use the JFCB, which contains all the DCB information coded in the DCB parameter on the DD card, and the data set label to complete the DCB coded in the program. This is discussed in the section on the execution of the program. In the JFCB is stored the data set name (DSNAME), label type, sequence number, volume serial numbers, space allocation information (from the SPACE parameter), disposition and all the DCB information coded in the DD card. All JFCB's for a job step are chained together; the SCT points to the JFCB queue (through the step I/O table, which we shall not discuss). Hence the total entry in SYS1.SYSJOBQE created by the READER/INTERPRETER for our job would appear as in Figure 19.

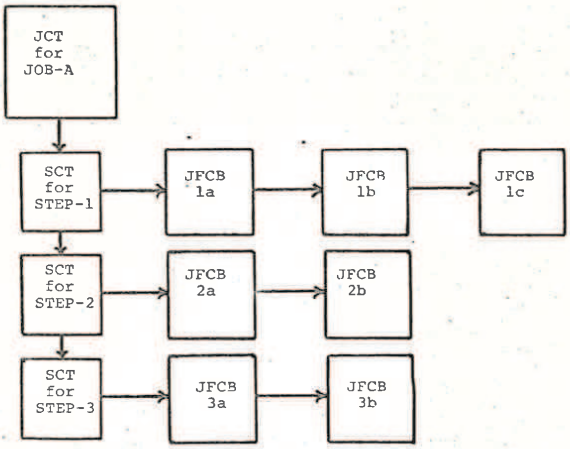

Figure 19:
Complete entry built by a READER/INTERPRETER in SYS1.SYSJOBQE for a job containing thre steps (SCT's STEP-1, STEP-2, STEP-3). STEP-1 contains three DD cards (JFCB's 1a, 1b, 1c), STEP-2 contains two (JFCB's 2a and 2b) and STEP-3 contains two (JFCB's 3a and 3b)

When the READER/INTERPRETER encounters SYSIN data (that is, DD * data) it SPOOL's the data to an IEFDATA data set on a direct access device and builds the JFCB for the DD * card to point to the direct access data set.

When a DD card referencing SYSOUT data (i. e. a SYSOUT=x card) is encountered, the READER/INTERPRETER allocates space for it in the OUTPUT QUEUE; notice that SPACE is allocated, it is not ENQUEUED or placed in the OUTPUT QUEUE.

At this time the READER/INTERPRETER has completed building the queue entry for the job in SYS1.SYSJOBQE, and now the job is ENQUEUED into the INPUT QUEUE by the READER/INTERPRETER. The system is now, for the first time, aware of the job's presence in the system. The INPUT QUEUE is kept in priority sequence, and the job's queue entry consists of a pointer to its JCT. The INPUT QUEUE resides in SYS1.-SYSJOBQE.

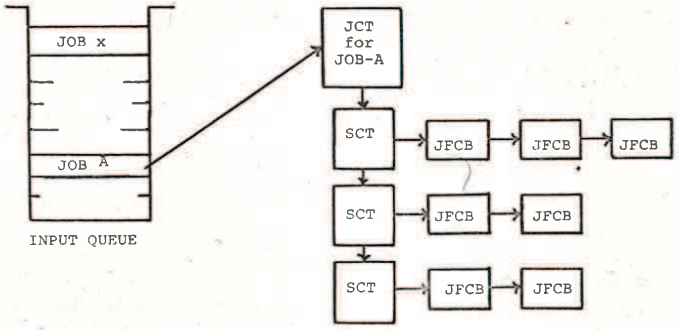

Figure 20: Complete INPUT QUEUE entry for JOB-A.

This sequence continues until either the READER/INTERPRETER is stopped or the input stream is exhausted. Many READER/INTERPRETER's may, and probably will be, be active at one time, each reading a separate job stream.

In summary, the READER/INTERPRETER:

* Reads the JOB STREAM;
* Builds control tables in SYS1.SYSJOBQE from JCL;
* SPOOL's SYSIN data;
* Reserves space for SYSOUT data;
* ENQUEUES the job into the INPUT QUEUE.

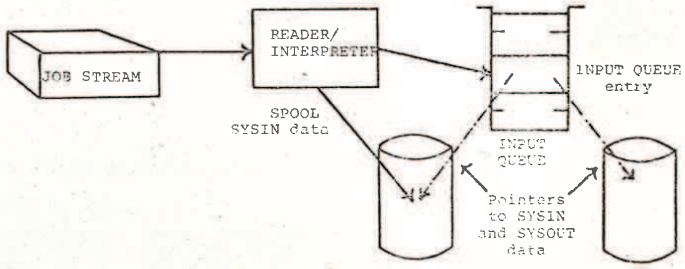

Figure 21: The READER/INTERPRETER

THE INITIATOR:

Once a job has been placed into the INPUT QUEUE, there it remains, until it becomes the next job for execution (i. e. all jobs of a higher priority have been started). The INITIATOR selects jobs from the INPUT QUEUE in priority sequence for initiation. The INITIATOR finds all information for selection and initiation by following the pointer in the job queue to the job's JCT, SCT's and JFCB's; these contain all information necessary to the INITIATOR.

From the JCT the INITIATOR determines the region requirements from the region field. The INITIATOR will request a region the size of which is determined by the larger of the region parameter in the JCT (filled in by the READER/INTERPRETER if not present in the JOB card) or the size of the main INITIATOR modules to be brought into the region. That is, a region will not be allocated smaller than the INITIATOR size. This also takes place for each step, the value being taken from the step's SCT. The INITIATOR then removes the job's entry from the INPUT QUEUE.

Into this just allocated region the resident portion of the INITIATOR fetches the non-resident INITIATOR modules and ATTACH'es to these modules. The INITIATOR queries any condition codes passed at termination by a previous job step against those in the JCT and next SCT for possible termination conditions. The INITIATOR (in the region) now examines the JFCB's for this step (through the JFCB chain found through this step's SCT) and allocates devices and DASD space. This allocation is done through the JFCB's and UCB's; the TIOT is built in the SYSTEM QUEUE SPACE at this time.

After I/O device and space allocation, the INITIATOR goes to the SCT to determine the program to be initiated. The INITIATOR then issues an XCTL to this program; this brings the program into the region on top of the INITIATOR, overlaying it, and places the program's RB as the only RB in the ACTIVE RB QUEUE. The INITIATOR has supplied a SAVE area in the region through a GETMAIN (protect key non-zero). A similar technique is used to relocate the PARM information from the SCT to the region.

One INITIATOR exists per region; it is created and dedicated to that region through the START INIT operator command.

In summary, the INITIATOR:

* Selects jobs from the INPUT QUEUE for execution;
* DEQUEUES the job from the INPUT QUEUE;
* Allocates a region for the job;
* Allocates I/O and DASD space - builds the TIOT;
* XCTL's to the problem program.

PROGRAM EXECUTION:

Control of problem program execution is, of course, handled by the program itself. Occasionally, however, the program will issue SVC's to request supervisor assistance for particular functions, such as performing I/O, bringing other programs into the region etc. It is important to understand the general flow of control during these SVC's, and in some detail for two particular SVC's, OPEN and CLOSE.

When a program issues an SVC, the SVC interrupt handler receives control. This interrupt handler determines what is to be done, and routes control to the proper SVC routine. For a request for a type I SVC control is routed via a branch, and the routine is basically transparent to the user, as no interrupts are permitted during execution of a type I SVC. When a type II, III or IV SVC is issued, however, the interrupt handler builds an SVRB for the proper routine, and enqueues it as the active RB in the active RB queue. That is, the RB of the program which issued the SVC becomes the second most active RB, and is chained, via the RETURN pointer, to the newly created SVRB, which is now the active RB. If the SVC routine is resident, it will then be given control; if transient, it will be loaded (if not already present) into one of the SVC transient areas in the NUCLEUS and given control. Return will be made in the normal manner to the issuing RB's program via the RETURN pointer in the SVRB.

Two type IV SVC's are often enough the place of a programmer's Waterloo to warrent separate discussion; these are OPEN and CLOSE.

OPEN is an SVC designed to locate and attach a data set to the program, and make it ready and available for processing. CLOSE is the logical converse of OPEN; it detaches the data set from the program, and makes it no longer available for processing by that program.

OPEN performs a great deal of its processing in what is called the forward and backward merge of the DCB, JFCB and the data set label; here a great number of errors in data management coding will be discovered. As previously mentioned (in the section on control blocks), the DCB is the final repository of most of the logical characteristics of a data set. The forward and backward merges of OPEN fill in information in the DCB which was either unavailable or not included at assembly or compile time; this information is supplied from the DD card, via the JFCB, and the data set label, if present. The JFCB, recall, contains the DCB information which was coded on the DD card; it was placed in the JFCB by the READER/ INTERPRETER.

The FORWARD MERGE consists of two steps:

1) The ZERO fields in the JFCB are filled in from the data set label if possible; note that only ZERO fields are merged. Data already present in the JFCB CANNOT be overridden;

2) The ZERO fields in the DCB are filled in from the JFCB; note that, as before, only the ZERO fields are merged. Data coded into the DCB CANNOT be overridden.

At this time the DCB is complete, and the two step BACKWARD MERGE begins, consisting of:

1) For INPUT data sets, the ZERO fields of the JFCB are filled in from the DCB; note that no fields are overridden.
For OUTPUT data sets, ALL JFCB fields except DSORG are OVERRIDDEN from the DCB;

2) For DASD OUTPUT data sets, all fields in the DSCB are overridden by the JFCB; for INPUT data sets, the labels are already existent, and remain inviolate.

After the merges, OPEN builds and fills in the DEB's for the data sets in the SYSTEM QUEUE SPACE.

CLOSE is the logical converse of OPEN. CLOSE creates any required trailer labels, and detaches the data set from the program by dequeuing the data set's DEB from the DEB queue.

THE TERMINATOR :

Execution of a job step terminates when the program controlled by the highest level RB in the active RB queue issues its RETURN. This returns control to the resident portion of the INITIATOR, which XCTL's to the transient portion of the INITIATOR, consisting of the TERMINATION routines, which overlays the program in the region.

This TERMINATOR performs any accounting functions and routines. It then examines the JFCB's for the step and disposes of the data sets (i. e. DELETE's, catalogues, PASS'es or KEEP's them). Any I/O devices no longer needed are de-allocated. If this was not the last SCT on the SCT queue for this job (i. e. it was not the last step), the TERMINATOR returns control to the resident portion of the INITIATOR which begins all over with the examination of the next SCT and continues as described in the section on the INITIATOR. If this just terminated step was the last step in the job (i. e. the last SCT), the TERMINATOR ENQUEUE's all SYSOUT data sets in the OUTPUT QUEUE in priority sequence. Note that all SYSOUT from the job is enqueued at one time, at job end; recall that the READER/INTERPRETER reserved space for this SYSOUT data, which was written by the problem program. Now, for the first time, the system is aware of this SYSOUT data.

After the termination of the last step of a job, the TERMINATOR returns to the resident portion of the INITIATOR, which selects the next job for initiation from the INPUT QUEUE, thus starting again the entire sequence described in the section discussing the INITIATOR. This sequence continues until either the INITIATOR is stopped, or the INPUT QUEUE is empty, at which time the INITIATOR waits for the next entry to be ENQUEUED, when the sequence begins again.

In summary, the TERMINATOR:

* Performs the accounting function;
* Handles data set disposition;
* Frees I/O devices no longer needed;
* If this was the last step in the job, ENQUEUE's, by priority, all SYSOUT data sets into the OUTPUT QUEUE, then returns to the INITIATOR;
* If this was not the last step in the job, returns to the INITIATOR for initiation of the next step.

THE OUTPUT WRITER:

An OUTPUT WRITER is started for a SYSOUT class; this writer is then dedicated to that class' OUTPUT QUEUE. For instance, if an OUTPUT WRITER is started for output class A, it will be dedicated to the OUTPUT QUEUE containing the entries for SYSOUT=A data sets.

The SYSOUT WRITER selects the top entry in the OUTPUT QUEUE; these entries were enqueued, recall, by the TERMINATOR at job end. These are selected in priority sequence. Once selected, the OUTPUT WRITER then SPOOL's the data set to the specified device (usually a printer or punch). This data set can consist of SYSOUT= data and system messages (SMB's) such as allocation and disposition messages for data sets.

The OUTPUT WRITER then DEQUEUE's this data set's entry from the OUTPUT QUEUE; it then returns to the first function and selects the next entry in the OUTPUT QUEUE for output.

The OUTPUT WRITER continues this sequence until either stopped or there are no more entries in its OUTPUT QUEUE for SPOOL'ing, at which time it waits for the next entry to be ENQUEUE'd, when it begins the sequence again.

In summary, the SYSOUT WRITER:

* Selects a completed data set for SPOOL'ing from the OUTPUT QUEUE;
* Writes (or SPOOL's) the data set;
* DEQUEUE's the entry from the OUTPUT QUEUE;
* Returns to the first step.

COMMENTS:

It is perhaps important to remember that these three functions, the READER/INTERPRETER, INITIATOR/TERMINATOR and SYSOUT WRITER, are operating continuously, asynchronously and independently, which is part of the power of the MVT system. Their only real communication with one another is through the INPUT QUEUE and the OUTPUT QUEUE's.

CORE DUMPS IN MVT

INTRODUCTION:

There are generally five kinds of core dumps in an MVT system.

An INDICATIVE dump consists of a single printed line on the MSGCLASS data set, containing the system completion code. It is produced automatically, without control cards and marks the termination of the job. In many cases, such as step timing violation, this single line is adequate for determining the cause of abnormal termination.

A UDUMP or SYSUDUMP is a full core dump listing all control blocks for the user's region (i. e. TCB's, RB's, DEB's etc.), and the user's region itself. In addition, any routines being used in the LINK PACK AREA are dumped. A SYSUDUMP is given when a terminal error occurrs and a //SYSUDUMP DD card has been included in the JCL for the current step. This is usually a large data set, so care should be exercised not to exceed space allocation in a SYSOUT data set. A SYSUDUMP marks the termination of the job. As this is the most common dump used in an MVT environment, it is this dump that the remainder of this paper discusses.

When a //SYSABEND DD card is used rather than a //SYSUDUMP DD card, a terminal error causes an ABEND dump. ABEND dumps are identical to UDUMPS except that the NUCLEUS and a TRACE TABLE are also included. As the NUCLEUS is likely to be very large, this is a very large data set and care should be taken not to overrun a SYSOUT data set. The NUCLEUS is usually of little help to the average programmer, and an ABEND dump should be taken only when a UDUMP is inadequate.

A SNAP dump may be produced by an assembler language programmer. A SNAP macro causes a dump to be taken which is similar to an ABEND dump; the job, however, then continues, rather than terminating. The SNAP macro also specifies which control blocks and areas of core are to be dumped. For further details, see the OS/360 Supervisor and Data Management Macro Instructions manual.

The fifth kind of dump is the stand-alone or unformatted dump. It is used when an error causes the operating system to become inoperable. Not an OS/360 program, it simply dumps all of core, with little or no formatting of control blocks.

What causes an ABEND error? ABEND is actually an SVC; in fact it is the "unlucky" SVC 13. An ABEND SVC is issued when either the processing program or the supervisor is unable to continue with the job. The reader should notice that we refer to ABEND and UDUMP dumps, which are caused by the ABEND SVC.

The ABEND is usually issued by the control program in response to some interrupt, such as a program interrupt, from which the job cannot recover. For example, if the program generates an address outside its region and tries to execute an MVC to this address, a program interrupt will occur with a protection violation interrupt code. This interrupt will be handled by the MVT program interrupt handler in the NUCLEUS. Upon determination of the cause of the interrupt (i. e. a protection violation), the interrupt handler then determines if the error is recoverable (it is not). Hence an ABEND is to be issued, so the interrupt handler next determines which task caused the error (i. e. which region). At this point the interrupt handler issues the ABEND for the program.

Two things should be noted from the preceding paragraph. First that the executing program in error did not issue the ABEND; it was issued by the MVT interrupt handler in the NUCLEUS. This is usually the case, and for this reason the PSW at entry to the ABEND processor will rarely contain the address of the actual error; it will contain the address of the MVT routine in the NUCLEUS which actually issued the SVC 13, the ABEND SVC. Secondly, a considerable amount of processing occurred in the interrupt handler before the ABEND was issued. That is, the actual ABEND was quite removed from the actual error.

The reader should be aware, however, that the problem program may issue its own ABEND, which make the PSW and general registers at entry to ABEND the program's own. Several language processors produce controlling routines which do this, such as IBCOM produced by FORTRAN. IBCOM, for instance, intercepts most interrupts, and occasionally issues its own ABEND.

THE MVT CORE DUMP IN GENERAL:

The MVT SYSUDUMP is divided into six general sections (see figures 22a, 22b, 22c, and 22d):

1) General status, indicative and TCB area, consisting of the top of the dump through the TCB section;

2) Region contents, showing ACTIVE RB QUEUE, LOAD LIST, CDE's (contents directory) and XL's (extents list);

3) The I/O section, consisting of the DEB QUEUE and TIOT;

4) Storage management and queue control blocks (the MSS, D-PQE, FBQE and QCB TRACE);

5) Register contents, consisting of the SAVE AREA TRACE and REGS AT ENTRY TO ABEND;

6) Load modules, consisting of the load modules being used by the task.

There are a few fields in the general and TCB area of which we should be aware before continuing. While many examples of dumps have been included, it is strongly urged that the reader have a dump readily at hand for his perusal.

Look at the TCB entry in Figure 23; this is followed by the address of the TCB - note this address, as we shall use it later. The following fields are of interest:

RBP is the pointer to the active RB; note this address matches that of the last SVRB in the ACTIVE RB QUEUE;

DEB points to the first DEB in the DEB queue;

TIO is the TIOT address;

CMP 80322000 contains the system completion codes; positions three through five contain the SYSTEM COMPLETION CODE (i. e. 322 which matches that printed at the top of the dump), and positions six through eight contain the user code, if any (none in this case);

PK-FLG contains in the first position the user protection key; in this example it is "E";

LLS points to the beginning of the LOAD LIST;

JLB contains the address of the JOBLIB DCB, if present.

AN APPROACH TO DEBUGGING:

When debugging a program using a dump, knowing the layout and contents of the dump is not always adequate; what is needed is a method of attack. The debugger is usually overwhelmed by the wealth of data displayed; he must have a plan for debugging to prevent his thrashing aimlessly through assorted control blocks. The author has evolved a plan for "normal" debugging which has proved generally useful: it usually brings one, in an orderly fashion, to a place where productive debugging can be accomplished. Most of the time the plan will determine the error itself.

Based on the old newspaper dictum of "Who, Where, Why, When and What," the author has developed his approach into two "The five W's approaches to debugging." Plan one, THE NEWSPAPER APPROACH TO PROGRAMMING ERRORS, is used for tracking down the general programming error, while plan two, THE NEWSPAPER APPROACH TO DATA MANAGEMENT ERRORS, is an alternate, aimed at tracking down I/O errors.

THE NEWSPAPER APPROACH TO PROGRAMMING ERRORS:

When approaching a core dump there are generally five things in which we are interested: these are the same five things a newspaper reporter wishes to know. They are:

1) WHY does OS/360 think we erred;

2) WHEN in our processing did we terminate;

3) WHO was executing - ourselves or OS;

4) WHERE in storage did we fail;

5) WHAT actually caused the error.

We shall begin interpreting our dump in the order listed.

Why:

The first thing we should examine is the reason OS/360 gives for producing the dump. OS tries to tell us why he thinks we erred; this is not always the actual reason for the ABEND.

At the top of the dump are displayed the job name, step name and a COMPLETION CODE, either SYSTEM or USER. A USER code is the code supplied by the programmer when he issues his own ABEND macro; the reason is in his own program. Normally a SYSTEM code will be listed; this indicates that OS/360 issued the ABEND. The meanings of these codes are listed in the Messages and Completion Codes manual.

Often this code itself is sufficient to determine the cause of the error. For example:

COMPLETION CODE 804 tells us that insufficient core was available for a GETMAIN macro. This usually indicates that the program to be loaded is too large; that is, the region allocated is too small;

COMPLETION CODE 001 informs us of an I/O error. In this case we might use the alternate approach to I/O errors discussed later;

COMPLETION CODE 806 indicates that a module was not found by the BLDL SVC. This usually occurs during the FETCH to a program. It indicates that either the incorrect program library was specified, or that we failed during a preceding link edit step. We may have merely mis-spelled the program name.

COMPLETION CODE 0Cx indicates a program check and will probably require further analysis.

If examination of the reason OS/360 gives for the dump is not adequate, we proceed to the next step. In our example, we ABEND'ed because we exceeded the time allowed for the job step.

When:

We should like to know WHEN in our processing we developed our error. This usually means in what module were we when something occurred which caused an ABEND to be issued. To determine this we need to know what modules were in core, and where they resided in our region.

We begin by examining the ACTIVE RB QUEUE (Figure 24a), which tells us which programs are in our region and being used. The ACTIVE RB QUEUE is listed with the oldest RB first, and the newest RB last. That is, the last RB to have control of the TCB is the RB on the bottom of the ACTIVE RB QUEUE. Notice that this is the RB whose address is contained in the RBP field of the TCB at the top of the dump.

Two kinds of RB's normally appear in the ACTIVE RB QUEUE: PRB's for programs and SVRB's for supervisor call routines. Let us examine the RB queue, starting with the most recent RB (i. e. the RB on the bottom of the queue).

The NEWEST RB is an SVRB controlling a module of ABDUMP (SVC 51). ABDUMP does the actual formatting and dumping of the control blocks for the region; this is of course the most recent RB, as it is printing the dump we are reading.

The next RB (second from bottom on the queue) is an SVRB controlling a module of ABEND (SVC 13). ABEND was called (by someone) by the issuing of an SVC 13. ABEND goes to the TIOT to check whether a SYSUDUMP or SYSABEND DD card was supplied by the user. If none was present, an INDICATIVE dump is produced and the job is terminated. If such a card is found (one obviously was, or we would not have a dump) then ABEND issues an SVC 51 to bring in ABDUMP which actually formats and prints the dump.

Hence, the RB which had control of the TCB at the time the ABEND was issued is the third most recent; that is, the third RB from the bottom of the ACTIVE RB QUEUE represents the program active at the time of the ABEND.

Now that we know which RB was in control of the TCB at the time of ABEND, let us map out the modules in our region to get an idea of the region's make up. Let us glance first at some of the fields in a PRB (see Figure 24a).

PRB 018C88 is the address of the PRB in the SYSTEM QUEUE SPACE;

APSW 400FF162 is the last half of the program's OLD PSW at the time of the last interrupt;

FL-CDE 00018DD8 contains, in the last six positions, the address of the corresponding CDE for this PRB. In our example this is 018DD8;

PSW FFE5000D 40003716 is the RESUME PSW.

Now, to map out the region, we go to the corresponding CDE's for the RB's (the address of each CDE is in the left hand column in the CDE section of the core dump). The CDE contains the following information (for our example, we are using the first CDE at location 018DD8):

018DD8 is the CDE address; note this is the CDE for our third most recent RB;

NCDE 000000 is the address of the Next Contents Directory Entry; note that, starting at the bottom of the CDE queue, each NCDE points to the next CDE up the queue. The last element on the queue has an NCDE field of zeroes, marking it as the last;

ROC-RB 00018C88 shows the address of the corresponding RB for this CDE in the last six digits. Note that this points back to our third most recent RB;

NM PDS tells us the load module name for this PRB - in this case PDS;

EPA 0B2C80 tells us the Entry Point Address of the module;

XL/MJ 018E18 gives us a pointer to the extent list (XL) element for this CDE (i. e. for this module). More on the extent list in a moment.

Hence, at this point we now know the names and entry point addresses for all the modules in our region and those we are using in the LINK PACK AREA.

We should like to know the EXTENTS of all these modules - i. e. their beginning and ending addresses. To find these for each module we find the corresponding extent list element for each CDE. Let us examine the XL entry for the module PDS (see Figure 24b).

018E18 in the left hand column is the address of this XL entry. Note that this is the address pointed to in the XL/MJ for the PDS CDE;

80003B80 in the fourth column contains the length, in bytes, of the module (the "8" in the first position signifies that this is the only entry for this module);

000B2C80 in the fifth column contains the starting address of this module (note: Not the entry point address).

Upon examination of the CDE list and the XL queue, we notice a great number of entries which have no corresponding RB. Note, for instance, that all CDE's except the one for module PDS have a ROC-RB pointer of zeroes; this implies that there is no RB for this CDE and XL pair. CDE's with zero RB pointers correspond to LOAD'ed modules;

that is, modules which were the object of a LOAD macro. Recall that for such a module, a LOAD'ed module, the LLE (Load List Element) is the equivalent of the RB. Let us examine the LOAD LIST (see Figure 26).

The list goes across the page rather than up and down. The first entry contains:

NE 00018258 points to the Next Element.

RSP-CDE 020180A8 points to the corresponding CDE in the last six positions; note that 0180A8 is the CDE for IGC0A05A.

The CDE for IGC0A05A points to the proper XL which gives us our starting address and length. We rarely use the LOAD LIST, as the CDE and XL contain most of the needed information.

We now can map out the core used by our job step; for illustrative purposes let us do it. We list the modules in ascending core addresses.

Module	Occupies
PDS	0B2C90 through 0B6700
IGC0A05A	0CD1E0 through 0CD7FC
IGG019AC	0D1918 through 0D1A00
IGG019BB	0FEB40 through 0FEB98
IGG019BA	0FEB98 through 0FED18
IGG019CI	0FED80 through 0FEDF8
IGG019CD	0FEDF8 through 0FF000
IGG019CC	0FF070 through 0FF0C8
IGG019AR	0FF0C8 through 0FF138
IGG019AQ	0FF138 through 0FF1B0
IGG019AK	0FF1B0 through 0FF288

Notice that some modules are contiguous in core. IGG019BB is contiguous with IGG019BA; IGG019CI with IGG019CD, while IGG019CC, IGG019AR, IGG019AQ and IGG019AK are all contiguous.

Upon examination of the list, it becomes clear that modules PDS, IGC0A05A and IGG019AC reside in our region, while the rest lie in the LINK PACK AREA at the extreme high end of core.

In summary, to discern WHEN in our processing we were ABEND'ed we:

* Examine the third most recent RB in the ACTIVE RB QUEUE; this RB had control of the TCB at the time the task was ABEND'ed;

* Map out the region, by examining the CDE queue and XL list for module names, entry points, starting addresses and module lengths.

Who:

Because we know that the third most active, or recent, RB is that which was in control of the TCB at the time of the error does not mean that the error occurred in the corresponding program code. Recall that transfer is made to a LOAD'ed routine via a branch instruction; that is, no supervisor action is involved as in a LINK, ATTACH or XCTL. For this reason, the supervisor is unaware of the transfer, and the RB queue will not reflect this branch. This is particularly true of the I/O access routines, all of which are LOAD'ed modules. In other words, in the case of the common READ or WRITE in a program, the program BRANCHES to the I/O routine, which is a LOAD'ed module. If an error occurs in such a LOAD'ed routine, the RB queue will not reflect that LLE, but the RB for the program which issued the BRANCH instruction.

To determine where the ABEND was issued, in your program or in a LOAD'ed routine (usually a system routine) check the RESUME PSW in your RB (i. e. the third most active RB). If the last six positions (in our case 003716) point to an address outside your program (they do), check the LOAD'ed programs to determine if it falls in one of them; if the address falls within one of these LOAD'ed modules, we may assume that the ABEND was issued in that routine. In our example, the address 003716 does not lie within any of our routines - it is clearly well within the NUCLEUS. The RESUME PSW is the address (plus four) of the ABEND instruction; that is, it gives us the address of the SVC 13. This address (003716) matches the PSW AT ENTRY TO ABEND address; this would follow, as the ABEND macro causes an SVC interrupt. A high proportion of the time the RESUME PSW will point to an address within the NUCLEUS; the ABEND was issued by an interrupt handler. For instance, the I/O interrupt handler or program check interrupt handler determined that the task cannot be continued and must be abnormally terminated. In our example, the time interval for the job step expired, and the timer routine issued the SVC 13 at address 003716.

The other PSW in a PRB is the APSW; this is the last half of the user's old PSW at the time of the last interrupt (the APSW field in an SVRB is discussed later). Since these two PSW's (the APSW and RESUME PSW) usually differ, let us discuss their contents in more detail.

The RESUME PSW is loaded with the user's OLD PSW at the time of the interrupt; that is, when an interrupt is taken, say an I/O interrupt, the I/O interrupt handler stores the program's OLD PSW in the RESUME PSW slot in the module's RB. This PSW will point to the instruction following the last instruction executed. The APSW field in the RB is not touched at this time. The same will be true of a program check interruption or SVC interrupt.

Now, if the interrupt handler decides it must ABEND the task, such as in the case of an uncorrectable I/O error, the interrupt handler branches to the NUCLEUS routine ABTERM which moves the last half of the RESUME PSW, which marks the last instruction executed under control of the RB, into the APSW field of the RB, and MOVES THE ADDRESS OF AN SVC 13 ABEND INSTRUCTION IN THE NUCLEUS INTO THE RESUME PSW ADDRESS FIELD. Then ABTERM returns control to the task. Control is returned, of course, by using the RESUME PSW, which now contains the address of an ABEND SVC 13 in the NUCLEUS. Hence an ABEND is issued; ABEND does not alter the RESUME PSW or APSW fields in the RB. The APSW field will contain, in this instance, the instruction address following the interrupted instruction, and the RESUME PSW field will contain the address of the SVC 13 in the NUCLEUS.

Now let us consider the case where we in our code issue an ABEND SVC 13. Like any interrupt handler, the SVC interrupt handler places the user's OLD PSW (containing, in this case, the address of our SVC 13) into the RESUME PSW slot in our RB. Since WE issued the ABEND, we did not branch to the ABTERM routine, as the interrupt handlers do, which stores the right half of the RESUME PSW into the APSW, this field (the APSW) remains unchanged, and its contents are unpredictable.

Let us list the sequence of events in this extremely important function.

1) An interrupt occurs (I/O, PROGRAM, EXTERNAL or SVC);

2) The appropriate interrupt handler stores the program's OLD PSW, marking the point of interruption, into the RESUME PSW slot in the program's RB;

Then performs one of the following:

3a) For a normal interrupt, with no errors detected, and which was NOT an SVC 13 interrupt, the interrupt handler returns to the program using the RESUME PSW; the APSW is untouched;

or

3b) If an error condition is discovered, the interrupt handler branches to ABTERM which moves the right half of the RESUME PSW to the APSW field, inserts the address of a NUCLEUS SVC 13 instruction into the RESUME PSW, then returns control to the program by restoring the RESUME PSW. This causes an SVC 13, thereby going to the explanation of step 3c below;

or

3c) For an SVC 13 interrupt, calls the ABEND routine. This routine does not touch the RESUME PSW or the APSW. The RESUME PSW remains as in 2 above, pointing to the SVC 13 instruction.

To further reinforce this concept, let us examine the contents of the APSW and RESUME PSW in some specific instances.

* For a PROGRAM CHECK INTERRUPT:
  ** APSW contains the address of the instruction following the error; that is, the user's PSW at the time of the program interrupt.
  ** RESUME PSW contains the address of the SVC 13 in the NUCLEUS. The program check interrupt handler branched to ABTERM in the NUCLEUS which inserted this address in the RESUME PSW, then returned control to the user via this RESUME PSW.

* For an I/O INTERRUPT:
  ** APSW contains the address of the instruction following the last user instruction executed; that is, the instruction at the time of the I/O interrupt.
  ** RESUME PSW contains the address of the SVC 13 in the NUCLEUS. The I/O interrupt handler branched to ABTERM in the NUCLEUS which inserted this address into the RESUME PSW, then returned control to the user via this RESUME PSW.

* For an EXTERNAL INTERRUPT:
  ** APSW contains the address of the instruction following the last instruction executed before the timer interrupt. This address should be within the user's or a LOAD'ed program.
  ** RESUME PSW contains the address of the SVC 13 in the NUCLEUS. The external interrupt handler branched to ABTERM in the NUCLEUS which inserted this address into the RESUME PSW, then returned control to the user via this RESUME PSW.

* For an ABEND issued by the user's code or in a LOAD'ed routine:
  ** APSW - unpredictable, as ABTERM was not entered. ABTERM is the only routine which alters this field.
  ** RESUME PSW contains the address of the instruction following the SVC 13, placed there by the SVC interrupt handler. This should be within your code or LOAD'ed code.

* For an ABEND issued during and by a type II, III or IV supervisor call:
  ** APSW is used for a different purpose in an SVRB. This use will be discussed shortly.
  ** RESUME PSW indicates the address of the instruction following the SVC. This will be an address within the SVC routine; it will not lie in the user's code.

In our example, the APSW points to 0FF162, which is the address following a WAIT instruction (SVC 1), while the RESUME PSW points to the SVC 13 in the NUCLEUS. The module IGG019AQ issued the WAIT SVC; the SVC interrupt handler placed this address into the RESUME PSW. Hence when the event occurred, the program would RESUME at the address following the WAIT, as this is the address in the RESUME PSW. In this case, however, the event never occurred, and the job step timed out. This caused an external interrupt; the timer routine determined that the job was to be abnormally terminated, and branched to ABTERM in the NUCLEUS, which moved the last half of the RESUME PSW, which points to the WAIT, to the APSW, which we see now. It then placed the address of an SVC 13 in the NUCLEUS into the RESUME PSW. This is 003714. ABTERM then returned to the program PDS, via the RESUME PSW. This, of course, caused an SVC 13; the SVC interrupt handler placed PDS's OLD PSW into the RESUME PSW (the 003716 we now see) and we ABEND'ed.

We have, thus far, blithely assumed that our third most active RB was a PRB - indeed, this has been our example. Alas, this is not always true. Many times we call upon a system function, such as OPEN or CLOSE, which is a type II, III or IV SVC, and this routine discovers serious errors, and is unable to continue, necessitating its issuing an ABEND. As type II, III and IV SVC's enter SVRB's into the ACTIVE RB QUEUE, we now have the case of an SVRB as the third most active on the queue, and our PRB as the fourth, or even further removed.

There are two fundamental differences in our approach to an SVC, or SVRB. First, as SVC's are either resident in the NUCLEUS, or operate from the SVC transient areas in the NUCLEUS, we do not have these modules available in a UDUMP, as the NUCLEUS is not dumped. Secondly, the format of an SVRB is different from that of a PRB. This clearly necessitates a slightly different approach.

Let us first examine the fields of an SVRB. We quickly note the absence of an FL-CDE entry; this makes sense, as there are CDE's only for routines in our region or LOAD'ed routines in the LINK PACK AREA, and an SVC is neither. We note also that the other fields are present, such as the RB address, the RESUME PSW and the APSW; but that the APSW looks very odd. The APSW looks "odd" as it is used for a different purpose in an SVRB; it contains the SVC module name for type III and IV SVC's. The name is represented in the hexadecimal form of the zoned decimal name. Let us examine, for example, our two SVRB APSW fields.

The bottom SVRB we know represents ABDUMP. The last two zoned digits in the APSW contain the signed SVC number. ABDUMP is SVC 51; note that the last four hex digits in the APSW are F5C1 which are the characters +51, i. e. the SVC number. Note in the next to last SVRB, the last four hex digits in the APSW are F1C3, which are the signed characters +13, the SVC number of ABEND. The first character (two hex digits) is the sequential number of the load of the SVC, starting with zero (hex F0). Notice that the first character of the ABDUMP APSW is F1 which is the character 1, implying that this is the second load of ABDUMP. In the ABEND SVRB's APSW this is F2, the character 2, implying this is the third load of the ABEND SVC.

As the APSW field of an SVRB is used differently and indeed would be useless if used as in a PRB without the module dumped, we must approach the case of an SVRB as the third most active RB differently. This will be discussed in the section on What.

In summary, then, to answer our question WHO:

* Check if the third most active RB is an SVRB; if so check the APSW, as above, to determine in which system routine we ABEND'ed;

* If the third most active RB is a PRB, check the address portion of the RESUME PSW. If this points to an address higher than the entry point address corresponding to the PRB, check the CDE's and XL's for the region to determine in which module we executed the SVC 13 ABEND. In this case the RESUME PSW pinpoints the SVC 13 in the region's code;

* If the RESUME PSW points to an address lower than our PRB's entry point address, we continue to the next step analysing WHERE we ABENDED.

Where:

When determining the actual instruction and location where the error occurred, we need consider two cases: When the third most recent RB on the active RB queue is a PRB; and the case when this is an SVRB.

We have already discussed in some detail the case of a PRB being the third most recent RB in the preceding section. We examine the RESUME PSW in the PRB; if it contains an address less than any in the region (reflected in the CDE's and XL's), we probably blew in a system routine which issued a branch to the ABTERM routine in the NUCLEUS. The address of the instruction following the last instruction executed should be in the APSW field of the PRB. This is actually

the last half of the user PSW at the time of interrupt, so the instruction length field will be in the first two bits. To find the last instruction, subtract the length of the next instruction from this APSW address value. In our example, the address portion of the APSW is 0FF162; the length is 4 which means a length of one half word (i. e. of bits 0 and 1 of the last half of the PSW, only bit 1 is on, signifying one half word length). Hence, 0FF162 less 2 points to 0FF160; in our case the 0A01 WAIT SVC.

If the RESUME PSW points to an address within our region, we locate the module, as discussed previously, through the CDE's and XL's for the region. This implies that the SVC 13 ABEND was given by code in our region or the LINK PACK AREA. The RESUME PSW should point to the byte following the 0A0D SVC 13 instruction; the APSW is probably meaningless.

The case of an SVRB being the active RB at the time of error requires a slightly different approach. These routines usually issue their own ABEND SVC 13; hence the RESUME PSW should point to the SVC location. Also we have seen that the APSW field of an SVRB has a different use; in type III and IV SVC's it contains the SVC name, as explained in the preceding section, or, if the SVC ABENDED, it contains the SVC's OLD PSW during execution of ABEND or ABTERM; it occassionally contains zeroes. For type II SVC's, the APSW will contain either the OLD PSW during execution of ABEND or ABTERM, or zeroes. Another basic difference to be faced is the fact that the code for the SVC does not appear in our dump; we cannot examine the module to locate the actual point of error. We must try to discern the error from the FUNCTION of the SVC; further techniques in SVC debugging are discussed in the section on data management debugging.

Note, though, that the RESUME PSW for the fourth most active RB should point just past the SVC which called this SVC routine, giving us the SVC number, from which we can determine the SVC's function. This also tells us where we were when we issued the SVC. These two facts are often enough to determine the cause of the error.

In summary, if the third most recent active RB is a PRB;

* Examine the PRB's RESUME PSW address;

* If the RESUME PSW points at an address outside our code, the APSW should point to the last executed instruction in our program;

* If the RESUME PSW points to an address within our code, this should be the address of the ABEND SVC 13. The contents of the APSW are unpredictable.

In the case of the third most active RB being an SVRB:

* The RESUME PSW should point to the SVC routine's ABEND instruction;

* The APSW should contain the SVC routine's PSW during ABEND, or the SVC name;

* The RESUME PSW of the preceding RB should point to the SVC instruction which called this routine;

* Find the function of the SVC and try to surmise the error from this and the location of the SVC in the calling program.

What:

Much as the author would like to be able to espouse a "plan" to pinpoint the actual cause of the error, he of course cannot. What he will attempt to do is supply the programmer (and/or debugger) with some additional bits of information which he might need in order to continue.

Perhaps the most useful information which is desired are the contents of the registers at the time of the error. Let us therefore examine the register SAVE areas, and the SAVE AREA TRACE.

One of the first things incumbent upon a CALL'ed program is SAVE'ing the registers in the SAVE area provided for him by the CALL'ing program, and pointed at by register 13. The CALL'ed program is also responsible for constructing the FORWARD and BACKWARD CHAIN's so as to chain all SAVE areas in the region together. One of the prime reasons for this chain's construction is that the ABDUMP routine will print out a SAVE AREA TRACE of all SAVE areas in the region, thus making it far easier to find the register contents upon entry to and exit from any routine used by the region. Let us look at the SAVE AREA TRACE (see Figures 24c and 24d).

The first line, PDS WAS ENTERED VIA LINK, is self explanatory; following this line is the SAVE AREA TRACE for our job step, reconstructed from the FORWARD and BACKWARD chains. Let us examine the first SAVE area.

SA 0D17B0 marks this as the SAVE area at location 0D17B0; notice that this lies outside any of our modules, but well within the region (reference the region map in the section on WHEN). This is the "highest level" SAVE area, it was set up with a GETMAIN by the INITIATOR. When the INITIATOR issued the XCTL to the program PDS, it had loaded register 13 with the address of this SAVE area. The first program initiated will use this SAVE area; in this case our program PDS. The SAVE that PDS executes will store the previous program's (in this case the INITIATOR's) registers in this area; that is, this SAVE area contains the INITIATOR's registers. This is an

important point: The SAVE area in a program will (if used) contain that program's registers. Some of the fields of interest in a SAVE area are discussed below.

WD1 00000000 are the contents of the first word of the SAVE area.

HSA 00000000 contains the address of the next higher, or previous, SAVE area. As the INITIATOR is the first program in the region, there is none higher, hence the address of zeroes. This is the BACKWARD CHAIN.

LSA 000B2CB0 points to the next lower SAVE area; in this case, the SAVE area within our program, PDS. Note that in our example the next SAVE area is indeed at 000B2CB0; this is the FORWARD CHAIN.

RET 000085DA displays the contents of the standard RETURN register, register 14. This is the address RETURN'ed to by a RETURN macro. The INITIATOR placed in register 14 the address of its resident portion in the NUCLEUS.

EPA 010B2C80 shows the contents of register 15, the ENTRY POINT REGISTER. Notice that 0B2C80 is the entry point of PDS (check the EPA field in the CDE for PDS).

R0 through R12 show registers 0 through 12; register 13's contents, the SAVE area address, is shown immediately following the SA field.

Examining the next SAVE area, SA 0B2CB0, we note that the HSA does indeed point to the next higher SAVE area, at 0D17B0; the LSA has indeterminate contents, as there is no lower SAVE area.

Examining the next SAVE area, SA 0B2CB0, we note that the HSA does indeed point to the next higher SAVE area, at 0D17B0; the LSA has indeterminate contents, as there is no lower SAVE area.

Continuing in our SAVE area trace, we find INTERRUPT AT 003716; this is the address portion of the RESUME PSW in the RB for PDS. PROCEEDING BACK VIA REG 13 indicates that the following SAVE areas are those starting from the lowest (the current value in register 13) to the highest following the BACKWARD CHAIN. In our case these match those in the FORWARD CHAIN; this is not necessarily true, for if the two chains do not correspond neither will the SAVE areas.

Now, the question is, what were my register contents at the time of the error? One is tempted to say,"They are displayed in the lowest SAVE area; that is, in our example, SA 0B2CB0." But this is not always true. Let us examine the usage of this lowest SAVE area.

Who uses this SAVE area? Well, if we use no systems routines, such as the LOAD'ed routines in our example, or if we have not yet progressed that far, the answer is, "No one." In this case, fields

R0 through R12 would contain no meaningful information. Well, what if we do use some of these routines? These routines are branched to, and they do save some registers, but they do not always save ALL the registers; often they save only those registers which they alter. The different routines may save different registers, so our SAVE area may well contain a composite of register contents, some from the time one routine was entered, some from the time another was entered. The point is that we cannot always rely on this SAVE area to contain our registers unless we know that the routine saved them all. And what of the registers the LOAD'ed routine was itself using? Where might they be? To answer these questions we need delve a little more deeply into the system's use of SAVE areas.

When an interrupt occurs, the interrupt handler stuffs the user's registers into a SAVE area in low core. All the interrupt handlers begin in a masked state, so there is no chance of these registers being overlayed. When the interrupt handlers decide what is to be done, they dispose of these registers. If control is to be returned to the user, they are simply restored, and the user continues as before; this is usually the case of an I/O interrupt. If the interrupt handlers find that a service needs to be performed, such as OPEN or CLOSE, or if they have determined that an ABEND need be issued, they first build an SVRB (recall, in our dump we have an SVRB for ABEND and one for ABDUMP). Into this SVRB the interrupt handlers stuff the user's registers they have so long held in lower core. These, then, are our registers; they will be contained in ABEND's SVRB. Let us therefore examine ABEND's SVRB.

Recall that the SVRB for ABEND is the one at 017CC0. Notice the third and fourth lines in this RB indicate RG 0-7 and RG 8-15; these are our registers. Let us note some interesting things about these registers in our example. First, register 13 contains 000B2CB0; this, notice, is the address of the SAVE area in PDS, which is correct, as register 13 is to point to the current SAVE area. Notice that registers 2, 12 and 13 contain addresses very close together and slightly above the entry point of PDS; these could well be base registers. Register 8 contains 000FF138, and the last instruction executed, recall, was just before 00FF162 (note the APSW in the RB for PDS). We determined, recall, that we were ABEND'ed while in routine IGG019AQ (notice that the entry point address in the CDE for IGG019AQ is 0FF138). This module must have a base register; it appears that it was using register 8 (IGG019AQ does indeed use register 8 as a base register).

Observe that at the bottom of the SAVE AREA TRACE is a SAVE area entitled REGS AT ENTRY TO ABEND, and that these match those in the SVRB for ABEND. These are indeed the same, as ABDUMP picks up these registers from this SVRB.

If we ABEND with an SVRB as the third most recent RB in the active RB queue, then the registers belonging to the program which issued the SVC will be in this third most recent RB, and the SVC's register contents will be in ABEND's SVRB. This becomes extremely important when we ABEND in a data management SVC such as OPEN or CLOSE, as will be explained in the next section.

Summary:

In summary, then, the NEWSPAPER APPROACH TO PROGRAMMING ERRORS consists of the following steps:

* Check the completion code in the dump to determine why OS/360 thinks we erred;

* Find which program was in control of the TCB at the time of the ABEND. This will be the third most active RB in the active RB queue;

* Determine the location of the ABEND instruction which caused the dump. The RESUME PSW in the third most recent RB should point to this instruction; this address should match that in the PSW AT ENTRY TO ABEND. Check if this SVC was issued within the code of the region; this may require mapping the region from the CDE's and XL's. If this RESUME PSW points within the region's code, then check which module issued it, and discover from documentation why the routine issues such a code;

* If the RESUME PSW in the third most recent RB in the active RB queue points to an address outside the region's code, it is probably in the NUCLEUS. If this is the case, then the APSW in this RB points to the last instruction executed in the region's code before the interrupt which caused the ABEND to be issued;

* When the third most active RB is an SVRB, then the SVC number is contained in the APSW field of this RB. In this case, the RESUME PSW points to the last instruction executed before the ABEND SVC 13 (indeed this is the ABEND SVC 13);

* The user's register contents at the time of the last interrupt will be displayed in the ABEND SVRB register save area if the third most active RB is a PRB. If the third most active RB is an SVRB, this SVC's register contents will be held in the ABEND SVRB's save area. The program's registers at the time of the SVC which called the type II, III or IV routine will be contained in this SVC's SVRB register save area;

* If the indication is at this time that the error was an I/O error, or occurred in an I/O service, such as OPEN or CLOSE, continue on to the section on the NEWSPAPER APPROACH TO DATA MANAGEMENT ERRORS.

THE NEWSPAPER APPROACH TO DATA MANAGEMENT ERRORS:

Many times when debugging, we determine that the error occurred in, or was caused by, a data management routine. This might mean an improperly declared data set or data definition, bad data, or some logical misuse of data management. In addition to, or in place of, our questions concerning programming errors, we have five basic questions we ask concerning what are generally termed "I/O errors." They are:

1) WHY does OS/360 think we erred;

2) WHO caused the error - i. e. in which data set were we processing at the time of the error;

3) WHEN did the ABEND occur; that is, had we successfully OPEN'ed or CLOSE'd the data set;

4) WHERE were we in the data set; that is, at which record or block;

5) WHAT actually happened with respect to hardware and software indicators.

Let us begin interpreting our dump with respect to these questions.

Why:

As in the case of a programming error, OS will give us a completion code, this time indicating an I/O error. For examples, let us consider the following codes:

COMPLETION CODE 001 tells us that an I/O error occurred; further study will be necessary to determine the cause;

COMPLETION CODE 400 indicates that some or all of the I/O control blocks such as the IOB, DCB, DEB or ECB are incorrect; further study will be necessary to determine which and why;

COMPLETION CODE 213 usually informs us that a data set's DSCB could not be found.

Many times this code alone is enough to allow us to determine the cause of the error. Code 213 usually means that the DD card is in error, pointing to the incorrect DASD volume, or the DSNAME is incorrect. If this is not the case, we proceed to the next step.

Who:

We wish to determine WHO has the error; that is, which data set.

At this point we might wish to map out the I/O control blocks to gain a better idea of the available information.

Recall that the DEB is the communication center for a data set's control blocks; we logically should begin with the DEB queue. Each DEB is nicely displayed (see figure 25), with the storage addresses in the left hand column. To find the beginning of the DEB queue we examine the DEB field in the TCB; this points to the beginning of the first DEB in the DEB queue. Our example's TCB DEB pointer contains 00017064. Look at this location in our first DEB; notice that it contains 000169E8, which is the address of the TCB. A DEB always begins with a pointer back to the TCB; in this way we can verify that we have found the beginning of the DEB. The code in front of this pointer to the TCB is called the DEB prologue, and contains a work space for portions of IOS.

The word following the TCB pointer contains a pointer to the next DEB in the DEB queue; notice that our DEB pointers do point to the beginnings of the DEB's, each DEB beginning with a pointer to the TCB. The last DEB pointer contains a DEB pointer of zeroes to indicate that it is the last DEB on the queue. Five words past the DEB pointer, or six words past the TCB pointer, is the address of the related DCB; two words past the DCB pointer, or eight words past the TCB pointer, is a pointer to the allocated UCB for this DCB. At this point we have the addresses of the UCB's and DCB's of all OPEN'ed data sets (OPEN, recall, builds the DEB's).

Following the DEB queue in the dump is a display of the job step's TIOT. The third column lists the DD card ddnames for the step; this is followed by the TTR for the JFCB corresponding to that DD card. The last column contains the address of the allocated UCB for this DD card. Sometimes we can make the correspondance of DD card to DCB through the UCB pointers in the DEB and TIOT. This cannot always be done, as many data sets may be allocated on a single direct access device, giving many duplicate UCB pointers; in our example, however, all UCB's except the SYSUDUMP and SYNPRINT UCB's are unique, so the relationship may be attempted. We note, then, that the first DEB has a DCB at 0D1FA0 which corresponds to the UCB at 001444, the SYSUDUMP DD card (as the DCB lies outside our program, it is not the SYNPRINT DCB); the second DEB has a DCB at 0B51B0 corresponding to the UCB at 001830 and the WRTANS data set. The third DEB's DCB is at 0B512C with a UCB at 001800 for the READ002 data set, while the fourth DEB has its DCB at 0B5210 for the UCB at 001860 for the ANSFILE data set. The other data sets shown in the TIOT have not been OPEN'ed, as there are no DEB's for them.

DCB's which have not been OPEN'ed may be found by scanning the EBCDIC portion of the dump for the associated ddnames for the DCB, which are contained in bytes hex 28 through 30.

Now, in the DCB's we find another pointer of interest: That of the associated IOB. This pointer is contained at varying locations depending upon the logical organization of the DCB (hex 44 for BSAM, BPAM, QSAM and BDAM; hex 1C for QTAM, BTAM and GAM). Hence we can now find the IOB's for the OPEN'ed data sets.

Looking at the IOB's we find a pointer to the actual channel program to perform the I/O. This pointer is at hex 10 within the IOB.

At this point we have a fairly complete map of the control blocks for the OPEN'ed data sets. The ddnames, DEB's, UCB's, DCB's, IOB's and CCW lists have been located.

Now, what we are after is WHO caused the error; that is, which DCB or data set. There are a number of techniques to determine this. If the interrupt which caused the ABEND occurred during an I/O access routine, examine register 1 in the ABEND SVRB (i. e. the problem program's registers); it may still contain the address of the DCB being operated upon. Register 2 may point to the work area (if any) and 14 and 15 were probably stored in the problem program's SAVE area. Registers 14 and 15 are used as the return and entry registers for I/O routines. If register 1 does not point to a DCB then scan the DCB's for a possible error flag. Hex 31 in the DCB contains the error flags; if bit zero, or bits zero and one are on, then the data set represented by the DCB encountered an error. This scanning of DCB's is usually the technique used to detect an error DCB for an error discovered at I/O interrupt time, such as a data check etc.

If the third most recent RB in the active RB queue is an SVRB for OPEN (SVC 19), then to identify the data set which was being OPEN'ed at the time of the ABEND, we should know the register usage of OPEN; these registers will be contained in the register save area of the ABEND SVRB. OPEN uses register 2 to point to the DCB being OPEN'ed; register 3 is used as the base register, while 4 points to the OPEN work area. Register 5 is a pointer to the parameter list passed to OPEN, while 7 contains the address of the current entry in the parameter list. The address of the TIOT is contained in register 9, while the UCB and DEB addresses are contained in registers 10 and 11 respectively. From these register contents we are usually able to tell which DCB we were trying to OPEN.

In summary, to determine in which data set we encountered the error:

* Map out the DCB's and IOB's from the DEB queue;
* Check register 1 of the problem program at the time of ABEND - it may still point to the sought after DCB;
* Check byte 31 in the DCB's for error flags;
* If the error occurred during OPEN, register 2 of OPEN points to the DCB being OPEN'ed.

When:

When determining when a data management error occurred, we are trying to determine when with respect to OPEN, CLOSE, READ or WRITE. The actual data output, if any, will aid us in determining this. For more concrete indication, however, we examine the DCB. Hex 28 BEFORE OPEN is performed will contain the DD name; OPEN overlays this field with other information. If this field has been overlaid, check the byte at hex 30 in the DCB; bit 3 is set to one if an OPEN has been successfully performed. If OPEN was successful, we can examine this byte to determine if the last I/O performed was a READ or WRITE (bit zero) or a READ BACKWARD (bit one), or whether a tape mark has been sensed (bit five). Hex 31 in the DCB tells us of any error conditions. We also have a hint as to the status of the data set by the DEB; OPEN'ed data sets have a DEB, others have not.

Where:

When determining WHERE we blew in processing a data set, we wish to know which record we were processing at the time of the error. Hex 4C in the DCB points to the current or next logical record in the I/O buffer. Block count for tape processing is also contained in the DCB.

What:

WHAT caused the error depends upon a great many things; the author can only supply additional information to allow the debugger to determine the actual cause of the ABEND.

If the error was caused by a logical inconsistency in the data definition, recall that the DCB is the final repository of the logical data definition, containing such things as LRECL, DSORG, BLKSIZE, etc.

Perhaps as important is the pointer in the DCB to the IOB. The IOB contains a wealth of information as to the physical status of the data set. In the IOB are contained such things as the last sense bytes read. Recall that sense bytes are read from a control unit after the I/O supervisor determines that an I/O error has occurred. The CSW (Channel Status Word) is stored after each I/O interrupt; this tells whether the I/O was successful, and the reason for the interrupt (channel end, device end, unit check etc.). If the CSW indicates an error, IOS issues a sense command which is sent to the appropriate control unit. From two to six bytes of detailed information are read into storage, indicating the specific cause of the error (see the Programmer's Guide to Debugging SRL for a chart of sense byte meanings). The first two of these sense bytes are placed

into the IOB (all the sense bytes are placed into the appropriate UCB).

The second half of the CSW is also stored in the IOB; this tells the reason for the interrupt, as mentioned before. Perhaps as important is the residual byte count shown in the CSW byte field. This is the difference between the number of bytes which where to be manipulated according to the CCW, and the actual number of bytes acted upon. This is extremely useful in the case of a wrong length record error, or in determining the end of a variable length or undefined record.

Byte four in the IOB contains the completion code from the ECB indicating whether the I/O was successful; this is the only place which will indicate a DEB violation (the attempt at I/O outside the limits prescribed in the DEB).

Summary:

In summary, then, the NEWSPAPER APPROACH TO DATA MANAGEMENT ERRORS with a dump consists of the following steps:

* Check the completion code in the dump to determine why OS/360 thinks we erred;
* Find the DCB which identifies the data set in error, by the contents of register one, or scanning the DCB's for error flags;
* If the error occurred during OPEN, check register two of OPEN for the DCB address;
* Check the DCB for OPEN flags, READ/WRITE flags, tape mark etc.;
* Look at the present or next data record for data errors;
* From the DCB find the IOB for:
    ** Completion code from the ECB;
    ** Last half of the CSW;
    ** Sense bytes;
    ** Actual CCW's.

APPENDIX A

EVENT SYNCHRONIZATION:

When many events in a computer are taking place simultaneously (such as channel programs, etc.), some technique must be found to synchronize these events, or else utter chaos reigns. In MVT, in addition to many events occurring concurrently in the computer hardware, different programs and routines are also executing asynchronously in storage during multi-programming.

The reader will recall that the different events taking place concurrently in the S/360 hardware are synchronized through the interrupt mechanizm. For example, when the CPU wishes an asynchronous, or concurrent, event to take place, such as I/O, it communicates this desire to the channel via a START I/O command. The channel immediately tells the CPU whether the command was accepted and started by supplying the CPU with a condition code in the current PSW. Assuming the command was started successfully, the CPU and channel continue with their different and divergent tasks, with no relationship to or knowledge of the other's progress. When the channel finishes its assigned task, it notifies the CPU of its completion by causing an I/O interrupt, and storing a channel status word (CSW) containing the reason for the interrupt. In this way the concurrent hardware functions have been synchronized.

In an OS/360 multi-programming environment we have, in addition to the hardware concurrency, programming concurrency. That is, we have different programs in different regions executing concurrently; between regions, however, there is little need for communication, as these are independent jobs. The user may, however, multi-task within his own region - this will demand synchronization of these concurrent tasks. The same is true of I/O buffering; IOS must have a way of communicating to the program that the channel has completed its task, and the outcome of this task. The communication technique for both of these needs is handled identically, through the ECB, and the WAIT and POST macros.

Event synchronization takes place through the medium of the ECB (Event Control Block). This is a full word, bit zero being the wait bit, bit one the post or completion bit, bits 2 through 31 containing the completion code or RB address.

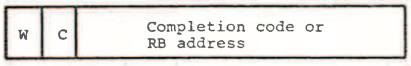

Figure 27: The Event Control Block.

When a program issues a request for a service that will require synchronization, such as a READ, it supplies the asynchronous routine with the address of one or more ECB's which have been set to zeroes. The ECB now becomes associated with this asynchronous event.

Now, when synchronization becomes necessary, a WAIT (SVC 1) macro is issued by the program; this macro points to the ECB's upon which to be waited. The WAIT routine then examines the ECB; if the completion flag is on, the event has occurred, and WAIT immediately returns control to the instruction following the WAIT macro in the program. If the completion flag is not on, the event has not occurred; the WAIT routine then turns the wait flag on, places the address of the WAIT'ing program's RB into the completion code field of the ECB, and places the program into the WAIT state. That is, a task switch is performed, the CPU being given to the next task which is ready to continue processing. The WAIT'ing program will remain in this state until the event occurs or the task is cancelled.

When the awaited event occurs, such as the completion of a physical I/O event, the WAIT'ing program must be notified. This will be done through the ECB. When the asynchronous routine, such as the I/O routine or ATTACH'ed task, determines that an event has occurred which requires synchronization, it issues a POST SVC for synchronization and notification of the WAIT'ing routine. The POST macro points to the ECB to be POST'ed.

POST examines the ECB; if the wait flag is off, which implies that the event has not had a WAIT issued for it, the complete bit is set on, the completion code is placed in the ECB, and control is returned to the next ready task. In this case the subsequent WAIT issued upon this event will be an effective NO-OP, as explained previously. If the wait flag in the ECB is on, implying that some program is WAIT'ing on this event, the completion code is placed in the ECB, and the task which was WAIT'ing is placed in the ready queue for dispatching. The program to be placed in the ready queue was found from the RB address which was placed in the ECB by the WAIT routine. When the task is dispatched, the RESUME PSW will be used; the SVC interrupt handler placed the OLD PSW in this slot at the time the WAIT SVC was issued. This will contain the address of the instruction following the WAIT SVC 1. Hence processing will continue at the address following the WAIT SVC.


```
JOB / 21854          STEP EXEC1           TIME 121442   DATE  3302                                  PAGE    1

COMPLETION CODE      SYSTEM = 322

PSW AT ENTRY TO ABEND  FFE5000D 40003716

TCB  0169E8  RBP 00017080   PIF     00000000   DEB 00017064   TIO 00016080   CMP   80322000   TRN 00000000
             MSS 020195A8   PK-FLG  E0854409   FLG 0000181B   LLS 000180A0   JLB   00000000   JPQ 000180A8
             FSA 01001780   TCB     00017558   TME 00000000   JST 000169F8   NTC   00000000   OTC 00016300
             LTC 00000000   TOE     00000000   ECB 0001BC00   STA 00000000   D-PQE 0001A6C8   SQS 00016648

ACTIVE RBS

PRB  018C88  RESV   00000000   APSW   400FF162   WC-SZ-STAB 01040082   FL-CDE 000180D8   PSW FFE5000D 40003716
             Q/TTR  00000000   WT-LNK 000169F8

SVRB 017CC0  TAB-LN 000803A8   APSW   F2F0F1C3   WC-SZ-STAB 0012D002   TQN   00000000   PSW 00040033 50002CBA
             Q/TTR  00000D13   WT-LNK FF018C88
             RG 0-7    00000001   000D14EC   000B512C   000D14E8   000CFFFF   000CEFFF   00000051   00000028
             RG 8-15   000FF138   00000000   000D0AEE   000D0995   00084C86   00082CH0   40001900   600049DA
             EXTSA     000021BE   000D1FA0   20000000   000D1DF8   FF030000   00017D3C   00017044   E2E8E2C9
                       C5C1F0F1   C9C5C1E0   C1C2C5D5   C4403110

SVRB 0170B0  TAB-LN 001803A3   APSW   F1F0F5C1   WC-SZ-STAB 00120002   TQN   00000000   PSW FF040001 400FEB64
             Q/TTR  00000027   WT-LNK 00017CC0
             RG 0-7    00000000   00017020   800029CC   00003640   000169F8   00017CC0   00018000   00000000
             RG 8-15   000169F8   40002942   000169F8   000D1FA0   000160B0   00017044   40002C2A   00000000
             EXTSA     000F0C00   E3D6C340   00017124   E2E8E2C3   E3D3C740   E2E8E2C3   E3D3C740   4780C1A6
                       00FFFFF8   02001000   700050F0   C4A205EF

LOAD LIST

      NE 00018258   RSP-CDE 020180A8        NE 00018260   RSP-CDE 0101A3A0        NE 000184A8   RSP-CDE 0101A2A0
      NE 00018970   RSP-CDE 0101A270        NE 00018F58   RSP-CDE 0101A430        NE 00018E80   RSP-CDE 0101A3D0
      NE 000192E0   RSP-CDE 0301A360        NE 00019410   RSP-CDE 0201A300        NE 00019470   RSP-CDE 02017270
      NE 00000000   RSP-CDE 0201A400

CDE

   0180D8    ATR1 08   NCDE 000000   ROC-RB 00018C88   NM PDS        USE 01   EPA 082C80   ATR2 20   XL/MJ 018E18
   0180A8    ATR1 30   NCDE 017270   ROC-RB 00000000   NM IGC0A05A   USE 02   EPA 0CD1E0   ATR2 20   XL/MJ 018248
   01A3A0    ATR1 B8   NCDE 01A3D0   ROC-RB 00000000   NM IGG019CD   USE 03   EPA 0FEDE8   ATR2 20   XL/MJ 01A390
   01A2A0    ATR1 B9   NCDE 01A2D0   ROC-RB 00000000   NM IGG019BA   USE 03   EPA 0FF898   ATR2 20   XL/MJ 01A290
   01A270    ATR1 B8   NCDE 01A2A0   ROC-RB 00000000   NM IGG019BB   USE 03   EPA 0FF840   ATR2 20   XL/MJ 01A260
   01A430    ATR1 B0   NCDE 01A460   ROC-RB 00000000   NM IGG019AK   USE 02   EPA 0FF100   ATR2 20   XL/MJ 01A420
   01A3D0    ATR1 B0   NCDE 01A400   ROC-RB 00000000   NM IGG019AR   USE 06   EPA 0FF0C8   ATR2 20   XL/MJ 01A3C0
   01A360    ATR1 B1   NCDE 01A3A0   ROC-RB 00000000   NM IGG019CC   USE 06   EPA 0FF070   ATR2 20   XL/MJ 01A350
   01A300    ATR1 B0   NCDE 01A330   ROC-RB 00000000   NM IGG019CI   USE 03   EPA 0FF000   ATR2 20   XL/MJ 01A2F0
   017270    ATR1 30   NCDE 0180D8   ROC-RB 00000000   NM IGG019AC   USE 02   EPA 0D1918   ATR2 20   XL/MJ 017260
   01A400    ATR1 B0   NCDE 01A430   ROC-RB 00000000   NM IGG019AQ   USE 04   EPA 0FF138   ATR2 20   XL/MJ 01A3F0

XL                                LN        ADR        LN        ADR        LN        ADR
```

Figure 22a: MVT SYSUDUMP - Page 1 of 4.

```
018E18   SZ 00000010   NO 00000001      A0003B80    00082C80
018248   SZ 00000010   NO 00000001      A000061C    000CD1E0
01A390   SZ 00000010   NO 00000001      80000208    000FFDF8
01A2D0   SZ 00000010   NO 00000001      80000180    000FFB98
01A260   SZ 00000010   NO 00000001      80000058    000FEA40
01A470   SZ 00000010   NO 00000001      80000008    000FF1D0
01A3C0   SZ 00000010   NO 00000001      80000070    000FF0C8
01A350   SZ 00000010   NO 00000001      80000058    000FF070
01A2F0   SZ 00000010   NO 00000001      A0000078    000FFD80
017260   SZ 00000010   NO 00000001      800000E8    00001918
01A3F0   SZ 00000010   NO 00000001      A0000078    000FF138

DEB

017040   00000C2F 00000C2E 00000C2E 00000C2E    00000C2E 00000000 32000000 00002AE0   *.....2.....*
017060   0E000000 000169E8 04016E24 88000000    8F000000 01000000 18000000 EF001FA0   *.....Y.....*
017080   04017040 10001444 00000000 00000000    00130028 0001F0F4 C2C2C2C1 C3C41800   *... ..........04BBBACD..*
0170A0   0A0218E7 07FF0000 0083D600 C612211F                                         *...X.....O.F.....205A.....*

DEB

016E00   00000C2E 00000C2F 00000C2E 00000C2E    00000C2F 00000000 00000000 00019BF0   *.....*
016E20   00000000 0301A9E8 04016F94 C8000000    0F000000 01000000 18000000 FF085180   *.....Y....H.....*
016E40   02016E00 33001830 00010000 C1D9C1D2    C3C3C002 00000000 FF040000 00005A34   *.....ARAKCC.....*
016E60   00000000 00016F68                                                           *.....8.....*

DEB

016E60                                          00000C2F 00000C2E 00000C2E 000FED80   *.....8.....*
016E80   00000C2E 00000000 00000000 00009BE0    00000000 0401A9E8 04017624 48000000   *.....Y....H.....*
016EA0   00000000 01000000 18000000 EF085120    02016E70 33001800 00010000 C1D8C1C3   *.....AQAC*
016EC0   C3C9C3C3 00000000 00000000 00000000    000B1A10 00000000                     *CICC.....*

DEB

017600   00000C2E 00000C2E 00000C2F 000FED80    00000C2F 00000000 00000000 00009BE0   *.....*
017620   00000000 04016F88 04000000 C8000000    00000000 01FFFFFF 18000000 EF085210   *.....Y....H.....*
017640   02017600 33001A60 0001Q000 C1D9C1C3    C3C9C3C3 00017978 00000000 04FF1294   *.....AQACCICC.....*
017660   0000F3D3 C7017970                                                           *..TLG.....*

TIOT   JOB  A0001854    STEP EXEC1
       DD             14040100    PGM=*.DD      001E0C00    A0001204
       DD             14040100    SYSUDUMP      000F3700    A0001444
       DD             14040100    SYNFILE       00123600    A0001794
       DD             14040100    SYNPRINT      000E1900    A0001444
       DD             14020100    READ002       00123C00    A0001800
       DD             14040100    WRITAYS       001D2F00    A0001830
       DD             14050100    ANSFILE       001D3100    A0001860
       DD             14050100    WRITOOI       001D3500    A0001890
```



```
       DD             14040100    WRITEIN       00210600    A0001444
       DD             14040100    READCRD       001D3700    A0001404

MSS              ************ SPQE ************      ***************** DQE *****************      ******** FQE ********
                 FLGS  NSPQE     SPID     DQE          BLK       FQE       LN        NDQE          NFQE        LN
       0195A8    00    019680    251      018DC8       00082800  00082800  00004000  00000000      00000000    00000480
       019680    80    01A6C8    000      017960
       017960    60    000000    000      018DE0       00001000  00001768  00000800  00018490      00001000    00000048
                                                       00000000  00000000  00001000  00017190      00000000    00000430
                                                       000CF000  000CF000  00001000  00017F88      00000000    00000030
                                                       000CF000  000CE000  00001000  00017250      00000000    00000678
                                                       000CA800  000CA800  00000800  00000000      00000000    00000C9
       01A6C8    40    000000    252      01AFC8       00001800  00001A00  00000800  00017CA0      00001A00    00000540
                                                       000CD000  000CD000  00000800  00017968      00000000    00000118
                                                       000CC000  000CCA00  00000800  00000000      00000000    000001F0
                                                                                                   00000000    00000140

D-PQE  0001A6C8   FIRST 00018E88   LAST 0001AEA8
PQE  018F88   FFB 00086800   LFB 00086800   NPQ 00000000   PPQ 00000000
              TCB 00016300   RSI 0001F800   RAD 00082800   FLG 00

FRQE 086800   NFB 00018E88   PFB 00018E88   SZ 00016000

QCB TRACE

MAJ 01A6F8   NMAJ 00018A00   PMAJ 0000A2C0   PMIN 00019480   NM  SYSDSN

MIN 019480   FQEL 00019700   PMIN 0001A6F8   NMIN 00019070   NM FF  CTSRCH

             NQEL 00000000   PQEL 00019480   TCB  00016300   SVRB 0001AF20

MIN 019070   FQEL 00019440   PMIN 00019480   NMIN 00018FA8   NM FF  SYS1.SMRTLIB

             NQEL 00000000   PQEL 80019070   TCB  00016300   SVRB 00016F20

MIN 018FA8   FQEL 00019200   PMIN 00019070   NMIN 00018FE8   NM FF  SYS1.LINKLIB

             NQEL 00000000   PQEL 8001AEA8   TCB  00016300   SVRB 00016F20

MAJ 017978   NMAJ 00000000   PMAJ 0001AA00   FMIN 00017758   NM  SYS1EA01

MIN 017758   FQEL 00017CA0   PMIN 00017978   NMIN 00000000   NM F0  IEA

             NQEL 00000000   PQEL 00017758   TCB  000169E8   SVRB 00016F68

SAVE AREA TRACE

PDS      WAS ENTERED VIA LINK

SA   001780  WD1 00000000   HSA 00000000   LSA 00082C80   RET 000085DA   EPA 01087C80   RO  FD000006
```

Figure 22b: MVT SYSUDUMP - Page 2 of 4

Figure 22c: MVT SYSUDUMP - Page 3 of 4

```
                                                                                                PAGE 0
          R1  000D17F8   R2  0001ACC8   R3  5001ACDC   R4  00015EC8   R5  00016300   R6  00016078
          R7  00017008   R8  0001ACB0   R9  00008000   R10 00000114   R11 00015EC8   R12 600FF41A

SA  082C80  WD1 00070003   HSA 000D1780   LSA 00080003   RET 4C0020A8   EPA 00090003   R0  50084F76
            R1  120D191A   R2  00082D69   R3  0008512C   R4  00088944   R5  40082C86   R6  00083CA6
            R7  00082FDB   R8  0000005C   R9  000830A0   R10 00082E9B   R11 400D19D0   R12 000FF138

INTERRUPT AT 003716

PROCEEDING BACK VIA REG 13

SA  082C80  WD1 00070003   HSA 000D1780   LSA 00080003   RET 4C0020A8   EPA 00090003   R0  50084F76
            R1  120D191A   R2  00082D69   R3  0008512C   R4  00088944   R5  40082CA6   R6  00083CA6
            R7  00082FDB   R8  0000005C   R9  000830A0   R10 00082F9A   R11 400D19D0   R12 000FF138

PDS       HAS ENTERED VIA LINK

SA  0D1780  WD1 00000000   HSA 00000000   LSA 00082C80   RET 000085DA   EPA 01082C80   R0  FD000006
            R1  000D17F8   R2  0001ACCA   R3  5001ACDC   R4  00015ECA   R5  00016300   R6  00016070
            R7  00017008   R8  00018C80   R9  000080D0   R10 00000114   R11 00015EC8   R12 600FF41A

REGS AT ENTRY TO ABEND

    FLTR 0-6      000014440001ACB4      000000010001ACEC                00016FA000000000      0009000907C4E240

    REGS 0-7      00000001   000D14EC   0008512C   000D14E8             000CEFFF   000CEFFF   00000051   00000028
    REGS 8-15     000FF138   00000000   000D0AFF   00000995             00084C86   00082CA0   400D19D0   600049DA

LOAD MODULE   PDS

082C80   90ECD00C 05305000 302E18CD 41D0302A   50DC0008 07005840 301E58C0 302247F0   *........................ ........0*
082CA0   44F20000 00083C86 00084C86 4C0020SC   00070003 000D1780 00080003 4C0020A8   *.?................................*
082CC0   00090003 50084F76 120D191A 00082D69   0008512C 00088944 40082CA6 00083CA6   *.......................... .......*
082CE0   00082FDB 0000005C 000830A0 00082F90   400D19D0 000FF138 00110003 5C00205A   *................. .....1.........*
082D00   F0F8F140 40E9D6D7 C8C1E460 F0F0F3F8   60F1F0F8 F8404040 40404040 40404040   *081  ZOKHA4.0038.1888             *
082D20   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040   *                                  *
082D40   40404040 40404040 40404040 40404040   40404040 40F0F0F0 F0F2F040 40002208   *                        000020 ...0*
082D60   400022DC F0F6F140 40E9C1D2 C8C1E860   F0F0F2F3 60F1F1F2 F440F2F1 40F2D603   *....061  ZAKHA8.0023.1124 21 SOL*
082D80   E5C5D5E3 40C5E7E3 D9C1C3E3 C9D6D540   D6C640C1 40C3C8C5 D3C1E3C5 40C3D6D4   *VENT EXTRACTION OF A CHELATE COM*
082DA0   D7D6E4D5 C440D6C6 40C140E3 D9C1C3C5   40404040 40404040 404AF0F0 F0F2F534   *POUND OF A TRACE          .00025.*
082DC0   00150015 4000374C 08AA5000 000008AA   68088874 088AC008 8AF80000 00088994   *..........................Y......*
082DE0   088A4000 000008A9 00000000 00802C00   0000088C 7C0A885C 00000008 897C0889   *.. ................................*
082E00   F4008BA0 00000000 00000000 00000000   00000000 00000000 00000000 0D000000   *4.................................*
082E20   00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000   *..................................*
082E40   00000011 AA000000 00007FFF C9B0D20F   C0C1C547 D207C602 00098978 00010000   *............I.K..II.K.F..........*
082E60   00000007 00000009 00184040 40400000   00000000 0000002C F14021C6 47B0C640   *..........    ..........1 .F..F   *
082E80   40404006 00088BA0 00088C8C 00088D0F   00088978 000CF000 4790C6F9 F9F9F9F9   *   1..................0...F99999*
082EA0   F9F9F9F9 F9F9F9F9 F9F9F9F9 00C9050A   F4C9D9F8 40D50640 40404040 40C8C1E2   *999999999999.INQUIRY NO.     HAS*
082EC0   40D50640 C8C5C1C4 C5D940C3 C1D9C44B   C3E3E2D9 C3C4C1E3 C5E3C5D9 D4E24096   * NO HEADER CARD.CTSRCDATETERMS O*
082EE0   D540E3C1 D7C5C510 47B0C3E0 5030C504   58B020C4 47F0C5B4 5930C4FC 00002000   *N TAPE...C...F....D.0F...D.....*
082F00   00004000 07000790 00088BA0 00080808   000D1658 F0F8F1F0 F10F0009 59F0F030   *.. ............0....08101....00.*
082F20   00000000 0057001F F0F8F140 40E9D7C3   C6C1E760 F0F0F5F9 60F0F2F9 F9404040   *.........081  ZPCFAX.0059.0299   *
```

Figure 22d: MVT SYSUDUMP - Page 4 of 4.

56

```
JOB /  11854        STEP EXEC1           TIME 121442   DAT'  1302                                   PAGE    1

COMPLETION CODE      SYSTEM = 322

PSW AT ENTRY TO ABEND   FFE5000D 40003716

TCB  0169E8  RBP 00017080   PIE     00000000   DEB 00017064   TIO 000160B0   CMP  80322000   TRN 00000000
             MSS 020195A8   PK-FLG  F0854409   FLG 00001818   LLS 000180A0   JLB  00000000   JPQ 000180A8
             FSA 010D1780   TCB     00017558   TME 00000000   JST 000169F8   NTC  00000000   OTC 00016300
             LTC 00000000   TOE     00000000   ECB 00018CD0   STA 00000000   D-PQE 0001A6C8  SQS 00016698

ACTIVE RBS

PRB  01AC88  RESV    00000000   APSW   400FF162   WC-SZ-STAB 01040082   FL-CDE 000180D8   PSW FFE5000D 40003716
             Q/TTR   00000000   WT-LNK 000169F8

SVRB 017CC0  TAB-LN 000803A8   APSW   F2F0F1C3   WC-SZ-STAB 0012D002   TQN    00000000   PSW 00040033 50002C8A
             Q/TTR  0000001?   WT-LNK FF01AC88
             RG 0-7    00000001   000D14EC   0008512C   000D14E8   000CEFFF   000CEFFF   00000051   00000028
             RG 8-15   000FF138   00000000   000D0AEE   000D0995   00084C86   00082CA0   400D19D0   600049DA
             EXTSA     000021BE   000D1FA0   20000000   000D1DFA   FF030000   0001703C   00017044   E2C8E2C9
                       C5C1F0F1   C9C5C1E8   C1C2C5D5   C4403110

SVRB 017080  TAB-LN 001803A3   APSW   F1F0F5C1   WC-SZ-STAB 0012D002   TQN    00000000   PSW FF040001 400FE864
             Q/TTR  00000027   WT-LNK 00017CC0
             RG 0-7    00000000   00017020   800029CC   00003640   000169F8   00017CC0   000180D8   00000000
             RG 8-15   000169F8   40002942   000169F8   000D1FA0   000160B0   00017044   40002C2A   00000000
             EXTSA     000E0C00   E306C340   00017124   E2E8E2C3   E3D3C740   E2E8E2C3   E3D3C740   4780C1A6
                       00FFFFF8   D2001000   70005AF0   C4A205EF

LOAD LIST

        NE 00018258   RSP-CDE 020180A8         NE 00018260   RSP-CDE 0101A3A0         NE 000184A8   RSP-CDE 0101A2A0
        NE 00018970   RSP-CDE 0101A270         NE 00018F58   RSP-CDE 0101A430         NE 00010LB0   RSP-CDE 0101A3D0
        NE 000192E8   RSP-CDE 0301A360         NE 00019410   RSP-CDE 0201A300         NE 00019470   RSP-CDE 02017270
        NE 00000000   RSP-CDE 0201A400

CDE

     0180D8     ATR1 08   NCDE 000000   ROC-RB 00018C88   NM PDS        USE 01   EPA 082C80   ATR2 20   XL/MJ 018E18
     0180A8     ATR1 30   NCDE 017270   ROC-RB 00000000   NM IGC0A05A   USE 02   EPA 0CD1E0   ATR2 20   XL/MJ 018248
     01A3A0     ATR1 B8   NCDE 01A3D0   ROC-RB 00000000   NM IGG019CD   USE 03   EPA 0FFDE8   ATR2 20   XL/MJ 01A390
     01A2A0     ATR1 B9   NCDE 01A2D0   ROC-RB 00000000   NM IGG019BA   USE 03   EPA 0FF898   ATR2 20   XL/MJ 01A290
     01A270     ATR1 B8   NCDE 01A2A0   ROC-RB 00000000   NM IGG019BB   USE 03   EPA 0FFB40   ATR2 20   XL/MJ 01A260
     01A430     ATR1 80   NCDE 01A460   ROC-RB 00000000   NM IGG019AK   USE 02   EPA 0FF1B0   ATR2 20   XL/MJ 01A420
     01A3D0     ATR1 80   NCDE 01A400   ROC-RB 00000000   NM IGG019AR   USE 06   EPA 0FF0C8   ATR2 20   XL/MJ 01A3C0
     01A360     ATR1 81   NCDE 01A3A0   ROC-RB 00000000   NM IGG019CC   USE 06   EPA 0FF070   ATR2 20   XL/MJ 01A350
     01A300     ATR1 80   NCDE 01A330   ROC-RB 00000000   NM IGG019C1   USE 03   EPA 0FFD80   ATR2 20   XL/MJ 01A2F0
     017270     ATR1 30   NCDE 0180D8   ROC-RB 00000000   NM IGG019AC   USE 02   EPA 001910   ATR2 20   XL/MJ 017260
     01A400     ATR1 80   NCDE 01A430   ROC-RB 00000000   NM IGG019AQ   USE 04   EPA 0FF138   ATR2 20   XL/MJ 01A3F0

XL                                   LN        ADR        LN        ADR        LN        ADR
```

Figure 23: MVT SYSUDUMP - The TCB.

57

```
JOB / 11854          STEP EXEC1          TIME 121442   DAT! 1307                                        PAGE    1

COMPLETION CODE      SYSTEM = 322

PSW AT ENTRY TO ABEND   FFE50000 40003716

TCB  0169E8   RBP 00017080    PIF    00000000   DEB 00017064   TIO 0001A080   CMP  80322000   TRN 00000000
              MSS 020195A8    PK-FLG E0854409   FLG 0001B1B    LLS 000180A0   JLB  00000000   JPQ 000180A8
              FSA 010D1780    TCB    00017558   TME 00000000   JST 000169E8   NTC  00000000   OTC 00016300
              LTC 00000000    IOE    00000000   ECB 00018CD0   STA 00000000   D-PQE 0001A6C8  SQS 00016688

ACTIVE RBS

PRB  018C88   RESV    00000000   APSW   400FF162   WC-SZ-STAB 01040082   FL-CDE 00018008   PSW FFE50000 40003716
              Q/TTR   00000000   WT-LNK 000169E8

SVRB 017CC0   TAB-LN  000803AB   APSW   F2F0F1C3   WC-SZ-STAB 00120002   TQN    00000000   PSW 00040033 50002C8A
              Q/TTR   00000013   WT-LNK FF018C88
              RG 0-7    00000001   000D14EC   00005120   000014E8   000CEFFF   000CEFFF   00000051   00000028
              RG 8-15   000FF138   00000000   00000AEE   00000995   00084C86   00082C80   400D19D0   600049DA
              EXTSA     000021BE   000D1FA0   20000000   000D1DF8   FF030000   00017D3C   00017D44   E2E8E2C9
                        C6C1F0F1   C9C5C1E8   C1C2C5D5   C4403110

SVRB 0170B0   TAB-LN  00180A03   APSW   E1F0F5C1   WC-SZ-STAB 00120002   TQN    00000000   PSW FF040001 400FE864
              Q/TTR   00000027   WT-LNK 00017CC0
              RG 0-7    00000000   00017020   800079CC   00003640   000169E8   00017CC0   00018008   00000000
              RG 8-15   000169F8   40002942   000169F8   000D1FA0   00016080   00017D44   40002C7A   00000000
              EXTSA     000F0C00   E3D6C340   00017124   E2E8E2C3   E3D3C740   E2E8E2C3   F3D3C740   4780C1A6
                        00FFFFF8   02001000   700050F0   C4A205EF

LOAD LIST

     NE 00018258   RSP-CDE 020180A8        NE 00018260   RSP-CDE 0101A3A0        NE 000184A8   RSP-CDE 0101A2A0
     NE 00018970   RSP-CDE 0101A270        NE 00018F58   RSP-CDE 0101A430        NE 00018E80   RSP-CDE 0101A300
     NE 000192E8   RSP-CDE 0301A360        NE 00019410   RSP-CDE 0201A300        NE 00019470   RSP-CDE 02017270
     NE 00000000   RSP-CDE 0201A400

CDE

     018008    ATR1 08   NCDE 000000   ROC-RB 00018C88   NM PDS        USE 01   EPA 082C80   ATR2 20   XL/MJ 018E18
     0180A8    ATR1 30   NCDE 017270   ROC-RB 00000000   NM IGC0A05A   USE 02   EPA 0CD1E0   ATR2 20   XL/MJ 018748
     01A3A0    ATR1 B8   NCDE 01A300   ROC-RB 00000000   NM IGG019CD   USE 03   EPA 0FEDF8   ATR2 20   XL/MJ 01A390
     01A2A0    ATR1 B9   NCDE 01A200   ROC-RB 00000000   NM IGG019BA   USE 03   EPA 0FFB98   ATR2 20   XL/MJ 01A290
     01A270    ATR1 B8   NCDE 01A2A0   ROC-RB 00000000   NM IGG019BB   USE 03   EPA 0FFB40   ATR2 20   XL/MJ 01A260
     01A430    ATR1 80   NCDE 01A460   ROC-RB 00000000   NM IGG019AK   USE 02   EPA 0FF1B0   ATR2 20   XL/MJ 01A420
     01A3D0    ATR1 80   NCDE 01A400   ROC-RB 00000000   NM IGG019AR   USE 06   EPA 0FF0C8   ATR2 20   XL/MJ 01A3C0
     01A360    ATR1 81   NCDE 01A3A0   ROC-RB 00000000   NM IGG019CC   USE 06   EPA 0FF070   ATR2 20   XL/MJ 01A350
     01A300    ATR1 80   NCDE 01A330   ROC-RB 00000000   NM IGG019C1   USE 03   EPA 0FFDB0   ATR2 20   XL/MJ 01A2F0
     017270    ATR1 30   NCDE 018008   ROC-RB 00000000   NM IGG019AC   USE 02   EPA 0D1918   ATR2 20   XL/MJ 017260
     01A400    ATR1 80   NCDE 01A430   ROC-RB 00000000   NM IGG019A0   USE 04   EPA 0FF138   ATR2 20   XL/MJ 01A3F0

XL                                    LN          ADR          LN          ADR          LN          ADR
```

To XL

Figure 24a: MVT SYSUDUMP - The Active RB Queue and CDE List.

58

```
                                                                                                   PAGE 0

     018E18    SZ 00000010    NO 00000001        80003880    00002C80
     018748    SZ 00000010    NO 00000001        8000061C    000CD1E0
     01A390    SZ 00000010    NO 00000001        80000208    000FFDF8
     01A290    SZ 00000010    NO 00000001        800001A0    000FFB98
     01A260    SZ 00000010    NO 00000001        80000058    000FFB40
     01A420    SZ 00000010    NO 00000001        80000008    000FF1B0
     01A3C0    SZ 00000010    NO 00000001        80000070    000FF0C8
     01A350    SZ 00000010    NO 00000001        80000058    000FF070
     01A2F0    SZ 00000010    NO 00000001        8000007A    000FFDB0
     017260    SZ 00000010    NO 00000001        800000E8    000D1918
     01A3F0    SZ 00000010    NO 00000001        80000078    000FF138

DEB

017040    00000C2F 00000C2E 00000C2F 00000C2E    00000C2E 00000000 3200000C 00002BE0    *................................*
017060    0E000000 000169E8 04016E24 88000000    8F000000 01000000 18000000 EF0D1FA0    *.......Y........................*
017080    04017040 10001444 0000000C 0000000D    0013002B 0001F0F4 C2C2C2C1 C3C41800    *.....................04BBBACD..*
0170A0    0A0218E7 07FF0000 00830600 C612211F                                           *...X......O.F.......205A........*

DEB

016E00    00000C2E 00000C2F 00000C2E 00000C2E    00000C2F 00000000 00000000 00019BF0    *................................*
016E20    0D000000 030169E8 04016F94 C8000000    0F000000 01000000 18000000 FF085180    *.......Y....H...................*
016E40    02016E00 33001830 00010000 C1D9C1D2    C3C3C002 00000000 FF040000 00005A34    *............ARAKCC..............*
016E60    00000000 00016F68                                                             *...........8....................*

DEB

016E60                                           00000C2F 00000C2E 00000C2E 000FED80    *...........8....................*
016E80    00000C2E 00000000 00000000 00009BE0    00000000 040169E8 04017624 48000000    *.........................Y......*
016EA0    00000000 01000000 18000000 EF08512C    02016E70 33001800 00010000 C1D8C1C3    *............................AQAC*
016EC0    C3C9C3C3 00000000 00000000 00000000    00081B10 000000D0                      *CICC............................*

DEB

017600    00000C2E 00000C2E 00000C2E 000FED80    00000C2E 00000000 00000000 00009BE0    *................................*
017620    00000000 040169E8 04000000 C8000000    00000000 01FFFFFF 18000000 EF085210    *.......Y....H...................*
017640    02017600 33001060 00010000 C1D8C1C3    C3C9C3C3 00017970 00000000 04FF1294    *............AQACCICC............*
017660    0000F303 C7017970                                                             *..TLG...........................*

TIOT   JOB  A0001854   STEP EXEC1
     DD              14040100    PGM=*.DD      001E0C00    80001204
     DD              14040100    SYSUDUMP      000F3700    80001444
     DD              14040100    SYNFILE       00123600    80001294
     DD              14040100    SYNPRINT      000E3900    80001444
     DD              14020100    READ002       00123C00    80001800
     DD              14040100    WRITANS       0012F00     80001830
     DD              14050100    ANSFILE       00103100    80001860
     DD              14050100    WRIT001       00103500    80001890
```

Figure 24b: MVT SYSUDUMP - The XL List.

59

```
               14040100   WRITELN    00210600   80001444
 DD            14040100   READCRD    001D3700   800014D4

MSS            ************ SPQE ************    *************** DQE ***************    ******* FQE ********
               FLGS  NSPQE    SPID    DQE        BLK      FQE      LN       NDQE        NFQE       LN

      0195A8   00    019680   251    01ADC8     00082800 00082800 00004000 00000000    00000000   00000480
      019680   80    01A6C8   000    017960
      017960   60    000000   000    01ADF0     000D1000 000D1768 00000800 00018480    000D1000   00000048
                                                                                       00000000   00000430
                                                00000000 00000000 00001000 00017190    00000000   00000678
                                                000CF000 000CF000 00001000 00017F88    00000000   00000030
                                                000CF000 000CE000 00001000 00017250    00000000   00000678
                                                000CD800 000CD800 00000800 00000000    00000000   000000C8
      01A6C8   40    000000   252    018FC8     000D1800 000D1A00 00000800 00017CB0    000D1800   000005A0
                                                                                       00000000   00000118
                                                000CD000 000CD000 00000800 00017968    00000000   000001F0
                                                000CC800 000CC800 00000800 00000000    00000000   000001A0

D-PQE  0001A6C8   FIRST 00018E88   LAST 0001AE88
PQF  018F88    FFB 000B6800   LFB 000B6800   NPQ 00000000   PPQ 00000000
               TCB 00016300   RSI 0001F800   RAD 00082800   FLG 00

FBQE 0B6800    NFB 00018E88   PFB 00018E88   SZ 00016000

QCB TRACE

MAJ 01A6F8    NMAJ 00018A00    PMAJ 0000A2C0    FMIN 00019480    NM   SYSDSN

MIN 019480    FQFL 00019700    PMIN 0001A6F8    NMIN 00019070    NM FF   CTSRCH

              NQFL 00000000    PQEL 00019480    TCB  00016300    SVRB 0001AF20

MIN 019070    FQFL 00019440    PMIN 00019480    NMIN 00018FA8    NM FF   SYS1.SORTLIB

              NQFL 00000000    PQEL 80019070    TCB  00016300    SVRB 00016F20

MIN 018FA8    FQFL 00019200    PMIN 00019070    NMIN 00018FE8    NM FF   SYS1.LINKLIB

              NQFL 00000000    PQEL 8001BEA8    TCB  00016300    SVRB 00016F20

MAJ 017978    NMAJ 00000000    PHAJ 0001BA00    FMIN 00017758    NM   SYSIEA01

MIN 017758    FQFL 00017CA0    PMIN 00017978    NMIN 00000000    NM FO   IEA

              NQEL 00000000    PQFL 00017758    TCB  000169E8    SVRB 00016F68

SAVE AREA TRACE

PDS       WAS ENTERED VIA LINK

SA   0D1780   WD1 00000000   HSA 00000000   LSA 00082C80   RET 000085DA   EPA 01082C80   R0  FD000006
```

To next SAVE area

Figure 24c: MVT SYSUDUMP - The SAVE Area Trace.


```
              R1   000D17F8   R2   0001ACCA   R3   5001ACDC   R4   00015FCA   R5   00016300   R6   0001607A
              R7   00017008   R8   0001BCD0   R9   00008000   R10  00000114   R11  00015FCA   R12  600FF41A

SA   082C80   WD1  00070003   HSA  000D1780   LSA  00090003   RET  4C0020AA   EPA  00090003   R0   50084F76
              R1   12001918   R2   00082069   R3   0008512C   R4   00088944   R5   40082C86   R6   00083CA6
              R7   00082FDB   R8   0000005C   R9   000830A0   R10  00082E98   R11  40001900   R12  000FF138

INTERRUPT AT 003716

PROCEEDING BACK VIA REG 13

SA   082C80   WD1  00070003   HSA  000D1780   LSA  00080003   RET  4C0020AA   EPA  00090003   R0   50084F76
              R1   12001918   R2   00082069   R3   0008512C   R4   00088944   R5   40082C86   R6   00083CA6
              R7   00082FD8   R8   0000005C   R9   000830A0   R10  00082F98   R11  40001900   R12  000FF138

PDS      WAS ENTERED VIA LINK

SA   0D1780   WD1  00000000   HSA  00000000   LSA  00082C80   RET  000085DA   EPA  01082C80   R0   FD000006
              R1   000D17F8   R2   0001ACCA   R3   5001ACDC   R4   00015ECA   R5   00016300   R6   0001607A
              R7   00017008   R8   00018C80   R9   000080D0   R10  00000114   R11  00015ECA   R12  600FF41A

REGS AT ENTRY TO ABEND

     FLTR 0-6      000014440001ACB4      000000010001ACEC               00016FA000000000      00090009D7C4E240

     REGS 0-7      00000001   000D14EC   0008512C   000D14E8            000CEFFF   000CEFFF   00000051   00000028
     REGS 8-15     000FF138   00000000   000D0AFF   000D0995            00084C86   00082C80   40001900   600049DA

LOAD MODULE   PDS

082C80   90ECD00C 05305000 302F18CD 4100302A   50DC0008 07005840 301E58C0 302247F0   *.......................  .......0*
082CA0   44F20000 00083C86 00084C86 4C00205C   00070003 000D1780 00080003 4C0020AA   *.2.............................*
082CC0   00090003 50084F76 12001918 00082069   0008512C 00088944 40082C86 00083CA6   *........................ .......*
082CE0   00082FD8 0000005C 000830A0 00082F98   40001900 000FF138 00110003 5C002058   *................ .....l.........*
082D00   F0F8F140 40E9D6D2 C8C1F460 F0F0F3F8   60F1F8F8 F8404040 40404040 40404040   *081  ZOKHA4.0038,1888           *
082D20   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040   *                                *
082D40   40404040 40404040 40404040 40404040   40404040 40F0F0F0 F0F2F040 4D0022D8   *                      000020 ...Q*
082D60   4D00220C F0F6F140 40E9C1D2 C8C1F860   F0F0F2F3 60F1F1F2 F440F2F1 40F20603   *....061  ZAKHAB.0023.1124 21 SOL*
082D80   E5C5D5E3 40C5E7E3 D9C1C3E3 C9D6D540   D6C640C1 40C3C8C5 D3C1E3C5 40C3D6D4   *VENT EXTRACTION OF A CHELATE COM*
082DA0   D7D6E4D5 C440D6C6 40C140E3 D9C1C3C5   40404040 40404040 404AF0F0 F0F2F534   *POUND OF A TRACE          .00025.*
082DC0   00150015 4000374C 008A5000 000008BA   680BB874 008AC008 8AE80000 00088994   *..........................Y......*
082DE0   088A4000 00000888 00000000 0008ZC00   0000088C 7C0088SC 00000008 897C0889   *.. .............................*
082E00   F40B8BA0 00000000 00000000 00000000   00000000 00000000 00000000 00000000   *4...............................*
082E20   00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000   *................................*
082E40   00000011 AA000000 00007FFF C980020F   CDF1C547 D207C602 00088978 00010000   *...........I.K..IF.K.I..........*
082E60   00000007 00000000 00184040 40400000   00000000 0000002C F14021C6 4780C640   *..........      ..........I .F.F *
082E80   404040C6 000888A0 00088C8C 000080DF   00088978 000CF000 4780C6F9 F9F9F9F9   *   F.................0...F99999*
082EA0   F9F9F9F9 F9F9F9F9 F9F9F9F9 00C9D5D8   F4C9D9E8 40D5064B 40404040 40C8C1E2   *999999999999.INQUIRY NO.     HAS*
082EC0   40D50640 C8C5C1C4 C5D940C3 C1D9C44B   C3E3E2D9 C3C4C1E3 C5E3C5D9 D4E24096   * NO HEADER CARD.CTSRCDATETERMS O*
082EE0   D540E3C1 D7C5C510 4780C3E0 5030C504   58B020C4 47F0C584 5830C4FC 00002000   *N TAPE...C...F....D.0I...D.....*
082F00   00004000 07000700 00088BA0 000080B0   0000165B F0F0F1F0 F10F0008 58F0F030   *.. ............0....08101....00.*
082F20   00000000 0057001F F0F8F140 40E907C3   C6C1E760 F0F0F5F9 60F0F2F9 F9404040   *........081  ZPCFAX.0059.0299   *
```

Figure 24d: MVT SYSUDUMP - The SAVE Area Trace.

Figure 25: MVT SYSUDUMP - The DEB QUEUE.

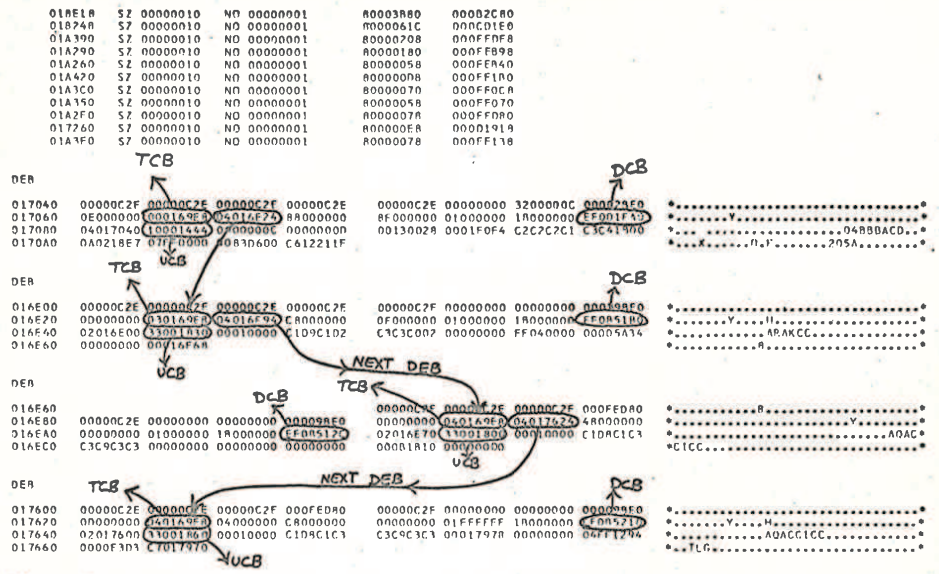

Figure 26: MVT SYSUDUMP - The LOAD LIST.